



RAID 2019

PROCEEDINGS



22nd International Symposium on Research in Attacks, Intrusions and Defenses

September 23–25, 2019
Beijing, China

**Proceedings of the
22nd International Symposium on
Research in Attacks, Intrusions and Defenses**

**September 23–25, 2019
Beijing, China**

Gold Sponsors



Silver Sponsors



Bronze Sponsor



Organizers



© 2019 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 978-1-939133-07-6

Organizing Committee

Honorary Chairs

Dengguo Feng, *Institute of Software, CAS*
Dan Meng, *State Key Laboratory of Information Security,
Institute of Information Engineering, CAS*

General Chair

Purui Su, *Institute of Software, CAS*

Vice General Chairs

Kai Chen, *Institute of Information Engineering, CAS*
Qi Li, *Tsinghua University*

Program Committee Chair

Thorsten Holz, *Ruhr-Universität Bochum*

Program Committee Co-Chair

Manuel Egele, *Boston University*

Local Arrangement Chair

Yi Yang, *Institute of Software, CAS*

Publication Chair

Zhi Wang, *Florida State University*

Travel Grant Chair

Kun Sun, *George Mason University*

Publicity Chair

Chao Zhang, *Tsinghua University*

Program Committee

Kevin Borgolte, *Princeton University*
Lorenzo Cavallaro, *King's College London*
Kai Chen, *Institute of Information Engineering, Chinese
Academy of Sciences*
Adrian Dabrowski, *UC Irvine*
Sanjeev Das, *University of North Carolina at Chapel Hill*
Martin Degeling, *Ruhr University Bochum*
Tudor A. Dumitraş, *University of Maryland*
Carrie Gates, *Bank of America*
Guofei Gu, *Texas A&M University*
Johannes Kinder, *Bundeswehr University Munich*
Engin Kirda, *Northeastern University*
Katharina Krombholz, *CISPA*
Andrea Lanzi, *University of Milan*
Tim Leek, *MIT Lincoln Laboratory*
Corrado Leita, *Lastline*
Wojciech Mazurczyk, *Warsaw University of Technology*
Jelena Mirkovic, *USC-ISI*
Nick Nikiforakis, *Stony Brook University*

Rishab Nithyanand, *University of Iowa*
Paul Pearce, *Georgia Tech / Facebook*
Roberto Perdisci, *University of Georgia*
Jason Polakis, *University of Illinois at Chicago*
Michalis Polychronakis, *Stony Brook University*
Kaveh Razavi, *Vrije Universiteit Amsterdam*
Konrad Rieck, *TU Braunschweig*
William Robertson, *Northeastern University*
Christian Rossow, *CISPA*
Ahmad Sadeghi, *TU Darmstadt*
Brendan Saltaformaggio, *Georgia Institute of Technology*
Angelos Stavrou, *George Mason University*
Kurt Thomas, *Google*
Stijn Volckaert, *KU Leuven*
Zack Weinberg, *Carnegie Mellon University*
Michael Weissbacher, *Square Inc.*
Dongpen Xu, *University of New Hampshire*
Chao Zhang, *Tsinghua University*

Steering Committee

Johanna Amann, *International Computer Science Institute*
Michael Bailey, *University of Illinois at Urbana Champaign*
Davide Balzarotti, *Eurecom Graduate School and Research Center*
Marc Dacier, *Eurecom Graduate School and Research Center*
Zhiqiang Lin, *The Ohio State University*
Mathias Payer, *École Polytechnique Fédérale de Lausanne (EPFL)*
Michalis Polychronakis, *Stony Brook University*
Angelos Stavrou, *George Mason University*

Message from the RAID 2019 Program Co-Chairs

Welcome to the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)!

We are proud to present the proceedings of RAID 2019—this year is an important step for RAID, we are excited to be in China for the first time in the entire history of the symposium. This year, the symposium received 166 submissions of which 37 were accepted (22% acceptance rate). As in previous years, a double-blind reviewing process was used to ensure that the reviewers remained unaware of the authors' names or affiliations during the discussion. Each paper received at least three reviews and the final decision for each paper was made during an in-person PC meeting following the IEEE Symposium on Security and Privacy in San Francisco (CA), in May 2019.

The quality and commitment of the Program Committee is paramount to the success of any conference. This year, we had a very international PC—more than 40% of the PC members were from outside the US and about 20% were from government, industry, or a mix. The PC was a mix of veteran PC members who have already served several times on the RAID PC, and many new faces that 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.

Last year, RAID introduced an annual Best Paper award. A subset of five PC members was 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!

An exciting change this year is that we switched to an open access license to publish the proceedings of the conference. For the first time, the proceedings are published by USENIX, and all papers are available online on the opening day of RAID 2019. 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 its organization. The sponsors, therefore, play a very important role and ensure that the registration fees remain very reasonable. We want to take this opportunity to thank our Gold Sponsors, Qi An Xin Group, Baidu Security, and Dawning Information Industry Co., Ltd.; our Silver Sponsors, Kryptowire, Topsec Network Technology Inc., and Inspur Power Commercial Systems Co. Ltd.; and our Bronze Sponsor, Nsfocus Information Technology Co., Ltd., for their generous sponsorships to RAID 2019!

We are, of course, very grateful to the Honorary Chairs Dengguo Feng from the Institute of Software at CAS and Dan Meng from the Institute of Information Engineering at CAS, the General Chair Purui Su from the Institute of Software at CAS, the Vice General Chairs Kai Chen from the Institute of Information Engineering at CAS and Qi Li from Tsinghua University, and their assembled team for ensuring that the conference runs smoothly. Special thanks go to the Local Arrangements Chair, Yi Yang, also from the Institute of Software, CAS; to the Publication Chair, Zhi Wang, from Florida State University; to the Travel Grant Chair Kun Sun from George Mason University; and to the Publicity Chair, Chao Zhang from Tsinghua University. Without their help, this conference could not have taken place. 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!

Thorsten Holz and Manuel Egele
RAID 2019 PC Chairs

**RAID 2019: 22nd International Symposium on
Research in Attacks, Intrusions and Defenses**
September 23–25, 2019
Beijing, China

Software Security

- Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing** 1
Jinghan Wang, *University of California, Riverside*; Yue Duan, *Cornell University*; Wei Song, Heng Yin,
and Chengyu Song, *University of California, Riverside*
- On Design Inference from Binaries Compiled using Modern C++ Defenses**17
Rukayat Ayomide Erinfolami, Anh T Quach, and Aravind Prakash, *Binghamton University*
- DECAF++: Elastic Whole-System Dynamic Taint Analysis** 31
Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin, *University of California, Riverside*

Understanding Attacks

- Towards a First Step to Understand the Cryptocurrency Stealing Attack on Ethereum** 47
Zhen Cheng, *Zhejiang University*; Xinrui Hou, *Xidian University*; Runhuai Li and Yajin Zhou, *Zhejiang University*;
Xiapu Luo, *The Hong Kong Polytechnic University*; Jinku Li, *Xidian University*; Kui Ren, *Zhejiang University*
- Fingerprinting Tooling used for SSH Compromisation Attempts** 61
Vincent Ghiütte, Harm Griffioen, and Christian Doerr, *TU Delft*
- Timing Patterns and Correlations in Spontaneous SCADA Traffic for Anomaly Detection** 73
Chih-Yuan Lin and Simin Nadjm-Tehrani, *Linköping Universitet*

Defenses

- USBESAFE: An End-Point Solution to Protect Against USB-Based Attacks** 89
Amin Kharraz, *University of Illinois at Urbana Champaign*; Brandon L. Daley and Graham Z. Baker, *MIT Lincoln
Laboratory*; William Robertson and Engin Kirda, *Northeastern University*
- Minimal Kernel: An Operating System Architecture for TEE to Resist Board Level Physical Attacks** 105
Shijun Zhao, *Institute of Software Chinese Academy of Sciences*; Qianying Zhang, *Capital Normal University
Information Engineering College*; Yu Qin, Wei Feng, and Dengguo Feng, *Institute of Software Chinese Academy
of Sciences*
- ScaRR: Scalable Runtime Remote Attestation for Complex Systems** 121
Flavio Toffalini, *Singapore University of Technology and Design*; Eleonora Losiouk and Andrea Biondo, *University of
Padua*; Jianying Zhou, *Singapore University of Technology and Design*; Mauro Conti, *University of Padua*

Embedded Security

- Toward the Analysis of Embedded Firmware through Automated Re-hosting** 135
Eric Gustafson, *UC Santa Barbara*; Marius Muench, *EURECOM*; Chad Spensky, Nilo Redini, and Aravind Machiry,
UC Santa Barbara; Yanick Fratantonio, Davide Balzarotti, and Aurelien Francillon, *EURECOM*; Yung Ryn Choe,
Sandia National Laboratories; Christopher Kruegel and Giovanni Vigna, *UC Santa Barbara*
- CRYPTOREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices** 151
Li Zhang, *Jinan University*; Jiongyi Chen, *The Chinese University of Hong Kong*; Wenrui Diao and Shanqing Guo,
Shandong University; Jian Weng, *Jinan University*; Kehuan Zhang, *The Chinese University of Hong Kong*
- PAtt: Physics-based Attestation of Control Systems** 165
Hamid Reza Ghaeini, *Singapore University of Technology and Design*; Matthew Chan, *Rutgers University*; Raad
Bahmani and Ferdinand Brasser, *TU Darmstadt*; Luis Garcia, *University of California, Los Angeles*; Jianying Zhou,
Singapore University of Technology and Design; Ahmad-Reza Sadeghi, *TU Darmstadt*; Nils Ole Tippenhauer, *CISPA,
Helmholtz Center for Information Security*; Saman Zonouz, *Rutgers University*

(continued on next page)

COMA: Communication and Obfuscation Management Architecture 181
Kimia Zamiri Azar, Farnoud Farahmand, Hadi Mardani Kamali, Shervin Roshanisefat, and Houman Homayoun,
George Mason University; William Diehl, *Virginia Tech*; Kris Gaj and Avesta Sasan, *George Mason University*

Privacy Enhancing Techniques

PRO-ORAM: Practical Read-Only Oblivious RAM 197
Shruti Tople, *Microsoft*; Yaoqi Jia, *Ziliqa Research*; Prateek Saxena, *NUS*

The DUSTER Attack: Tor Onion Service Attribution Based on Flow Watermarking with Track Hiding 213
Alfonso Iacovazzi, *ST Engineering-SUTD Cyber Security Laboratory, Singapore University of Technology and Design*;
Daniel Frassinelli, *CISPA, Helmholtz Center for Information Security, Germany*; Yuval Elovici, *Department of Software
and Information Systems Engineering and Cyber Security Research Center, Ben-Gurion University of the Negev, Israel,*
and *iTrust—Centre for Research in Cyber Security, Singapore University of Technology and Design, Singapore*

TALON: An Automated Framework for Cross-Device Tracking Detection 227
Konstantinos Solomos, *FORTH*; Panagiotis Ilia, *University of Illinois at Chicago*; Sotiris Ioannidis, *FORTH*;
Nicolas Kourtellis, *Telefonica Research*

Android Security I

Analysis of Location Data Leakage in the Internet Traffic of Android-based Mobile Devices 243
Nir Sivan, Ron Bitton, and Asaf Shabtai, *Ben Gurion University of the Negev*

Kindness is a Risky Business: On the Usage of the Accessibility APIs in Android 261
Wenrui Diao, *Shandong University*; Yue Zhang and Li Zhang, *Jinan University*; Zhou Li, *University of California, Irvine*;
Fenghao Xu, *The Chinese University of Hong Kong*; Xiaorui Pan, *Indiana University Bloomington*; Xiangyu Liu, *Alibaba
Inc.*; Jian Weng, *Jinan University*; Kehuan Zhang, *The Chinese University of Hong Kong*; XiaoFeng Wang, *Indiana
University Bloomington*

Automatic Generation of Non-intrusive Updates for Third-Party Libraries in Android Applications 277
Yue Duan, *Cornell University*; Lian Gao, Jie Hu, and Heng Yin, *University of California Riverside*

Machine Learning & Watermarking

Exploiting the Inherent Limitation of L_0 Adversarial Examples 293
Fei Zuo, Bokai Yang, Xiaopeng Li, Lannan Luo, and Qiang Zeng, *University of South Carolina*

NLP-EYE: Detecting Memory Corruptions via Semantic-Aware Memory Operation Function Identification 309
Jianqiang Wang, *Shanghai Jiao Tong University*; Siqi Ma, *CSIRO DATA61*; Yuanyuan Zhang and Juanru Li, *Shanghai
Jiao Tong University*; Zheyu Ma, *Northwestern Polytechnical University*; Long Mai, Tiancheng Chen, and Dawu Gu,
Shanghai Jiao Tong University

Robust Optimization-Based Watermarking Scheme for Sequential Data 323
Erman Ayday, *Case Western Reserve University and Bilkent University*; Emre Yilmaz, *Case Western Reserve University*;
Arif Yilmaz, *Bilkent University*

Malware

Smart Malware that Uses Leaked Control Data of Robotic Applications: The Case of Raven-II Surgical Robots 337
Keywhan Chung and Xiao Li, *University of Illinois at Urbana-Champaign*; Peicheng Tang, *Rose-Hulman Institute of
Technology*; Zeran Zhu, Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, and Thenkurussi Kesavadas, *University of Illinois
at Urbana-Champaign*

SGXJail: Defeating Enclave Malware via Confinement 353
Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss, *Graz University of Technology*

Fluorescence: Detecting Kernel-Resident Malware in Clouds 367
Richard Li, *University of Utah*; Min Du, *University of California Berkeley*; David Johnson, Robert Ricci, Jacobus Van der
Merwe, and Eric Eide, *University of Utah*

DNS Security

Now You See It, Now You Don't: A Large-scale Analysis of Early Domain Deletions 383
Timothy Barron, Najmeh Miramirkhani, and Nick Nikiforakis, *Stony Brook University*

HinDom: A Robust Malicious Domain Detection System based on Heterogeneous Information Network with Transductive Classification 399

Xiaoqing Sun, Mingkai Tong, and Jiahai Yang, *Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China*; Liu Xinran, *National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China*; Liu Heng, *China Electronics Cyberspace Great Wall Co., Ltd, Beijing, China*

DomainScouter: Understanding the Risks of Deceptive IDNs 413

Daiki Chiba, Ayako Akiyama Hasegawa, and Takashi Koide, *NTT Secure Platform Laboratories*; Yuta Sawabe and Shigeki Goto, *Waseda University*; Mitsuaki Akiyama, *NTT Secure Platform Laboratories*

Attacks

Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC 427

Wei Song, *Institute of Information Engineering, CAS*; Peng Liu, *Pennsylvania State University*

Time and Order: Towards Automatically Identifying Side-Channel Vulnerabilities in Enclave Binaries 443

Wubing Wang, Yinqian Zhang, and Zhiqiang Lin, *The Ohio State University*

Application level attacks on Connected Vehicle Protocols 459

Ahmed Abdo, Sakib Md Bin Malek, and Zhiyun Qian, *University of California, Riverside*; Qi Zhu, *Northwestern University*; Matthew Barth and Nael Abu-Ghazaleh, *University of California, Riverside*

Security in Data Centers and the Cloud

S3: A DFW-based Scalable Security State Analysis Framework for Large-Scale Data Center Networks 473

Abdulhakim Sabur, Ankur Chowdhary, and Dijiang Huang, *Arizona State University*; Myong Kang, Anya Kim, and Alexander Velazquez, *Naval Research Lab*

Container-IMA: A privacy-preserving Integrity Measurement Architecture for Containers 487

Wu Luo, Qingni Shen, Yutang Xia, and Zhonghai Wu, *Peking University, Beijing, China*

Fingerprinting SDN Applications via Encrypted Control Traffic 501

Jiahao Cao, *Tsinghua University and George Mason University*; Zijie Yang, *Tsinghua University*; Kun Sun, *George Mason University*; Qi Li, Mingwei Xu, *Tsinghua University*; Peiyi Han, *Beijing University of Posts and Telecommunications*

Android Security II

Exploring Syscall-Based Semantics Reconstruction of Android Applications 517

Dario Nisi, *EURECOM*; Antonio Bianchi, *University of Iowa*; Yanick Fratantonio, *EURECOM*

Towards Large-Scale Hunting for Android Negative-Day Malware 533

Lun-Pin Yuan, *Penn State University*; Wenjun Hu, *Palo Alto Networks Inc.*; Ting Yu, *Qatar Computing Research Institute*; Peng Liu and Sencun Zhu, *Penn State University*

DroidScraper: A Tool for Android In-Memory Object Recovery and Reconstruction 547

Aisha Ali-Gombe, *Towson University*; Sneha Sudhakaran, *Louisiana State University*; Andrew Case, *Volatility Foundation*; Golden G. Richard III, *Louisiana State University*

Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing

Jinghan Wang[†], Yue Duan[‡], Wei Song[†], Heng Yin[†], and Chengyu Song[†]

[†]UC Riverside [‡]Cornell University

[†]{jwang131,wsong008}@ucr.edu, {heng,csong}@cs.ucr.edu [‡]yd375@cornell.edu

Abstract

Coverage-guided greybox fuzzing has become one of the most common techniques for finding software bugs. Coverage metric, which decides how a fuzzer selects new seeds, is an essential parameter of fuzzing and can significantly affect the results. While there are many existing works on the effectiveness of different coverage metrics on software testing, little is known about how different coverage metrics could actually affect the fuzzing results in practice. More importantly, it is unclear whether there exists one coverage metric that is superior to all the other metrics. In this paper, we report the first systematic study on the impact of different coverage metrics in fuzzing. To this end, we formally define and discuss the concept of *sensitivity*, which can be used to theoretically compare different coverage metrics. We then present several coverage metrics with their variants. We conduct a study on these metrics with the DARPA CGC dataset, the LAVA-M dataset, and a set of real-world applications (a total of 221 binaries). We find that because each fuzzing instance has limited resources (time and computation power), (1) each metric has its unique merit in terms of flipping certain types of branches (thus vulnerability finding) and (2) there is no grand slam coverage metric that defeats all the others. We also explore combining different coverage metrics through cross-seeding, and the result is very encouraging: this pure fuzzing based approach can crash at least the same numbers of binaries in the CGC dataset as a previous approach (Driller) that combines fuzzing and concolic execution. At the same time, our approach uses fewer computing resources.

1 Introduction

Greybox fuzzing is a state-of-the-art program testing technique that has been widely adopted by both mainstream companies such as Google [45] and Adobe [47], and small startups (e.g., Trail of Bits [48]). In the DARPA Cyber Grand Challenge (CGC), greybox fuzzing has been demonstrated to be more effective compared to other alternatives such as symbolic execution and static analysis [8, 15, 34, 37, 39].

Greybox fuzzing generally contains three major stages: seed scheduling, seed mutation, and seed selection. From a set of seed inputs, the seed scheduler picks the next seed for testing. Then, more test cases are generated based on the scheduled seeds through mutation and crossover in the seed mutation stage. Finally, test cases of good quality are selected as new seeds to generate more test cases in the future rounds of fuzzing. Among these stages, seed selection is the most important one as it differentiates greybox fuzzing from blackbox fuzzing and determines the goal of the fuzzer. For example, when the goal is to improve coverage, we use a coverage metric to evaluate the quality of a test case, and when the goal is to reach a particular code point, we can use distance to evaluate the quality of a test case [2]. Note that although previous studies [14, 17] have shown that better coverage of test suite is not directly related to a better quality of the tested software, the observation that under-tested code is more likely to have bugs still holds. For this reason, coverage-guided greybox fuzzing still works very well in practice.

Although various techniques have been proposed to improve greybox fuzzing at the seed scheduling stage [2, 3, 27, 29] and the seed mutation stage [21, 28, 29, 37, 54], very few efforts focus on improving seed selection. Honggfuzz [40] only counts the number of basic blocks visited. AFL [38] utilizes an improved branch coverage that also counts how many times a branch is visited. Angora [7] further extends the branch coverage to be context-sensitive. More importantly, many critical questions about coverage metrics remain unanswered.

First, *how do we uniformly define the differences among different coverage metrics?* Coverage metrics can be categorized into two major categories: code coverage and data coverage. Code coverage metrics evaluate the uniqueness among test cases at the code level, such as line coverage, basic block coverage, branch/edge coverage, and path coverage. Data coverage metrics, on the other hand, try to distinguish test cases from a data accessing perspective, such as memory addresses, access type (read or write), and access sequences. While many new metrics have been proposed individually in recent works,

there is no systematic and uniform way to characterize the differences among them. Apparently, different coverage metrics have very distinct capability of differentiating test cases, which we refer to as *sensitivity*. For example, block coverage could not tell the difference between visits to the same basic block from different preceding blocks, while branch coverage can. Therefore, branch coverage is more sensitive than block coverage. A systematic and formal definition of *sensitivity* is essential as it can not only tell the differences among current metrics but also guide future research to propose more metrics.

Second, *is there an optimal coverage metric that outperforms all the others in coverage-guided fuzzing?* Although sensitivity provides us a way to compare the capability of two coverage metrics in discovering interesting inputs, a more sensitive coverage metric does not always lead to better fuzzing performance. More specifically, fuzzing can be modeled as a multi-armed bandit (MAB) problem [51] where each stage (seed selection, scheduling, and mutation) has multiple choices, and the ultimate goal is to *find more bugs* with a limited time budget. A more sensitive coverage metric may select more inputs as seeds, but the fuzzer may not have enough time budget to schedule all the seeds or mutate them sufficiently. Implementation details such as how coverage is actually measured can further complicate this problem. For instance, a previous study [12] has shown that hash collisions could reduce the actual sensitivity of a coverage metric. A systematic evaluation is essential to understand the relationship between sensitivity and fuzzing performance better.

Third, *is it a good idea to combine different metrics during fuzzing?* Hypothetically, if different coverage metrics have their own merits during fuzzing, then it would make sense to combine them so that different metrics could contribute differently. This question is also crucial as it motivates different thinking and may lead to strategies for improving fuzzing.

To answer the questions mentioned above, we conduct the first systematic study on the impact of coverage metrics on the performance of coverage-guided fuzzing. In particular, we formally define and discuss the concept of *sensitivity* to distinguish different coverage metrics. Based on the different levels of sensitivity, we then present several representative coverage metrics, namely “basic branch coverage,” “context-sensitive branch coverage,” “n-gram branch coverage,” and “memory-access-aware branch coverage,” as well as their variants. Finally, we implement six coverage metrics in a widely-used greybox fuzzing tool, AFL [38], and evaluate them with large datasets, including the DARPA CGC dataset [4], the LAVA-M dataset [42], and a set of real-world binaries. The highlighted findings are:

- Many of these more sensitive coverage metrics indeed lead to finding more bugs as well as finding them significantly faster.

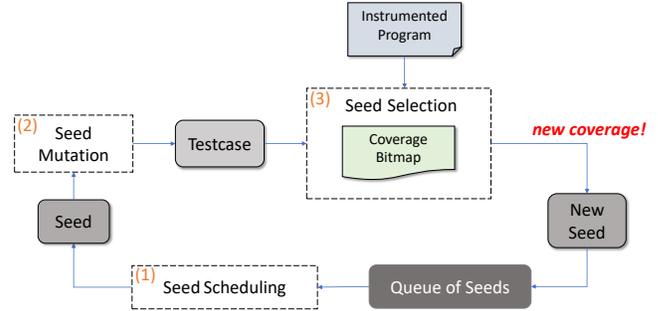


Figure 1: The workflow of coverage-guided greybox fuzzing.

- Different coverage metrics often result in finding different sets of bugs. Moreover, at different times of the whole fuzzing process, the best performer may vary. As a result, there is no grand slam coverage metric that can beat others.
- A combination of these different metrics can help find more bugs and find them faster. Notably, using less computing resources, a combination of fuzzers with different coverage metrics is able to find at least the same amount of bugs in the CGC dataset as Driller, a hybrid fuzzer augmented AFL with concolic execution did [35].

To facilitate further research on this topic, we have made the source code and dataset available at <https://github.com/bitsecurerlab/afl-sensitive>.

2 Background

In this section, we provide the background information about coverage-guided greybox fuzzing, with a focus on the seed selection.

2.1 Coverage-guided Greybox Fuzzing

Coverage-guided greybox fuzzing generates inputs (or test cases) incrementally via a feedback loop. Specifically, there are three main stages, as illustrated in Figure 1. (1) *Seed scheduling*: a seed is picked from a set of seeds according to the scheduling criteria. (2) *Seed mutation*: within a limited time budget, new test cases are generated by performing various mutations on the scheduled seed. (3) *Seed selection*: each generated test case is fed to the program under test and evaluated based on the coverage metric; if the testcase leads to new coverage, it will be selected as a new seed. As this feedback loop continues, more coverage will be reached, and hopefully, a test case will trigger a bug.

2.2 Seed Selection

A seed selection strategy determines the trend and speed of the evolution of the fuzzing process. Essentially, a good seed

selection strategy needs to solve two essential problems: (1) how to collect coverage information and (2) how to measure the quality of test cases.

Coverage Information Collection. AFL instruments the program under test to collect and compute the coverage. There are two instrumentation approaches. When the source code of the program under test is available, a modified `Clang` compiler is used to insert the coverage calculation logic into the compiled executable at assembly level (normal mode) or intermediate representation level (fast mode). When the source code is not available, a modified user-mode QEMU is used to run the binary code of the tested program directly, and the coverage calculation logic is inserted during the binary translation phase. VUzzer [29] uses PIN [41] to perform binary instrumentation to collect the information. HonggFuzz [40] and kAFL [31] use hardware branch tracers like Intel Process Tracing (PT) to collect coverage information and DigTool [25] uses a hypervisor to collect coverage information from OS kernels.

Test Case Measurement. The quality of test cases is measured by leveraging coverage metrics. HonggFuzz [40] and Vuzzer [29] use basic block coverage metric that tracks visits of basic blocks. AFL [38] uses an improved branch coverage metric that could differentiate the visits to the same block from different preceding blocks. LibFuzzer [43] can use either block coverage or branch coverage. A more recent work Angora [7] extends the branch coverage metric with a calling context. Another important aspect is how the metric is really measured. Since coverage is measured during the execution of each test case, fuzzers usually prefer simpler implementations to improve the fuzzing throughput. For example, AFL identifies a branch using a simple hash function (Equation 1). Unfortunately, this approximation could reduce the effective sensitivity of a coverage metric due to hash collisions [12].

3 Sensitivity and Coverage Metrics

In this section, we formally define and discuss the concept of the sensitivity of a coverage metric. Accordingly, we present several coverage metrics that have different sensitivities.

3.1 Formal Definition of Sensitivity

When comparing different coverage metrics, a central question is “is metric A better than metric B ?” To answer this question, we need to take a look at how a mutation-based greybox fuzzer finds a bug. In mutation-based greybox fuzzing, a bug triggering test case is reached via a chain of mutated test cases. In this process, if an intermediate test case is deemed “uninteresting” by a coverage metric, the chain will break and the bug triggering input may not be reached. Based on this observation, we decide to define *sensitivity* as a coverage metric’s ability to preserve such mutation chains.

To formally describe this concept, we first need to define a coverage metric as a function $\mathcal{C} : (\mathcal{P} \times \mathcal{I}) \rightarrow \mathcal{M}$, which produces a measurement $M \in \mathcal{M}$ when running a program $P \in \mathcal{P}$ with an input $I \in \mathcal{I}$. Given two coverage metrics C_i and C_j , C_i is “more sensitive” than C_j , denoted as $C_i \succ C_j$, if

- (1) $\forall P \in \mathcal{P}, \forall I_1, I_2 \in \mathcal{I}, C_i(P, I_1) = C_i(P, I_2) \rightarrow C_j(P, I_1) = C_j(P, I_2)$, and
- (2) $\exists P \in \mathcal{P}, \exists I_1, I_2 \in \mathcal{I}, C_j(P, I_1) = C_j(P, I_2) \wedge C_i(P, I_1) \neq C_i(P, I_2)$

The first condition means, for any program P , if any two inputs I_1 and I_2 produce the same coverage measurement using C_i ; then they must produce the same measurement using C_j , i.e., C_j is always not more discriminative than C_i . The second condition means, there exists at least a program P such that two inputs I_1 and I_2 would produce the same measurement using C_j but different measurements using C_i , i.e., C_i can be more discriminative than C_j .

3.2 Coverage Metrics

In this subsection, we introduce several coverage metrics and their approximated measurement. Then we compare their sensitivity.

Branch Coverage Branch coverage is a straightforward yet effective enhancement over block coverage, which is the most basic one that can only tell which code block is visited. By involving the code block preceding the currently visited one, branch coverage can differentiate the visits of the same code block from different predecessors. Branch here means an edge from one code block to another one.

Ideally, branch coverage should be measured as a tuple (*prev_block*, *cur_block*), where *prev_block* and *cur_block* stand for the previous block ID and the current block ID, respectively. In practice, branch coverage is usually measured by hashing this tuple (as key) into a hash table (e.g., a *hit_count* map). For example, the state-of-the-art fuzzing tool AFL identifies a branch as:

$$block_trans = (prev_block \ll 1) \oplus cur_block \quad (1)$$

where branch ID is calculated as its runtime address. The *block_trans* is then used as the key to index into a hash map to access the *hit_count* of the branch, which records how many times the branch has been taken. After a test case finishes its execution, its coverage information is compared with the global coverage information (i.e., a global *hit_count* map). If the current test case has new coverage, it will be selected as a new seed.

Although branch coverage is widely used in mainstream fuzzers, its sensitivity is low. For instance, considering a

branch within a function that is frequently called by the program (e.g., `strcmp`). When the branch is visited under different calling contexts, branch coverage will not be able to distinguish them.

N-Gram Branch Coverage After incorporating one preceding block in branch coverage, it is intuitive to incorporate more preceding basic blocks as history into the current basic block. We refer to this coverage metric as *n-gram branch coverage*, where n is a configurable parameter that indicates how many continuous branches are considered as one unit, and any changes of them will be distinguished. When $n = 0$, n -gram branch coverage is reduced to block coverage. On the opposite extreme, when $n \rightarrow \infty$, n -gram branch coverage is equivalent to path coverage because it incorporates all preceding branches into the context and any change in the execution path will be treated differently.

Ideally, n -gram branch coverage should be measured as a tuple $(block_1, \dots, block_{n+1})$. For efficiency, we propose to hash the tuple as a key into the `hit_count` map as $(prev_block_trans \ll 1) \oplus curr_block_trans$, where

$$prev_block_trans = (block_trans_1 \oplus \dots \oplus block_trans_{n-1}) \quad (2)$$

In other words, we record the previous $n - 1$ block transitions (calculated as in Equation 1) and XOR them together, left shift 1 bit, and then XOR with the current block transition.

Now an interesting question is: *what is the best value for n ?* If n is too small, it might be almost the same as branch coverage. If n is too large, it may cause seed explosion (a similar phenomenon as path explosion). Fuzzing progress would be even slower due to the enormous amount of seeds.

To answer this question empirically, we adapt AFLFast to n -gram branch coverage where n is set to 2, 4, and 8. We will evaluate these settings in §4.

Context-Sensitive Branch Coverage A function lies between a basic block and a path with respect to the granularity of code. Therefore, calling context is another important piece of information that can be incorporated as part of the coverage metric, which allows a fuzzer to distinguish the same code executed with different data. We refer to this coverage metrics as “context-sensitive coverage metric.”

Ideally, context-sensitive branch coverage metric should be measured as a tuple $(call_stack, prev_block, curr_block)$. For efficiency, we define a calling context $call_ctx$ as a sequence of program locations where function calls are made in order:

$$call_ctx = \begin{cases} 0 & \text{initial value} \\ call_ctx \oplus call_next_insn & \text{if call} \\ call_ctx \oplus ret_to_insn & \text{if ret} \end{cases} \quad (3)$$

Then the key-value pair stored in the bitmap will be now calculated as $call_ctx \oplus block_trans$.

Initially, the calling context value $call_ctx$ is set to 0. Then during the program execution, when encountering a `call` instruction, we XOR the current $call_ctx$ with the instruction’s position immediately next to the call instruction and store the result in $call_ctx$. Similarly, when encountering a `ret` instruction, we XOR the current $call_ctx$ with the return address. In this way, a small value $call_ctx$ efficiently accumulates function calls made in sequence and eliminates function calls that have returned.

Memory-Access-Aware Branch Coverage In addition to leveraging extra control flow information as stated above, data flow information also deserves to be considered. Based on the intuition that a primary focus of fuzzing is to detect memory-corruption vulnerabilities, memory access information can be of great help in measuring coverage. Fundamentally, memory corruption exhibits an erroneous memory access behavior. Therefore, it makes sense to select seeds that exhibit distinct memory access patterns.

In general, this memory-access aware coverage metric is more sensitive than branch coverage. Because if a new test case reaches a branch that has been covered by prior test cases, but at least one new memory location is accessed, this test case will still be considered as “interesting” in memory-access aware coverage metric and kept as a seed.

There can be many ways to characterize memory access patterns. In this paper, we investigate one design option. We instrument memory access operations of the program under test, and define each memory access as a tuple $(type, addr, block_trans)$, where $type$ represents access type (read or write), $addr$ is the accessed memory location, and $block_trans$ means after which branch this memory access is performed.

For efficiency, we propose to calculate the hash key as $(block_trans \oplus mem_ac_ptn)$, where

$$mem_ac_ptn = \begin{cases} mem_addr & \text{if read} \\ mem_addr + half_map_size & \text{if write} \end{cases} \quad (4)$$

Note that reads are distinguished from writes by allocating their keys to different half regions of the map.

Since memory corruption is mainly caused by memory writes, it is meaningful to investigate a variant of memory access coverage: “memory-write-aware branch coverage.” That is, we only instrument and record memory writes, but not reads, making it less sensitive.

3.3 Sensitivity Lattice

Obviously, \succ is a strict partial order, because it is asymmetric (if $C_1 \succ C_2$, by no means $C_2 \succ C_1$), transitive (if $C_1 \succ C_2$ and $C_2 \succ C_3$, then $C_1 \succ C_3$), and irreflexive ($C_i \succ C_i$ is not possible). However, it is not a total order, because it is possible that two metrics are not comparable.

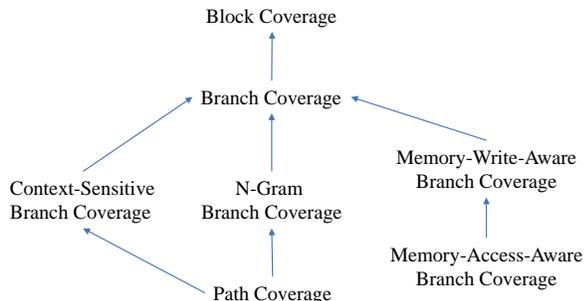


Figure 2: Sensitivity Lattice for Coverage Metrics

As a result, we can draw a sensitivity lattice for the coverage metrics discussed above. Figure 2 shows this lattice. Block coverage is the least sensitive metric, compared to the rest, so it appears on the top. Immediately below is branch coverage. It is more sensitive than block coverage. Then below branch coverage are the three coverage metrics that incorporate different extra information on top of branches.

However, there is no direct comparison among these three coverage metrics, because each of them extends branch coverage in different dimensions: context-sensitive branch coverage incorporates calling context, n-gram branch coverage integrates n-1 preceding block transitions, and memory-access-aware branch coverage includes memory accesses. We can always construct a program and two inputs, such that the same coverage measurement is produced for one metric, but two different coverage measurements are produced for another.

For different values of n in n-gram branch coverage, i -gram is more sensitive than j -gram if $i > j$. Ultimately, path coverage is more sensitive than n-gram branch coverage and context-sensitive branch coverage.

Interestingly enough, we cannot compare path coverage with either memory-access-aware branch coverage or memory-write-aware branch coverage. Path coverage is not necessarily more sensitive because two inputs may follow the same path but exhibit different memory access patterns.

It is noteworthy that the coverage metrics presented here are a few representative ones but are by no means complete. We hope this work can stimulate research on developing more coverage metrics and obtaining a deeper understanding of their impact.

4 Evaluation

To answer the research questions raised in §1, we implemented all the coverage metrics mentioned in §3 except the basic branch coverage, which is already implemented in AFL. We then conducted comprehensive experiments to evaluate the performance of different coverage metrics. Moreover, to better understand how different coverage metrics working together could affect fuzzing; we also evaluate the combination of them.

Table 1: real-world applications used in evaluation.

Applications	Version	Applications	Version
objdump+binutils	2.29	readelf+binutils	2.29
strings+binutils	2.29	nm+binutils	2.29
size+binutils	2.29	file	5.32
gzip	1.8	tiffset+tiff	4.0.9
tiff2pdf+tiff	4.0.9	gif2png	2.5.11
info2cap+ncurses	6.0	jhead	3.0

4.1 Implementation

In this study, since our primary goal is to fuzz binaries without source code, we choose to add our instrumentation based on user-mode QEMU. For instance, for context-sensitive branch coverage, we instrument `call` and `ret` instructions to calculate calling context, and for memory-access-aware branch coverage, we instrument memory reads and writes. For n-gram branch coverage, we use a circular buffer to store the last n-block transitions, for efficient n-gram calculation.

For convenience, in the remainder of this paper, we use the following abbreviations to represent different metrics: `bc` represents the existing branch coverage in AFL, `ct` represents context-sensitive branch coverage, `mw` is short for memory-write-aware branch coverage, and `ma` represents memory-access-aware branch coverage. For n-gram branch coverage, we choose to implement three versions: 2-gram, 4-gram and 8-gram, and use `n2`, `n4`, and `n8` for their abbreviations.

Furthermore, we adopted the seed scheduling of AFLFast [3] in our implementation. Since AFLFast inclines to allocate more fuzzing time on newly generated seeds, different coverage metrics will make a greater impact on fuzzing performance.

4.2 Dataset

We collect binaries from DARPA Cyber Grand Challenge (CGC) [4]. There are 131 binaries from CGC Qualifying Event (CQE) and 74 binaries from CGC Final Event (CFE), and thus 205 ones in total. These binaries are carefully crafted by security experts to utilize different kinds of techniques (e.g., complex I/O protocols and input checksums) and embed vulnerabilities in various ways (e.g., buffer overflow, integer overflow, and use-after-free) to comprehensively evaluate various vulnerability discovery techniques.

We also choose the LAVA-M Dataset [11, 42], which consists of four GNU coreutils programs (`base64`, `md5sum`, `uniq`, and `who`) for evaluation. Each of these binaries is injected with a large number of specific vulnerabilities. As a result, we treat these injected vulnerabilities as ground truth and use them to evaluate different coverage metrics.

In addition to the two datasets above, we also manage to collect 12 real-world applications with their latest versions (Table 1) and assess the performance of different coverage metrics in practice with them.

4.3 Experiment Setup

Our experiments are conducted on a private cluster consisting of a pool of virtual machines. Each virtual machine has a Ubuntu 14.04.1 operating system equipped with 2.3 GHz Intel Xeon processor (24 cores) and 30GB of RAM. As fuzzing is a random process, we followed the recommendations from [20] and performed each evaluation several times for a sufficiently long period.

The tests are mainly focused on the CGC dataset. Specifically, each coverage metric is tested with every binary of the CGC dataset in the dataset using two fuzzing instances for 6 hours (i.e., similar to one instance running 12 hours). We chose this fuzzing time because almost all of the bugs found by fuzzer in CQE and CFE were reported within the first six hours. Moreover, in order to take the randomness of fuzzing into account, each test is performed ten times. The total evaluation time is around 60 days. For binaries with initial sample inputs, we utilized them as initial seeds; otherwise, we used an empty seed.

For the LAVA-M dataset, we tested each coverage metric separately for 24 hours and three times. We used the seed inputs provided by this benchmark and dictionaries of constants extracted from the binary as suggested in [44]. For the real-world dataset, we tested each coverage metric for 48 hours, with two fuzzing instances, and for six times. We used the example inputs from AFL as seeds whenever possible; otherwise with an empty seed.

4.4 Evaluation Metrics

To answer the question of whether there is an optimal coverage metric, we propose three metrics to quantify the experimental results and evaluate the performance of the presented coverage metrics:

- **Unique crashes.** A unique crash during fuzzing implies that a potential bug of the binary has been found. For the CGC dataset, each binary is designed to have a single vulnerability, so we did not perform any crash deduplication. For the LAVA-M dataset, each bug is assigned with a unique ID which is used for crash deduplication. For the real-world dataset, we utilize the hash of each crash’s backtrace for deduplication.
- **Time to crash.** This metric indicates how fast a given binary can be crashed by a fuzzer and is mainly for the CGC dataset. Because a CGC binary only has one vulnerability, this metric can be used to measure the efficiency of fuzzing with different coverage metrics.
- **Seed count.** A more sensitive coverage metric is more likely to convert a testcase into a seed, and thus the number of unique seeds may be larger. Therefore, this metric quantifies the sensitivity of each coverage metric in a practical sense.

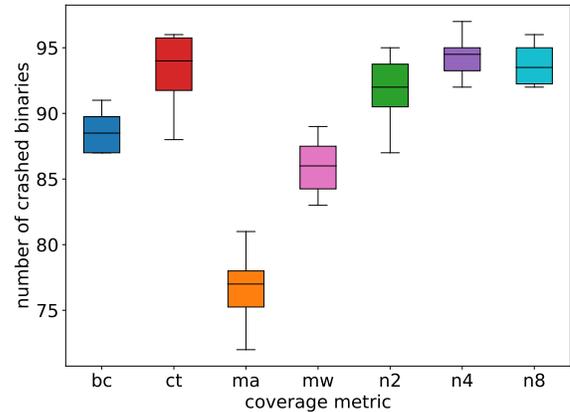


Figure 3: Number of crashed CGC binaries. Because each binary only has one vulnerability, this number is equivalent to the total number of unique crashes.

4.5 Comparison of Unique Crashes

CGC dataset Figure 3 summarizes the number of crashed CGC binaries for each coverage metric across ten rounds of trials. Overall, the baseline metric *bc* crashed about 89 binaries on average and 91 binaries at most. Except for *ma* and *mw*, all other more sensitive coverage metrics (*ct*, *n2*, *n4*, *n8*) outperform *bc*. This result is encouraging: sensitivity does play an important role in finding crashes. However, as demonstrated by *mw* and *ma*, too much sensitivity could also have a negative impact on fuzzing performance. The reason is, more sensitive metrics will select more test cases as seeds (§4.7); when the time budget is limited, each seed will get less time to mutate or not get scheduled at all.

Next, we investigated each coverage metric’s ability to trigger individual bug/crash – is there any bug that is only triggered by one or a subset of the evaluated metrics but not the rest? To answer this question, we conducted a pairwise comparison on crashed binaries (Table 2). For each pair of coverage metrics *i* (in the row) and *j* (in the column), we first count the number of binaries that were only crashed by *i* but not by *j*, denoted as the number after the “/”. Since such

Table 2: Pairwise comparisons (row vs. column) of uniquely crashed CGC binaries.

	bc	ct	ma	mw	n2	n4	n8	others
bc	0/0	0/6	0/15	0/11	0/6	0/6	0/5	0/2
ct	9/13	0/0	9/23	10/15	6/12	3/6	4/8	1/3
ma	2/3	3/4	0/0	2/3	4/6	4/5	2/3	1/1
mw	6/8	2/5	0/12	0/0	3/8	2/7	3/5	0/2
n2	4/4	0/3	7/16	4/9	0/0	0/2	0/2	0/0
n4	9/12	3/5	12/23	8/16	8/10	0/0	0/5	0/1
n8	9/10	6/6	13/20	10/13	7/9	2/4	0/0	0/0
all	19/21	10/14	20/33	19/24	18/23	11/15	9/16	11/0

Table 3: Number of unique bugs found by different coverage metrics on the LAVA-M dataset

	bc	ct	ma	mw	n2	n4	n8	Listed
base64	45	45	44	45	45	45	45	44
md5sum	54	58	35	43	59	58	51	57
uniq	29	29	29	20	29	29	29	28
who	261	255	301	231	166	159	299	2136

differences could be caused by randomness, we conducted a second experiment focusing on the impact of sensitivity. Specifically, during fuzzing, we recorded the chain of seeds that led to each crashing test case. Each chain starts with the initial seed and ends with the crashing test case. Afterward, for each pair of coverage metrics (i, j), we checked whether each seed along the chain selected by i would also be selected by j as seed, without any additional mutation (i.e., fuzzing). In this process, we also discarded additional sensitivity (non-binary `hit_count`) and insensitivity (key collision) introduced in implementation. The result is denoted as the number before the “/” in each cell of Table 2. For example, entry (ct, bc) indicates that there were 13 binaries crashed by ct but not by bc, within which 9 crashes have at least one seed along the crashing chains that will be dropped by bc. Similarly, entry (bc, ct) indicates that 6 binaries crashed by bc are not crashed by ct, of which however none of the seeds along the crashing chain will be dropped by ct. Besides, for a metric k , entry (all, k) indicates the number of binaries crashed by at least one of the other coverage metrics but not by k and entry (k , others) indicates the number of binaries only crashed by k but not by any other coverage metrics. Finally, entry (all, others) indicates the number of binaries crashed by at least one of all the seven coverage metrics.

We can see that the difference between any two coverage metrics is considerable. More importantly, there is no single winner that beats everyone else. Even for ma, although it crashes the smallest amount of binaries in total, it contributes 2 unique crashed binaries beyond bc, and 3, 2, 4, 4, and 2 unique crashed binaries beyond ct, mw, n2, n4, and n8 respectively, of which the crashes have at least one seed along the crashing chains that will be dropped by the other metric. In other words, every coverage metric can make its own and unique contribution. This observation further motivates us to study the combination of different coverage metrics. We will discuss more in §4.8.

LAVA-M dataset Table 3 summarizes the bugs found on LAVA-M dataset by different coverage metrics, while the last column represents the number of bugs listed by LAVA authors. Compare to the CGC dataset, the LAVA-M dataset is not very suitable for our goal. In particular, most injected bugs are protected by a magic number, which is very hard to be solved by random mutation and cannot reflect unique abilities of different coverage metrics. Although we have followed the suggestions from [44] and used dictionaries of constant

Table 4: Number of unique crashes found by different coverage metrics in the real-world dataset.

	bc	ct	ma	mw	n2	n4	n8
gif2png	4	4	3	4	5	4	4
info2cap	1446	1063	481	99	568	933	943
objdump	–	–	–	–	1	1	–
size	–	1	–	–	1	1	1
nm	–	1	–	1	–	–	1

(magic) numbers extracted from the binary, we still cannot rule out the differences caused by not being able to solve the magic number. For binary base64, md5sum, and uniq, the difference between different coverage metrics is small, except for ma in md5sum and mw in uniq. For binary who, it is surprising that in addition to n8, ma also finds much more unique bugs than bc and other three metrics, despite its poor performance on the CGC dataset.

Real-world dataset There are many crashes found for binaries in the real-world dataset. We use the open-source tool afl-collect [49] to de-duplicate these crashes and identify unique crashes. Overall, we have successfully found unique bugs in 5 real-world binaries as listed in Table 4. It is worth noting that for binary objdump, size, and nm, only our newly proposed coverage metrics find unique bugs.

4.6 Comparison of Time to Crash

CGC dataset Since most CGC binaries only contain one bug, we then measure the time to first crash (TFC) for different coverage metrics across the ten rounds of trials. The accumulated number within a 95% confidence of binaries crashed over time is shown in Figure 4. The x-axis presents time in seconds while the y-axis shows the accumulated number of binaries crashed. For example, we can see that n4 almost manages to crash more binaries than other coverage metrics in the first hour (3600 seconds) and ma performs the worst among them. We also see that all of the proposed coverage metrics other than ma and mw can help find crashes in binaries more quickly than the original AFL (bc). Moreover, although n4 does not find the most crashes, it is the best one during the early stage (30 to 90 minutes). After 90 minutes, ct surpasses it and becomes one of the best performers. For the time each coverage metric spends on crashing individual binaries, please refer to Figure 11 in Appendix.

LAVA-M dataset Figure 5 presents the number of unique bugs found over time by different coverage metrics on the four binaries. We can see that the newly proposed coverage metrics outperform bc on all four binaries. Although ma is slower than others, it finally finds the same number of unique bugs on binary base64 and unique. On binary who, ma even finds quite more unique bugs. Moreover, ct and n8 perform stably well across four binaries, and the latter one performs

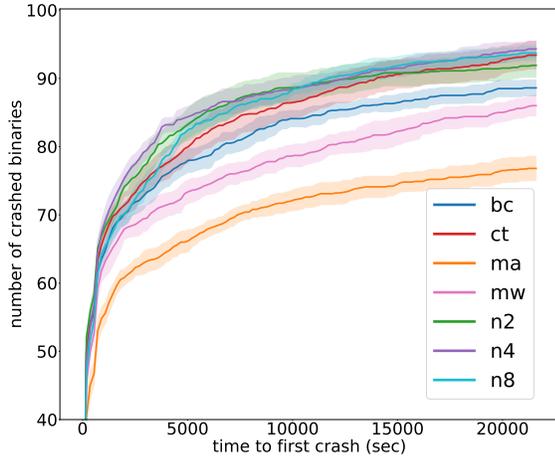


Figure 4: Number of binaries crashed over time during fuzzing on the CGC dataset. The x-axis presents in seconds and the y-axis shows the number of binaries whose TFC (time-to-first-crash) were within that time.

extremely well on binary `who`: it finds the largest number of unique bugs and much faster than the rest.

Real-world dataset Similarly, Figure 6 shows the number of unique bugs over time found by different coverage metrics on the five crashed binaries in the real-world dataset. We can see that except for `info2cap`, `bc` either finds unique bugs much more slowly than others or does not find any bugs at all. In addition, there is no global trend about which coverage metric is the fastest one to find bugs across the five binaries.

4.7 Comparison of Seed Count

CGC dataset We collect the number of seeds selected for each binary using different coverage metrics and report the mean number within a 95% confidence among the ten runs. Figure 7 displays the cumulative distribution of the numbers of generated seeds. A curve closer to the top left in the figure implies that in general fewer seeds are generated for binaries with the corresponding coverage metric.

We had several observations from the result. First, `ma` was significantly more sensitive than the rest coverage metrics. It selects several orders of magnitude more seeds than the others. While most of these seeds are stepping stones for more meaningful mutations that lead to final crashes, too many of them would hurt the fuzzing performance because the differences among most of the seeds are so tiny that they are unlikely to result in any new bug. Second, for `n`-gram branch coverage, as `n` increases from 1 (`bc`) to 8, the number of seeds increases correspondingly, although the lines for `bc` and `n2` are too close to each other. This phenomenon meets our expectation, as $n8 \succ n4 \succ n2 \succ bc$. Third, while in theory, we cannot compare `ct` with `n`-gram regarding their sensitivities, we observe that the seed count distribution for `ct` is between `n4` and `n8`, at least for the CGC dataset. Fourth,

in theory, $ma \succ mw \succ bc$. We indeed observe these relations in the form of seed counts for `ma`, `mw`, and `bc`.

Table 5: The numbers of seeds generated by different coverage metrics on the LAVA-M dataset.

	bc	ct	ma	mw	n2	n4	n8
base64	208	170	16372	200	196	273	425
md5sum	706	497	75323	71131	474	719	4958
uniq	104	52	43928	50178	77	92	153
who	223	144	14183	16511	190	271	470

LAVA-M dataset Table 5 lists seed counts generated by each coverage metric on the four binaries in the LAVA-M dataset. We can see that the observations for the CGC dataset still hold in general, with some outliers. For instance, the seed counts of `ct` on all four binaries are smaller than those of `bc`. These numbers are not statistically significant, given such a small-scale dataset.

Table 6: The numbers of seeds generated by different coverage metrics on the real-world dataset.

	bc	ct	ma	mw	n2	n4	n8
file	38	38	38	19462	38	38	38
gif2png	1039	2037	151008	29606	804	1665	3840
gzip	1305	1340	124253	65035	1002	1875	5446
info2cap	4966	12555	76048	30136	4802	8831	17104
objdump	6015	42625	49401	126578	4978	8756	22914
readelf	8461	15317	91982	63009	8758	15425	35429
strings	61	62	1619	59	69	68	131
tiff2pdf	834	883	143902	2841	724	1108	2395
tiffset	2	2	2	2	2	2	2
size	2117	4860	111978	143693	1605	3003	10278
nm	12566	49307	133460	73386	5947	10174	23322
jhead	384	284	75328	29229	362	576	1376

Real-world dataset Table 6 lists seed counts generated by each coverage metric on the 12 real-world binaries. We can draw similar observations as on the CGC and LAVA-M datasets with some exceptions: the seed count distribution for `ct` is no longer between `n4` and `n8` in general.

4.8 Combination of Coverage Metrics

From the evaluation results above, we observe that each coverage metric has its unique characteristics in terms of crashes found and crashing times. This observation leads us to wonder whether combining fuzzers with different coverage metrics together would find more crashes and find them faster. To answer this question, we consider two options for combination: (1) fuzzers with different coverage metrics are running in parallel and synchronizing seeds across all metrics periodically (i.e., cross-seeding); and (2) fuzzers with different coverage metrics are running in parallel but independently, as the baseline to show whether cross-seeding really helps.

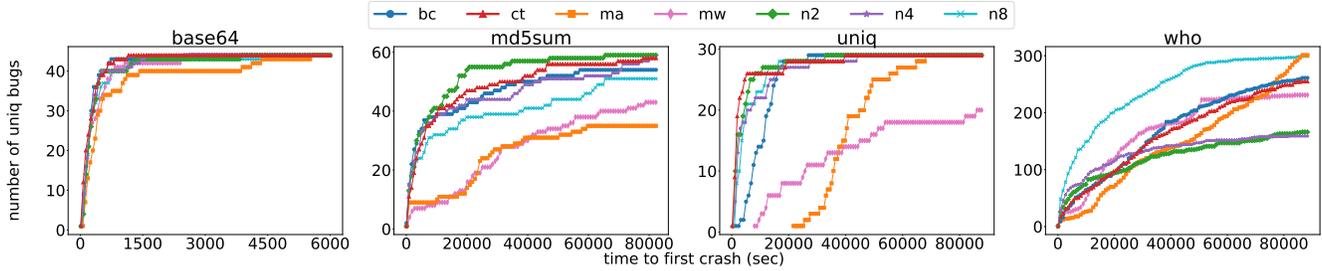


Figure 5: Number of unique bugs found over time during fuzzing on the LAVA-M dataset.

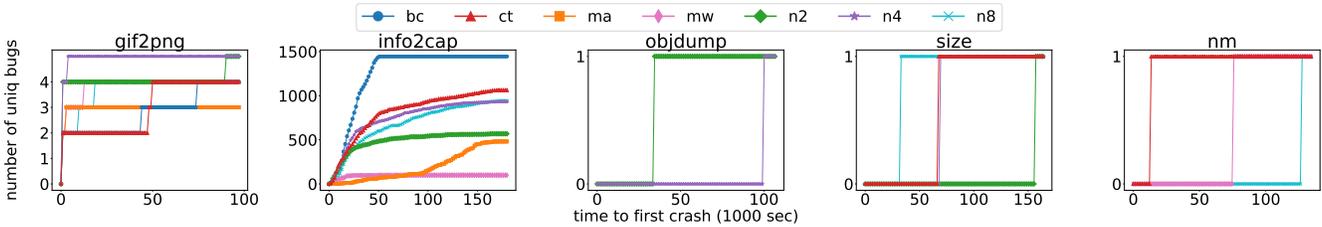


Figure 6: Number of unique crashes found over time on real-world dataset. The x-axis presents TFC in 1000 seconds.

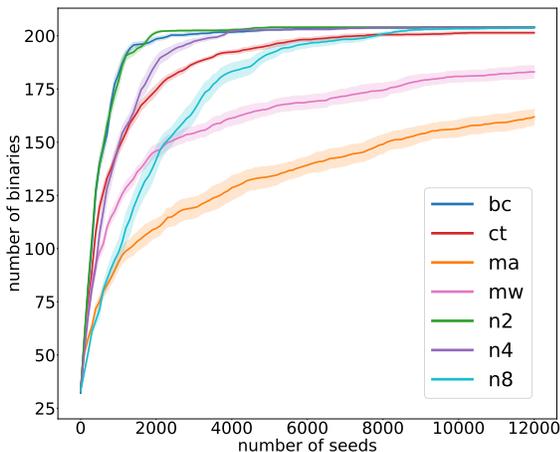


Figure 7: Partial CDFs of seeds generated by different coverage metrics on the CGC dataset. A curve closer to the top left indicates fewer generated seeds.

To study these two options, we create three configurations of 14 fuzzing instances: (a) all 14 fuzzing instances with `bc` and seed synchronization; (b) 2 fuzzing instances for each of the 7 different coverage metrics with seed synchronization only within the same metric; and (c) 2 fuzzers for each of the 7 different coverage metrics with seed synchronization across all metrics (i.e., cross-seeding).

CGC dataset We run the three configurations each for six hours, and for three times to get median results on the CGC dataset. Figure 8 illustrates the number of binaries crashed over time for the three configurations. We can make the following observations. First, both combination options outperform the baseline by large margins, with respect to both the number of crashed binaries and crash times. The combination without cross-seeding (configuration b) crashes 78 CQE bina-

ries, 31 CFE binaries, and 109 binaries in total. The one with cross-seeding (configuration c) crashes 77 CQE binaries, 33 CFE binaries, and 110 in total. Meanwhile, the baseline only crashes 64 CQE binaries, 30 CFE binaries, and 94 in total. It is a notable achievement: the hybrid fuzzer Driller [35] was able to crash 77 CQE binaries after 24 hours with the help of concolic execution, where each binary is assigned to four fuzzing instances and all binaries share a pool of 64 CPU cores for concolic execution, using totally 12,640 CPU hours ($131 \text{ binaries} \times 4 \text{ cores} \times 24 \text{ hours} + 60 \text{ cores} \times 24 \text{ hours}$). Compared with Driller, we can achieve the same or even better results by pure fuzzing with less computing resources ($131 \text{ binaries} \times 14 \text{ cores} \times 6 \text{ hours} = 11,004 \text{ CPU hours totally}$)!

Second, the blue line and the red line cross at around 3 hours. At this cross point, 105 binaries have been crashed for both configurations. It implies that the combination with cross-seeding is able to crash 105 binaries much earlier than the one without cross-seeding.

LAVA-M and real-world datasets We also run the three configurations each for 24 hours on LAVA-M dataset, and each for 48 hours on the real-world dataset. Figure 9 and Figure 10 present the results. We observed that the combination without cross-seeding always outperforms the baseline (14 fuzzers with `bc` only) by large margins. On the other hand, the combination with cross-seeding has inconsistent performance across these nine binaries. In some cases, it is even worse than the baseline. Unlike the result for the CGC dataset, this result is not statistically significant. However, it does indicate that sometimes, the overhead of cross-seeding may outweigh its benefits. Xu et al. [52] have shown that cross-seeding overhead is significant in parallel fuzzing and propose OS-level modifications for improving fuzzing performance. It would be interesting to re-evaluate the performance of the combi-

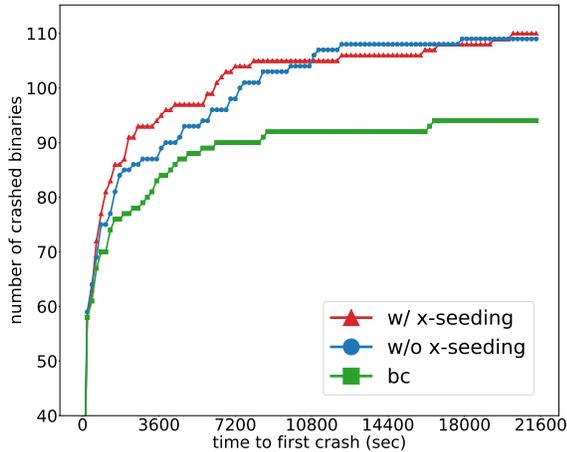


Figure 8: Number of binaries crashed during fuzzing tests by combining different coverage metrics on the CGC dataset.

nation with cross-seeding with these OS-level modifications. We leave it as future work.

In summary, *it is better to combine different coverage metrics with or without cross-seeding, which can help find more bugs and find them faster.*

5 Discussion and Future Work

In this section, we discuss several areas that can be potentially improved and explored in future work.

Precision and collision of coverage calculation For our presented coverage metrics, we adopt straightforward formulas for computing key-value pairs in the `hit_count` map, for the sake of efficiency, but at the cost of precision. For instance, Equation 3 uses a simple XOR for computing the calling context. As a result, it cannot differentiate a function being called twice from a function just returned. Similarly, Equation 2 XOR’s previous $n - 1$ block transitions together to compute n -gram branch coverage. This computation omits the exact order among these $n - 1$ block transitions, and thus loses precision. A related problem is hash collisions [12]. Simple formulas presented in this paper may end up computing the same key from two sets of different input values. Better formulas that improve precision and reduce collisions deserve more investigation. Note that although [12] has proposed a greedy algorithm to reduce collision, the proposed method only works for branch coverage and cannot be easily applied to other coverage metrics.

Application-aware coverage metric selection and resource allocation In Figure 2, we can see the presented coverage metrics are not in a total order in terms of sensitivity. This means different coverage metrics have either unique strength in breaking through a specific pattern of code like loops. From the evaluation results presented in §4, we also observe that (1) there is no “grand slam” metric that beats all

other metrics; and (2) even for metrics whose sensitivities are in total order (e.g., `bc`, `n2`, `n4`, `n8`), the most sensitive one is not always better. In this paper, we explored a simple combination of them and allocated computing resources equally among them. Because fuzzing can be modeled as a multi-armed bandit (MAB) problem [51] that aims to find more bugs with a limited time budget, previous work has shown how to improve the performance of fuzzing through adaptive mutation ratio [6]. Similarly, it might be possible to conduct static or dynamic analysis on each tested program to determine which coverage metric is more suitable. This decision may also change over time, so a resource allocation scheme might be useful to allocate computing resources among different coverage metrics dynamically.

6 Related Work

In §2 we have highlighted some related work on greybox fuzzing. In this section, we briefly discuss some additional work related to fuzzing.

Fuzzing was first introduced to test the reliability of UNIX utilities [22] in a blackbox way. Since then blackbox fuzzing has been widely used and developed that results in several mature tools such as Peach [46] and Zzuf [50]. There are many research works on improving it. For instance, Woo et al. [51] evaluate more than 20 seed scheduling algorithms using a mathematical model to find the one leading to the greatest number of found bugs within the given time budget. SYMFUZZ [6] optimizes the mutation ratio to maximize the number of found bugs given a pair of program and seed via detecting dependencies among the bit positions. Rebert et al. [30] propose an optimal algorithm of selecting a subset from a given set of input files as initial seed files to maximize the number of bugs found in a fuzzing campaign. Moon-Shine [23] develops a framework that automatically generates seed programs for fuzzing OS kernels via collecting and distilling system call traces.

Whitebox fuzzing aims to direct the fuzz testing via reasoning about various properties of the programs. Mayhem [5] involves multiple program analysis techniques, including concolic execution, to indicate the execution behavior for an input to find exploitable bugs. Taintscope [37] leverages dynamic taint analysis to identify checksum fields in input and locate checksum handling code in programs to direct fuzzing bypass checksum checks. BuzzFuzz [13] uses taint analysis to infer input fields affecting sensitive points in the code, which most often are parameters of system and library calls, and then make the fuzzing focus on these fields. MutaGen [18] aims to generate high-coverage test inputs via performing mutations on an input generator’s machine code and using dynamic slicing to determine which instructions to mutate. Redqueen [1] presents another approach to solve magic bytes and checksum tests via inferring input-to-state correspondence based on lightweight branch tracing. ProFuzz [53] tries to infer the

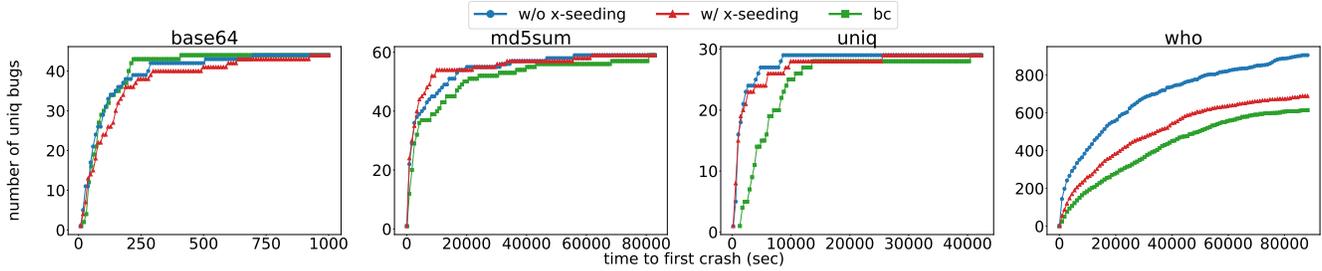


Figure 9: Number of unique bugs found over time by combining different coverage metrics on the LAVA-M dataset.

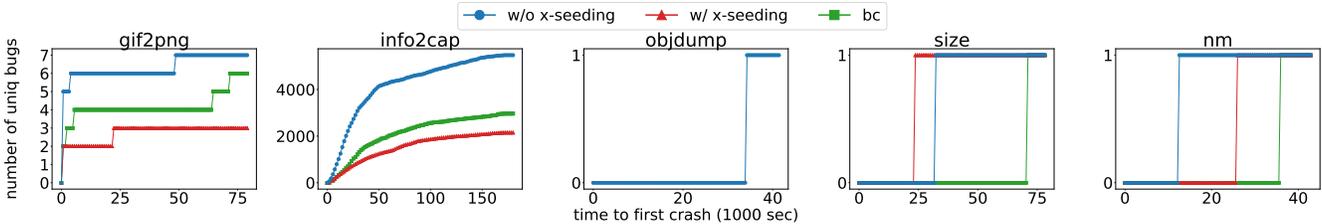


Figure 10: Number of unique bugs found over time by combining different coverage metrics for crashed real-world binaries.

semantic type of input bytes through the coverage information and apply different mutation strategies according to the type. Neuzz [32] approximates taint analysis by learning the input-to-branch-coverage mapping using a neural network, which can then predict what inputs bytes can lead to more coverage. Eclipse [9] identifies input-dependent branch predicates by checking which branches are affected when mutating an input byte; then uses binary search to flip the branch.

It is worth mentioning that recently whitebox fuzzing has been extensively explored in finding OS kernel and driver bugs. CAB-Fuzz [19] optimizes concolic execution for quickly exploring interesting paths to find bugs in COTS OS kernels. SemFuzz [54] uses semantic bug-related information retrieved from text reports to guide generating system call sequences that crash Linux kernels as Proof-of-Concept exploits. IMF [16] leverages dependence models between API function calls inferred from API logs to generate a program that can fuzz commodity OS kernels. DIFUZE [10] uses a specific interface recovered from statically analyzing kernel driver code to generate correctly-structured input for fuzzing kernel drivers.

The combination of whitebox fuzzing and blackbox/greybox fuzzing results in hybrid fuzzing. Pak’s master thesis [24] first uses symbolic execution to discover frontier nodes representing unique paths and then launches blackbox fuzzing to explore deeper code along the paths from these nodes. Stephens et al. [35] develop Driller that launches selective symbolic execution to generate new seed inputs when the greybox fuzzing could not make any new progress due to complex constraints in program branches. Furthermore, Shoshitaishvili et al. [33] extend Driller to incorporate human knowledge. DigFuzz [56] proposes a novel Monte Carlo based probabilistic model to prioritize paths for concolic execution in hybrid fuzzing. QSYM [55] designs a fast concolic execution en-

gine that integrates symbolic execution tightly with the native execution to support hybrid fuzzing.

In addition, Skyfire [36] proposes a novel data-driven approach to generate correct, diverse, and uncommon initial seeds for fuzzing to start with via leveraging knowledge including syntax features and semantic rules learned from a large scale of existing testcase samples. Xu et al. design new operating primitives to improve the performance of fuzzing with shortening the execution time for an input, especially when it runs on multiple cores in parallel [52]. T-Fuzz [26] develops transformational fuzzing that automatically detects and removes sanity checks making it get stuck in the target program to improve coverage and then reproduces true bugs in the original program via a symbolic execution-based approach.

7 Conclusion

In this paper, we conducted the first systematic study on the impact of coverage metrics on greybox fuzzing with the DARPA CGC dataset, the LAVA-M dataset, and real-world binaries. To this end, we formally define the concept of sensitivity when comparing two coverage metrics, and selectively discuss several metrics that have different sensitivities. Our study has revealed that each coverage metric leads to find different sets of vulnerabilities, indicating there is no grand slam that can beat others. We also showed a combination of different metrics helps find more crashes and find them faster. We hope our study would stimulate research on developing more coverage metrics for greybox fuzzing.

Acknowledgments

This work is supported, in part, by National Science Foundation under Grant No. 1664315, No. 1718997, Office of Naval Research under Award No. N00014-17-1-2893, and UCOP under Grant LFR-18-548175. 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] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.
- [2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [4] DARPA CGC. 2014. DARPA Cyber Grand Challenge Binaries. <https://github.com/CyberGrandChallenge>. (2014).
- [5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [6] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [7] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* 46, 3 (2011), 265–278.
- [9] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the 2019 International Conference on Software Engineering (ICSE)*. IEEE.
- [10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [11] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. ColIAFL: Path Sensitive Fuzzing. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [13] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE.
- [14] Gregory Gay, Matt Staats, Michael Whalen, and Mats PE Heimdahl. 2015. The Risks of Coverage-Directed Test Case Generation. *IEEE Transactions on Software Engineering* 41, 8 (2015), 803–819.
- [15] Patrice Godefroid, Michael Y Levin, David A Molnar, and others. 2008. Automated whitebox fuzz testing. In *Proceedings of the 2008 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.
- [16] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [17] Laura Inozemtseva and Reid Holmes. 2014. Coverage Is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM.
- [18] Ulf Kargén and Nahid Shahmehri. 2015. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM.
- [19] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyong Lee, Youngtae Yun, and Taesoo Kim. 2017. CAB-Fuzz: Practical Concolic Testing Techniques for COTS

- Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference*. USENIX.
- [20] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [21] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2008. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *IEEE International Conference on Dependable Systems and Networks (DSN)*. IEEE.
- [22] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.
- [23] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Security Symposium*. IEEE.
- [24] Brian S Pak. 2012. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. In *Master's thesis, School of Computer Science Carnegie Mellon University* (2012).
- [25] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtol: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *Proceedings of the 26th USENIX Security Symposium*. USENIX.
- [26] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [27] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [28] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596* (2017).
- [29] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.
- [30] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 24th USENIX Security Symposium*. USENIX.
- [31] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium*. USENIX.
- [32] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Learning. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [33] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [34] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. Bit-Blaze: A new approach to computer security via binary analysis. *Information Systems Security* (2008), 1–25.
- [35] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.
- [36] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven Seed Generation for Fuzzing. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [37] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [38] Website. 2018. American Fuzzy Lop (AFL) Fuzzer. <http://lcamtuf.coredump.cx/afl/>. (2018). Accessed: 2018-04.
- [39] Website. 2018. Angr: a framework for analyzing binaries. <https://angr.io/>. (2018). Accessed: 2018-04.
- [40] Website. 2018. honggfuzz. <http://honggfuzz.com/>. (2018). Accessed: 2018-04.

- [41] Website. 2018. Intel PIN Tool. <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>. (2018). Accessed: 2018-04.
- [42] Website. 2018. The LAVA Synthetic Bug Corpora. <https://moyix.blogspot.com/2016/10/the-lava-synthetic-bug-corpora.html/>. (2018). Accessed: 2018-04.
- [43] Website. 2018. libFuzzer. <https://llvm.org/docs/LibFuzzer.html>. (2018). Accessed: 2018-04.
- [44] Website. 2018. Of Bugs and Baselines. <https://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html>. (2018). Accessed: 2018-04.
- [45] Website. 2018. OSS Fuzz. <https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html>. (2018). Accessed: 2018-04.
- [46] Website. 2018. Peach Fuzzer. <https://www.peach.tech/>. (2018). Accessed: 2018-04.
- [47] Website. 2018. Security @ Adobe. <https://blogs.adobe.com/security/2012/05/a-basic-distributed-fuzzing-framework-for-foe.html>. (2018). Accessed: 2018-04.
- [48] Website. 2018. Trail of Bits Blog. <https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again>. (2018). Accessed: 2018-04.
- [49] Website. 2018. Utilities for automated crash sample processing/analysis. <https://github.com/rc0r/afl-utils>. (2018). Accessed: 2018-04.
- [50] Website. 2018. Zzuf: multi-purpose fuzzer. <http://caca.zoy.org/wiki/zzuf>. (2018). Accessed: 2018-04.
- [51] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [52] Wen Xu, Sanidhya Kashyap, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [53] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.
- [54] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [55] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium*. USENIX.
- [56] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

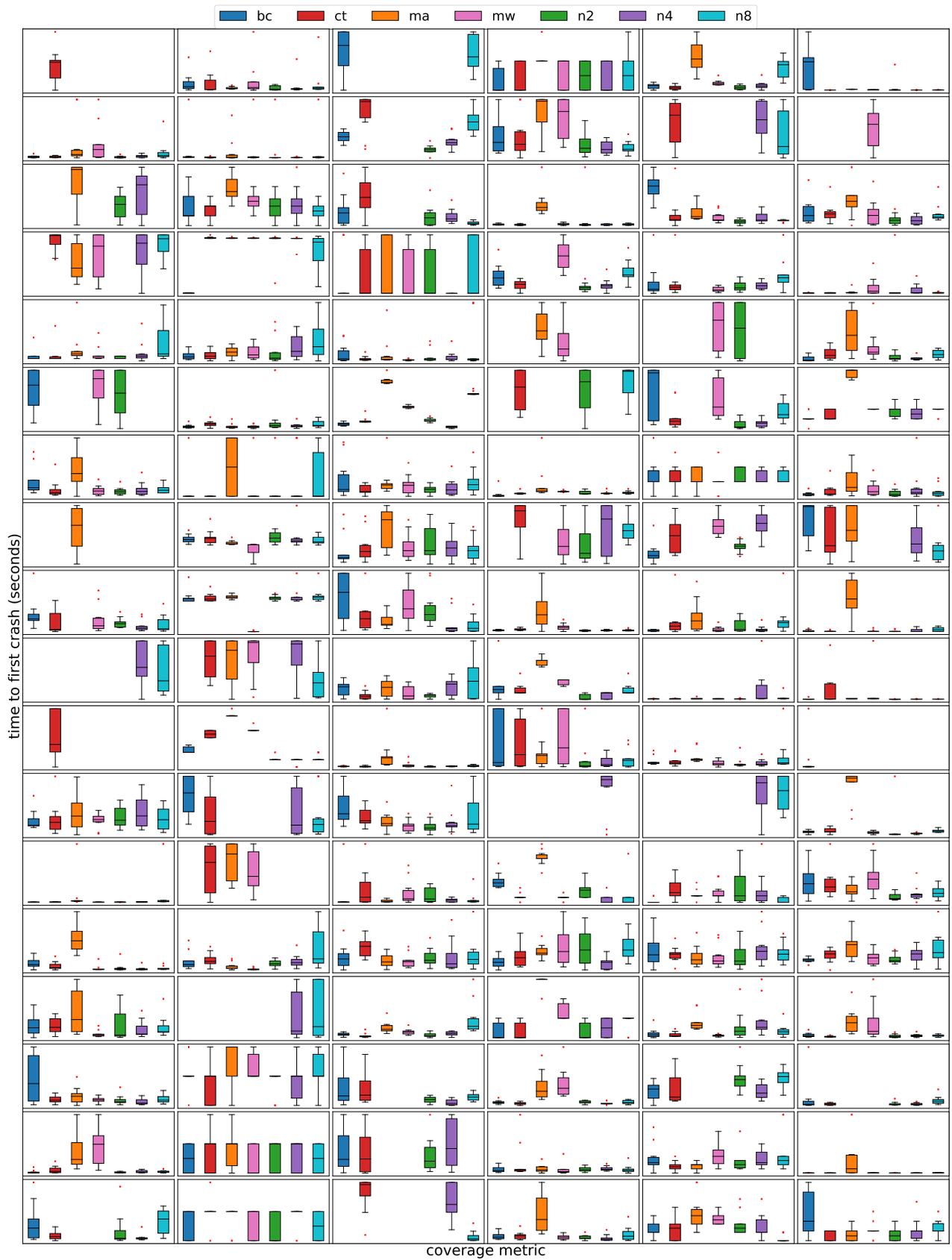


Figure 11: Time to first crash per binary of CGC dataset.

On Design Inference from Binaries Compiled using Modern C++ Defenses

Rukayat Ayomide Erinfolami
Binghamton University

Anh Quach
Binghamton University

Aravind Prakash
Binghamton University
{rerinfo1, aquach1, aprakash}@binghamton.edu

Abstract

Due to the use of code pointers, polymorphism in C++ has been targeted by attackers and defenders alike. Vulnerable programs that violate the runtime object type integrity have been successfully exploited. Particularly, virtual dispatch mechanism and type confusion during casting have been targeted.

As a consequence, multiple defenses have been proposed in recent years to defend against attacks that target polymorphism. Particularly, compiler-based defenses incorporate design information—specifically class-hierarchy-related information—into the binary, and enforce runtime security policies to assert type integrity.

In this paper, we perform a systematic evaluation of the side-effects and *unintended consequences* of compiler-based security. Specifically, we show that application of modern defenses makes reverse engineering and semantic recovery easy. In particular, we show that modern defenses “leak” class hierarchy information, i.e., design information, thereby deter adoption in closed-source software. We consider a comprehensive set of 10 modern C++ defenses and show that 9 out of the 10 at least partially reveal design information as an unintended consequence of the defense. We argue a necessity for design-leakage-sensitive defenses that are preferable for closed-source use.

1 Introduction

The benefits of C++ as an object-oriented language have prompted its wide use in commercial software. As a consequence, the under-the-hood mechanisms of the language implementation (e.g., virtual dispatch) have come under strict scrutiny from both the attackers and defenders. Particularly, practical attacks against C++ software that target control-flow hijacking through virtual dispatch [25, 26], and type confusion through static and dynamic casts [12, 17] have become commonplace. As such, in the last few years, the defense community has focused on defending against such attacks. Defenses both at source-code [4, 15, 16, 27] and binary levels [6, 20] have been proposed. Compiler-based defenses that

rely on source code utilize rich high-level class inheritance information available in the source code and construct strict integrity (control-flow integrity in the case of virtual dispatch and type integrity in the case of type confusion attacks) policies.

On the one hand, with access to source code, compiler-based defenses are precise and well-performing when compared to binary defenses, and so, recent research in protection of C++ software has primarily leaned towards compiler-based defenses [4, 15, 16, 27]. On the other hand, the unintended consequences (side effects) of such defenses have been overlooked, receiving little to no attention. In this paper, we systematically analyze 10 C++ compiler-based defenses to examine their effect on binary reverse engineering. Our results show that 9 out of 10 defenses reveal sensitive class hierarchy information as an unintended consequence. From a software design standpoint, designing class hierarchy is pivotal to a software’s success, and is therefore highly valuable.

From a security perspective, both control-flow-hijacking and type-confusion attacks originate from abuse of inheritance and polymorphism in C++. In essence, inheritance and polymorphism in C++ are defined through the classes and their relationship, i.e., the class inheritance tree. As a common characteristic, compiler-based defenses analyze the source code and extract the inheritance tree, and augment sufficient information into the binary that allows for runtime validation of—at least a subset of—the inheritance tree. Such defenses have been successful in thwarting virtual-dispatch and type confusion attacks with high precision and good performance, and have been welcomed by the community (e.g., [17]).

From a software development perspective, it is crucial to prevent reverse engineering, and challenges in binary analysis that prevent accurate reverse engineering are welcome [18]. Designing classes and their hierarchy is in the heart of C++ design. Software vendors—especially for complex software, invest huge resources in design, and take stringent measures to protect their code from plagiarism and reverse engineering.

In fact, commercial software commonly use obfuscation [18] to prevent reverse engineering. In particular, com-

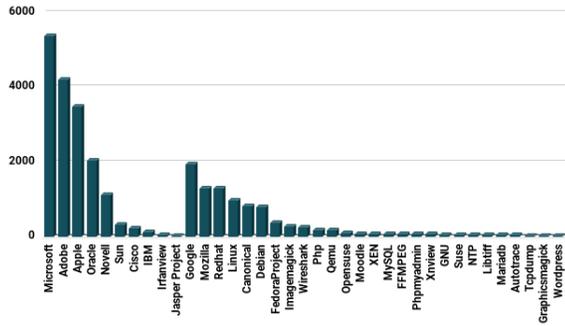


Figure 1: Vulnerabilities vs Manufacturer in the last 5 years. Source: cvedetails.com.

mercial C++ software often incorporates compile-time flag (`-fno-rtti`) to disable inclusion of inheritance-revealing runtime type information (RTTI) in the binary. RTTI includes special type-revealing data structures in the binary that allows for runtime class type resolution. In addition RTTI is known to impose undesirable runtime performance penalties [17]. Without RTTI, reverse engineering C++ binaries to recover design is considered hard and impractical for complex software [20]. This is partly due to the complexities in C++ language along with compiler optimizations that result in challenges for static and dynamic C++ program analysis. For example, intricacies arising due to dynamic dispatch of virtual functions necessitate use of indirect branches in the binary. Indirect branches pose dead-ends with respect to static analysis [28], and dynamic analysis is known to lack coverage. These challenges are a blessing-in-disguise to the closed-source vendor community.

Although modern compiler-based defenses render C++ software attack resilient, by very virtue of augmenting a binary with inheritance information, they reveal a significant aspect of design which vendors do not necessarily want to make public. Furthermore, Figure 1 shows that vulnerabilities reported in the last 5 years for closed-source software vastly outnumber the vulnerabilities reported in open-source products. Therefore, practical and impactful defenses must not only target the open-source community, but also address the concerns of the closed-source community in order to encourage adoption.

Intuitively, the addition of class inheritance information into the binary should aid in reverse engineering and design inference against modern C++ defenses, but so far, there has not been a systematic evaluation of an such effort. In this paper, we present *practical* design inference approaches against modern compiler-based C++ defenses. We show that a *significant* amount of class hierarchy information can be recovered from binaries protected by modern defenses with *high fidelity*. We considered a comprehensive list of 10 modern defenses, and based on source code availability, we evaluated 4 representative defenses. Based on our experimental findings and

the technical details of the remaining 6 defenses, we believe that class hierarchy information can be successfully recovered from embedded metadata of 9 out of 10 defenses. In each case, by modeling classes as nodes and inheritance relationships as edges, we were able to recover 95% of classes with edge-correctness of over 80%, and low inheritance graph edit distance. In spite of precise security, leaking of design information is counter to the interests of closed-source community. At the very least, *closed-source developers must endure real non-negligible risk with respect to design revelation in using modern C++ defenses*. Precise and well performing defenses against control-flow-hijacking and type-confusion attacks in closed-source C++ software are needed.

Our contributions can be summarized as follows:

1. We consider a comprehensive set of 10 modern C++ defenses, and show that 9 of them at least partially reveal design (class hierarchy tree) information as shown in Table 1. We provide systematic inference strategies for such defenses.
2. We recover directed class inheritance graph for popular open-source programs and show that the recovery is of high fidelity, i.e., 95% polymorphic classes recovered with over 80% edge correctness.
3. Although we primarily target polymorphic classes, we show that both polymorphic and non-polymorphic classes can be successfully recovered from binaries compiled using 2 out of 10 defenses.

The remainder of the paper is organized as follows. Section 2 provides the technical background followed by an overview of our approach in Section 3. We present the details of our design inference approach in Section 4 followed by evaluation in Section 5. Finally we present the related work, conclusion and acknowledgement in Sections 6, Section 7 and Section 8 respectively.

2 Background

2.1 C++ Polymorphism—Under the Hood

Virtual functions are at the heart of polymorphism. In order to implement polymorphism, C++ compilers utilize a per-class supplementary data structure called a “VTable” that contains a list of polymorphic (virtual) functions an object may invoke. The structure of a VTable is dictated by the C++ Application Binary Interfaces (ABIs) – Itanium [3] and MSVC [22]. For the rest of this paper, we refer to the Itanium ABI although the differences between the two ABIs are insignificant to our inference approach. A VTable is allocated for each polymorphic class (i.e., a class that contains virtual functions, or inherits from class(es) that contain virtual function, or inherits a class virtually). Within the constructor of a polymorphic class, a

Table 1: Recoverability of Class Hierarchy Tree (CHT) from binaries compiled using various defenses. “Inference depends on callsite” means that design inference on that defense depends on callsite information, “Provides direction info” means that the class hierarchy metadata embedded by defense also includes direction of inheritance, “CHT Recovery” means how much of the embedded class hierarchy tree can be recovered”

Scheme	Category	Inference depends on callsite?	Provides direction info?	CHT Recovery
SafeDispatch [15]	Polymorphic classes-aware	✗	✓	Full
FCFI [27]	Polymorphic classes-aware	✗	✓	Full
Shrinkwrap [11]	Polymorphic classes-aware	✗	✓	Full
OVT [4]	Polymorphic classes-aware	✓	✓	Full
VIP [7]	Polymorphic classes-aware	✓	✗	Full
VTrust [29]	Polymorphic classes-aware	✓	✗	Partial
CaVer [17]	All classes-aware	✓	✓	Full
TypeSan [12]	All classes-aware	✗	✓	Full
HexType [16]	All classes-aware	✓	✓	Full
CFIXX [19]	Polymorphic classes-aware	✗	✗	Low

```

//Read-only Code
Callsite (A *obj):
    vptr = *(obj)
    vfnptr = *(vptr + offset)
    call vfnptr //Indirect call

```

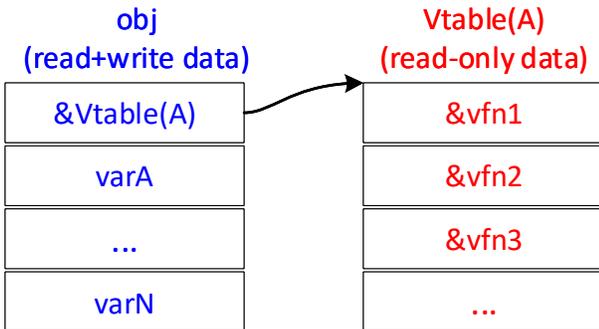


Figure 2: Virtual dispatch mechanism in C++.

pointer to the VTable – called “virtual table pointer” or vptr – is stored as an implicit member in object instance. In case of multiple inheritance, wherein a class derives from more than one polymorphic base class, the VTable for derived class comprises of a group of two or more VTables. The group comprising of the primary and secondary VTables is collectively called the complete-object VTable for the derived class. The complete-object VTable for an object is a comprehensive representation of polymorphic capabilities of an object. For more information on the structure of a VTable, we refer readers to the Itanium ABI [3].

Each virtual function and its polymorphic variants (i.e., functions that override the virtual function) are stored at a

fixed predefined offset. While the offset is known at compile-time, the concrete virtual function is resolved at runtime. At a virtual function dispatch site (as shown in Figure 2), the compiler embeds code to access the VTable from the object, offsets into the VTable and retrieves the appropriate virtual function, and finally invokes the virtual function. Because the virtual functions are resolved at runtime, an indirect call instruction is used to accomplish the dynamic dispatch.

2.2 Attacks Against C++

The mechanisms surrounding virtual function dispatch, particularly storage and retrieval of virtual function pointers, have been targets of exploits. Based on the nature of vulnerability and the exploitation, we classify attacks into two categories.

2.2.1 VTable Hijacking

In this case, an attacker corrupts the object (or creates fake objects) such that the VTable pointer points to an illegitimate location. An example of virtual dispatch along with the sample object and the VTable layout is provided in Figure 2. The code and the VTables are usually protected by allocating them in non-writable sections. However, the objects themselves are located in writable region (heap, stack or the data region). Typically, the attacker exploits a memory corruption vulnerability to overwrite the VTable pointer within the object. Such a pointer could either point to an attacker-injected VTable (VTable injection), or an offset in an existing legitimate VTable. In both cases, virtual function pointer resolution at the virtual dispatch callsite results in a corrupted pointer. Invocation of corrupted pointer leads to arbitrary code execution.

2.2.2 Unsafe Casting

In this case, the attacker takes advantage of unsafe and inconsistent casting operations in code. C++ supports multiple types of casting operations—*static_cast*, *reinterpret_cast*, *const_cast* and *dynamic_cast*. Particularly, when an object allocated to a base class is casted to a derived class, it results in invalid downcasting. Virtual function calls using such a downcasted derived class pointer can lead to arbitrary code execution when the virtual function offset is larger than the size of the base class VTable.

2.3 Compiler-Based C++ Defenses

2.3.1 Defenses Against VTable Abuse

Defenses in this category, except CFIXX, harness the result of class hierarchy analysis (CHA) to defend against VTable abuse. They restrict the functions or vptrs allowed at a callsite to those in the specific hierarchy of the callsite’s static type. While FCFI [27] builds the result of CHA into a table specifying valid vptrs for any given callsite, SafeDispatch [15] builds for both valid vptrs and valid functions. Shrinkwrap [11] modifies FCFI by removing redundant vptr entries in the metadata table, thereby providing stricter defense. This metadata is embedded into the binary to be looked up at runtime. OVT/IVT [4] attempts to reduce the overhead of vptr validation at runtime by reordering VTables in preorder traversal such that VTables within the same hierarchy are laid out contiguously. A range can therefore be assigned to specify the valid VTables in a given hierarchy. The range assigned to a type tells the set of its subclasses. OVT adds paddings in between VTables so that vptr validation can be done in a simple range check. IVT eliminates these paddings by interleaving VTable entries. More details about the specific metadata embedded by these defenses is presented in Section 4.2.

At compile time, VTrust [29] generates hash values to tag functions that are legitimate for each callsite, basically functions defined by classes in the same hierarchy. Hash values are computed using function name, argument type list, qualifiers and most base class which defined each function. At runtime, the hash value at the caller callsite is compared with that at the beginning of the callee. All polymorphic functions have the same hash values. VIP [7] supplements pointer analysis with CHA to provide a more precise set of valid vptrs. Pointer analysis makes it possible to identify types that are likely to be used at a given callsite during execution. This information helps to further restrict the set obtained from CHA. CFIXX [19] enforces Object Type Integrity (OTI) which dynamically tracks object type and enforces its integrity against arbitrary writes. OTI protects key application control flow data from being corrupted. CFIXX keeps track of the only valid object at a callsite at runtime, it does not embed metadata of class hierarchy.

2.3.2 Defenses Against Unsafe Casting

CaVer [17] uses a runtime type tracing mechanism called type hierarchy table (THTable) to dynamically verify type casting operations for polymorphic and non polymorphic classes. Given a pointer to an object allocated as type T, the THTable stores the set of all possible types to which T can be casted (i.e. its base classes) including its phantom classes. TypeSan [12] improves CaVer by providing lower runtime overhead and higher detection coverage. It uses two data structures: type layout table and type relation tables which serve the same purpose as CaVer’s THTable but useful for optimizing type checks. These data structures also specify the base classes of the representing class.

HexType is an improvement over TypeSan and CaVer. It provides two folds of improvement, 1) higher coverage of typecasting operations and 2) lower performance overhead. The first is achieved by considering more instances of object creation which include the use of new operator, placement new, and when an already constructed object is hard-coded. The second is achieved by improving the method of object tracing. HexType and CaVer use the same data structures.

3 Overview

Design Inference. In this work, we propose systematic approaches to infer design information, specifically class inheritance tree from a protected C++ binary.

3.1 Compiler Defense Categories

We apply our design inference approach to a diverse and comprehensive set of C++ defenses against control-flow-hijacking and type-confusion attacks. In fact, all compiler-based defenses that result in binaries adhering to the ABI are susceptible to design inference.

Based on how design information is embedded in the binary, we group solutions into two categories.

C1: Explicit Design Information Inclusion: Solutions in this category explicitly embed design information into a given binary. Validating an object’s type (e.g., object used at a given callsite) simply requires searching the information stored in the binary. In order to reverse such binaries, we extract the embedded information (typically in read-only data sections) and process them to extract class hierarchy information. FCFI [27], ShrinkWrap [11], SafeDispatch [15], CaVer [17] and TypeSan [12] are examples of defenses in this category.

C2: Implicit Design Information Inclusion: These solutions transform the original design information into forms that makes verification faster or reduce the amount of work required to provide protection. For example, OVT [4] encodes class hierarchy information by choreographed ordering of existing VTables thereby eliminating the need to embed class

hierarchy information in a separate section. Similarly, VIP [7] and VTrust [29] are examples of defenses in this category.

3.2 Scope and Assumptions

Our design recovery approaches are applicable to *all* known source-code based C++ defenses against attacks that abuse polymorphism-related mechanisms in C++.

Since most C++ compiler-based defenses adhere to the Itanium ABI, our solution targets defenses that adhere to the Itanium ABI. Although, due to the high similarity between Itanium and MSVC ABIs, our technique will also apply to potential solutions on MSVC. IVT [4] is a compiler-based defense that violates the ABI. Therefore, IVT can not interoperate with binaries that adhere to the ABI. It is outside our scope.

3.3 High-Level Approach

At a high level, our approach comprises of two steps. First, given a protected C++ binary, we extract all the polymorphic classes from it. This is simply the set of all complete-object VTables in the binary. Each complete-object VTable maps to a specific polymorphic type in the program. Recall that CaVer and TypeSan represent classes, polymorphic and non-polymorphic, as THTables. Each THTable contains information such as base classes of the representing class. Second, for each type, we generate a *Type Congruency Set* or (TC_{Set}), which is simply a set of types that are congruent or exhibit a *is-a* relationship with the type. That is:

$$TC_{Set}(X) = \{Y, \text{ where } Y \text{ is-a } X \text{ and } X, Y \text{ are polymorphic types } \}$$

Intuitively, $TC_{Set}(X)$ of a type X contains the type X itself and all the classes that derive from X because derived classes exhibit *is-a* relationship to their bases. The TC_{Set} provides a commonality between different defenses. Defense policies for both virtual call dispatch and type casting are based on polymorphic relationship between types, which is succinctly captured by TC_{Set} .

Given the TC_{Set} for all the polymorphic types in the binary, we construct a directed inheritance graph for the binary.

Does the underlying compiler matter? Our approach relies solely on the C++ ABI that a compiler adheres to, and not on specific compiler features. Further, as a proof of concept, we evaluate defenses that adhere to Itanium ABI, which is implemented in two popular compilers GCC and LLVM. Given that the high-level design and data structures (e.g., offset-to-top, RTTI) in Itanium [3] and MSVC [22] ABIs are largely similar, we believe that with insignificant changes to our approach, defenses that adhere to the MSVC ABI can also be targeted.

3.4 Key Challenges

Unavailability of Runtime Type Information. Without access to symbol information, inferring class hierarchy is a hard but important part of C++ reverse engineering. Prior efforts have relied on the RunTime Type Information (RTTI) to recover inheritance structure. However, RTTI is an optional data structure that can be omitted if `typeid` and `dynamic_cast` are not used in the code. In fact, due to the rich semantic information contained within RTTI, along with performance penalties incurred in using `dynamic_cast`, commercial closed-source software often exclude RTTI information and hinder reverse engineering.

In the absence of RTTI, past efforts have relied on the sizes and contents of VTables for reverse engineering class hierarchy. For example, Fokin et al., [8, 9] derive rules between two polymorphic classes, and apply them to infer inheritance. These approaches either (1) lack in precision [1, 5], or (2) lack in directionality of class inheritance graph [20].

From a software vendor's point of view, such inaccuracies are preferable, and hinder reversing of software design. Yet, from an end-user point of view, such inaccuracies result in coarse-grained CFI policies that introduce an attack surface in the form of redundant edges in the CFG.

Compiler Optimizations. Function inlining and removal of unused code become aggressive with higher levels of optimization. These occurrences limit the amount of information available for reverse engineering. For instance, constructors give indications about the base classes of a class as well as the direction of inheritance. However, the compiler could either inline them or even remove them completely, thereby making their identification difficult and incomplete. This eventually results in incomplete or over approximation of class relations.

4 Design Inference

4.1 VTable Accumulation and Grouping

As a first step, we extract all the complete-object VTables in the binary. Complete-object VTables provide an unlabeled representation of all the polymorphic classes in the binary. In essence, they represent the nodes in the class inheritance tree of the program.

Extracting VTables from a binary is a previously studied problem [10, 21, 30]. The well defined nature of VTables, particularly the existence of mandatory fields [3], provides a robust signature that can retrieve all the VTables in the binary. Objects are initialized with a `vptr` in the constructors and destructors of polymorphic classes. These VTable addresses appear as immediate values. We scan the binary for immediate values that point to readonly sections of memory. We then examine each of those addresses for VTables, and accumulate all the VTables. Our approach to VTable recovery

is similar to the one presented in vfGuard [21]. However, the recovered VTables (using vfGuard approach) include both the primary and secondary VTables. This means that one or more VTables in the gathered list map to a single polymorphic class in the program. Therefore, we merge the primary and all corresponding secondary VTables to obtain a set comprising of only the complete-object VTables. We implement Algorithm 1 to group recovered VTables into complete-object VTables.

Algorithm 1 is based on two key observations:

- Modern compilers (g++ and LLVM-clang) layout primary and secondary VTables for a derived class in sequential order.
- Since offset-to-top represents the displacement from the top of the object to a sub-object, a value of 0 represents primary VTable, and a non-zero value represents secondary VTable.

Given a set of VTables, we first sort the VTables in increasing order of vptrs. Then, we merge the primary VTables with succeeding (zero or more) secondary VTables to form the complete object VTables. VTable grouping for OVT is done differently. This is necessary because OVT reorders VTable which separates secondary VTables from their corresponding primary VTables. Each VTable is placed in a tree where the most base class it inherits from is the root. We adopt two techniques to obtain the complete-object VTables for OVT binaries: thunk and constructor analysis, they are discussed in Section 4.4.

Algorithm 1 GroupVTables groups VTables in \mathcal{V} to form complete-object VTables \mathcal{V}'_{Group} where each VTable in \mathcal{V}'_{Group} is a Primary VTable, and contains a list of zero or more Secondary VTables.

```

1: procedure GROUPVTABLES( $\mathcal{V}$ )
2:    $\mathcal{V}'_{Sorted} \leftarrow$ 
3:    $sort\_increasing(\mathcal{V})$ 
4:    $\mathcal{V}'_{Group} \leftarrow \emptyset$ 
5:   for each  $V_T$  in  $\mathcal{V}'_{Sorted}$  do
6:     if  $V_T.OffsetToTop == 0$  then
7:        $V_{Primary} \leftarrow V_T$ 
8:        $\mathcal{V}'_{Group}.append(V_{Primary})$ 
9:     else
10:       $V_{Primary}.Secondaries.append(V_T)$ 
11:    end if
12:  end for
13:  return  $\mathcal{V}'_{Group}$ 
14: end procedure

```

4.2 Generating Type Congruency Set

The format in which design information is represented in the binary determines how the TC_{Set} can be recovered. FCFI, Shrinkwrap and SafeDispatch use similar representation, while CaVer and TypeSan use a different representation. TC_{Set}

for FCFI, Shrinkwrap and OVT comprises of complete-object VTables while that for CaVer and TypeSan comprises of THTables.

4.2.1 C1 Defenses

Algorithm 2 shows how metadata is recovered for C1 defenses.

Algorithm 2 GatherTCSet for C1 defenses

```

1: procedure GATHERTCSET( $sAddr, eAddr$ )
2:    $\mathcal{M}\mathcal{D}_{Tables} \leftarrow \emptyset$ 
3:    $addr \leftarrow sAddr$ 
4:   while  $addr \leq eAddr$  do
5:      $addr \leftarrow get\_next\_8bytes\_aligned\_addr(sAddr)$ 
6:     if  $is\_valid\_metadata\_start(addr)$  then
7:        $\mathcal{M}\mathcal{D}_{Table} \leftarrow \emptyset$ 
8:       while  $!is\_valid\_metadata\_end(addr)$  do
9:          $key \leftarrow extract\_next\_key(addr)$ 
10:         $\mathcal{M}\mathcal{D}_{Table}.append(key)$ 
11:         $addr \leftarrow get\_next\_8bytes\_aligned\_addr(addr)$ 
12:      end while
13:    end if
14:     $\mathcal{M}\mathcal{D}_{Tables}.append(\mathcal{M}\mathcal{D}_{Table})$ 
15:  end while
16:  return  $\mathcal{M}\mathcal{D}_{Tables}$ 
17: end procedure

```

FCFI and Shrinkwrap. The VTable maps embedded by FCFI and Shrinkwrap are simply arrays of pointers to VTable with nothing indicating its start or end. However, the VTable maps are initialized with calls to a function named `__VLTRegisterSetDebug`, which contains details about the VTable map addresses and their lengths. As shown in listing 1, the first argument to this function is a pointer to the VTable map for a given class while the second argument is the length of the map. Once the first and second arguments to this function are identified, we can recover the complete inheritance graph for the program. After identification of these callsites and extraction of VTable map addresses and length, we locate all the VTable maps and use the length to decide the end of each map. Since Shrinkwrap puts primary and secondary VTables in different VTable maps, we use VTable grouping to locate the primary VTable of any secondary VTable we find in a map.

Listing 1: Function used to initialize VTable maps

```

...
1 init &vtable_map
2 init length_of_vtable_map
...
3 call __VLTRegisterSetDebug
...

```

CaVer and TypeSan. The data structures embeded by these defenses are well defined, starting and ending with easily distinguishable entries and stored in the data section. CaVer and TypeSan represent matadata differently but the idea is

the same, hence, we generally refer to metadata representation in both as THTables. We first perform a linear sweep through the data section to gather THTables. For CaVer, the first field in a THTable is the length of allowable casts for the class which it represents. The next field is the key for the representing class. Next set of fields are keys to the THTables of the base classes or phantom class. The final field is the name of the representing class. TypeSan divides the operation performed with THTable into two phases, using two different data structures. Even though they are laid out differently, they both specify the inheritance relationship among classes. In some cases, each structure may contain partial information for a given class, by combining them we get the complete information.

4.2.2 C2 Defenses

OVT. OVT performs both range and alignment check in a single branch. To obtain TC_{Set} for binaries compiled using OVT, we identify callsites to extract vptrs corresponding to specific types as well as the range of its subclasses. Listing 2 shows OVT’s instrumentation at virtual function callsites. As explained in Section 2, each type has an associated range which specifies the number of its subclasses. OVT computes the position of the runtime type of an object within a subtree (hierarchy) by first subtracting its vptr “\$vptr” from that of its static type “\$a” and then performing a right bit rotation using a literal l . It then compares the resulting value with the maximum range for the object’s static type, which should be greater. $(\$b - \$a) \gg l$ equates to the maximum range in a given subtree (class hierarchy), where $\$b$ is the last vptr in that subtree. If the static type of the object is a leaf class, i.e. has no subclass, its vptr is compared with the vptr of the type used at runtime for equality. That vptr is used if the equality check is successful, otherwise a violation is reported.

We retrieve the range for each class by following the sequence of operations performed by OVT. The range as well as the vptr are used to recover the $TC_{Set}(X)$. Specifically, we group all the VTables within the identified range into $TC_{Set}(X)$ for the callsite’s static type. For every secondary VTable found in a given range, we obtain the corresponding primary VTable using thunk or constructor analysis. Algorithm 3 shows how we identify and extract the vptrs and ranges of pointer types.

VTrust and CFIXX. VTrust like OVT implicitly embeds class hierarchy into the binary by assigning similar hash values to functions defined by classes in the same hierarchy. The information provided is sufficient to group related classes into sets (similar to Marx), but does not provide direction of inheritance. CFIXX keeps track of the single correct object type that can be used for a virtual call, it neither provides information about classes in the same hierarchy nor direction of inheritance. Therefore, VTrust reveals only partial CHT in

Algorithm 3 GatherTCSet for OVT

```

1: procedure GATHERTCSET(tsAddr, teAddr)
2:   addr ← tsAddr
3:   while addr ≥ teAddr do
4:     Idec ← dec_instr(addr)
5:     if  $\neg$ (isDerefObjectVptrInst(Idec)) then
6:       continue
7:     end if
8:     addr ← get_next_addr(addr)
9:     Idec ← dec_instr(addr)
10:    if  $\neg$ (isMoveTypeVptrInst(Idec)) then
11:      continue
12:    end if
13:    vptr ← Idec.opr[2]
14:    addr ← get_next_addr(addr)
15:    Idec ← dec_instr(addr)
16:    if isOVTCompInstr(Idec) then
17:       $\mathcal{VT}_{Tree}.append(vptr, 1)$ 
18:    continue
19:    end if
20:    addr ← get_next_addr(addr)
21:    Idec ← dec_instr(addr)
22:    if  $\neg$ (isOVTDiffInstr(Idec)) then
23:      continue
24:    end if
25:    addr ← get_next_addr(addr)
26:    Idec ← dec_instr(addr)
27:    if  $\neg$ (isOVTDiffRInstr(Idec)) then
28:      continue
29:    end if
30:    addr ← get_next_addr(addr)
31:    Idec ← dec_instr(addr)
32:    if  $\neg$ (isOVTCompInstr(Idec)) then
33:      continue
34:    end if
35:     $\mathcal{VT}_{Tree}.append(vptr, I_{dec}.opr[2])$ 
36:  end while
37:  return  $\mathcal{VT}_{Tree}$ 
end procedure

```

comparison with binaries without defense, whereas CFIXX reveals no more information than binaries without defense.

4.3 Inferring Inheritance

We combine all recovered type congruency set into the class hierarchy for the entire binary. TC_{Sets} having similar entries are merged into a single tree, which still maintains the edges and the direction of inheritance.

4.4 Thunk and Constructor Analysis

Thunk and constructor analysis are done to identify the corresponding primary VTable of a secondary VTable found in subtrees of binaries compiled using OVT. This is necessary because OVT reorders all sub VTables of a given class into the various subtrees they belong to, which makes VTable grouping (explained in subsection 4.1) not applicable.

Every function in a secondary base class that is redefined by the derived class has a thunk entry in the Base-in-Derived

Listing 2: Callsite generated by OVT

```

...
// $a = vptr of most base class in subtree
1 $diff = $vptr - $a
2 $diffR = rotr $diff, 1
// ($b - $a) >> 1 equates to max range for $a
3 cmp $diffR, ($b - $a) >> 1
...

```

VTable. Hence, we link every secondary VTable which contains a thunk entry to the primary VTable containing the function it references. This method is certain to identify the correct primary of a given secondary VTable, however, a thunk might not always be present. For this reason, we also employ grouping VTables through constructor analysis.

One of the operations performed within a constructor is to write the vptrs (primary and secondary) of the class whose object is to be constructed in the memory space allocated for that object. To group VTables using constructor analysis, we first scan for constructors in the .text section of the binary. Next, we extract all valid vptrs from each constructor. Finally, we identify the primary VTable by looking at the offset-to-top entry which must be zero.

4.5 Executables vs Shared Libraries

Design information recoverable from shared libraries tend to be more comprehensive and accurate compared to executables. The scope of use of shared libraries is unknown at compile time. This prevents compilers from inlining their functions and makes it necessary to retain all useful information. However, executables get inlined aggressively since the scope of their use is visible to the compiler during compilation. For instance, CaVer and TypeSan embed THTables while FCFI embeds VTable maps for virtually all classes in a library.

4.6 Callsite Analysis

Callsite analysis is necessary for C2 defenses which implicitly embed class hierarchy information at callsites. The completeness of our analysis for OVT, for example, depends greatly on the number of base classes which have at least one corresponding callsite. The availability of this information at callsites makes it possible for us to identify subtrees which are merged to build the class hierarchy for the entire binary.

4.7 Handling Template Classes

Binaries generated by both GCC and Clang contain separate VTables for each template implementation in the binary, so is the ground truth obtained using GCC’s -fdump-class-hierarchy. All the compiler-based defenses targeted in this work modify either GCC or Clang, therefore they also treat template classes similarly. Since our ground truth matches

the binaries being analyzed, the presence of template classes does not affect the precision recorded.

4.8 Graph Similarity Measure

To show the effectiveness of our approach, we need to quantify the similarity of the class inheritance recovered from hardened binaries with the ground truth. This problem can be reduced to a graph matching problem. Classes in the hierarchy are represented as nodes, while relationships among them are represented as edges. GEDEVO [14] is an evolutionary algorithm which uses Graph Edit Distance (GED) as optimization model for finding the best similarity between two graphs. GEDEVO produces high quality result even on large graphs, this makes it appropriate for this work since our evaluation set includes large programs like Spidermonkey. GED is a general model for solving graph matching problem, it is defined as the minimal amount of modification needed in graph G_1 to make it isomorphic to graph G_2 [2, 14].

For our evaluation, we modeled class inheritance as a directed graph containing pairs of (baseClass, derivedClass) interaction networks. Considering two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and a one-to-one mapping f between nodes V_1 and V_2 . The GED between G_1 and G_2 induced by mapping f is as follows:

$$GED_f(G_1, G_2) = | \{ (u, v) \in E_1 : (f(u), f(v)) \notin E_2 \} \cup \{ (u', v') \in E_2 : (f'(u'), f'(v')) \notin E_1 \} |$$

By definition, $GED_f(G_1, G_2)$ counts inserted or deleted edges induced by the mapping f , which can be easily extended to depict node/edge dissimilarities. The raw GED for a mapping is calculated from the number of removed and added edges required to transform one graph into the other graph. The raw counts are summed up and normalized, producing the actual GED score.

Even though GEDEVO is dedicated to biological network, it internally utilizes GED for optimization which makes it applicable to general graph comparison problems outside computational biology.

Result Interpretation. GED score ranges from 0 to 1, a value close to 0 shows high similarity while a value close to 1 shows high dissimilarity. GEDEVO also computes the Edge Correctness (EC) of found edges in the two graphs. An EC value of 100% is the highest and it is possible when two graphs are either isomorphic or sub-isomorphic. GED score and Edge correctness represent the precision of recovery and accuracy of inheritance direction respectively.

5 Evaluation

We evaluate our solution on both executables and libraries. Our test suite include SPEC CPU 2006 C++ benchmarks, Spidermonkey, and four shared libraries — CplusplusThread, Attic-c-hdfs-client, LibEbml, and LibMatroska. Our choice

of SPEC CPU programs is based on the programs evaluated by the C++ defenses.

Spidermonkey and the libraries are chosen to show the impact of our solution on real world complex applications. For binaries compiled with FCFI and OVT we consider only polymorphic classes while for CaVer and TypeSan we consider both polymorphic and non polymorphic classes. In our evaluation, we answer the following questions:

1. How accurate and precise are the class hierarchies extracted from binaries hardened with compiler-based defenses?
2. How much design information do compiler-based defenses embed within binaries?
3. Can we recover a more complete class hierarchy from binaries hardened with modern C++ defenses, than state of the art binary analysis tools?

All evaluation was performed on a system with Intel Core i7-4790 CPU @ 3.60GHz x 8 and 32GiB of Memory, running Ubuntu 14.04 LTS with Linux Kernel 4.10.0. All binaries were compiled using GCC under O0 optimization.

Defenses Evaluated. FCFI, Shrinkwrap and SafeDispatch use similar techniques, therefore, we chose FCFI to represent the group. Also, Hextype is only an improvement over CaVer, therefore, we chose CaVer for this evaluation. We were unable to evaluate VTrust and VIP because of the unavailability of their source code.

5.1 Recovery for Polymorphic Classes

Table 2 shows the recovery rate, GED score and Edge correctness of polymorphic classes compared with the ground truth. We constructed the ground truth by using `-fdump-class-hierarchy` option of `g++` which dumps VTables, their layout and inheritance relationship during compilation.

On the average, we obtained a GED score of 0.08, 0.21, 0.09 and 0.35 and Edge correctness of 98.26%, 89.97%, 96.31% and 81.14% for FCFI, OVT, TypeSan and CaVer respectively. The ground truth graph for Namd has four polymorphic classes with only two edges. Classes `PairCompute` and `SelfCompute` have `ComputeNonbondedUtil` as their base class. We were unable to recover `ComputeNonbondedUtil` from the Namd binary compiled with OVT this is the reason its GED score is 1 and its Edge correctness 0%. Also, we could not compile Spidermonkey with TypeSan and DealII, Libbml and Libmatroska with OVT. The recovery from binaries compiled with CaVer are not as precise and accurate as the other defenses, this is because CaVer has been shown to have low coverage [12] [16]. Not all objects are protected, hence fewer design information is embedded in the binary. Also, the GED score for Spidermonkey compiled with FCFI is not as close to zero as other programs, this is because FCFI omits design information for abstract classes.

5.2 Recovery for Both Polymorphic and Non-polymorphic Classes

CaVer and TypeSan protect both polymorphic and non-polymorphic classes, hence we evaluate the precision and accuracy of our recovery for all classes in the programs compiled with these defenses. TypeSan represents class names with hash values in the binary, while CaVer uses actual names. To construct the ground truth for TypeSan, we modified its source code to output the mapping (className, hashValue) during compilation. The results tabulated in Table 3 show the number of THTables recovered from TypeSan and CaVer.

The column "# Keys found" represents the total number of keys found in all THTables, however, some keys do not have a specific THTable in the binary, they are only present within other THTables. For TypeSan, all keys found have a corresponding THTable, but that is not the case for CaVer. CaVer protects only a subset of objects created at runtime, as a result, it does not dump metadata for those unprotected objects in the binary.

5.3 Class Hierarchy Tree Recoverable from the Amount of Information Embedded

Table 4 shows the number of unique classes whose metadata is present in the binary. The aim of this evaluation is to show that even though some defenses do not embed a corresponding metadata for every class in the binary, the ones present are enough to build an accurate class hierarchy. Unlike FCFI and TypeSan, CaVer and OVT embed metadata for classes depending on their use in the program. CaVer embeds a THTable for a class only if at least one instance of that class is created. Since we rely on callsite for OVT, the metadata of a class is made available only if an indirect call is made using an object of that class. However, the absence of metadata for certain classes will have no impact on our recovery as long as their keys (for CaVer) or VTables (for OVT) are found in the binary. For binaries compiled with CaVer, the THTables of classes with no base class are not necessary to build class hierarchy since they contain just one key and name. They do not give any information about inheritance. For binaries compiled with OVT, callsites for classes with no derived class are also not necessary since their range is 1. We refer to such classes as leaf nodes in Table 4. Only the non-leaf nodes are important for reconstructing class hierarchy. On the average we found callsites for 66% of non-leaf nodes, except for Namd whose only non-leaf node has no corresponding callsite. Note that we only consider polymorphic classes for OVT.

5.4 Comparison Against Marx

Marx [20] is a state-of-the-art reverse engineering tool for class hierarchy recovery which infers relationship among classes by heuristics. VTables are taken as classes since they

Table 2: Evaluation result for precision and accuracy of class hierarchy recovered from FCFI, OVT, TypeSan(TS) and CaVer(CV). GT is the Ground Truth obtained by using GCC’s -fdump-class-hierarchy option. VTS&G is VTable Scanning + Grouping, i.e the total number of VTables recovered simply by using VTable Scanning and Grouping without relying on information embedded by the defenses. OVT did not compile DealII.

Programs	GT(polymorphic)	Total Recovery				VTS&G	GED Score				Edge Correctness			
		FCFI	OVT	TS	CV		FCFI	OVT	TS	CV	FCFI	OVT	TS	CV
Spidermonkey	807	795	780	-	521	805	0.33	0.35	-	0.38	88.34	76.98	-	68.80
Xalanc	975	958	673	913	531	857	0.02	0.23	0.15	0.43	100	94.66	86.76	68.94
Soplex	29	29	29	29	22	30	0.05	0.07	0.13	0.18	100	95.24	95.45	94.12
Povray	32	28	26	28	14	30	0.1	0.33	0.11	0.45	100	91.67	100	100
Omnetpp	112	110	105	111	110	107	0	0.21	0.1	0.32	100	81.25	97.06	70.59
dealII	874	717	-	687	98	746	0.12	-	0.16	0.86	97.70	-	94.91	46.70
Namd	4	4	0	4	4	3	0	1	0	0	100	0	100	100
CplusplusThread	11	11	9	11	8	11	0	0.09	0	0.2	100	100	100	100

Table 3: Evaluation result for the number of THTables and unique classes extracted from TypeSan(TS) and CaVer(CV) binaries for both polymorphic and non polymorphic classes. GT is the Ground truth obtained from the (hash, name) mapping which TypeSan and CaVer generate during compilation. *Keys found* is the number of unique keys found through the THTables. *THTables recovered* is the number of THTables found. TypeSan did not compile Attic-c-hdfs-client. FN and FP represent false negatives and positives in THTable recovery respectively.

Programs	GT		Keys found		THTables recovered		FP Keys		FN Keys	
	TS	CV	TS	CV	TS	CV	TS	CV	TS	CV
Xalanc	3075	1591	3162	1562	3162	1013	126	0	39	29
Deal	2332	448	2366	437	2366	263	124	0	90	11
Omnetpp	244	178	236	171	236	150	15	0	23	7
Soplex	115	54	123	50	123	37	12	0	4	4
Povray	249	33	211	32	211	27	14	0	52	1
Namd	21	10	29	10	29	9	8	0	0	0
CplusplusThread	26	14	23	14	23	7	1	0	3	0
Attic-c-hdfs-client	-	651	-	544	-	439	-	0	-	107
LibEbml	92	38	85	35	85	29	1	0	7	3
LibMatroska	286	293	303	288	303	262	18	0	1	5

are the only artifact left in the binary that can uniquely identify classes. Marx performs overwrite analysis which is based on the heuristic that in a constructor, the vptr of a base class gets overwritten by that of its derived class and vice versa in a destructor. Therefore, two vptrs that overwrite each other are said to be related. It also checks VTable function entries to take advantage of the possibility that a derived class may not redefine every virtual function it inherits from its base class. Therefore, if two VTables contain pointers to the same function(s) at the same offset, they are said to be related. In order to improve coverage, Marx also performs inter-procedural data flow analysis, which uses forward edge analysis to resolve indirect control flow (to improve coverage for overwrite analysis) and backward edge analysis to take return values into account. Finally, it performs inter-modular data flow, which considers hierarchy from libraries to obtain a more comprehensive result. Even though these heuristics give strong indication about class relations, Marx only reconstructs

class hierarchy as a plain set, direction of inheritance is not inferred. According to the authors, class relations recovery is a hard problem in itself and information regarding the direction of the relation is not available in binaries [20].

We analyzed xalancbmk with Marx, Figure 3 shows its representation of a subset of the inheritance graph generated. Marx was able to correctly identify related vptrs, but they are only grouped into a set. In Figure 4, we show a mapping between the ground truth and our analysis. The figure shows that not only are we able to infer class relations, we are also able to infer the direction of inheritance correctly. In the figure, base classes are above the derived classes and the broken lines show nodes in the ground truth corresponding to nodes in our analysis while the solid lines show relationship between classes.

Table 4: Evaluating the amount of class hierarchy information OVT reveals at callsites. Column with “# types with associated callsites” contains the number of classes that has at least one corresponding callsite, “#overall types recovered + VTable scanning” contains number of classes recovered from the binary using both callsite information and VTable scanning, “#leaf node” contains number of classes with no base class, and “#non-leaf node” contains number of classes with at least one base class.

Program	From Analysis				GT	
	# types with associated callsites	#overall types recovered + VTable scanning	#leaf node	#non-leaf node	#leaf node	#non-leaf node
Spidermonkey	91	768	19	72	636	171
Xalanc	427	847	275	152	668	307
Soplex	23	30	12	11	17	12
Povray	14	30	6	8	20	12
Omnetpp	45	104	27	18	85	27
Namd	0	3	0	0	3	1
CplusplusThread	4	11	0	4	6	5



Figure 3: Marx’s representation of a subset of the inheritance graph generated from Xalanc

6 Related Work

6.1 C++ Attacks and Defenses

Due to the use of computed code pointers in dynamic dispatch, multiple attacks have targeted C++ programs including the recent COOP [26] attack that reuses existing virtual functions in order to accomplish malicious computation. With insufficient semantics in the binary, binary-level solutions – VTint [30], TVIP [10], vfGuard [21], RECALL [5] introduce imprecision in the defense. However, source-code based solutions (i.e., [4, 11, 15, 17, 27, 29]) must embed inheritance information into the binary in order to improve precision and offer interoperability, and therefore reveal design information.

6.2 C++ Reverse Engineering

VCI [6] reconstructs the class hierarchy of a program using constructor only analysis. Constructors give useful information about inheritance as well as its direction [23], however, due to constructor inlining, there are inadequate constructors in the binary to carry out a comprehensive static analysis. The

evaluation reported for this work shows a high number of false positive and false negative inferred inheritance relationships. Marx [20] uses heuristics to reconstruct class hierarchy, such as, overwrite analysis, similar VTable entries and return values. For comprehensive class hierarchy, it also analyzes libraries used by executables. Even though Marx can infer relationship with high precision, it ignores the direction of inheritance.

Fokin et al. [9] depends on rules including analyzing VTable sizes, checking for pure virtual functions, checking for parameters of virtual functions and checking if the vptr of a class is overwritten by that of another class. However they are also not able to give precise class hierarchy.

OOAnalyzer [24] combines traditional binary analysis, symbolic analysis and Prolog-based reasoning to group methods into classes (for both polymorphic and non-polymorphic classes). Methods called on the same pointers are identified and then reasoning rules are applied to decide if they belong to the same classes. The authors mentioned that inheritance can be assigned using class size and VTable sizes. However, no evaluation was done regarding this, therefore, we cannot confirm if OOAnalyzer can indeed decide inheritance.

TVIP [10], ensures that a VTable pointer points to the read only section before it is used, leveraging the fact that VTables are always located in read only memory. Class hierarchy was not used. Similarly, VTint [30] is developed on the basis that all legitimate VTables are stored only in the read only memory. They identified the possibility of an attacker reusing existing VTables which are in read only memory. They suggested that this can be handled by leveraging class hierarchy information. TypeArmor [28] determines an approximate number of arguments prepared at callsite and the number of arguments that a function expects. This information is used to restrict the functions that any given callsite can target.

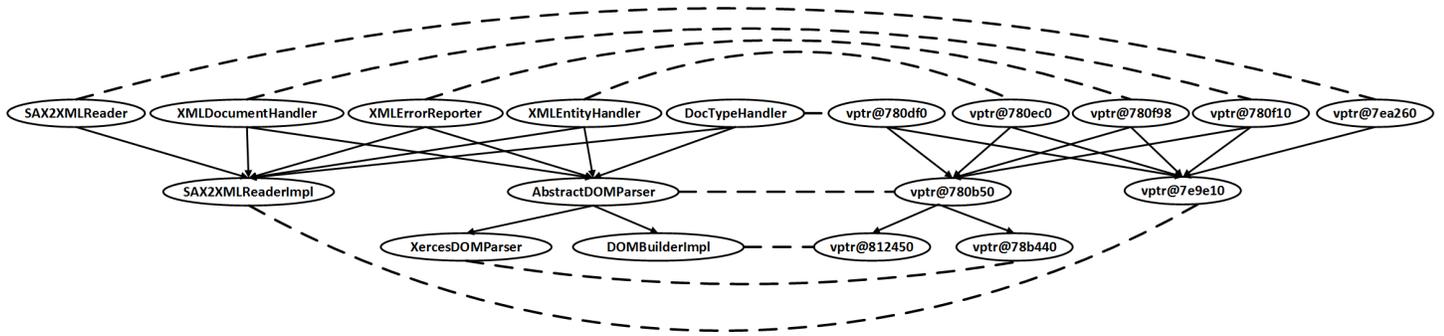


Figure 4: A mapping between a subset (for the reason of space) of the recovered class hierarchy from the ground truth and our analysis of Xalanc compiled with OVT.

7 Conclusion

In Summary, we have shown that most modern C++ defenses 1. embed metadata in the binary 2. and the metadata can be easily recovered. This makes reverse engineering the binary easier. In order to address this problem, we suggest the following:

1. Transform the metadata: This can be done either by encrypting the metadata or randomizing their layout so that recovering them does not reveal any meaningful information.
2. Avoid embedding metadata: CFIXX [19] and μ CFI [13] achieve this by dynamically deciding the only valid target of a callsite. Instead of statically identifying possible targets of a callsite, CFIXX dynamically tracks object type using a policy called Object Type Integrity. This makes it possible to identify the single allowable target, thereby providing better integrity. Similarly, μ CFI proposes another CFI called Unique Code Target which relies on Intel Processor Trace to dynamically record data. This data is used to augment points-to analysis which is used to identify the only allowable target of a callsite.

By incorporating design-revealing information in the binary, modern C++ defenses arguably pose a deterrent to usability in commercial products. While their precision in security is certainly appealing to the open-source software, it comes at the cost of privacy, which may not be acceptable in commercial software.

8 ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their valuable feedback. This research was supported in part by Of-

fice of Naval Research Grant #N00014-17-1-2929, National Science Foundation Award #1566532, and DARPA award #81192. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

- [1] “The IDA Pro Disassembler and Debugger,” <https://www.hex-rays.com/products/ida/>.
- [2] *A Novel Software Toolkit for Graph Edit Distance Computation*, 2013.
- [3] “Itanium C++ ABI,” <http://refspecs.linuxbase.org/cxxabi-1.83.html>, Revision: 1.83.
- [4] D. Bounov, R. G. Kıcı, and S. Lerner, “Protecting C++ dynamic dispatch through vtable interleaving,” in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS’16)*, 2016.
- [5] D. Dewey and J. T. Giffin, “Static detection of C++ vtable escape vulnerabilities in binary code,” in *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS’12)*, 2012.
- [6] M. Elsabagh, D. Fleck, and A. Stavrou, “Strict Virtual Call Integrity Checking for C++ Binaries,” in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS’17)*, 2017.
- [7] S. Fan, L. Xiaokang, X. Yulei, and J. Xiangke, “Boosting the Precision of Virtual Call Integrity Protection with Partial Pointer Analysis for C++,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’17)*, 2017.

- [8] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina, "SmartDec: Approaching C++ Decompilation," in *18th Working Conference on Reverse Engineering (WCRE)*, 2011.
- [9] A. Fokin, K. Troshina, and A. Chernov, "Reconstruction of class hierarchies for decompilation of C++ programs," in *14th European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.
- [10] R. Gawlik and T. Holz, "Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs," in *Proceedings of 30th Annual Computer Security Applications Conference (ACSAC'14)*, 2014.
- [11] I. Haller, E. Göktas, E. Athanasopoulos, G. Portokalidis, and H. Bos, "ShrinkWrap: VTable Protection without Loose Ends," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, 2015.
- [12] I. Haller, Y. Jeon, P. Hui, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "TypeSan: Practical Type Confusion Detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [13] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.
- [14] R. Ibragimov, M. Malek, J. Guo, and J. Baumbach, "GEDEVO: An Evolutionary Graph Edit Distance Algorithm for Biological Network Alignment," in *German Conference on Bioinformatics 2013*, 2013.
- [15] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," in *Proceedings of 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.
- [16] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "Hex-Type: Efficient Detection of Type Confusion Errors for C++," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.
- [17] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.
- [18] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03)*, 2003.
- [19] Nathan Burow and Derrick McKee and Scott A. Carr and Mathias Payer, "CFIXX: Object Type Integrity for C++ Virtual Dispatch," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, 2018.
- [20] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, "MARX : Uncovering Class Hierarchies in C++ Programs," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*, 2017.
- [21] A. Prakash, X. Hu, and H. Yin, "vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.
- [22] J. Ray, "C++: Under the Hood," <http://www.openrce.org/articles/files/jangrayhood.pdf>, 1994.
- [23] P. V. Sabanal and M. V. Yason, "Reversing C++," *Blackhat Security Conference*, 2007.
- [24] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, "Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.
- [25] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS'07)*, 2007.
- [26] F. Shuster, T. Tendyck, C. Liebchen, L. Davi, A.-r. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming, On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of 36th IEEE Symposium on Security and Privacy (Oakland'15)*, 2015.
- [27] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *Proceedings of 23rd USENIX Security Symposium (USENIX Security'14)*, 2014.
- [28] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level," in *Proceedings of IEEE Symposium on Security and Privacy (Oakland'16)*, 2016.

[29] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, “VTrust: Regaining Trust on Virtual Calls,” in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS’16)*, 2016.

[30] C. Zhang, C. Song, Z. K. Chen, Z. Chen, and D. Song, “VTint: Defending Virtual Function Tables’ Integrity,” in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS’15)*, 2015.

DECAF++: Elastic Whole-System Dynamic Taint Analysis

Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin
University of California, Riverside
{adava003,zqi020,yuq, hengy}@ucr.edu

Abstract

Whole-system dynamic taint analysis has many unique applications such as malware analysis and fuzz testing. Compared to process-level taint analysis, it offers a wider analysis scope, a better transparency and tamper resistance. The main barrier of applying whole-system dynamic taint analysis in practice is the large slowdown that can be sometimes up to 30 times. Existing optimization schemes have either considerable baseline overheads (when there is no tainted data) or specific hardware dependencies. In this paper, we propose an elastic whole-system dynamic taint analysis approach, and implement it in a prototype called DECAF++. Elastic whole-system dynamic taint analysis strives to perform taint analysis as least frequent as possible while maintaining the precision and accuracy. Although similar ideas are explored before for process-level taint analysis, we are the first to successfully achieve true elasticity for whole-system taint analysis via pure software approaches. We evaluated our prototype DECAF++ on nbench, apache bench, and SPEC CPU2006. Under taint analysis loads, DECAF++ achieves 202% speedup on nbench and 66% speedup on apache bench. Under no taint analysis load, DECAF++ imposes only 4% overhead on SPEC CPU2006.

1 Introduction

Dynamic taint analysis (also known as dynamic information flow tracking) marks certain values in CPU registers or memory locations as *tainted*, and keeps track of the tainted data propagation during the code execution. It has been applied to solving many program analysis problems, such as malware analysis [17, 28, 29], protocol reverse engineering [5], vulnerability signature generation [24], fuzz testing [26], etc.

Dynamic taint analysis can be implemented either at the process level, or at the whole system level. Based on process-level instrumentation frameworks such as Pin [19], Valgrind [23], and StarDBT [3], process-level taint analysis tools like LibDft [16], LIFT [25], Dytan [7] and Minemu [4] keep track of taint propagation within a process scope. Whole-system taint analysis tools (e.g., TaintBochs [6], DECAF [12]

and PANDA [10]) are built upon system emulators (e.g., Bochs [20] and QEMU [2]), and as a result can keep track of taint propagation throughout the entire software stack, including the OS kernel and all the running processes. Moreover, whole-system dynamic taint analysis offers a better transparency and temper resistance because code instrumentation and analysis are completely isolated from the guest system execution within a virtual machine; in contrast, process-level taint analysis tools share the same memory space with the instrumented process execution.

However, these benefits come at a price of a much higher performance penalty. For instance, the most efficient implementation of whole-system taint analysis to our knowledge, DECAF [12], incurs around 6 times overhead over QEMU [12], which itself has another 5-10 times slowdown over the bare-metal hardware. This overhead for tainting is paid constantly no matter how much tainted data is actually propagated in the software stack.

To mitigate such a performance degradation introduced by dynamic taint analysis, some systems dynamically alternate between the execution of program instructions and the taint tracking ones [13, 25]. For instance, LIFT [25] is based on the idea of alternating execution between an original target program (fast mode) and an instrumented version of the program containing the taint analysis logic. Ho et al. proposed the idea of *demand emulation* [13], that is, to perform taint analysis via emulation only when there is an unsafe input.

Despite the above, there are still some unsolved problems in this research direction. First, LIFT [25] works at the process level, which means LIFT has the aforementioned shortcomings of process level taint analysis. Moreover, LIFT still has to pay a considerable overhead for checking registers and the memory in the fast mode. Second, the demand emulation approach [13] has a very high overhead in switching between the virtualization mode and the emulation mode [13]. Third, some optimization approaches depend on specific hardware features for acceleration [4, 16, 25].

In this paper, we propose solutions to solve these problems, and provide a more flexible and generally applicable

dynamic taint analysis approach. We present DECAF++, an enhancement of DECAF with respect to its taint analysis performance based on these solutions. The essence of DECAF++ is *elastic* whole-system taint analysis. Elasticity, here, means that the runtime performance of whole-system taint analysis degrades gracefully with the increase of tainted data and taint propagation. Unlike some prior solutions that rely on specific hardware features for acceleration, we take a pure software approach to improve the performance of whole-system dynamic taint analysis, and thus the proposed improvements are applicable to any hardware architecture and platform.

More specifically, we propose two independent optimizations to achieve the elasticity: elastic taint status checking and elastic taint propagation. DECAF++ elasticity is built upon the idea that if the system is in a safe state, i.e., there is no data from taint sources, there is no need for taint analysis as well. Henceforth, we access the shadow memory to read the taint statuses only when there is a chance that the data is tainted. Similarly, we propagate the taint statuses from the source to the destination operand only when any of the source operands are tainted.

We implemented a prototype dubbed DECAF++ on top of DECAF. Our introduced code is around 2.5 KLOC including both insertions and modifications to the DECAF code. We evaluated DECAF++ on nbench, SPEC CPU2006, and Apache bench. When there are tainted bytes, we achieve 202% (18% to 328%) improvement on nbench integer index, and on average 66% improvement on apache bench in comparison to DECAF. When there are no tainted bytes, on SPEC CPU2006, our system is only 4% slower than the emulation without instrumentation.

Contributions In summary, we make the following contributions:

- We systematically analyze the overheads of whole system dynamic taint analysis. Our analysis identifies two main sources of slowdown for DECAF: taint status checking incurring 2.6 times overhead, and taint propagation incurring 1.8 times overhead.
- We propose an elastic whole-system dynamic taint analysis approach to reduce taint propagation and taint status checking overhead that imposes low constant and low transition overhead via pure software optimization.
- We implement a prototype based on elastic tainting dubbed DECAF++ and evaluate it with three benchmarks. Experimental results show that DECAF++ incurs nearly zero overhead over QEMU software emulation when no tainted data is involved, and has considerably lower overhead over DECAF when tainted data is involved. The taint analysis overhead of DECAF++ decreases gracefully with the amount of tainted data, providing the elasticity.

2 Related Work

2.1 Hardware Acceleration

Related works on taint analysis optimization focus on a single architecture [4, 16, 25]. Henceforth, they utilize the capabilities offered by the architecture and hardware to accelerate the taint analysis. LIFT uses x86 specific LAHF/SAHF instructions to accelerate the context switch between the original binary code and the instrumented code. Minemu uses X86 SSE registers to store the taint status of the general purpose registers, and fails if the application itself uses these registers. libdft uses multiple page size feature on x86 architectures to reduce the Translation Lookaside Buffer (TLB) cache miss for the shadow memory. In this work, we stay away from these hardware-specific optimizations, and rely only on software techniques to make our solution architecture agnostic.

2.2 Shadow Memory Access Optimization

Minimizing the overhead of shadow memory access is crucial for the taint analysis performance. Most related works reduce this overhead by creating a direct memory mapping between the memory addresses and the shadow memory [4, 16, 25]. This kind of mapping removes the lookup time to find the taint status location of a given memory address. The implementation of the direct mapping requires a fixed size memory structure. This fixed size structure to store the taint status is practical only for 32-bit systems; to support every application, such an implementation requires 32 TB of memory space on 64 bits systems [16]. Even on 32-bit systems, the implementation usually incurs a constant memory overhead of around 12.5% [16]. Minemu furthers this optimization and implements a circular memory structure that rearranges the memory allocation of the analyzed application. The result is that it quickly crashes for applications that have a large memory usage [4]. In this work, we aim to follow a dynamically managed shadow memory that can work not only for applications with large memory usage but also for 64-bit applications.

In addition to the above, LIFT [25] coalesces the taint status checks to reduce the frequency of access to the shadow memory. To this end, LIFT needs to know ahead of time what memory accesses are nearby or to the same location. This requires a memory reference analysis before executing a trace. LIFT scans the instructions in a trace and constructs a dependency graph to perform this analysis. LIFT reports that this optimization is application dependent (sometimes no improvement), and depends on what percentage of time the taint analysis is required for the application. LIFT is a process level taint analysis tool, and in case of whole system analysis, we expect the required taint analysis percentage for a program to be very low in comparison to the size of the system. Henceforth, we do not expect this optimization to

be very useful for whole system analysis given the constant overhead of performing memory reference analysis for the entire system.

2.3 Decoupling

Several related works reduce the taint analysis overhead by decoupling taint analysis from the program execution [14, 15, 18, 21, 22]. ShadowReplica [14] decouples the taint analysis task and runs it in a separate thread. TaintPipe [22] parallelizes the taint analysis by pipelining the workload in multiple threads. StraightTaint [21] offloads the taint analysis to an offline process that reconstructs the execution trace and the control flow. LDX [18] performs taint analysis by mutating the source data and watching the change in the sink. If the sink is tainted, the change in the source would change the sink value. LDX reduces the overhead by spawning a child process and running the analysis in the spawned process on a separate CPU core. RAIN [15] performs on-demand dynamic information level tracking by replaying an execution trace when there is an anomaly in the system. RAIN reduces the overhead by limiting the replay and the analysis to a few processes in the system (within the information flow graph) based on a system call reachability analysis. Our work complements these works as we aim to separate the taint analysis from the original execution.

2.4 Elastic Tainting

Elastic taint propagation Similar ideas to elastic tainting have been explored in the previous works [13, 25]. Qin et al. introduced the idea of fast path optimization [25]. Fast path optimization is based on the notion of alternating execution between a target program and an instrumented version of the program including the taint analysis logic. The former is called check execution mode, and the latter is called track execution mode. Qin et al. presented this idea for process level taint analysis. We build upon this idea for whole system analysis. While the intuition behind both elastic tainting and fast optimization is the same, our work advances Qin et al.’s work for the whole system.

Our first novelty is that we reduce the overhead in the check mode. LIFT [25] checks registers and memory locations of every basic block regardless of the mode. Note that this overhead in system level analysis is a major issue because it affects every process (and the kernel) as we show in §3.3. We reduce the overhead by releasing DECAF++ from checking registers in the check mode and instead monitor the taint sources, data from input devices or memory locations, directly. Combining this with our low overhead taint checking, we reduce the overhead in the check mode to nearly zero as we show in §6.4. Our second novelty is that, unlike LIFT [25], we implement elastic tainting in an architecture agnostic way. Meeting this requirement while obtaining a low overhead is technically

challenging. As an example, LIFT uses a simple jump to switch modes while such a jump in our case would panic the CPU since a single guest instruction might break into several host binary instructions that need to be executed atomically.

Finally, the effectiveness of elastic taint propagation for whole system taint analysis has not been investigated before. As we show in §6, this optimization for whole-system taint analysis is application dependent and needs to be accompanied with a taint status checking optimization. Our work is the first that shows the elastic property through comprehensive evaluations, and provides a means to compare the elastic taint propagation with elastic taint status checking.

Elastic taint status checking Ho et al. present the idea of demand emulation [13] that has elements in common with our elastic taint status checking. The demand emulation idea is to perform taint analysis via emulation only when there is an unsafe input e.g. network input in their case. Otherwise, the system is virtually executed without extra overhead. Demand emulation idea looks enticing because the virtualization overhead is usually lower than emulation. However, as Ho et al. state, the transition cost between virtualization and emulation is quite high and possibly offsets the speedup gained through the virtualization. In contrast, as we show in §6.4, our elastic tainting incurs almost zero transition overhead. Further, Ho et al. had to modify the underlying target operating system to provide efficient support for demand emulation. In contrast, in this work, we implement elastic tainting using only emulation without any modifications to the target systems, and show substantial improvements in real world applications of taint analysis.

3 Whole-System Dynamic Tainting

In this section, we introduce a basic background knowledge on DECAF, mainly focusing on its taint analysis functionality related to this work. For further details on QEMU, the underlying emulator, we refer the readers to [Appendix A](#).

3.1 Taint Propagation

DECAF defines how instructions affect the taint status of their operands. Going to the details of the rules for every instruction is out of the scope of the current work; an interested reader can refer to [27]. Just to give an idea, *mov* instructions in x86 result in a corresponding *mov* of the taint statuses from the source to the destination (see [Figure 1](#)). What is important about the DECAF taint propagation is that DECAF inserts a few instructions before every Tiny Code Generator (*TCG*) IR instruction that do the following:

- Read the taint status of the source operand. The taint status depending on the operand type can be in the shadow registers, temporary variables or in the shadow memory.

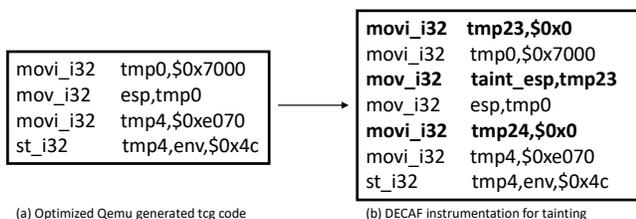


Figure 1: DECAF tcg instrumentation to apply tainting for the instruction `mov $0x7000, %esp`. (a) shows the tcg IR after translating the guest `mov` instruction, and (b) shows the tcg IR after applying the taint analysis instrumentation.

- Decide the taint status of the destination operand based on the instruction tainting rule. To implement the tainting rule, a few TCG IRs are inserted before each IR to propagate the taint status.
- Write the taint status of the destination operand to its shadow variable.

3.2 Shadow Memory

DECAF stores the taint statuses of the registers and the memory addresses respectively in global variables and in the shadow memory (allocated from heap). DECAF does not make any assumptions about the memory and can support any application with any memory requirements. The shadow memory associates the taint statuses with *guest physical addresses*. This is a key design choice because the taint analysis is done at the system level, and hence, virtual addresses point to different memory addresses in different processes.

DECAF stores the taint statuses in a two-level tree data structure. The first level points to a particular page. The second level stores the taint statuses for all the addresses within a physical page. This design is based on the natural cache design of the operating systems, and hence makes use of the temporal and spatial locality of memory accesses.

DECAF instruments QEMU memory operations to maintain the shadow memory. Instrumentation is in fact on the Tiny Code Generator (TCG) Intermediate Representation (IR). DECAF instruments the IR instruction for memory load, `op_qemu_ld*`, and the IR instruction for memory store, `op_qemu_st*`. For load, DECAF loads the taint status of the source operand of the current instruction along with the memory load operation. For store, DECAF stores the taint status of the destination operand of the current instruction to its corresponding shadow memory along with the memory store operation. In both cases, the load (or store) is to (or from) a global variable named `temp_idx`.

3.3 Taint Analysis Overhead

We analyzed the current sources of slowdown in DECAF. There are three sources of slowdown in DECAF:

- The QEMU emulation overhead that is not inherent to the DECAF taint analysis approach but rather an inevitable overhead that enables dynamic whole system analysis. That said, QEMU is faster than other emulators like Bochs [20] by several orders of magnitude [2].
- The taint propagation overhead as explained in §3.1.
- The taint status checking as explained in §3.2.

After applying DECAF tainting instrumentation, the final binary code is on average 3 times the original QEMU generated code according to Table 1. Clearly, the additional inserted instructions (after the instrumentation) impose an overhead.

Table 1: The statistical summary of the blowup rate after the instrumentations. The numbers show the ratio of the code size to a baseline after an instrumentation. QEMU baseline is the guest binary code, and DECAF baseline is QEMU IR code. DECAF increases the already inflated QEMU generated code size around 3 times on average.

System	Component	Min	Median	Mean	Max
QEMU	lifting binary to tcg	3.33	6.75	7.13	28.00
DECAF	taint checking & propagation	1.25	3.12	2.94	5.14

We systematically analyzed the overheads of DECAF framework, i.e., taint propagation and taint status checking. To measure each overhead, we isolated the codes from DECAF that would cause the overhead by removing other parts. For taint propagation overhead measurement, we removed the shadow memory operations by disabling the memory load and store patching functionality of DECAF that adds the shadow memory operations. For taint status checking overhead measurement, we removed the taint propagation functionality from DECAF by deactivating the instrumentation that implements the taint propagation rules.

We measured the performance of the isolated versions of DECAF using nbench benchmark on a windows XP guest image with a given 1024MB of RAM. The experiment was performed on an Ubuntu 18 i686 host with a Core i7 3.5GH CPU and 8GB of RAM. Figure 2 illustrates the result of our analysis. Figure 2 reports the geometric mean of the nbench reported indexes normalized using a baseline. The baseline is the DECAF without the taint analysis functionality outright, that is, DECAF with only Virtual Machine Introspection (VMI). The result shows that on nbench, taint analysis slows down the system 400%. But more important than that, Figure 2 shows that taint propagation alone slows down the system about 1.8 times while taint status checking alone adds a 2.5 times slowdown.

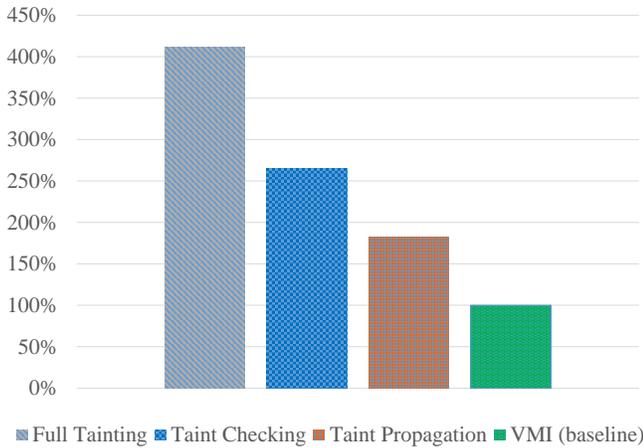


Figure 2: Breakdown of overhead given the DECAF VMI as baseline.

4 Elastic Taint Propagation

4.1 Overview

Elastic taint propagation aims to remove the taint propagation overhead whenever possible. It is based on the intuition that taint analysis can be skipped if the taint analysis operation does not change any taint status value. Taint analysis operations can possibly result in a change only when either of the source or the destination operand of an instruction is tainted.

Two modes Based on the above intuition, we define two modes with and without the taint propagation overhead. We name the mode with taint propagation operations *track mode*, and the mode without taint propagation operations *check mode*. At any given time, the execution mode depends on whether any CPU register is tainted.

Mode transition When the system starts, no tainted data exists in the system, so the system runs in the check mode. The execution switches to the track mode when there is an input from a taint source. The taints propagate in the track mode until the propagation converges and the shadow registers are all zero (clear taint status). At this point, the execution switches back to the check mode. Finally, either based on an input or a data load from a tainted memory address the execution switches back to the track mode. Figure 3 shows when transition occurs between the track and the check mode.

4.2 Execution Modes

An execution mode determines the way a block should be instrumented. Each mode has its set of translation blocks. Further, each mode has its own cache tables. The execution in a mode can flow only within the same mode translation blocks. This means that blocks only from the same mode would be

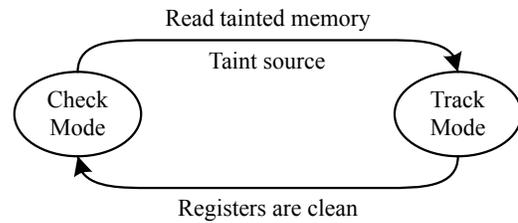


Figure 3: State transitions between check mode and track mode

chained together. We determine the mode using a flag variable. Based on the mode, the final code would be instrumented with or without the taint propagation instructions. The generated code will be reused for execution based on the execution mode unless invalidated. A cache invalidation request invalidates the generated codes regardless of the mode. The set of translation blocks and code caches virtually form an exclusive copy of the translated code for the execution mode.

Check mode The generated code in the check mode is the original guest code (program under analysis) plus the instrumentation code for memory load and memory store. For memory load, shadow memory is checked, and if any byte is tainted, the mode is switched to the track. In §5, we further explain how we efficiently perform this checking. For memory store, the destination operand taint status will be cleaned because any propagation in this mode is safe.

Track mode Code generated in track mode is the same as the one generated in the original DECAF. Readers can refer to §3.1 and §3.2 for further details.

4.3 Transition

A key challenge after having two execution modes in place is to decide when and how the transition between the two modes should occur. The transition between the two modes should affect neither the emulation nor the taint analysis correctness. The execution should immediately stall in the check mode and resume in the track mode when there is a data load from a taint source. Further, this mode switch should happen smoothly without panicking the CPU.

We need to monitor data flow from the taint sources and the shadow registers for timely transition between the modes. In the check mode, we only monitor the taint sources. This is a key design choice for performance because monitoring registers for timely transition is very costly. Note that to implement register monitoring in the check mode, we would need to check the register taint statuses before *every instruction*. Thus, in the check mode, to reduce the overhead, we only monitor the taint sources without losing the precision. In the track mode, we can check the registers less often because

longer execution in this mode neither affects the taint analysis precision nor the emulation correctness.

Input devices monitoring The taint sources are generally memory addresses of the *input* devices like keyboards or network cards. For the input devices, DECAF++ relies on the monitoring functionality implemented in DECAF. However, we slightly modify the code to raise an exception whenever there is an input. This exception tells the system that it is in an unsafe state because of the user input and the track mode should be activated if not before.

Memory monitoring In addition to the input devices, we should also monitor the *memory load* operations. This is because after processing the data from an input device, the data might propagate to other memory locations and pollute them. We need to track the propagation in the check mode as soon as a tainted value is loaded from memory for further processing. Efficient design and implementation of memory monitoring is a key to our elastic instrumentation solution. We elaborate on how we do this efficiently in §5.

Registers monitoring Monitoring the registers is a key to identifying when we can stop the taint propagation. If none of the registers carry a tainted value, no machine operations except the memory load can result in a tainted value. Henceforth, we can safely stop the execution in the track mode and resume the execution in the check mode while carefully monitoring the memory load operations as explained earlier.

We point out that monitoring registers in the track mode has low overhead. This is because in the track mode, we can tolerate missing the exact time that the registers are clean without affecting either the safety or the precision (we would propagate zero). Therefore, DECAF++ can check the cleanliness of the registers in the track mode at block (instead of instruction) granularity.

We check the registers' taint status either after the execution of a chain of blocks, or when there is an execution exception (including the interrupts). Our experiments confirm that this is a fine granularity given its lower overhead comparing to an instruction level granularity approach. If all the registers have a clean taint status, we resume the execution for the next blocks in the check mode.

Transition from check mode to track mode Unlike the transition from the track mode to the check mode (always in the beginning of a block), the transition from the check to the track can happen anywhere in the block depending on the position of an I/O read or a load instruction. However, the execution of a single guest instruction should start from the beginning to the end, note that a single guest instruction might

be translated to several TCG instructions¹; otherwise, the result of the analysis would be both invalid and unsafe. This is because the block code copies in each mode are different and the current instruction might have dependency on the former instructions that are not executed in the current mode. To have a smooth transition, the following steps should be followed:

- (1) Restore CPU state: before we switch the code caches, we need to restore the CPU state to the last successfully executed instruction. We need to restore the CPU state to avoid state inconsistency. Since the corresponding execution block in the other mode is different, we can not resume the execution from the same point; the same point CPU state is not consistent with the new mode block instructions. To restore the CPU state, we re-execute the instructions from the beginning of the block to the last successfully executed guest instruction. This will create a CPU state that can be resumed in the other mode.
- (2) Raise exception: after restoring the CPU state, we emulate a custom exception: *mem_tainted*. We set the exception number in the *exception_index* of the emulated CPU data structure. After that, we make a long jump to the QEMU main loop (*cpu-exec* loop).
- (3) Switch mode: in the QEMU main execution loop, we check the *exception_index* and change the execution mode if the exception is *mem_tainted*. We switch the mode by changing mode flag value that instructs us how we should instrument the guest code (track or check).

After switching the modes, QEMU safely resumes the corresponding block execution in the new mode because we restored the CPU state in the step 1.

5 Elastic Taint Status Checking

The main idea behind reducing the taint status checking overhead is to avoid unnecessary interactions with the shadow memory. In DECAF, the taint status checking happens for every memory operation. However, we can avoid the overhead per memory address if we perform the check for a larger set of memory addresses. Thus, if the larger set doesn't contain any tainted byte we can safely skip the check per address within that set. The natural sets within a system are physical memory pages.

DECAF++ scans physical pages while loading them in TLB, and decides whether or not to further inspect the individual memory addresses. We modified the TLB filling logic of QEMU according to Figure 4. The modifications are highlighted. The figure illustrates that if the page contains any tainted byte, DECAF++ sets a shadow memory handler for

¹ For instance, a single x86 "ADD m16, imm16" instruction will be translated to three TCG IR instructions; one load, one add and one store

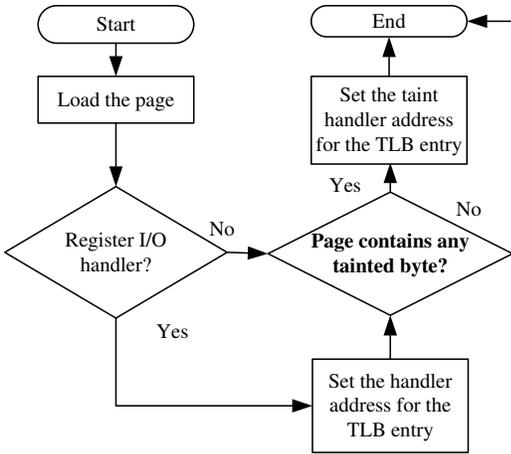


Figure 4: Fill TLB routine

the page through some of the TLB entry control bits. Afterwards, whenever this page is accessed, DECAF++ redirects the requests to the shadow memory handler. In the following paragraphs, we explain how we handle memory load and store operations separately because of their subtle differences.

Memory load During memory load operations, we should load the taint status of the source memory address operands as well. We load the taint status value from the shadow memory only when the TLB entry for the page contains the shadow memory handler. In other cases, we can safely assume that the taint status is zero. Based on this notion, we modify the QEMU memory load operation logic as shown in Figure 5a. In particular, two cases might occur:

- If the TLB entry control bits for the page contain the shadow memory handler, the address translation process for the memory load operations results in a TLB miss. In the TLB miss handler routine, if the control bits indicate that the shadow memory handler should be invoked we do so and load the taint status for the referenced address from the shadow memory. If the execution is not already in the track mode, and the loaded status is not zero we quickly switch to the track mode.
- If the address translation process for the load operation results in a TLB hit, we check whether we are in the track mode, and if so we load zero as the taint value status. Otherwise, we don't need to load the taint status since it will not be used for taint propagation.

Based on the locality principle, a majority of accesses should go through the fast path shown in the Figure 5a. Our elastic taint status checking is designed not to add overhead to this fast path, and hence a performance boost is expected.

Memory store During memory store operations, we should store the taint status of the source operand to the shadow mem-

ory address of the destination referenced memory address. Similar to memory loads, we perform the shadow memory store operation only when the TLB entry control bits for the referenced address page indicate so. That said, there is a subtle difference that makes memory stores costlier than memory loads. Figure 5b shows how we update the shadow memory alongside the memory store operations. In particular, there will be three cases:

- If the source operand for the store operation has a zero taint status, and the page TLB entry does not indicate a tainted page, we do nothing. This happens both in the check mode all the time, and in the track mode when the page to be processed does not contain any tainted byte.
- If the page TLB entry flags us to inspect the shadow memory, we check the TLB control bits in the TLB miss handler and update the shadow memory if the shadow memory handler is set (see Figure 5b).
- If the taint status of the source operand is not zero, even if the page is not registered with a shadow memory handler, we still update the shadow memory. This can only happen in the track mode, because in the check mode there is no taint propagation, and hence the source operand taint status is always zero. We update the TLB entry when this is the first time there is a non-zero taint value store to the page. Since the page now contains at least a tainted byte, all the next memory operations involving this page should go through the shadow memory handler. We update the TLB entry and register the shadow memory handler for the future operations.

Propagation of non-tainted bytes A special case of the taint status store is when a tainted memory address is overwritten with non-tainted data. This case happens when the TLB entry for the page flags shadow memory operation even when the source operand is not tainted. In such a case, the memory address taint status would be updated to zero but the page is still processed as unsafe; the memory operations still will go through the taint handler. For performance reason, we do not immediately reclaim the data structure containing the taint value, but rely on a garbage collection (see Appendix B) mechanism that would be activated based on an interval. The page remains unsafe until the garbage collector is called. The garbage collector walks through the shadow memory data structure and frees the allocated memory for a page if no byte within the page is tainted. After this point, Fill TLB routine will not set the taint handler for the page anymore, and any processing involving the page will take the fast path.

6 Evaluation

In this section, we evaluate DECAF++, a prototype based on the elastic whole system dynamic taint analysis idea. DECAF++ is a fork of DECAF project including the introduced

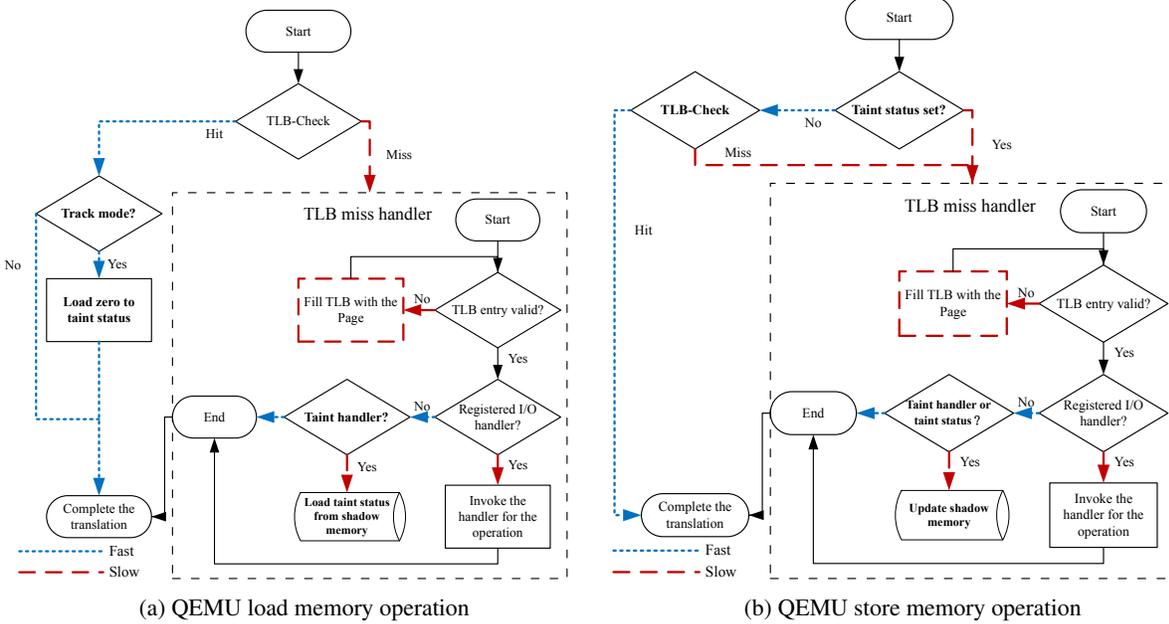


Figure 5: Elastic shadow memory access workflow

optimizations. Overall, our changes (insertion or deletion of code) to develop our prototype on top of DECAF does not exceed 2.5 KLOC. We defined two compilation options that selectively allows activating elastic taint propagation or elastic taint status checking. We evaluate DECAF++ to understand:

1. How effective each of our optimization, i.e., elastic taint propagation and elastic taint status checking and altogether is in terms of performance.
2. Whether our system achieves the elastic property for different taint analysis applications, that is a gradual degradation of performance based on the increase in the number of tainted bytes.
3. What the current overheads of DECAF++ are and whether we can address the shortcomings of the previous works [13, 25], i.e., reducing the overhead in the check mode and in the transition between the two modes.

6.1 Methodology

We measure the performance metrics using standard benchmarks under two different taint analysis scenarios in §6.2 and §6.3. In both scenarios, a virtual machine image is loaded in DECAF++ and a benchmark measures the performance of the virtual machine while the taint analysis task is running.

To answer (1), we measure the performance of DECAF++ with different optimizations, i.e., with elastic taint propagation dubbed as *Propagation*, with elastic taint status checking dubbed as *Memory* and with the both dubbed as *Full* and compare them. To answer (2) and evaluate the elastic property, we introduce a parameter in our taint analysis plugin that

adjusts the number or the percentage of tainted bytes. Plotting the performance trend based on this parameter values allows answering question (2). Finally to answer (3), we measure the overhead of the frequent or costly tasks in our implementation. In the rest of this section, we describe the details and answer (1) and (2) in §6.2 and §6.3. In §6.4, we answer (3).

6.2 Intra-Process Taint Analysis

In this scenario, we track the flow of information within a single process. For this experiment, we use nbench benchmark [11]. We track the flow of information within the nbench programs using a taint analysis plugin we developed for DECAF. The goal is to be able to report the performance indexes measured by nbench while the taint analysis task is running. In the next paragraphs, we explain the configurations for the experiment, and at the end of this section we report the results.

nbench Understanding nbench is important since our taint analysis plugin instruments it. nbench has 10 different programs. These programs implement a popular algorithm and measure the execution time on their host. Although the underlying algorithms are different, they all follow the same pattern regarding loading the initial data. They all create an array of random values and then run the algorithm on that array. The arrays are allocated from the heap, and the random generator is a custom pseudo random generator.

Taint analysis plugin The taint analysis plugin instruments nbench programs and taints a portion of the initial input data based on a given parameter. Although this is not trivial, we do

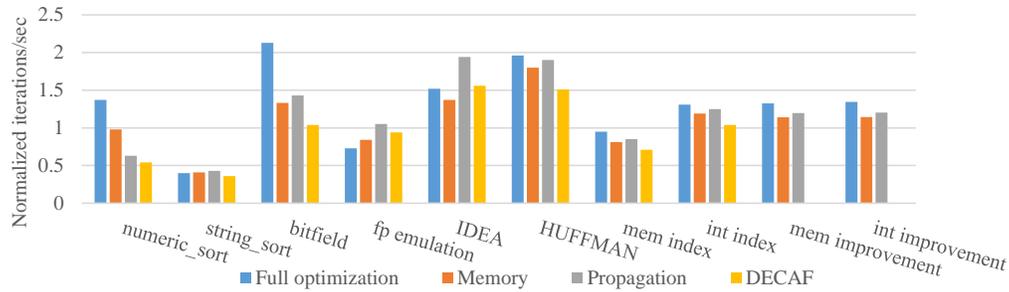


Figure 6: Comparing DECAF and DECAF++ performance.

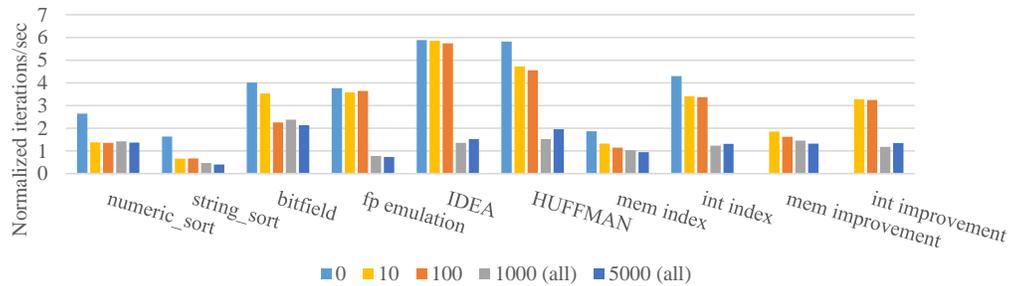


Figure 7: Evaluation of DECAF++ with full optimization under different number of tainted bytes; performance values are normalized by nbench based on a AMD k6/233 system.

not go into the details. We just mention that we record the address of the allocated array from the heap, and taint a portion of the array right after the random initialization. The portion size depends on an input parameter that we call *taint_size*. For instance, *taint_size*=100 means that 100 bytes of the array (from the beginning) used in the running programs of nbench are tainted using the plugin. After the instrumentation, DECAF automatically tracks the propagation from the the taint sources to other memory locations.

Experiments setup We measure the performance of each solitary optimization feature (and together) of DECAF++ using nbench² on a loaded windows XP guest image. The image is given 1024MB of RAM. The experiment was performed on an Ubuntu18 i686 host with a Core i7 3.5GH CPU and 8GB of RAM. The reported indexes by nbench are the mean result of many runs (depending on the system performance). Further, nbench controls the statistical reliability of the results and reports if otherwise. Finally, note that since all the measurements are conducted on the VM after it is loaded, the VM overhead would be a constant that is the same for all the measurements.

Result Figure 6 shows the performance of different optimization in DECAF++ in comparison to DECAF. The results

²Three programs Fourier, NEURAL NET and LU DECOMPOSITION from the nbench did not reflect any change in their reported numbers so we removed them from analysis. Also due to cross compilation, Assignment test did not work on Windows XP.

in this figure answers question (1) for this scenario. Overall, when the entire program inputs are tainted, combining elastic taint propagation and elastic taint status checking (full optimization) achieves the best performance. However, the performance is application dependant and sometimes a single optimization can achieve better performance than both combined, e.g. HUFFMAN and IDEA.

Figure 7 shows the performance of the DECAF++ when both optimization are activated for varied *taint_size* values. This figure answers question (2): DECAF++ has the elastic property, that is, the performance degrades based on the number of tainted bytes. On average, in comparison to DECAF, DECAF++ achieves on average 55% improvement (32% to 86%) on the nbench memory index and 202% (18% to 328%) on the nbench integer index.

Further, we can see from Figure 7 that results for *taint_size*={10,100} are similar and differ from the result for *taint_size*={1000,5000}. For *taint_size*={10,100}, since the tainted bytes are adjacent, the track mode activates for a sequence of bytes and then quickly switches back to the check mode. Also since *taint_size*={10,100} is well below a page size, the shadow memory access penalty would be low because often the tainted bytes will be within a single page. However for *taint_size*={1000,5000}, almost the entire nbench programs input array is tainted that results in frequent execution in the track mode and shadow memory access penalty.

In addition to the above, an interesting observation is the performance degradation for the sort algorithms. The performance degrades abruptly while *taint_size* changes from zero

to greater values. This is because of the behavior of the sort algorithm, that is, frequently moving an element in the array. This behavior results in polluting the entire array quickly and hence degrading the performance abruptly.

6.3 Network Stack Taint Analysis

In this scenario, the taint analysis tracks the flow of information from the network throughout the entire system and every process that accesses the network data. Performing taint analysis in this level is only possible using whole-system taint analysis tools like DECAF. Since the taint analysis affects the entire system, the need of having an elastic property would be more necessary.

Honeypots are an instance of the applications that can greatly benefit from the elastic property. Previous studies show that the likelihood of the malice of a network traffic can be predetermined [8]. Therefore, a honeypot can adopt a policy to achieve taint analysis only for network traffic that are expected to be malicious. Elastic property helps such systems to boost their performance based on their policy.

We measure transfer rate and throughput based on a parameter called `taint_perc` (instead of `taint_size` in the previous experiment) that defines the percentage of network packets to be tainted. This is because percentage, here, better represents the real applications. For instance, for honeypots, the `taint_perc` can be easily derived based on the taint policy.

Taint analysis plugin Our taint analysis plugin taints the incoming network traffic based on the `taint_perc` parameter. We implemented this plugin using the callback functionality of the DECAF. Our plugin registers a callback that is invoked whenever the network receive API is called. Then, based on the `taint_perc` parameter, our plugin decides whether to taint the payload or not.

Experiments setup The experiments were performed on an Ubuntu16.04 LTS host with a Core i7 6700 3.40GHz×8 CPU and 16GB of RAM. The guest image was Ubuntu 11.10 and it was given 4GB of RAM. For throughput measurement, we use Apache 2.2.22. We isolate the network interface between the server (guest image running Apache) and the client (the host machine) to reduce the network traffic noise that might perturb the results. That said, there is still a large deviation in the throughput because of the non-deterministic interrupt processing behavior of the system. We rely on significantly different values considering the standard deviation to draw conclusions.

We use netcat to measure the transfer rate. Our measurement is based on the transfer rate for 200 netcat requests of size 100KB. We use apache bench [1] to measure the throughput of an apache web server on the guest image. We execute Apache bench remotely from the host system with a fixed 10000 request parameter. Apache bench sends 10000 requests

Table 2: Network transfer rate of solitary features of DECAF++ (and together as Full) on Netcat; the throughput is the mean of 5 measurements for a range of tainted bytes.

Tainted Bytes	Implementation	Transfer Rate (MB/S)	Standard Deviation
40KB - 50KB	Full	3.57	18%
	Memory	3.60	10%
	Propagation	3.43	8%
20KB - 30KB	Full	5.70	3%
	Memory	4.25	14%
	Propagation	3.70	12%
0KB	Full	5.31	5%
	Memory	5.24	7%
	Propagation	4.12	8%
0KB - 50KB	DECAF	3.70	9%
0KB - 50KB	QEMU	6.00	3%

and reports the average number of completed requests per second. For both transfer rate and the throughput, we repeat the experiments for each `taint_perc` parameter value 5 times and report the average and the relative standard deviation.

Transfer rate result Table 2 reports the result of our transfer rate measurement using netcat. The results show the transfer rate for three `taint_perc` parameter values: when every packet is tainted (40KB - 50KB tainted bytes), when half of the number of packets are tainted (20KB - 30KB) and finally when no packet is tainted. Note that although every request is 100KB, only a portion of the packet is payload, and not the entire 100KB payload would be live in the system at the same time; this is why eventually only around 50KB is tainted. *The results show a substantial 54% improvement when only half of the incoming packets are tainted.* This is only 5% less than the QEMU transfer rate that is the maximum we can achieve. There is no improvement when every packet is tainted but this is expected because taint propagation and taint status checking have to be constantly done.

Throughput result Figure 8 illustrates the result of our throughput measurement for apache server using apache bench. The figure shows that the full optimization achieves the best throughput. Answering (1), full optimization and elastic taint status checking outperform DECAF for all values of `taint_perc`, and elastic taint propagation outperforms DECAF when `taint_perc` is below 1%. Figure 9 shows the performance of the full optimization based on the percentage of the tainted bytes. Although DECAF++ has the elastic property and there is improvement in all cases (answering (2)), it is more tangible for `taint_perc` values less than 1%. We point out two points on why `taint_perc` values seem very small. First, the number of

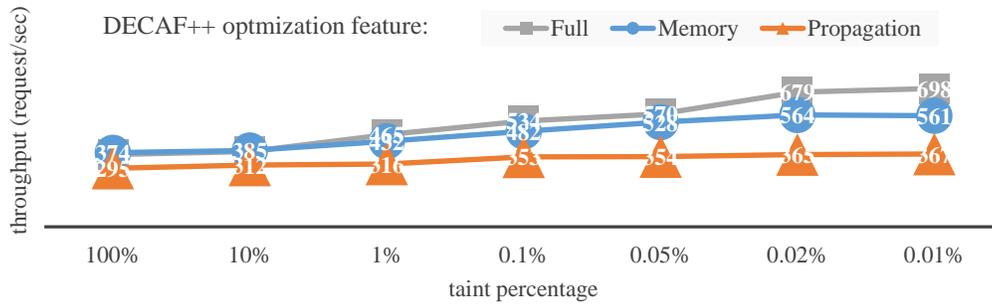


Figure 8: Throughput of solitary optimization features (and together) of DECAF++ in. The reported numbers are the mean of 5 measurements and the relative standard deviation are in range of [2%,18%] for Full, [1%,15%] for Mem and [1%,14%] for the Propagation. DECAF and QEMU 1.0 throughput are 320 and 815 request/sec.

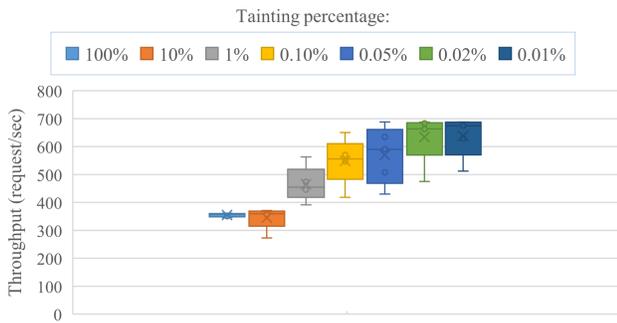


Figure 9: Evaluation of the DECAF++ on Apache bench. Each candlestick shows 5 measurements of throughput (request/sec) for a percentage of tainted packets.

tainted bytes do not linearly decrease with taint_perc. Second, these even seemingly small taint_perc values represent the real world scenarios. For instance for security applications, the attacks are anomaly cases and the percentage of suspicious packets are well below 1%. Overall, DECAF++ achieves an average (geometric) 60% throughput improvement in comparison to DECAF. When there are no tainted bytes, our system is still around 18% slower than QEMU because of the network callbacks.

6.4 Elastic Instrumentation Overhead

In this section, we evaluate DECAF++ to answer (3) and understand whether we could address the shortcomings of the previous works that are high overhead in the check mode of LIFT [25] or high transition overhead of [13].

Check mode overhead Elastic taint analysis imposes an overhead even when there are no tainted bytes. In case of LIFT [25], it's the registers taint tag check at the beginning of every basic block and further memory tag checks before memory instructions. For DECAF++, our evaluation using SPEC CPU2006 and nbench illustrated in Figure 10 and Fig-

Table 3: The nbench evaluation of DECAF++ by removing the potential overheads

Procedure	Memory index	Relative STD	Integer Index	Relative STD
Baseline	4.04	1%	4.36	1%
Full	3.86	2%	4.17	2%
Mode checking	4.04	2%	4.34	3%
Load operations patching	3.87	1%	4.23	1%
Page check in TLB fill	3.74	1%	4.14	1%

ure 11 shows that DECAF++ imposes around 4% overhead even when there is no taint analysis task, that is, running only in the check mode. This overhead is in comparison to when tainting functionality is completely disabled. DECAF++ introduces a few overheads in comparison to this case. These overheads are:

- Checking the mode before the code translation and in the memory operations
- Patching the memory load operations in the track mode
- Checking the status of the page throughout the TLB filling process to register the taint status handler

We measured the effect of each of these overheads on indexes reported by nbench by removing the code snippets attributed to these functionalities. The results of these measurements are listed in Table 3. Removing none of the overheads except the mode checking has a substantial effect on the performance. This is because mode checking is frequently done along with every memory load and store operation. It goes unsaid that this overhead is inevitable.

Transition Overhead The transition from the check mode to the track mode imposes an overhead as discussed in section 4.3. This overhead is the major issue with [13]. However, our measurement shows that this overhead is negligible for DECAF++. We measured the transition overhead by recording

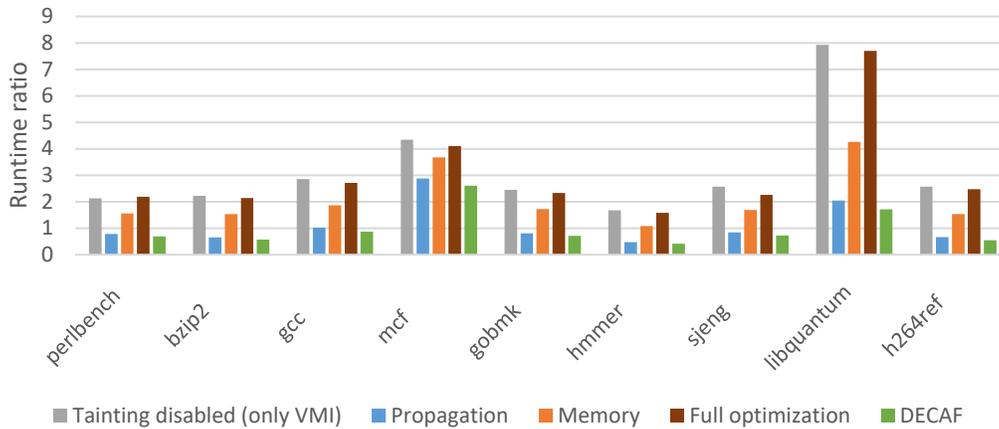


Figure 10: SPEC CPU2006 for different implementations

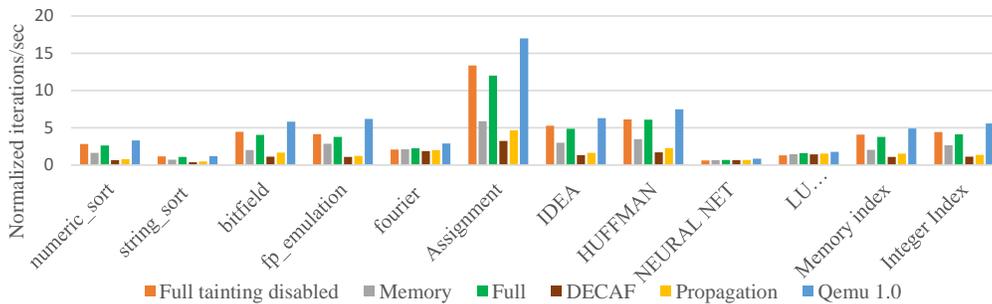


Figure 11: DECAF++ check mode overhead on nbench

the time it takes to change the mode and execute the same instruction that was executing before the transition occurred. Our measurement was performed during nbench execution, and every input byte was tainted. We repeated the measurement 10 times. The average transition time is 0.031% of the overall benchmark execution time with 0.007% relative standard deviation.

7 Conclusion

In this work, we introduced elastic tainting for whole-system dynamic taint analysis. Elastic tainting is based on elastic taint propagation and elastic taint status checking that accordingly address DECAF taint propagation and taint status checking overhead by removing unnecessary taint analysis computations when the system is in a safe state. We successfully designed and implemented this idea on top of DECAF in a prototype dubbed DECAF++ via pure software optimization. We showed that elastic tainting helps DECAF++ achieve a substantial better performance even when all inputs are tainted. Further, we showed how elastic taint propagation and elastic taint checking optimization each and together contribute to the performance improvement for different applications.

Our elastic tainting addresses the shortcomings of the previous works that are either high overhead when there's no tainted bytes or high transition cost when there is some. As a

result, DECAF++ has an elastic property for both information flow within a process and information flow of a network input throughout the system. We believe whole-system dynamic taint analysis applications like intrusion detection systems and honeypots can greatly benefit from the elastic property. On one hand, this is because these systems are constantly online and taint analysis affects the entire system constantly. On the other hand, these systems can filter benign traffic and focus on the taint analysis of a small portion of the traffic that are likely to be malicious.

Acknowledgement

We thank the anonymous reviewers for their insightful comments on our work. This work is partly supported by Office of Naval Research under Award No. N00014-17-1-2893. 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] apache HTTP server benchmarking tool. <https://httpd.apache.org/docs/2.4/programs/ab.html>.

- [2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATC '05)*, pages 41–41, 2005.
- [3] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *Proceedings of the International Symposium on Code generation and Optimization (CGO '06)*, pages 333–345. IEEE Computer Society, March 2006.
- [4] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world’s fastest taint tracker. In *Proceedings of the 14th International Symposium On Recent Advances in Intrusion Detection (RAID'11)*, pages 1–20, Berlin, Heidelberg, September 2011. Springer.
- [5] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.
- [6] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium (Security '04)*, pages 321–336, August 2004.
- [7] James Clause, Wanchun Li, and Alessandro Orso. Dy-tan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 196–206, New York, NY, USA, July 2007. ACM.
- [8] Ali Davanian. Effective granularity in internet badhood detection: Detection rate, precision and implementation performance. Master’s thesis, University of Twente, August 2017.
- [9] Peter J Denning. The locality principle. In *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*, pages 43–67. World Scientific, 2006.
- [10] 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 (PPREW-5)*, page 4. ACM, December 2015.
- [11] Cornelia Cecilia Eglantine. Nbench. 2012.
- [12] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*, pages 248–258. ACM, July 2014.
- [13] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 29–41, New York, NY, USA, 2006. ACM.
- [14] Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. Shadowreplica: efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pages 235–246. ACM, 2013.
- [15] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 377–390, New York, NY, USA, October 2017. ACM.
- [16] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12)*, volume 47, pages 121–132. ACM, 2012.
- [17] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, pages 1691–1708, New York, NY, USA, October 2017. ACM.
- [18] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pages 503–515. ACM, March 2016.
- [19] 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 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200. ACM, June 2005.

- [20] Darek Mihocka and Stanislav Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *1st Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA'08, Beijing*, page 32, 2008.
- [21] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, pages 308–319. IEEE, August 2016.
- [22] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium (Security '15)*, pages 65–80, August 2015.
- [23] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, volume 42, pages 89–100. ACM, June 2007.
- [24] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, volume 5, pages 3–4, 2005.
- [25] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 135–148. IEEE, 2006.
- [26] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium (NDSS'17)*, February 2017.
- [27] LK Yan, A Henderson, X Hu, H Yin, and S McCamant. On soundness and precision of dynamic taint analysis. *Dep. Elect. Eng. Comput. Sci., Syracuse Univ., Tech. Rep. SYR-EECS-2014-04*, 2014.
- [28] Heng Yin, Zhenkai Liang, and Dawn Song. Hook-Finder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.
- [29] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pages 116–127. ACM, 2007.

A QEMU

DECAF is built on top of QEMU [2]. Therefore, to understand DECAF and DECAF++, Qemu knowledge is required. QEMU provides binary instrumentation functionality in an architecture agnostic way via emulation. Qemu emulates the execution of a target binary e.g. a virtual machine image. This means CPU, memory and other hardware are emulated for the target binary. CPU is represented using a *vcpu* data structure that contains all the CPU registers. Providing Memory Management Unit (MMU) is more complicated but the idea is to emulate memory for the target through a software approach known as software MMU (softMMU). We elaborate on softMMU at the end of this section.

Binary translation Qemu first needs to translate the target binary to understand how to emulate it. QEMU loads a guest executable, and translates the binary one block at a time. QEMU translates the binary into an Intermediate Representation (IR) named Tiny Code Generator (*tcg*). The result of this translation is stored in a data structure called *Translation Block* (TB).

Code generation After the translation, Qemu generates an executable code from the translation block. This generated code is written to a data structure called *code cache*. In essence, code cache is an executable page that allows dynamic execution of code. After code generation, Qemu executes the code and updates emulated CPU and memory.

Cache table Qemu stores the result of code translation and generation in a cache to speedup the emulation. Then, before future execution of the same program counter, the cache table would be consulted and the request would be resolved using cache.

Block chaining In addition to the above, Qemu employs another optimization to speed up the emulation. We mentioned that the unit of translation and execution for Qemu is basic blocks. However, after translation of consecutive code blocks, Qemu chains them together and forms a trace. This process is known as block chaining and is implemented by placing a direct jump from the current block to the next one. A trace can be executed without interruption for translation.

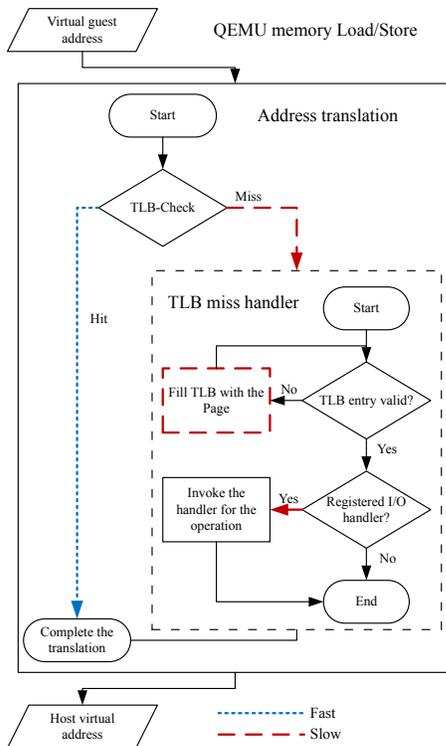


Figure 12: QEMU software memory management through `op_qemu_ldlst*` operations.

Cache invalidation The generated code blocks may be invalidated to stop further fast execution using cache table. There are different reasons for this. Two main reasons that a code cache would be invalidated are (a) the code cache is full, and (b) the code has been modified and the previous generated code is not valid for re-execution.

Software MMU QEMU software memory management unit (softMMU) translates Guest Virtual Addresses (GVA) to the Host Virtual Address (HVA). This process happens at runtime and within guest memory operations. QEMU generates load and store IR operations for machine level memory operations. QEMU tcg IR store operation, i.e., `op_qemu_st`, stores the content of a register to a given virtual guest address. Qemu tcg load operation, i.e., `op_qemu_ld`, loads the content of a memory address to a given register.

QEMU implements a software Translation Look-aside Buffer (TLB). Hardware TLB maps virtual pages within a process to their corresponding physical pages in the memory. In the Qemu software TLB, the mapping is indeed between the GVA to the HVA. Through address translation and software TLB, QEMU ensures that every guest virtual address (regardless of the process) is addressable in the host QEMU process space.

Figure 12 depicts what happens at runtime in the QEMU memory load and store operations. QEMU needs to make sure that the software TLB contains a valid entry for the following page that will be accessed. To this end, it checks whether the page index portion of the address is valid in the software TLB. If yes, and the page is not registered as a memory mapped I/O, it can be safely (without page fault) accessed. This is the fastest case, and it is expected that based on the *locality principle* [9], a majority of memory operations go through this path. If the page is not present, or the page is registered as memory mapped I/O then the memory operation is much slower.

B DECAF Garbage Collection

A memory address may become tainted, and then may be overwritten through a non-tainted data propagation. In such a case, the shadow memory can be de-allocated. DECAF does not reclaim it immediately for performance reasons. Immediate reclaiming requires DECAF to explore all the leaves (memory addresses) of a parent node (a physical page) for every memory operation that is very costly.

DECAF relies on a garbage collector to reclaim the unused shadow memory. DECAF will handle the unused memory by calling the garbage collector based on an interval. The garbage collector is called in the QEMU `main_loop` that runs in a separate thread. Experimentally, garbage collector is set to walk the shadow memory pages every 4096 times the main loop runs.

The garbage collector walks through the shadow memory and checks every parent and all its leaves. It will return an unused leaf to a memory pool. This pool will use the leaf for another taint status storage. Finally, the garbage collector will return a parent to the memory pool if none of its children is in use.

Towards a First Step to Understand the Cryptocurrency Stealing Attack on Ethereum

Zhen Cheng^{1,2*} Xinrui Hou^{2,4*} Runhuai Li^{1,2} Yajin Zhou^{1,2†} Xiapu Luo³ Jinku Li⁴ and Kui Ren^{1,2}

¹Zhejiang University

²Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies

³The Hong Kong Polytechnic University

⁴Xidian University

Abstract

We performed the first systematic study of a new attack on Ethereum that steals cryptocurrencies. The attack is due to the unprotected JSON-RPC endpoints existed in Ethereum nodes that could be exploited by attackers to transfer the Ether and ERC20 tokens to attackers-controlled accounts. This study aims to shed light on the attack, including malicious behaviors and profits of attackers. Specifically, we first designed and implemented a honeypot that could capture *real* attacks in the wild. We then deployed the honeypot and reported results of the collected data in a period of six months. In total, our system captured more than 308 million requests from 1,072 distinct IP addresses. We further grouped attackers into 36 groups with 59 distinct Ethereum accounts. Among them, attackers of 34 groups were stealing the Ether, while other 2 groups were targeting ERC20 tokens. The further behavior analysis showed that attackers were following a three-steps pattern to steal the Ether. Moreover, we observed an interesting type of transaction called zero gas transaction, which has been leveraged by attackers to steal ERC20 tokens. At last, we estimated the overall profits of attackers. To engage the whole community, the dataset of captured attacks is released on <https://github.com/zjuicrs/eth-honey>.

1 Introduction

The Ethereum network [38] has attracted lots of attentions. Users leverage this platform to transfer the Ether, the official cryptocurrency of the network, or build DApps (decentralized applications) using smart contracts. This, in turn, stimulates the popularity of the Ethereum network.

However, this popularity also attracts another type of users, i.e., attackers. They exploit the insecure setting of Ethereum clients, e.g., Go-Ethereum [7] and Parity [12], to steal cryptocurrencies. These clients, if *not properly config-*

ured, will expose a JSON-RPC endpoint *without any authentication mechanism enforced*. As a result, they could be remotely reached by attackers to invoke many privileged methods to manipulate the Ethereum account, *on behalf of the account holder who is using the client*¹. Though we have seen spot reports of stealing the Ether by hackers [11, 15], there is no systematic study of such an attack. In order words, not enough insights were provided on the attack.

To this end, we performed a systematic study to understand the cryptocurrency stealing attack on Ethereum. The purpose of our study is to shed lights on this attack, including detailed malicious behaviors, attacking strategies, and attackers' profits. Our study is based on *real* attacks in the wild captured by our system.

Specifically, we designed and implemented a system called *Siren*. It consists of a honeypot that listens the default JSON-RPC port, i.e., 8545, and accepts any incoming requests. To make our honeypot a reachable and valuable target, we register it as an Ethereum full node on the Internet and prepare a real Ethereum account that has Ethers inside. In order to implement an interactive honeypot, we use a real Ethereum client (Go-Ethereum in our work) as the back-end. We then redirect all the incoming RPC requests to the back-end, except for those that may cause damages to our honeypot. Our honeypot logs the information of the request, including the method that the attacker intends to invoke, and its parameters. For instance, our honeypot logs account addresses that attackers intend to transfer the stolen Ether into. We call these accounts as **malicious accounts**. We further crawl transactions from the Ethereum network and analyze them to identify **suspicious accounts**, which accept Ethers from malicious accounts. Though we are unaware of real owners of these accounts, they are most likely to be related to attackers since transactions exist from malicious accounts to them. We then estimate attackers' profits by calculating the income of malicious and suspicious accounts.

*Co-first authors with equal contribution.

†Corresponding author.

¹Note that, the RPC interface is intended to be used with proper authentication.

Findings We performed a detailed analysis on the data collected in a six-month period². Some findings are in the following.

- **Attacks captured** During a six-month period, our system captured 308.66 million RPC requests from 1,072 distinct IP addresses. Among these IP addresses, 9 of them are considered as the main source of attacks, since they count around 83.8% of all requests. One particular IP address 89.144.25.28 sent the most RPC requests, with a record of 101.73 million requests in total. In order to hide their real IP addresses, attackers were leveraging the Tor network [20] to launch attacks. We also observed that some IP addresses of worldwide universities were probing our honeypot, though none of them were invoking methods to steal cryptocurrencies. Most of these IP addresses are from the PlanetLab nodes [13].
- **Cryptocurrencies targeted** We grouped attackers based on IP addresses and target Ethereum accounts. These accounts are the ones that attackers transferred the stolen cryptocurrencies into. In total, attackers are grouped into 36 clusters with 59 distinct malicious accounts. Among them, attackers of 34 groups were stealing the Ether, while other 2 groups were targeting ERC20 tokens.
- **Steal the Ether** We observed that attackers were following a three-steps pattern to steal the Ether. They first probed potential victims, and then collected necessary information to construct parameters. After that, they launch the attack, either passively waiting for the account being unlocked by continuously polling the account state, or actively launching a brute-force attack to crack the user's password to unlock the account.
- **Steal ERC20 tokens** Besides the Ether, attackers were also targeting ERC20 tokens. We observed a type of transaction called zero gas transaction, in which the sender of a transaction does not need to pay any transaction fee. We find that attackers were leveraging this type of transactions to steal tokens from *fisher* accounts that intended to scam other users' Ethers, and exploiting the AirDrop mechanism to gain numerous bonus tokens.

Contributions In summary, this paper makes the following main contributions:

- We designed and implemented a system that can capture *real* attacks to steal cryptocurrencies through unprotected JSON-RPC ports of vulnerable Ethereum nodes.

- We deployed our system and reported attacks observed in a period of six months.
- We reported various findings based on the analysis of collected data. The dataset is released to the community for further study.

The rest of the paper is structured as the follows: we introduce the background information in Section 2 and present the methodology of our system to capture attacks in Section 3. We then analyze the attack in Section 4 and estimate profits in Section 5, respectively. We discuss the limitation of our work in Section 6 and related work in Section 7. We conclude our work in Section 8.

2 Background

In this section, we will briefly introduce the necessary background about the Ethereum network [38] to facilitate the understanding of this work.

2.1 Ethereum Clients and the JSON-RPC

An Ethereum node usually runs a client software. There exist several clients, e.g., Go-Ethereum and Parity. Both clients support remote procedure call (RPC) through the standard JSON-RPC API [4]. When these clients are being started with a special flag, they will listen a specific port (e.g., 8545), and accept RPC requests *from any host without any authentication*. After that, functions could be remotely invoked on behalf of the account holder of the client, including privileged ones to send transactions (or transfer cryptocurrencies). Note that, though the Ethereum network is a P2P network, attackers can discover and reach vulnerable Ethereum nodes directly through the HTTP protocol.

2.2 Ethereum Accounts

On the Ethereum platform, there exist two different types of accounts. One is externally owned account (EOA), and another one is smart contract account. An EOA account can transfer the Ether, the official currency in Ethereum, to another account. An EOA account can deploy a smart contract, which in turn creates another type of account, i.e., the smart contract account. A smart contract is a program that executes exactly as it is set up to by its creator (the smart contract developer). In Ethereum, the smart contract is usually programmed using the Solidity language [14], and executes on a virtual machine called Ethereum Virtual Machine (EVM) [38]. Both types of accounts are denoted in a hexadecimal format. For instance, the account address 0x6ef57be1168628a2bd6c5788322a41265084408a denotes an (attacker's) EOA account, while the address 0x87c9ea70f72ad55a12bc6155a30e047cf2acd798 denotes a smart contract.

²The dataset is released on <https://github.com/zjuicrsr/eth-honey>.

ERC20 tokens [1] are digital tokens designed and used on the Ethereum platform, which could be shared, exchanged for other tokens or real currencies, e.g., US dollars. The community has created standards for issuing a new ERC20 token using the smart contract. For instance, the smart contract should implement a function called `transfer()` to transfer the token from one account to another, and a `balanceOf()` function to query the balance of the token. The values of ERC20 tokens vary for different tokens at different times. For instance, the market capitalization of the Minereum token [8] was more than 7 million US dollars [9] in August, 2017, and is around 40,000 US dollars in March, 2019.

2.3 Transactions

Transactions can be used to transfer the Ether, or invoke functions of a smart contract. Specifically, the *to* field of a transaction denotes the destination, i.e., an EOA account or a smart contract. For a transaction to send the Ether, fields including *gas* and *gasPrice* specify the gas limit and the gas price of the transaction. Listing 1 (Section 4 on page 5) shows a real transaction to send the Ether to `0x63710c26a9be484581dcac1aacdd95ef628923ab`, a malicious EOA account captured by our system. If the transaction is used to invoke a function of a smart contract, then the *data* field specifies the name and parameters of the function to be invoked. Note that, a function is identified by a function signature, i.e., the four bytes of the Keccak hash of the canonical expression of the function prototype, including the function name, the parameter types. Listing 2 and 3 (Section 4 on page 6) shows the *data* field and the signature of the invoked function and its prototype.

Sending a transaction consumes *gas*, which is the name of the unit that measures the work needs to be done. It is similar to the use of a liter of fuel consumed when driving a car. The actual cost of sending a transaction (transaction fee) is calculated as the product of the consumed gas and the current gas price. The gas price is similar to the cost of each liter of fuel that is paid for filling up a car. The smallest unit of the Ether is Wei. A Gwei consists of a billion Wei, while an Ether consists of a billion Gwei. The amount of gas consumed in a transaction is accumulated during instruction execution. Since the operation of transferring the Ether is a sequence of fixed instructions, thus the consumed gas is always 21,000.

The transaction fee is paid by the sender to the miner, who is responsible for packing transactions into blocks and executing smart contract instructions. To earn a higher transaction fee, miners tend to pack the transaction with a higher gas price. Specifically, the sender of a transaction can specify the gas price in the field *gasPrice* (Section 4 on page 5) to boost the chance of the transaction being packed. We have observed a trend of higher gas price in the transactions to steal Ether (Figure 3 on page 8).

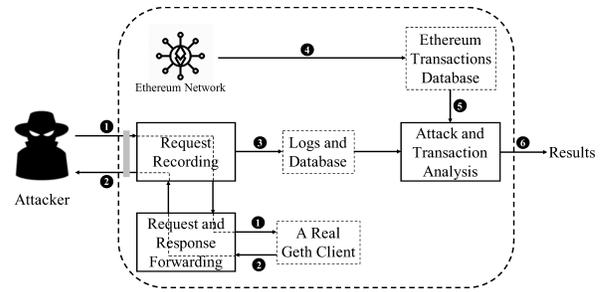


Figure 1: The overview of our system.

There are multiple RPC methods that could be remotely invoked to send (or sign) a transaction on behalf of the account holder, including `eth_sendTransaction` and `eth_signTransaction`. Note that the account needs to be unlocked before sending a transaction. This involves the process to enter the user’s password. Otherwise, the invocation of these methods will fail. In other words, in order to successfully steal the Ether from the victim’s account, his or her account should be in the state of being unlocked. In Section 4.3, we will show that attackers continuously monitor the victim’s account until it is unlocked by the user, or launch a brute-force attack using a predefined dictionary of popular passwords.

3 Methodology

Figure 1 shows the architecture of our system. In order to capture real attacks towards Ethereum nodes with unprotected HTTP JSON-RPC endpoints, we design and implement a system called Siren. Our system consists of a honeypot that listens the default JSON-RPC port, i.e., 8545. Any attempt to connect to this port of our honeypot will be recorded. Our system forwards the request to the back-end Ethereum client (1) and returns the results (2). The Ethereum client used in our system is the Go-Ethereum³. Our honeypot logs the RPC requests with parameters from the attacker, and then imports them into a database (3). To further estimate profits of attackers, we crawl transaction records from the Ethereum network (4), and identify suspicious accounts that are connected with attacker’s accounts. Our system combines the transaction records (5) of both malicious and suspicious accounts to generate the final result (6).

3.1 Ethereum Honeypot

In order to capture real attacks and understand attacking behaviors, we build a honeypot. It can interact with JSON-RPC requests that invoke APIs for malicious intents, e.g., transferring the Ether to attacker controlled malicious accounts. Our

³In this paper, if not otherwise specified, the Ethereum client we discussed is the Go-Ethereum.

honeypot logs the information of the API invocation, including the method name, parameters and etc., for later analysis. Moreover, our system confines the attacker's behaviors that APIs invoked cannot cause real damages to our honeypot.

To this end, we design a front-end/back-end architecture for our honeypot. Specifically, we implement a front-end that listens to the 8545 port and accepts any incoming HTTP JSON-RPC request from this port. We also have a real Ethereum client in the back-end, which runs as a full Ethereum node. This client accepts any *local* JSON-RPC request from the front-end. That means our real Ethereum client is not publicly available to the attackers. If the invoked API is inside a predefined whitelist, our front-end will forward the request to the back-end client, and then forward the response to the attacker. The APIs might bring a financial loss to our account are strictly forbidden. By doing so, our system protects the Ethereum node from being actually exploited, while at the same time, facilitates the information logging of the requests since all the requests need to go through the front-end.

However, there are several challenges that need to be addressed to make the system effective. For instance, the honeypot should behave like a real Ethereum node. Otherwise, attackers could be aware the existence of our honeypot and do not perform malicious activities. In the following, we will illustrate ways that our system leverages to attract attackers, and further describe how our honeypot works.

Respond the probe requests The default port number of the HTTP JSON-RPC service of an Ethereum node is 8545. Before launching a real attack, attackers usually send probe requests to check whether this port is actually open. For instance, the attacker invokes the `web3_clientVersion` method to check whether it is a valid Ethereum node. The front-end of our honeypot accepts any incoming JSON-RPC request, and responds with valid results, by relaying responses from the back-end Ethereum client.

Advise the existence of our Ethereum node In order to capture attacks, our system needs to attract attackers. There are two options for this purpose. One option is that we passively wait for attackers by responding to the probing request. However, this strategy is not efficient since the chance that our honeypot is happened to be scanned is low, given the large space of valid IP addresses. The second option is to actively attract attackers. Specifically, to make our Ethereum node (or our honeypot) visible to attackers, we register it on public websites that provide the list of full Ethereum nodes. The original purpose of this list is to speed up the discovery process of Ethereum nodes in the P2P network. However, this list provides valuable information to attackers, since they can find potential targets without performing time- and resource-consuming port scanning process. It turns out this strategy is really effective. Our honeypot receives incoming probe requests shortly after being registered on the list.

Pretend as a valuable target The main purpose of the attack is to steal cryptocurrencies. In order to make the attacker believe our honeypot is a valuable target, we create a real Ethereum account with the address `0xa33023b7c14638f3391d705c938ac506544b25c3` and transfer some amounts of Ether into this account. Since the Ethereum network is a public ledger, the amount of the Ether inside the account could be obtained by querying on the network. Our honeypot returns this account address to attackers if they invoke the `eth_accounts` method to get a list of accounts owned by our Ethereum node. We also return the real amount of Ether inside this account to attackers if the `eth_getBalance` method is invoked.

Emulate a real transaction After obtaining the information of the account owned by our Ethereum node and the balance of the account, attackers tend to steal the Ether by transferring it to accounts they controlled (malicious accounts). For instance, they could invoke the `eth_sendTransaction` method, which returns the hash value of a newly-created transaction. Attackers could check the return value of the method invocation to get the status of Ether transfer. To make the attacker believe that the transaction is being processing, while not actually transferring any Ether from our account, we do not actually execute the `eth_sendTransaction` method. Instead, we log the parameters of this method invocation, and return a randomly generated hash value to the attacker.

Log RPC requests Our honeypot logs the attacker's invoked methods, including the method name, parameters, along with the metadata of the attack, such as the IP address and the time. All the data will be saved into a log file, which will be imported into a database.

3.2 Data Collection and Analysis

After capturing attacks and malicious account addresses, we will estimate profits gained by attackers. Our system leverages transactions launched from these accounts to find more attacker-controlled accounts. For this purpose, we crawl the whole transactions from the Ethereum network.

Our system first downloads Ethereum transactions, then imports them into a database. After that, we can conveniently combine data captured by our honeypot and transactions from the Ethereum network to generate final analysis results.

4 Attack Analysis

In this section, we will illustrate the data we collected, the grouping process of attackers, and detailed information about attacks to steal the Ether and ERC20 tokens.

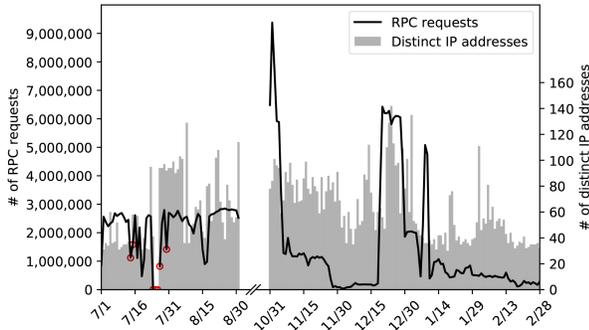


Figure 2: The number of daily RPC requests and distinct IP addresses captured by our honeypot. The seven red circles mean the data in these days is incomplete, either because our system was accidentally shut down, or the network was not stable.

4.1 Data Overview

We deployed our system on a virtual machine in Alibaba cloud, and collected the data in a period of six months, i.e., from July 1 to August 31 in the year 2018, and November 1, 2018 to February 28, 2019. Unfortunately, there are three days (from July 24 to July 26) when the virtual machine was accidentally shut down, and four days (July 14, 15, 27 and 30) when the network was not stable. During these days, the data is either missing or incomplete. Figure 2 shows the number of daily RPC requests and distinct IP addresses of these requests.

In total, our system observed 308.66 million RPC requests from 1,072 distinct IP addresses. In average, we received 1.72 million RPC requests each day (excluding the incomplete data.) In terms of IP addresses, the average daily number is 62. This number reached its peak value (142) on December 24, 2018.

Among these RPC requests, 9 different IP addresses are main sources of attacks, given that they contribute most of RPC requests in our dataset. These 9 IP addresses sent around 258.70 million requests in total, which counts around 83.8% of all requests. It's worth noting that, the IP address 89.144.25.28 sent the most RPC requests. It sent 101.73 million requests in total, accounting for 33.0% of the requests we received. We believe such an aggressive behavior is to increase the possibility of stealing the Ether, since the time window to transfer the Ether only exists when the user account is unlocked. We also observed that attacks from this IP address ceased in several days, e.g., from August 15 to 17.

RPC requests from universities worldwide Interestingly, some RPC requests are from IP addresses that belong to universities. Specifically, our honeypot received requests from 66 IP addresses of 39 universities in 13 countries or regions. Among them, 37 IP addresses are from universities in the USA. For instance, two IP addresses (146.57.249.98

```
// Date: Jul 1 20:44:09 GMT+08:00 2018
// Source IP: 89.144.25.28
{
  "jsonrpc": "2.0",
  "method": "eth_sendTransaction",
  "params": [
    //The account address of our honeypot.
    "from": "0xa33023b7c14638f3391d705c938ac506544b25c3",
    //Attacker's account address.
    "to": "0x63710c26a9be484581dcac1aacdd95ef628923ab",
    "gas": "0x5208",
    "gasPrice": "0x199c82cc00",
    "value": "0x2425f024b7fd000",
  ],
  "id": 739296
}
```

Listing 1: The captured attack and the associated account address in the *to* field.

and 146.57.249.99) belong to the University of Minnesota. We further used the reverse DNS lookup command to obtain the domain name associated with these IP addresses. It turns out that all of them are associated with the PlanetLab [13]. For instance, the domain names of the previous two IP addresses are planetlab1.dtc.umn.edu and planetlab2.dtc.umn.edu, respectively. Requests from these IP addresses are not performing malicious activities, e.g., transferring the Ether to other accounts. Most of them are merely probing our honeypot for information collection, e.g., invoking `eth_getBlockByNumber`. Though the exact intention of collecting such information is unknown, we believe these requests are mainly for a research purpose.

Abuse of the Tor network and cloud services Attackers are leveraging the Tor network and cloud services to hide their identities. For instance, some IP addresses belong to popular cloud services, e.g., Amazon, DigitalOcean and etc. Among the 1,072 distinct IP addresses, 370 of them are identified as Tor gateways [20] (299 of them performed malicious behaviors, e.g., trying to steal the Ether.) All these IP addresses belong to the second group (Section 4.2). They are from 104 different ISPs in 39 countries. Using the Tor network to hide the real IP addresses make the tracing of attackers more difficult.

4.2 Grouping Attackers Accounts

After collecting the data, our next step is to group attackers. However, due to the anonymity property of the Ethereum network, it is hard to group them based on their identities. In this paper, we take the following ways to group attackers.

First, we *directly* retrieve attackers' Ethereum accounts through the parameters of RPC requests, and group them based on these accounts. For instance, the parameter *to* of the method `eth_sendTransaction` denotes the destination account of a transaction that the Ether will be transferred into. Attackers use this method to transfer (steal) the Ether to their controlled accounts. Listing 1 shows the parameters (in the JSON format) of a captured malicious transaction launched by an attacker. The value of the *to*

Table 1: The result of grouping attackers.

#	Addresses	# of IP	# of RPC calls	Max calls per day	First capture date	Last capture date	Days of activity
1	Ox6a141e661e24c5e13fe651da8fe9b269fec43df0 Ox6e4cc3e76765bdc711cc7b5cbfc5bbfe473b192e Ox6ef57be1168628a2bd6c5788322a41265084408a Ox7097f41f1c1847d52407c629d0e0ae0fdd24fd58 Oxe511268ccf5c8104ac8f7d01a6e6eaaa88d84ebb	57	72,915,681	1,207,185	Jul. 1, 2018	Feb. 28, 2019	178
2	Ox581061c855c24ca63c9296791de0c9a1a5a44fcf Ox5fa38ab891956dd35076e9ad5f9858b2e53b3eb5 Ox8cacaf0602b707bd9bb00ceeda0fb34b32f39031 Oxab259c71e4f70422516a8f9953aaba2ca5a585ae Oxd9ee4d08a86b430544254ff95e32aa6fcc1d3163 Ox88b7d5887b5737eb4d9f15fcd03a2d62335c0670 Oxe412f7324492ead5eac30dcec2240553bf1326a	309	363,860	167,183	Jul. 2, 2018	Feb. 28, 2019	166
3	Oxd6cf5a17625f92cee9c6caa6117e54cbfbcceadf Ox21bdc4c2f03e239a59aad7326738d9628378f6af Ox72b90a784e0a13ba12a9870ff67b68673d73e367	14	13,315,318	365,878	Jul. 16, 2018	Feb. 25, 2019	78
4	Ox04d6cb3ed03f82c68c5b2bc5b40c3f766a4d1241 Ox63710c26a9be484581dcac1aacdd95ef628923ab	1	101,731,595	4,802,304	Jul. 1, 2018	Nov. 5, 2018	63
5	Oxb0ec5c6f46124703b92e89b37d650fb9f43b28c2	6	326,154	9,711	Jul. 2, 2018	Dec. 3, 2018	64
6	Ox1a086b35a5961a28bead158792a3ed4b072f00fe Ox73b4c0725c900f0208bf5f5eb36856abc520de26 Oxaf4778d8d05e9595d540d40607c16f677c73cca Oxec13837d5e4df793e3e33b296bad8c4653a256cb	3	6,791,438	3,346,904	Nov. 1, 2018	Feb. 28, 2019	21
7	Ox241946e18b9768cf9c1296119e55461f22b26ada	1	7,750,800	118,127	Jul. 2, 2018	Feb. 28, 2019	151
8	Ox8652328b96ff12b20de5fdc67b67812e2b64e2a6	2	3,569,924	281,975	Jul. 1, 2018	Jul. 30, 2018	28
9	Oxff871093e4f1582fb40d7903c722ee422e9026ee	1	3,522	1,128	Jul. 2, 2018	Aug. 31, 2018	27
10	Ox6230599f54454c695b5cd882064071fc39e6e562	1	13	13	Jul. 5, 2018	Jul. 5, 2018	1
11	Ox2c5129bdfc6f865e17360c551e1c46815fe21ec8	1	618	618	Jul. 5, 2018	Jul. 5, 2018	1
12	Oxeb29921d8eb0e32b2e7106afca7f53670e4107e5	1	5	5	Jul. 29, 2018	Jul. 29, 2018	1
13	Oxe231c73ab919ec2b9aaeb87bb9f0546aa47581b1	1	10	5	Jul. 4, 2018	Jul. 17, 2018	5
14	Ox5c8404b541881b9999ce89c00970e5e8862f8e88	3	80	46	Jul. 10, 2018	Jul. 15, 2018	5
15	Ox5e87bab71bbea5f068df9bf531065ce40a86be4	1	274	274	Jul. 11, 2018	Jul. 11, 2018	1
16	Ox97743cc5a168a59a86cf854cf04259abe736006a Ox9d6d759856bfcabf6f405f308d450b79e16dd4e2	3	235,213	71,521	Jul. 10, 2018	Jul. 17, 2018	8
17	Ox02a4347035b7ba02d7923885503313ecb817688 Oxcb31bea86c3becc1f62652bc8b211fe1bd7f8aed	3	11,246,017	175,417	Nov. 13, 2018	Feb. 28, 2019	97
18	Oxe128bb377f284d2719298b0d652d65455c941b5b	1	277	147	Nov. 12, 2018	Nov. 16, 2018	4
19	Oxb744d5f73d27131099efee0b70062de6f770a102	2	237,481	64,462	Dec. 18, 2018	Feb. 14, 2019	17
20	Ox0e0a930fb51c499b624d6ca56fdd9c95c5bf2e06 Ox2c022e9a0368747692b7bd532c435c7a78dc447d Ox3334f7f8bcf593794b01089b6ff4dc63fe023dfe Ox884aa595c10b3331ce551c2d9f905e52e21fa0bb Oxef462edb8880c4fd0738e4d3e9393660b9c5ac72	2	59,842	37,608	Aug. 4, 2018	Feb. 7, 2019	38
21	Ox9781d03182264968d430a4f05799725735d9844d	8	38,558	13,559	Aug. 28, 2018	Aug. 31, 2018	4
22	Ox98c6428fba6c0ff97570d822dd607f8a55080e5	6	270	140	Aug. 2, 2018	Aug. 5, 2018	2
23	Oxa0b0209a04398cb61d845148623e68b3eff8f8cb	1	135	135	Jul. 9, 2018	Jul. 9, 2018	1
24	Ox21d8976138a2b280d441fd7b12456a1193cb2baf	1	18,597	2,285	Aug. 10, 2018	Nov. 9, 2018	14
25	Oxfed69981c21b96ff37fc52f9e19849126624ddfd	5	963	825	Aug. 13, 2018	Aug. 19, 2018	3
26	Ox31c3ecd12abe4f767cb446b7326b90b1efc5bbd9	3	440,962	49,995	Nov. 1, 2018	Feb. 14, 2019	41
27	Ox5f622d88cd745ebb8ff2d4d6b707204c65243438	1	2,782	113	Nov. 1, 2018	Feb. 28, 2019	116
28	Oxf2565682d4ce75fcf3b8e28c002dfc408ab44374	1	9	3	Dec. 22, 2018	Jan. 11, 2019	5
29	Oxc97663c1156422e2ad33580adab45cad33cf7698	1	3,298	3,297	Feb. 3, 2019	Feb. 10, 2019	2
30	Oxc6c42a825555fbef74d21b3cb6bfd7074325c348	9	73,302	4,327	Nov. 4, 2018	Jan. 6, 2019	22
31	Ox454d7320d5751de29074a55ac95bbde312dd7615	1	11	11	Feb. 5, 2019	Feb. 5, 2019	1
32	Ox4e25e7e76dbd309a1ab2a663e36ac09615fc81eb	1	24	23	Jan. 15, 2019	Jan. 16, 2019	2
33	Oxa8a21375ca42dccc26237f3e861d58f88fe72eab2	1	256	256	Nov. 27, 2018	Nov. 27, 2018	1
34	Oxb703ae04fd78ab3b271177143a6db9e00bdf8d49	1	1,345	72	Dec. 30, 2018	Feb. 21, 2019	32
35	Ox0fe07dbd07ba4c1075c1db97806ba3c5b113cee0	11	536,612	27,062	Jul. 1, 2018	Feb. 28, 2019	144
36	Oxaa75fb2dcac2e3061a44c831baf0d4c2d4f92fd7 Oxffecf9e4c3e87987454f2392676ccd8b98b926f8	5	26,991	9,510	Jul. 16, 2018	Nov. 9, 2018	11

Table 2: Most used commands for probing.

Command	# of IP addresses	# of RPC requests
net_version	122	4,822,620
rpc_modules	81	3,815
web3_clientVersion	103	4,495,312
eth_getBlockByNumber	325	1,190,445
eth_blockNumber	225	27,019,686
eth_getBlockByHash	214	1,633

Table 3: Commands used to prepare attacking parameters.

Command	# of IP addresses	# of RPC requests
eth_accounts	615	27,040,164
eth_coinbase	64	87,442
personal_listAccounts	11	95
personal_listWallets	5	173,243
eth_gasPrice	21	63,133
eth_getBalance	493	93,585,372
eth_getTransactionCount	63	2,411,504

method, the attacker could find the right targets running the Ethereum mainnet. The *rpc_modules* command returns all enabled modules. By probing this information, attackers can get the information of enabled modules and then invoke the APIs inside each module accordingly. Besides the previously discussed two methods, other ones shown in Table 2 are also serving the purpose of collecting client information.

Step 2 - Preparing attacking parameters After locating potential victims, attackers need to prepare the necessary data to launch further attacks. In order to steal the Ether, the attacker needs to send an Ethereum transaction with valid parameters. Specifically, each transaction needs *from_address* and *to_address* as the source and destination of a transaction, and other optional ones including *gas*, *gasPrice*, *value* and *nonce*. In order to make the attack succeed, valid parameters should be prepared to steal the Ether.

- *from_address*: The *from_address* in the transaction is the victim's Ethereum account address. The attacker can obtain this value through invoking the following methods, including *eth_accounts*, *eth_coinbase*, *personal_listAccounts*, *personal_listWallets*.
- *to_address*: The *to_address* in the transaction specifies the destination of the transaction. Attackers will set this field to the account under their control.
- *value*: This is the value of Ether that will be transferred into the *to_address*. In order to maximize their income, the attacker tends to transfer all the Ether in the victim's account, leaving a small amount to pay the transaction

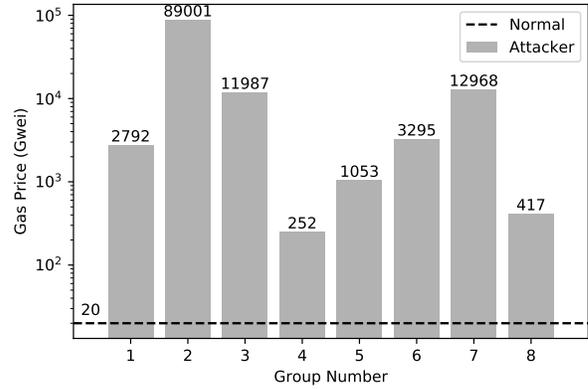


Figure 3: The comparison of the gas price in the transactions of attackers and normal users. The typical gas price is 21 Gwei, while the gas price of attackers' transactions is much higher.

fee. In order to get the balance of the victim's account, the method *eth_getBalance* is used.

- *gasPrice*: The attacker could set a high *gasPrice* to increase the chance of the transaction being executed (or packed into a block by miners.) For instance, the attacker (0x21bdc4c2f03e239a59aad7326738d9628378f6af) tends to use a much higher *gasPrice* in the transaction to steal Ether. Figure 3 shows the gas price of transactions from attackers and normal users. We will illustrate it later in this section.

Step 3 - Stealing Ether In order to successfully send a transaction, it needs to be signed using the victim's private key. However, the private key is locked *by default*, and a password is needed to unlock it. We observed two different behaviors that are leveraged by attackers to solve this problem.

- Continuously polling: Attackers continuously invoke the methods, i.e., *eth_sendTransaction* or *eth_signTransaction* in the background. If a legitimate user wants to send a transaction at the same time, then he or she will unlock the account by providing the password. This leaves a small time window that the attacker's attempt to send a transaction will succeed.

However, in order to successfully launch the attack, there are still two challenges. First, the time window is really small. Attackers should happen to invoke the method to send a transaction at the same time when the user is unlocking the account. To increase the chance of a successful attack, the operation to send the transaction should be very frequent. That's the reason of our observation that some attackers are repeatedly invoking the previously mentioned methods at a very high frequency, nearly 50

requests per-second. Second, since the attacker is sending the transaction at the same time with the user (i.e., when the user is unlocking his account), his transaction may fail if the user's transaction is accepted by the miner at first and the remaining balance of the account will not be sufficient for the attacker's transaction. In order to ensure that his transaction will be accepted by miner in a timely fashion, the attacker will use a much higher value of the *gasPrice* than normal transactions to *bribe* miners.

Figure 3 shows the gas price of transactions from attackers and normal users. Specifically, we first calculate the average gas price of captured transactions from the group 1 to 8 (the solid line in the Figure). Then we calculate the average gas price of transactions of normal users in six months (the dash line in the Figure). It turns out that the gas price from attacker's transactions is much higher (from 15 times to 4,500 times) than the value of a normal transaction. Setting a higher gas price can increase the speed that their transactions are packed into a block. This strategy is very effective, and we have observed several cases that the transaction with a higher *gasPrice* succeed, while the ones with lower *gasPrice* failed [18, 19].

- Brute force cracking: Besides the polling strategy, some attackers are leveraging the brute force attack to guess the password. Specifically, they try to unlock the account using the password in a predefined dictionary. Since the Ethereum client does not limit the number of wrong password attempts during a certain time period, this attack is effective if the victim uses a weak password. For instance, attackers from the group 11 leveraged this strategy and tried a dictionary with more than 600 weak passwords, e.g., `qwerty123456`, `margarita` and `192837465`. Another attacker from the group 1 took the same way, but only tried one password (`ppppGoog1e`). The reason why this specific password was used is unknown. However, we think it may be the default password for some customized Ethereum clients.

Interestingly, after a successful try to unlock the account, the attacker will set a relatively long timeout value by invoking the `personal_unlockAccount`. By doing so, the account will not be locked again in a long time period so that the attacker can perform further attacks much easier.

4.4 The Analysis of ERC20 Token Stealing

Attackers from two groups (group 35 and 36) are targeting ERC20 tokens. ERC20 is a technical standard used by smart contracts on the Ethereum network to implement exchangeable tokens [1]. The ERC20 token can be viewed as a kind of cryptocurrency that could be sold on some markets, thus becoming valuable targets.

Before illustrating the detailed attacking behaviors, we will first discuss an interesting type of transaction called

zero gas transaction, which we observed in our dataset. It exploits the packing strategy of some miners to send transactions without paying any transaction fee. By using this type of transaction, attackers could perform malicious activities to steal ERC20 tokens from addresses with leaked private key, or exploit the AirDrop mechanism of ERC20 smart contracts to gain extra bonus tokens with nearly zero cost.

Zero gas transaction Sending a transaction usually consumes gas (Section 2.3). The actual cost is calculated as the product of the amount of gas consumed and the current gas price. The amount of gas consumed during a transaction depends on the instructions executed in the Ethereum virtual machine, while the gas price is specified by the user who sends the transaction. If it is not specified, a default gas price will be used.

Interestingly, our honeypot captured many attempts of sending transactions with a zero value in the `gasPrice` field. This brings our attention for a further investigation. We want to understand whether such transactions could be successful, and the intentions for sending such transactions. After performing multiple experiments, transactions with a zero gas price received through the p2p network are not accepted by the miner, and will be discarded as invalid ones. However, if such a transaction is created and launched on the miner node itself (i.e., the node that successfully mines a new block is sending a zero gas transaction), then the transaction will be packed into the block by the miner and accepted by the network.

This explains the existences of such transactions captured by our honeypot. In particular, attackers were launching zero gas transactions on every vulnerable Ethereum node, in hope that the node is a miner node that is successfully mining a new block. Though the chance looks really slim, we found several successful cases in reality, e.g., the first several transactions in the block 5899499 [3]. Most of the transactions are transferring ERC20 tokens to the address `0x0fe07dbd07ba4c1075c1db97806ba3c5b113cee0`, which is a malicious account owned by the attacker in group 20.

Attack I: stealing tokens from *fisher* accounts The first type of attack is leveraging the zero gas transactions to steal tokens from *fisher* accounts. In order to understand this attack, we first explain what the *fisher* account is in the following.

The *fisher* account means that some attackers intentionally leak the private key of their Ethereum accounts on Internet. They also transfer some ERC20 tokens to the accounts as the bait. Since the private key of the account is leaked, other users could use the private key to transfer out the ERC20 tokens. However, there is one problem in this process. In order to transfer the ERC20 tokens, the account should have some Ethers to pay the transaction fee. As a result, one may transfer some Ethers into this account, in hope to get the ERC20 tokens. Unfortunately, after transferring the Ether into this

Table 4: The top ten ERC20 tokens that attackers are targeting.

ERC20 token addresses	# of RPC requests	Token name
0x1a95b271b0535d15fa49932daba31ba612b52946	11,788	MNE
0xee2131b349738090e92991d55f6d09ce17930b92	8,998	DYLC
0x0775c81a273b355e6a5b76e240bf708701f00279	8,099	BUL
0xbdeb4b83251fb146687fa19d1c660f99411eefe3	7,735	SVD
0x0675daa94725a528b05a3a88635c03ea964bfa7e	7,359	TKLN
0x87c9ea70f72ad55a12bc6155a30e047cf2acd798	7,058	LEN
0x4c9d5672ae33522240532206ab45508116daf263	5,510	VGS
0x23352036e911a22cfc692b5e2e196692658aded9	4,011	FDZ
0xc56b13ebbcffa67cfb7979b900b736b3fb480d78	2,219	SAT
0x89700d6cd7b77d1f52c29ca776a1eae313320fc5	1,708	PMD

account, the Ether will be transferred out to some accounts immediately by attackers. That’s the reason why such an account is called the *fisher* account. The main purpose of leaking the private key is to seduce others transferring Ether into the *fisher* account.

For instance, there is a *fisher* account whose address is 0xa8015df1f65e1f53d491dc1ed35013031ad25034 [2]. The attacker bought 75,000 ICX (a ERC20 token) as the fishing bait that values around 66,000 US dollars. *Occasionally*, the fisher released the private key of that account on the Internet. Anyone who transfers the Ether into this account and hopes to obtain the ICX token will be trapped to lose the transferred Ether.

Interestingly, by leveraging the zero gas transaction previously discussed, attackers could steal the ERC20 tokens in the *fisher* account. Specifically, attackers could send the transactions to transfer the ERC20 tokens in the *fisher* account with zero gas price. If the transaction is successful, then the attackers will obtain the ERC20 tokens without any cost.

In our dataset, the user in group 35 (the address is 0xf0e07dbd07ba4c1075c1db97806ba3c5b113cee0) was performing this type of attack. In total, the attacker sent 61,158 RPC requests, stealing 161 different types of ERC20 tokens. We show the detailed information of the top ten ERC20 tokens that this attacker is targeting in Table 4. We observed several different IP addresses (62.75.138.194, 77.180.167.78, 77.180.200.1, 92.231.160.88, 92.231.169.137, 95.216.158.152 and etc.) from this attacker.

Attack II: Exploiting the airdrop mechanism Airdrop is a marketing strategy that the token holders would receive bonus tokens based on some criteria, e.g., the amount of total tokens they hold. The conditions to send out bonus tokens depend on the individual token maintainer.

Some attackers are leveraging the zero gas transaction to obtain the free LEN tokens. Specifically, the LEN token has an airdrop strategy that if a new user A sends any amount of LEN token to the user B, then both A and B

will be rewarded with 18,895 LEN tokens. Hence, the attacker could create a large number of new accounts, and then transfer LEN tokens to the attacker’s address (address 0xffecffe94c3e87987454f2392676ccdb98b926f8 in group 36). By doing so, the new account will receive a bonus token, which will be transferred to the attacker’s account, while at the same time the attacker’s account will also receive the bonus. We observed many attempts of such transactions using zero gas price, with 7,058 different source account addresses and one destination address (the attacker’s address). This transaction does not consume any gas, and the attacker could be rewarded with ERC20 tokens. By using this method, the attacker even becomes a large holder of this token (2.4%) [16].

5 Transaction Analysis

After capturing malicious accounts and analyzing the detailed attackers’ behaviors, we further estimate profits of attackers. Though we can directly get the estimation by calculating the income of malicious accounts, attackers may use other account addresses that have not been captured by our honeypot. We call these addresses that are potentially controlled by attackers as suspicious accounts.

In our system, we take the following steps to detect suspicious accounts. The basic idea is if the attacker transfers the Ether from a malicious account to any other account, it is highly possible that the destination account is connected with the attacker. The attacker has no reason to transfer the Ether to an account that has no relationship with. Note that, the attacker could transfer the Ether to a cryptocurrency market, where he can exchange it with other types of cryptocurrencies or real currency. These markets should be removed from suspicious accounts in our study⁴.

To this end, we used a similar idea of the taint analysis [35] to find suspicious accounts. Specifically, we treat malicious accounts captured by our system as the taint sources, and propagate the taint tags through the transaction flows until reaching the taint sinks, i.e., the cryptocurrency markets. We also stop this process if the number of accounts traversed reaches a certain threshold. In our study, we use 3 as the threshold. All the accounts in the path from the taint source to the taint sink are considered tainted and suspicious, as long as the endpoint is a cryptocurrency market. Other nodes are marked as unknown ones, since we do not have further knowledge about whether the nodes are suspicious or not. Figure 4 shows an example of this process to detect the suspicious accounts from the malicious one 0xe511268ccf5c8104ac8f7d01a6e6eaaa88d84ebb. In this figure, the cryptocurrency market nodes are marked in the house symbol, and the original attacker we captured is

⁴We obtained the addresses of cryptocurrency markets from the Etherscan website [6].

Table 5: Our estimation of attackers' profits in Ether and US dollars. The price of one Ether is around 139 US dollars (March, 2019). We remove the addresses with zero profit from the table.

#	Addresses	Malicious		Plus Suspicious		
		Ether	USD	Ether	USD	
1	0x6a141e661e24c5e13fe651da8fe9b269fec43df0	116.91	\$16,280.23	814.45	\$113,412.00	
	0x6e4cc3e76765bdc711cc7b5cbfc5bbfe473b192e	56.16	\$7,820.34	794.67	\$110,657.59	
	0x6ef57be1168628a2bd6c5788322a41265084408a	37.79	\$5,261.74	1,420.06	\$197,743.19	
	0x7097f41f1c1847d52407c629d0e0ae0fdd24fd58	281.44	\$39,191.07	1,331.74	\$185,444.98	
	0xe511268ccf5c8104ac8f7d01a6e6eaaa88d84ebb	152.26	\$21,201.86	1,332.53	\$185,554.52	
	0x8652328b96ff12b20de5fdc67b67812e2b64e2a6	37.75	\$5,256.18	1,066.31	\$148,483.43	
	0xff871093e4f1582fb40d7903c722ee422e9026ee	0.00	\$0.69	9.34	\$1,300.01	
	2	0x5fa38ab891956dd35076e9ad5f9858b2e53b3eb5	48.24	\$6,716.88	94.28	\$13,129.14
		0x8caca0602b707bd9bb00ceeda0fb34b32f39031	0.00	\$0.14	10.66	\$1,483.80
0xab259c71e4f70422516a8f9953aaba2ca5a585ae		2.53	\$351.64	4.18	\$581.92	
0xd9ee4d08a86b430544254ff95e32aa6fcc1d3163		54.12	\$7,535.80	55.72	\$7,759.26	
0x88b7d5887b5737eb4d9f15fcd03a2d62335c0670		0.24	\$33.41	0.24	\$33.41	
0xe412f7324492ead5eacf30dcec2240553bf1326a		0.24	\$33.96	0.24	\$33.96	
0x241946e18b9768cf9c1296119e55461f22b26ada		1.53	\$213.74	1.53	\$213.74	
0x9781d03182264968d430a4f05799725735d9844d		50.32	\$7,006.89	61.47	\$8,560.18	
4		0x04d6cb3ed03f82c68c5b2bc5b40c3f766a4d1241	2.38	\$331.13	2.38	\$331.13
	0x63710c26a9be484581dcac1aacdd95ef628923ab	19.44	\$2,706.79	38.88	\$5,413.47	
	0xb0ec5c6f46124703b92e89b37d650fb9f43b28c2	0.87	\$120.84	1.64	\$227.89	
6	0x1a086b35a5961a28bead158792a3ed4b072f00fe	80.22	\$11,170.51	4,821.68	\$671,419.10	
	0x73b4c0725c900f0208bf5febb36856abc520de26	1.10	\$153.12	1.10	\$153.12	
	0xec13837d5e4df793e3e33b296bad8c4653a256cb	1.62	\$226.21	1.62	\$226.21	
11	0x2c5129bdfc6f865e17360c551e1c46815fe21ec8	113.93	\$15,864.25	506.86	\$70,580.16	
15	0x5e87bab71bbea5f068df9bf531065ce40a86ebe4	0.05	\$6.42	0.05	\$6.42	
17	0x02a4347035b7ba02d79238855503313ecb817688	4.30	\$598.46	4.30	\$598.46	
	0xcb31bea86c3becc1f62652bc8b211fe1bd7f8aed	0.21	\$29.19	0.21	\$29.19	
	0xd6cf5a17625f92cee9c6caa6117e54cbfbceaedf	2,030.19	\$282,704.44	2,030.19	\$282,704.44	
	0x21bdc4c2f03e239a59aad7326738d9628378f6af	357.78	\$49,820.26	58,692.91	\$8,172,988.20	
26	0x72b90a784e0a13ba12a9870ff67b68673d73e367	558.32	\$77,746.63	59,298.45	\$8,257,309.40	
	0x31c3ecd12abe4f767cb446b7326b90b1efc5bbd9	0.10	\$13.23	0.10	\$13.23	
28	0xf2565682d4ce75fcf3b8e28c002dfc408ab44374	173.99	\$24,228.78	866.10	\$120,604.60	
	0xb703ae04fd78ab3b271177143a6db9e00bdf8d49	8.02	\$1,116.77	8.02	\$1,116.77	
30	0xc6c42a825555fbef74d21b3cb6bfd7074325c348	1.50	\$208.36	1.50	\$208.36	
32	0x4e25e7e76dbd309a1ab2a663e36ac09615fc81eb	0.04	\$6.27	0.05	\$7.34	
Total		4,193.58	\$583,956.23	133,273.46	\$18,558,328.61	

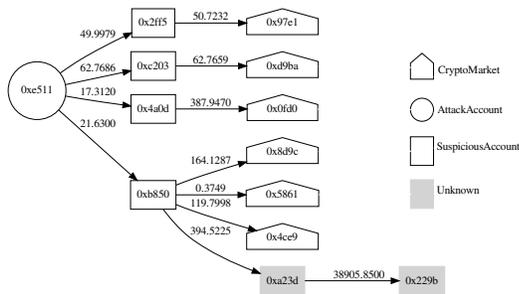


Figure 4: One example of detecting suspicious accounts through the transaction analysis. The house ones are the cryptocurrency markets, the circle one is the malicious account. The box ones without background color are suspicious accounts, while the ones with gray background are unknown accounts.

marked as a circle. Box nodes without background color are the suspicious accounts we identified, and others with gray background color are unknown addresses. The line between two nodes denotes transactions between them. We also put the number of Ether transferred above the line. In total, we identified 113 suspicious addresses, and 936 unknown ones, respectively.

After that, we estimate attackers' profits. We first calculate the lower bound of profits by only considering the income of the malicious accounts. Since our honeypot observed their behaviors of stealing the Ether, we have a high confidence that these malicious accounts belong to attackers. Then we add the income of suspicious addresses into consideration. Since these addresses are not directly captured by our honeypot, we do not have a hard evidence that they belong to attackers. However, they may be connected with or controlled by attackers. Table 5 shows the estimated profits. We remove the addresses with zero profit from the table (e.g., addresses in the group 10), and we do not count the attackers from the group 35 and 36 since they are targeting ERC20 tokens, whose value are hard to estimate due to the dramatic change of the token price. It's worth noting that, the actual income of attackers are far more than the value shown in the table since there are many attacks in the wild that were not captured by our system.

6 Discussion

Though we have adopted several ways to make our honeypot an interactive one, cautious attackers can still detect the existence of our honeypot and do not perform malicious activities thereafter. For instance, the attacker can first send a small amount of Ether to a newly generated address and then observe the return value (the transaction hash) of this

transaction. Since the transaction to send the Ether in our honeypot does not actually happen, the return value is an invalid one (a randomly generated value). The attackers can also simply send some uncommon commands and observe the return value to detect the honeypot. Nevertheless, it is an open research question to propose more effective countermeasures to improve the honeypot.

In this paper, we take a conservative way to detect suspicious accounts and estimate profits of attackers. Specifically, we leverage the knowledge of whether an address belongs to a cryptocurrency market and mark the tainted accounts whose Ether eventually flows into cryptocurrency markets as suspicious. However, the knowledge of the mapping between addresses and cryptocurrency markets is incomplete, since these addresses are manually labelled. Some suspicious accounts may be identified as unknown ones, hence introducing false negatives to our work. Moreover, our estimation is based on the attacker's addresses collected by our honeypot (in six months). There do exist attackers missed by our system, and profits of these attackers are not included in our estimation. We believe the total income of attackers in the wild is much higher than our estimation.

In this paper, attackers are exploiting the unprotected JSON-RPC interface to launch attacks. Though it is simple to fix the problem by changing the configuration of the Ethereum client, we are surprised by the fact that 7% of Ethereum nodes are still vulnerable. Specifically, we performed a port scanning to the 15,560 Ethereum public nodes [5] and found that around 1,000 of them are reachable through the RPC port without any authentication⁵. This fact demonstrated the severity of this problem, and advocates the need to have a better understanding of this issue in the community (the purpose of our work.)

7 Related work

Honeypot Honeypot systems have been widely used to capture and understand attacks by capturing malicious activities [22, 26, 33, 36]. For instance, HoneyD [33] is one of the best-known honeypot projects. It can simulate the network stack of many operating systems and arbitrary routine topologies, thus making it a highly interactive one. Collapsar can manage a large number of interactive honeypots, e.g., Honeypot farms. The concept of honeypot has been adopted to detect attacks to IoT devices [23, 25, 29, 32] and mobile devices [39]. Our system is working towards Ethereum clients, which have different targets with previous systems. The general idea of attracting attacks and logging behaviors are similar, but with different challenges.

Security issues of smart contracts One reason that Ethereum is becoming popular is its support for smart con-

⁵For ethical reasons, we did not perform any RPC calls that may cause damages to those nodes.

tracts. Developers can use contracts to develop decentralized apps (or Dapps), including the lottery game, or digital tokens. Since its introduction, security issues of smart contracts have been widely studied by previous researchers. Atzei et al. systematically analyzed the security vulnerabilities of Ethereum smart contracts, and proposed several common pitfalls when programming the smart contracts [21]. For instance, the stack size of the Ethereum virtual machine is limited, attackers could leverage this to hijack the control flow of the smart contracts. A system called teEther [28] is proposed to automatically generate the exploits to attack the vulnerable smart contracts. The evaluation showed that among 38,757 unique smart contracts, 815 of them could be automatically exploited.

To mitigate the threats, some tools to analyze the smart contracts [10, 24, 27, 31, 37, 40] or fix the vulnerable contracts [34] have been proposed. For instance, Oyente [30] is a system that can automatically detect smart contracts vulnerabilities using the symbolic execution engine. Moreover, it can make the smart contracts less vulnerable by proposing some new semantics to the Ethereum virtual machine. Sereum [34] automatically fixes the reentrancy vulnerabilities in smart contracts by modifying the Ethereum virtual machine. Erays [40] is a tool to analyze the smart contracts without the requirement of the source code. In particular, it translates the bytecode to the high-level code that is readable for manual analysis. Securify [37] automatically proves smart contract behaviors as safe or unsafe. Other similar tools to analyze smart contracts include Mythril [10] and Maian [31].

8 Conclusion

In this paper, we performed a systematic study to understand the cryptocurrency stealing on Ethereum. To this end, we first designed and implemented a system that captured *real* attacks, and further analyzed the attackers' behaviors and estimated their profits. We report our findings in the paper and release the dataset of attacks (<https://github.com/zjuicrs/eth-honey>) to engage the whole research community.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. Special thanks go to Siwei Wu, Quanrun Meng for their constructive suggestions and feedbacks. This work was partially supported by Zhejiang Key R&D Plan (Grant No. 2019C03133), the National Natural Science Foundation of China under Grant 61872438, the Fundamental Research Funds for the Central Universities, the Key R&D Program of Shaanxi Province of China under Grant 2019ZDLGY12-06. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

References

- [1] Erc20 token standard, 2018. https://theethereum.wiki/w/index.php/ERC20_Token_Standard.
- [2] Ethereum address 0xa8015df1f65e1f53d491dc1ed35013031ad25034, 2018. <https://etherscan.io/address/0xa8015df1f65e1f53d491dc1ed35013031ad25034>.
- [3] Ethereum block 5899499, 2018. <https://etherscan.io/block/5899499>.
- [4] Ethereum json rpc, 2018. <https://github.com/ethereum/wiki/wiki/JSON-RPC>.
- [5] Ethernodes, 2018. <http://www.ethernodes.org>.
- [6] Etherscan, 2018. <https://etherscan.io>.
- [7] Go ethereum, 2018. <https://github.com/ethereum/go-ethereum>.
- [8] Minereum, 2018. <https://www.minereum.com/>.
- [9] Minereum market capital, 2018. <https://coinmarketcap.com/currencies/minereum/>.
- [10] Mythril, 2018. <https://github.com/ConsenSys/mythril>.
- [11] Over \$20 million stolen in ethereum by hackers from unsecured nodes, 2018. <https://sensorstechforum.com/20-million-stolen-ethereum-hackers-unsecured-eth-nodes/>.
- [12] Parity, 2018. <https://www.parity.io/>.
- [13] Planetlab, 2018. <https://www.planet-lab.org/>.
- [14] Solidity, 2018. <https://solidity.readthedocs.io/>.
- [15] There's some intense web scans going on for bitcoin and ethereum wallets, 2018. <https://www.bleepingcomputer.com/news/security/theres-some-intense-web-scans-going-on-for-bitcoin-and-ethereum-wallets/>.
- [16] Token learnchain, 2018. <https://etherscan.io/token/0x87c9ea70f72ad55a12bc6155a30e047cf2acd798?a=0xffecffe94c3e87987454f2392676ccdb98b926f8>.
- [17] Transfer token balance, 2018. https://theethereum.wiki/w/index.php/ERC20_Token_Standard#Transfer_Token_Balance.
- [18] Ethereum transaction 0x8d95864cc7142ef883148d45697b49be2c30a4275ebbcdbd2684acd809258b6, 2019. <https://web.archive.org/web/20190307141313/https://etherscan.io/tx/0x8d95864cc7142ef883148d45697b49be2c30a4275ebbcdbd2684acd809258b64>.

- [19] Ethereum transaction 0xdc4fe5301b9544892fdd97f476c1ec7d3da3de5ab75972866b87b7991cafedf6, 2019. <https://web.archive.org/web/20190307141627/https://etherscan.io/tx/0xdc4fe5301b9544892fdd97f476c1ec7d3da3de5ab75972866b87b7991cafedf6>.
- [20] Whackersforhackers.com - real time ip block list (ipbl), 2019. <https://whackersforhackers.com/share/tor.txt>.
- [21] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust*, pages 164–186. Springer, 2017.
- [22] Paul Baecher, Markus Koetter, Thorsten Holz, Maximilian Dornseif, and Felix Freiling. The nepenthes platform: An efficient approach to collect malware. In *Proceedings of the 9th Recent Advances in Intrusion Detection*, 2006.
- [23] Alex Bredo. Honeypot farms. <https://github.com/alexbred0/honeypot-camera>.
- [24] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ang Chen, Kun Yang, Bin Hu, Tong Zhu, Shifang Deng, Teng Hu, Jiachi Chen, and Xiaosong Zhang. Dataether: Data exploration framework for ethereum. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, 2019.
- [25] Juan Guarnizo, Amit Tambe, Suman Sankar Bhunia, Martín Ochoa, Nils Tippenhauer, Asaf Shabtai, and Yuval Elovici. Siphon: Towards scalable high-interaction physical honeypots. In *Proceedings of 3rd ACM Workshop on Cyber-Physical System Security*, 2017.
- [26] Xuxian Jiang and Dongyan Xu. Collapsar: A vm-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, 2005.
- [27] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.
- [28] Johannes Krupp and Christian Rossow. Teether: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27th USENIX Security Symposium*, 2018.
- [29] Tongbo Luo, Zhaoyan Xu, Xing Jin, Yanhui Jia, and Xin Ouyang. Iotcandyjar: Towards an intelligent-interaction honeypot for iot devices. In *Blackhat*, 2017.
- [30] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, 2016.
- [31] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *CoRR abs/1802.06038*, 2018.
- [32] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, and Tsutomu Matsumoto. Iotpot: Analysing the rise of iot compromises. In *Proceedings of the 9th USENIX Workshop on Offensive Technologies*, 2015.
- [33] Niels Provos. A virtual honeypot framework. In *Proceedings of the 12nd USENIX Security Symposium*, 2004.
- [34] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *Proceedings of the 26th Network and Distributed System Security Symposium*, 2019.
- [35] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.
- [36] Lance Spitzner. Honeypot farms. <https://www.syman-tec.com/connect/articles/honeypot-farms>.
- [37] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*, 2018.
- [38] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.
- [39] Matthias Wählisch, Sebastian Trapp, Christian Keil, Jochen Schönfelder, Thomas C. schmidt, and Jochen Schiller. First insights from a mobile honeypot. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, 2012.
- [40] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: Reverse engineering ethereum’s opaque smart contracts. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

Fingerprinting Tooling used for SSH Compromisation Attempts

Vincent Ghiëtte, Harm Griffioen, and Christian Doerr
TU Delft, Cyber Threat Intelligence Lab
{v.d.h.ghiette, h.j.griffioen, c.doerr}@tudelft.nl

Abstract

In SSH brute forcing attacks, adversaries try a lot of different username and password combinations in order to compromise a system. As such activities are easily recognizable in log files, sophisticated adversaries distribute brute forcing attacks over a large number of origins. Effectively finding such distributed campaigns proves however to be a difficult problem.

In practice, when adversaries would spread out brute-forcing over multiple sources, they would likely reuse the same kind of software across all of these origins to simplify their operation and reduce cost. This means if we are able to identify the tooling used in these attempts, we could cluster similar tool usage into likely collaborating hosts and thus campaigns. In this paper, we demonstrate that it is possible to utilize cipher suites and SSH version strings to generate a unique fingerprint for a brute-forcing tool used by the attacker.

Based on a study using a large honeynet with over 4,500 hosts, which received approximately 35 million compromisation attempts over the period of one month, we are able to identify 49 tools from the collected data, which correspond to off-the-shelf tools, as well as custom implementations. The method is also able to fingerprint individual versions of tools, and by revealing mismatches between advertised and actually implemented features detect hosts that spoof identifying information. Based on the generated fingerprints, we are able to correlate login credentials to distinguish distributed campaigns. We uncovered specific adversarial behaviors, tactics and procedures, frequently exhibiting clear timing patterns and tight coordination.

1 Introduction

Secure Shell (SSH) is a widely used protocol to operate services on a remote host over a network. One of the commonly used services of SSH is remote terminal access, which allows a user to execute programs on a remote system. The protocol authenticates a user based on a public key or an username/password combination, which prohibits malicious users to connect and exploit the host.

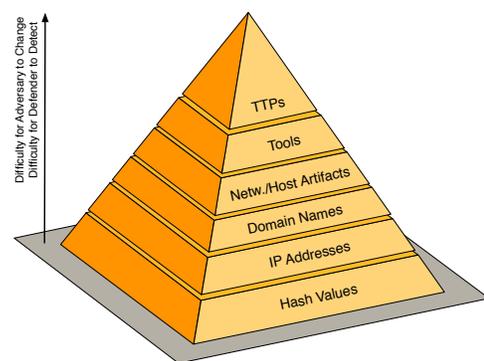


Figure 1: While basic Indicators of Compromise (IoC) are easy to gather and distribute, they are trivially changed by an adversary. For effective, more persistent detection it is necessary to assemble threat intelligence that covers behavioral features of the attacker. [3]

Due to the extensive use of the protocol, SSH is a popular target in brute forcing attacks. While system administrators are able to change the usernames and passwords used by the device, a lot of devices are still configured to use standard username and password combinations. As many devices are left with default configurations, simply trying a list of common username and password combinations proves effective enough for attackers to massively scan for, and attack SSH devices using this method.

While unsophisticated attackers would run through an extensive username/password candidate list in order to gain access, such behavior would be quickly visible in log files, and source addresses with repeated failed attempts are routinely blocked by intrusion detection systems (IDS) or monitoring systems such as fail2ban. Advanced adversaries would thus split the brute forcing out over multiple hosts, but in order to simplify the administration, usage and lower the cost, they would typically run a similar software across systems.

Current detection revolves mostly around simple indicators to detect malicious behavior. Virus scanners or intrusion detection systems for example rely on signatures and hashes to identify malicious activity, and also the IP addresses of malicious hosts scanning and brute forcing logins is enumerated and distributed in IP block lists. As explained by the so-called “pyramid of pain” [3] depicted in figure 1, these indicators are however trivially changed for an adversary, for example by simply recompiling a malware or moving the activity to a newly compromised host or proxy. In result, such information is unsuited to stop adversaries for long and at a broader scale. An alternative is the detection based on complex indicators, such as the systems or tools used or the tactics in a compromise, as they are much more difficult and costly for an adversary to change. If we can detect a particular software or modus operandi used for brute forcing SSH, we can reliably identify malicious activity, regardless of the IP address it is coming from and whether this address was participating in such activities before.

In this paper, we introduce methods for fingerprinting software stacks and tools used in SSH connections. This will help to study and follow the activities of adversaries, as an attacker will most likely distribute the same tool over a number of hosts to leverage the economies of scale. By detecting attacks by their used tools, attackers will have to change their software between campaigns, and even between different hosts. This greatly increases the cost for attackers, and can price them out of the system.

Our approach extracts session negotiation information such as the list and ordering of key exchange algorithms, cipher suites, or compression algorithms which are exchanged in clear text during the SSH session initiation. This means that using this approach, we do not need to interfere with the connection itself, meaning that the method is completely passive, and as the fingerprint is derived from the SSH handshake, we are able to identify brute forcing attempts even before the first password is sent to the system.

This paper makes the following contributions:

- We introduce the concept of fingerprinting to the SSH protocol and demonstrate based on a large corpus of 35 million brute forcing attempts that fingerprinting is suited to identify tools that are used by adversaries. By detecting attacks on this level, the cost for adversaries rises as they need to build new tools for every campaign.
- We deploy the technique to 4,500 honeypots with the aim of gaining cyber threat intelligence about the practices of adversaries. We empirically show the presence of 49 different tools, and show that a cluster of hosts relies on the same toolchains. We furthermore find evidence of large, distributed campaigns of collaborating hosts.

The remainder of this paper is structured as follows: Section 2 describes the state of the art in fingerprinting and SSH

brute forcing. Section 3 provides an overview of the SSH protocol and components necessary to introduce the proposed method. Section 4 explains the fingerprinting methodology. Section 5 provides details about the design and scale of our honeynet. The evaluation of our proposed method is presented in Section 6. Using our method, we find a large number of actors, each featuring different strategies, tactics and resources. Finally, Section 7 summarizes and concludes our work.

2 Related Work

As stated in the previous section, a sustainable cyber defense best focuses not on identifiers of specific malicious instances, but on characteristics that are constant over multiple instances. One way of generating these characteristics is to fingerprint the tools used by attackers. Our main claim in this paper is that we can extract fingerprints from the SSH connection negotiation that can be used to distinguish different tools. Two lines of related work are important to class the proposed work, first previous research on fingerprinting protocols, and second previous research in brute force detection.

First, while fingerprinting has not been done in SSH, differences in cipher suite strings have been used in the SSL/TLS protocol suite to identify server or client software. To fingerprint clients, Husak et al. [9] were able to infer the used client application based on the cipher suites that were used in the connection. The authors found that many applications support different cipher suites for establishing a connection, and some applications also send the cipher suites in a different order. Therefore, the authors were able to fingerprint client applications using only the cipher suites presented in the handshake. Durumeric et al. [4] applied the analysis of advertised clients (through the HTTP User-Agent) and implemented SSL/TLS handshakes to detect the nature of the client connecting, and thereby identify middleboxes that intercepted the TLS connection between client and server. Fingerprinting specific implementations is also possible by detecting specific patterns in which header fields [5] or packet payloads [6] are set and encapsulated in scan and attack traffic.

Fingerprinting the traffic sent through encrypted channels has been done by Sun et al. [18]. Their algorithm is able to identify which webpages are visited from the amount of traffic sent during the page load. Similar research by Korczynski et al. [11] uses Markov chains to generate fingerprints for different services based on the SSL session. Their research shows that they are able to fingerprint certain applications with a high confidence level. In the case of SSL, research has focused on fingerprinting clients and client behavior. Our SSH fingerprinting method leverages the same intuitions, but is tailored towards fingerprinting adversaries that are attempting to compromise a system.

Second, although there exists no prior work in the literature for fingerprinting SSH endpoints, a selection of previous studies have developed methods for detecting SSH brute forcers.

Hellemons et al. [7] have proposed an intrusion detection system method for detecting SSH intrusions using netflows. Similarly, Najafabadi et al. [14] propose a machine learning algorithm to detect brute force attacks in netflow data. The authors have validated their results on the SSH protocol and found that machine learning techniques perform well for detecting these brute force attacks. Nicomette et al. [15] clustered adversaries together based on attempted passwords. The authors find relationships between dictionaries, but at the same time notice that few dictionaries are shared across attackers. All these works focus on the detection of brute forcing attacks at the moment that the system is already under attack, however we show in the following that it is possible to obtain much information about the incoming request during the connection negotiation itself and before the password prompt is shown.

A selection of studies have investigated adversarial behavior after the successful compromise of a honeypot. Ramsbrock et al. [16] followed compromises made into four honeypots, and was able to derive a state machine to describe the actions of adversaries. Barron and Nikiforakis [2] investigated whether adversarial actions differed based on environmental factors, for example depending on the presence of real users on the systems and their usage of files. They were able to distinguish between human and bot login activity, and noticed humans did to a limited extent show interest in stored files while bots generally avoided significant interaction with the file system, and in half of the cases only proceeded to install a proxy gateway.

While proposed methods can identify brute force attacks, they do not allow for tool classification or for pre-emptively stopping these attacks. Given the current threat landscape, in which there is a high number of attackers, identifying attacks in an early stage before actual compromization is increasingly important. By forcing attackers to change their tools every attempt, the cost for attackers increases and many attackers will be priced out of the system. In this paper, we propose a method to fingerprint tools in use by adversaries, which can be used to track their activities over time, relate distributed attempts to the same toolchain and possibly actor, and thus gain a greater insight into the ecosystem as a whole.

3 The SSH Protocol

The secure shell protocol (SSH) is an established protocol for accessing services on a remote host, which is secured by an authentication procedure. In order for an attacker to enter login credentials it is first necessary to set up a secure protocol connection as specified in RFC4253 [20]. The main steps in setting up a secure communication channel between the attacker and its target are shown in figure 2 and go through three main phases.

First, after a TCP connection has been established, both parties exchange the version of the SSH protocol they are

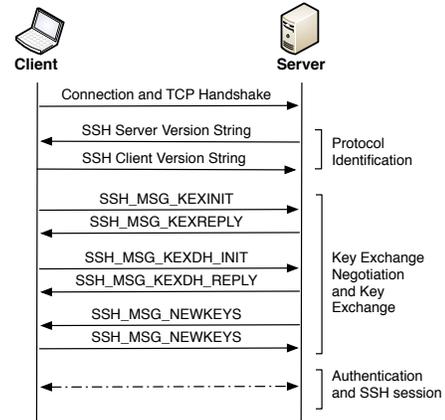


Figure 2: Schematic overview of the message exchanges in the establishment of an SSH session.

running in the protocol identification phase. Typical examples of sent version strings are SSH-2.0-JSCH-0.1.51 and SSH-2.0-paramiko_1.7.5. As will later be explained, the version exchange is one of the components used for profiling attackers.

As SSH provides an authenticated encryption tunnel, both sides need to negotiate key material for the connection. In the second phase of the protocol, client and server negotiate the key exchange mechanism to be used. This negotiation is initiated by the client through a key exchange initialization message (SSH_MSG_KEXINIT), which contains all the different key exchange algorithms, encryption algorithms, algorithms to compute a message authentication code, and algorithms for compressing the data the client is able to accept. The order of the advertised algorithms is of importance as the algorithms are advertised according to the host's preference, and thus both the presence and order of algorithms shared during this step of the SSH connection can be used to profile a connecting client. After both parties have sent and received the key exchange initialization message, the highest commonly preferred algorithms are selected for setting up a secure connection. Depending on which algorithms have been chosen, the rest of the key negotiation and key exchange procedure slightly varies. After the key negotiation, the actual key exchange is initialized by sending the SSH_MSG_KEXDH_INIT message, after which the key exchange algorithm is run. Once the algorithm is finished, each side signals using a SSH_MSG_NEWKEYS message that the secure connection is set up and ready to be used.

In the third phase, both client and server switch to an encrypted tunnel using the just negotiated key material and perform the authentication, during which the client sends its login credentials. Given the correct credentials, the SSH protocol then makes the requested resource on the remote host available to the client.

4 Fingerprinting Tooling

Brute forcing the login credentials to gain access to a shell generally requires a great amount of attempts due to the large amount of possible username/password combinations. Therefore, it is uneconomical for an attacker to perform this task manually, and he or she will likely resort to a tool in order to automate the login attempts. Depending on the knowledge of the attacker, he or she will utilize an existing tool or develop a new one.

If the attacker opts to use known tools, there is no shortage of available material. A quick search on any search engine yields an extensive list of SSH brute forcing tools. Most of these programs are advertised as penetration testing tools, used to assess the security of a network, for example to find servers that use weak login credentials. More so, entire articles and tutorials are dedicated to the usage of those tools, such as Hydra [8], Medusa [13], and Ncrack [12].

When writing an SSH brute forcing tool, one could create an implementation of the SSH protocol, or use a pre-built library that handles the SSH connection. The aforementioned brute forcing tools utilize different libraries implementing the SSH protocol, and only add the logic to perform the attack. Although the libraries are likely to adhere to the standards specified in the RFC describing the SSH protocol, minor differences in connection establishment can be witnessed. One of those difference is the announcement of the SSH version. Libraries such as libssh [1] and libssh2 [17] use a different version string to announce compatibility with the same version of SSH protocol. Both libraries add their name and release version into the SSH version string; libssh version 0.7.1 announces `SSH-2.0-libssh-0.7.1`, whereas libssh2 version 1.8.0 identifies itself as `SSH-2.0-libssh2_1.8.0`. While this provides a first, trivial angle to identify the tools used by attackers, we only use this information as a reference and later combine it with the capabilities implemented by a specific library. Thus, if an adversary is spoofing the version string, the announced version will not match the fingerprint of the advertised key exchange, encryption, MAC and compression algorithms anymore, which in combination yields an even more distinctive fingerprint for a specific brute forcing tool implementation.

Similarly to the announced version string, the information exchanged during the key exchange initialization varies between libraries. As discussed in the previous section, during the session establishment, different algorithm suites are announced in order of preference. Not all libraries support all key exchange, encryption, MAC, or compression algorithms. More so, not all libraries supporting identical algorithms will order them according to the same preference. This multitude of possible variation of the initialization message increases the likelihood of libraries implementing them differently, which

can be leveraged to identify the tool and/or underlying library in an incoming SSH connection request.

In order to compare different key exchange initialization messages, the four exchanged capabilities as discussed in Section 3 are used. The advertised

1. key exchange algorithms (kex),
2. symmetric encryption algorithms (enc),
3. message authentication code algorithms (mac), and
4. compression algorithms (comp)

are concatenated into a single capabilities string. Since we only care about exact matches in capabilities, we hash the concatenated string as a fingerprint for a specific tool. Consider the case of an out-of-the-box SSH server install on Ubuntu 16.04 Desktop, which comes preconfigured with the following configuration:

- *KEX algorithms:* `curve25519-sha256@libssh.org,ecdh-sha2-nistp256,ecdh-sha2-nistp384,ecdh-sha2-nistp521,diffie-hellman-group-exchange-sha256,diffie-hellman-group14-sha1`
- *Ciphers:* `chacha20-poly1305@openssh.com,aes128-ctr,aes192-ctr,aes256-ctr,aes128-gcm@openssh.com,aes256-gcm@openssh.com`
- *MAC algorithms:* `umac-64-etm@openssh.com,umac-128-etm@openssh.com,hmac-sha2-256-etm@openssh.com,hmac-sha2-512-etm@openssh.com,hmac-sha1-etm@openssh.com,umac-64@openssh.com,umac-128@openssh.com,hmac-sha2-256,hmac-sha2-512,hmac-sha1`
- *Compression algorithms:* `none,zlib@openssh.com`

Thus, we obtain the fingerprint through the MD5 hash of

```
curve25519-sha256@libssh.org,ecdh-sha2-nistp256,ecdh-sha2-nistp384,ecdh-sha2-nistp521,diffie-hellman-group-exchange-sha256,diffie-hellman-group14-sha1;chacha20-poly1305@openssh.com,aes128-ctr,aes192-ctr,aes256-ctr,aes128-gcm@openssh.com,aes256-gcm@openssh.com;umac-64-etm@openssh.com,umac-128-etm@openssh.com,hmac-sha2-256-etm@openssh.com,hmac-sha2-512-etm@openssh.com,hmac-sha1-etm@openssh.com,umac-64@openssh.com,umac-128@openssh.com,hmac-sha2-256,hmac-sha2-512,hmac-sha1;none,zlib@openssh.com
```

resulting in `9f735e5485614bcf6e88b9b848582965`.

Finding or building a tool for SSH brute forcing is only the first step for an attacker towards compromise - next the adversary needs to choose which login credentials to use during authentication. Here the attacker has three main choices:

First, an attacker can opt to incrementally, pseudo randomly, or randomly generate the username and password from a predefined set of characters. Given enough time, and assuming the attacker is not blocked after some attempts, this method is bound to provide access to the targeted machine. Second, an attacker can decide to generate the login credentials using a dictionary attack. In a dictionary attack, words are combined to form the password and or username. Third, an attacker can choose to use predefined login credentials and available password lists. As with SSH brute forcing tools, a quick search on any search engine reveals a plethora of hits advertising password list for download. These lists often contain known default passwords and usernames such as `admin` and `root`.

Here, again, the different options available for the attackers to generate the login credentials offers an opportunity to detect brute forcing campaigns, and we put forth the assumption that the same adversaries will most likely use the same or similar login credentials when trying to exploit systems. Under the premise that an attacker will use the same tool to attack multiple targets, the same fingerprints for a single IP address should be witnessed at different honeypots. While the combination of algorithms provides already a distinctive fingerprint for a specific tool, in the later part of our evaluation we further investigate the relationship between identical tools and password generation algorithms. If an attacker uses multiple machines or IP addresses with the same brute forcing tool and password generation algorithm, the previously described fingerprints can be used to cluster IP addresses belonging to the same attacker.

5 Data Collection

This section describes the setup of our honeypot infrastructure and the data acquisition strategy used in this paper. As the goal of our study is to demonstrate the possibility of fingerprinting the tools and techniques used by adversaries in SSH break-in attempts, we have designed a distributed honeypot system and exposed approximately 4,500 honeypots distributed over three /16 subnets to the open Internet.

5.1 Honeypot design

As you recall from section 3, the SSH protocol goes through three main phases in connection establishment: first, client and server announce their protocol versions to each other, second, the endpoints exchange their ciphering, MAC and compression capabilities and agree on a key, and finally, the tunnel is authenticated by a public key or password before an SSH session is established. While we would clearly expect to see differences in the behavior of adversaries after they have gained access to a particular machine, the SSH protocol is complex enough and contains several configuration options so that the tooling used by attacker may contain implementation differences, that will – as we show in the following –

allow recognition of a particular tooling *even before* the SSH protocol advances to the password prompt and an interactive session. The distributed infrastructure was therefore implemented as a honeypot that would negotiate a key exchange and an SSH session with the connecting client, display a login prompt and collect usernames and passwords, but never let any user in. While this simplifies containment and thus reduces the risk of operating such a system, to the connecting user it basically appears like a regular server and an incorrect password guess.

In order to pose as yet another open SSH server, it is essential that the honeypot itself blends in with existing installations found on the Internet, otherwise knowledgeable adversaries could soon identify instances running some honeypot software and avoid individual IPs or even subnets where honeypots were detected. Thus, it would be a major failure if a system meant to identify adversaries based on handshake fingerprints could be fingerprinted itself, which has been found to be an issue for existing common open source honeypots such as Kippo or Cowrie [19]. To avoid this problem, we connect the incoming session to an actual OpenSSH implementation running inside a container, which matched in terms of version strings, list of available algorithms and options and ordering a default Ubuntu 16.04 LTS installation. This way, an adversary probing the system for implementation deviations to unexpected inputs will observe no difference from a typical server, and to an adversary scanning the Internet for banners and key exchanges our honeypots will blend in with what one would expect when connecting to a popular deployed operating system.

5.2 Organizational Placement

Aside from being identifiable based on specific implementation characteristics, it would also be conceivable that adversaries could spot honeypots based on the way they are deployed and subsequently avoid them. For example, an apparently open SCADA system hosted on an Amazon EC2 cloud IP address should trigger some suspicion in a knowledgeable adversary. In our case, an entire block of consecutive IP addresses running SSH servers where otherwise nothing else is open in the network could similarly bias the results, and adversaries be motivated to evade such networks.

In order to create a believable posture and collect representative results, we deployed the honeypot in the enterprise network of an organization. This organization is connected with three /16 networks to the Internet, on these networks approximately 60,000 devices are active and incoming SSH traffic is not filtered by the firewall. These 60,000 official network hosts were of various types and origin, with a mix of servers, workstations, laptops and other mobile devices. While the foremost category would be constantly powered on and accessible, personal workstations, laptops or phones would only be powered on at select times.

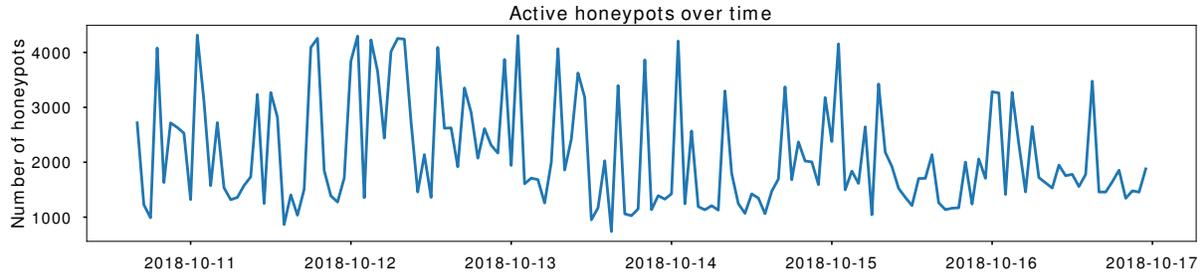


Figure 3: Number of active honeypots aggregated by hour over the course of one week in the study.

In this study, we spread the 4,500 honeypots randomly throughout the network ranges of the organization, so that they would be assigned IP addresses belonging to server, workstation or mobile host subnets. This meant that IP addresses belonging to our honeypot system were located in between sets of servers or regular user machines, thus an adversary exploring the network could not evade the honeypots by skipping select parts of the network ranges, and after scanning several “official” servers our honeypot would appear to the adversary as just another server in the same group. The routing rules of the organization were set up in such a way that IP addresses that were allocated to a host but *also* chosen for the honeypot were forwarded to the official host whenever powered on, and forwarded to the decoy as soon as the official host left the network. This way, adversaries interacting with the organization’s hosts would experience an instantaneous seamless handover to our honeypot infrastructure.

Figure 3 depicts the number of active honeypots in our deployment over the course of one week. The graph clearly shows a diurnal pattern stemming from a base population of approximately 1,500 active systems in server and workstation ranges, as well as an additional 3,000 decoys which only become activated when the mobile and temporary devices leave the network. Both components of our honeypot deployment strategy make it thus very challenging for an adversary to locate and evade our infrastructure.

6 Evaluation

This section evaluates the results of applying the fingerprinting methods on the dataset. Both the SSH versions string and the fingerprint based on the session negotiation are analyzed. In addition, we use time and password correlation to evaluate the hypothesis that attackers leverage multiple hosts to brute force SSH servers.

6.1 Available fingerprints

During the data collection period, a total of 107,793 hosts tried to brute force the login credentials of one or more honeypots

in our network. We only considered source IPs that completed at least one completed SSH key exchange towards our 4,500 honeypots during the entire month, thus excluding mere TCP SYN scans. Within this entire dataset, we observed a total of 123 SSH version strings, and identified 49 distinct MD5 hashes for different libraries and library versions in use.

While the analysis yielded a substantial count of different fingerprints and version strings, we also find that these instances are also surprisingly well spread across source hosts. The distribution of source IPs that use a particular fingerprint follows an exponential decay; while the most commonly used library fingerprint is used by 58% of the sources, already the bottom half of the top 10 fingerprints account only for fractions of a percent. Fingerprints thus have large amounts of variations, and are distinctively correlated to sources: more than 89% are only associated with one fingerprint over the entire observation period. Similar results apply to the version strings; the top 3 version strings are used by more than 75% of all source IPs, also here 90% of all sources only advertise one version string to our honeypot during the entire period.

The large body of fingerprints compared to the number of available tools, as well as the larger number of version strings compared to the amount of fingerprints matches our expectations how brute forcing tools are developed and used. First, as commonly used tools build on system libraries such as libssh or OpenSSH, major updates to the underlying system library implementing the SSH protocol will result in a new fingerprint, even though the adversary uses the same toolchain.

Second, tools (or their users) actively change the version string and configuration advertised by their toolchain, possibly in an attempt to evade detection by signatures. Examples of this are invalid encryption and mac algorithms such as `hmac-sha\x11` and `lowfIsh` in some of the key exchange messages. Software packages such as OpenSSH allow a user to configure the ciphers used in the key exchange message, even if these algorithms are not implemented in the library itself. Inspection of the source code of for example the libssh2 library reveals that the versions strings announced by the majority of the hosts matches with the one in the source code. When we further analyze the design of available brute

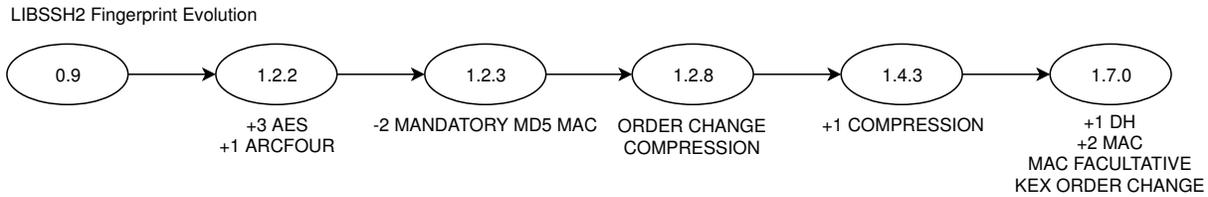


Figure 4: Evolution of the algorithms used by the libssh2 library to construct the SSH_MSG_KEXINIT message.

forcing tools, we can link common tools back to these libraries. Medusa [13], a tool for SSH brute forcing included in Kali linux, builds on this library in this selected version. The analysis of the 35 million compromisation attempts using the SSH version and key exchange fingerprinting leads to the conclusion that the majority of attackers use readily available tools to perform the brute forcing.

6.2 Fingerprints and libraries

Different tools generate different combinations in terms of SSH version strings and key exchange fingerprints. If two tools use different libraries implementing the SSH protocol, the announced SSH version string will likely be different. As seen in the data, the name of the SSH library is often included in the announced SSH version string.

Similarly, the fingerprint retrieved from the key exchange can also provide an indication of the tool used. Tools building upon different libraries will announce different algorithms during the key exchange. This is due to different libraries supporting different algorithms and announcing them in different order of preference. Therefore, due to slight variations in used libraries and implementations, tools can be linked to the fingerprints generated by processing the SSH version string and the key exchange initialization message.

In order to investigate the origin of the fingerprints and study user customization, we downloaded 10 commonly used SSH brute forcing tools to inspect which library is linked to realize the SSH protocol, and in addition mined the SSH version strings from all received handshakes for mentions of libraries or implementation stacks. This yielded a total of seven libraries that are utilized across SSH brute forcers we observed, namely (1) Granados, (2) JSCH, (3) libssh, (4) libssh2, (5) OpenSSH, (6) paramiko, and (7) the Erlang standard SSH implementation. For all these libraries, we downloaded every single release, as well as every intermediate version available in the software repositories and manually identified the location in the source code responsible for the SSH connection and parameter selection. A program then identified every intermediate/release version when this code was modified over the entire period of the softwares' past development, and we manually analyzed each changed code segment to extract how

this library would advertise itself as in this particular version. This yielded a set of (banner, MD5 hash) tuples, which is on the one hand dependent on the version of a library, on the other hand also on certain options and taken branches in the code, for example if certain other libraries or headers were available on the system where it would be compiled and thus activate `ifdef` blocks. Based on this, we generated a set of possible 57 banner-hash tuples for the seven aforementioned libraries, implemented in a particular software package at any point in time.

Figure 4 shows this evolution for the libssh2 library with respect to the construction of the SSH_MSG_KEXINIT message. While libssh2 saw several intermediate releases between version 0.9 and version 1.2.2, and the library advertises itself differently in between these two releases, the cryptographic routines remained unchanged during all of these updates leading to an identical fingerprint. In 1.2.2, three options for the data encryption using the AES cipher suite as well as the option to encrypt using RC4 were added, while in 1.2.3 the previously mandatory option to create a message authentication code using MD5 was removed. Later versions such as 1.2.8 only differed in the order they advertised the preference of algorithms. Similar version graphs were created for the other six libraries mentioned above, most of them appeared in our data set announcing different release versions.

When libraries are compiled, the supported algorithms might differ, depending on installed software packages that are required by the algorithms. These dependencies can greatly affect the number of supported algorithms during the key exchange initialization, which is why we have identified all different combinations possible. These also result in a unique fingerprint for a particular installation path and thus allow a peek into the configuration of the attack hosts.

Given the advertised library and version string we can then cross validate whether the fingerprint obtained from the handshake is consistent with the default behavior of the library, or whether some code changes or configuration changes were introduced. Interestingly, when we look at the 123 SSH version strings and 49 fingerprint hashes that we collected in our honeypots, we find that all 57 theoretically possible tuples from the software libraries were present. When we match the version string for a particular library and the fingerprint hash that *should* have been generated from the honeypot handshake,

we find that there is only a match for 26 out of the checked 57 library versions. This indicates that in 31 instances, more than half, the announced version string is spoofed. We manually verified these instances of spoofing, and found that while some of them are attempts to make the version string more generic, others modify the version number of a library or pretend to run a different software stack than the behavior of the library would in practice indicate. For example, a particular brute forcing tool would announce OpenSSH version 4.3, but announces a cipher suite that was not implemented in this particular version. Also, the order of algorithms for some advertised versions of libssh and libssh2 do not match the implementation of the library.

While spoofing of version string is common among attackers, given our tracking of code changes, we were able to trace back 26 out of the 31 spoofing instances to a library that is consistent with the behavior of the brute forcing tool. Overall, we find that we were able to identify more than 91% of the tools used to attack our honeynet using the fingerprint.

6.3 Collaborating hosts

As we have shown in the previous part, the combination of available key exchange algorithms, cipher suites, MAC and compression options together with the advertised version string does contain large amounts of entropy. As an additional verification that this fingerprint can serve as a measure to fingerprint the tooling itself, we look in this part into the behavior of the hosts exhibiting a particular fingerprint. If this relationship holds, we would expect the following two results: First, commonly available tools should see continuous usage, but within this set there could be groups of hosts that use the same tool in a specific way or with a similar behavior that could be clustered together. Second, given that we identified 31 mismatching version strings and fingerprints, we would expect some adversaries to have built custom tools for SSH brute forcing. As these are not publicly available, they should only be in used by a limited group of source hosts, thus the tool fingerprint could be used as a proxy to partially fingerprint the actor.

In the following, we will now investigate the different behaviors of the 49 different key exchange-cipher-MAC algorithm hashes that we initially discovered in our dataset. Figure 5 shows the activity of these fingerprints over the course of the experiment, for compactness of the figure and the discussion, each fingerprint has been assigned a numeric ID from 0 through 48. The number of hosts using a tool with a specific fingerprint at a given time is represented by the size of the marker in the plot, with the area of the markers being proportional to the number of unique hosts using a fingerprint per hour. In the figure we can readily identify five distinct behavioral patterns of fingerprint usage:

- *Popular, commonly available tools* such as Ncrack (fingerprint 30 in cyan), or SSHtrix (fingerprint 16 in red)

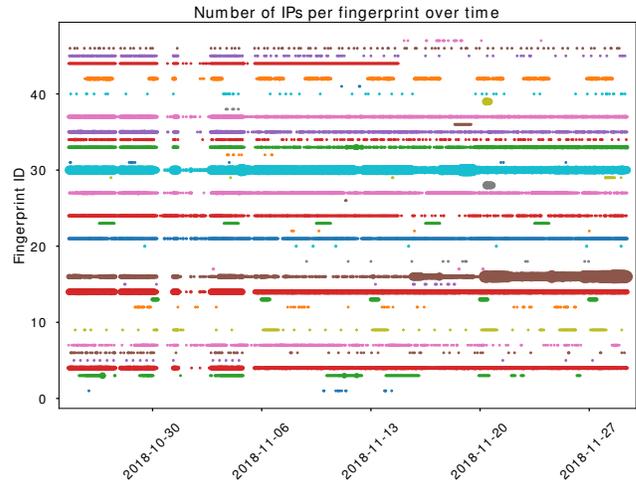


Figure 5: Scatter plot showing the number of hosts using a certain key exchange fingerprint over time. The number hosts is indicated by the size of the markers.

or Hydra (fingerprint 4 or 14 depending on the operating system it is installed on such as raspbian), see some continuous usage by a diverse and significant number of origins. In the next section, we further show that these groups can be separated by the password list configuration of the brute forcer into distinct subgroups that pursue a common strategy.

- *Custom tools* with a relatively uncommon or unique fingerprint are essentially only in use by distinct groups of adversaries. Often these are rolled out to a large amount of hosts, and from there explore remote hosts simultaneously with synchronized start and stop times. Consider for example fingerprint #23 indicated in green, which performs a weekly recurring scan of the address space, always using a similar amount of resources. Similar behaviors are shown by the groups of hosts using the tool with fingerprint #42 (orange) who scan every other day, or by the groups scanning bi-hourly using the tool with fingerprint #46 (brown).
- Among these, some *custom tools are run at low intensity* only by a select group of origins, which on average make less than 4 login attempts per hour. All installations with fingerprint #36 (purple) operate in this way, and may be classified as a slow brute forcing campaign following the description by Javed and Paxson [10].
- *Distributed hit & run campaigns* only occasionally surface, but then for a short period of time brute force many remote hosts with a large number of resources. Interestingly, we observe these hit & run campaigns to typically exhibit a unique fingerprint and thus employ custom

tooling for their activities, which makes these attempts and hosts participating in them easily identifiable by our proposed method. Examples of such fast, concentrated attempts are fingerprints 39 (light green) or fingerprint 29 (gray). The plot shows that both clusters become active during the same time period, around November 20th, and have approximately the same size. A closer inspection reveals that the hosts using the tool identified by fingerprint ID 29 are also using the tool identified by fingerprint ID 39. The IP addresses are located in 103 different /8 subnets indicating that the attacker has the knowledge, resource and intention to spread his or her infrastructure across the Internet, possibly in an attempt to avoid detection.

6.4 Password combinations

After the completion of the key exchange and session negotiation, the adversaries were presented a login prompt they could interact with. Past work such as Nicomette et al. [15] have used the entered user credentials to link individual login attempts into related clusters, and for example identified relationships between dictionaries but also noticed that only few of them were reused across adversaries. As discussed before, the economies of scale would imply that an attacker is most likely going to deploy the same setup and tooling on different hosts to launch attacks, thus the same tooling would result in an identical fingerprint, and thus help us gain deeper insights into the activities of the attackers, for example if they are splitting and distributing parts of dictionaries for brute forcing across collaborating hosts. In this section, we will discuss the relationship between groups found based on an identical banner and key exchange, and their associated password lists.

For each of the 49 fingerprints detected earlier, we extracted all SSH sessions from any host that exhibited this signature and assembled the set of credentials (username + password) during login attempts. Table 6.4 shows a selection of 8 fingerprints, which exemplarily shows the spectrum of different key exchange - password list behaviors found throughout the dataset. From the data we can distinguish three types of groups: First, we see clusters where tools and the credential list used are tightly linked together. Second, we clearly see select fingerprints in wide use, which focus on (subsets of) fixed password lists. Third, we observe groups of tooling, where hosts pursue brute forcing with diverse and customized password lists. We will discuss each of these three categories in the following subsections.

6.4.1 High credential / tool correlation

For each group advertising the same banner and using the same key exchange algorithm, the table lists the number of IP addresses matching this fingerprint and the number of

unique login credentials list used by an attacker belonging to the cluster. To provide a better understanding of the login credential lists used, the number of hosts using the 5 most frequently used credential lists are shown.

All clusters in this category exhibit a tight link between the fingerprint and the utilized password list. For example, the fingerprint `0df0d56bb50c6b2426d8d40234bf1826` of cluster 1 is sent by 684 hosts, however within this group only 9 different password lists are used. The vast majority of hosts in this cluster, 672 or 98.2%, always send the exact set of credentials to our honeypots, deviations of the cluster default occur only very infrequently among all remote hosts having connected to our honeypots. In addition to the strong link from a particular fingerprint to a credential list, also the reverse is true: no other attackers have been using this credentials list in the dataset. This would indicate that the proposed fingerprinting method can be used as a predictor for password usage.

6.4.2 Popular tool

The second category of behaviors we can distinguish in the fingerprint analysis are those tools which are widely deployed but are run similarly configured. In this particular category, we observe the presence of multiple common credential lists, from which hosts pick a subset and brute force all of our honeypots with the same credential set.

An example of this is cluster 5 with 86,805 IP addresses, which employed a surprisingly low number of 625 login credentials lists. The top five groups all settle on a permutation of *(admin, admin)*, *(admin, default)*, *(admin, password)* and move horizontally throughout our ranges. Other groups of hosts choose from lists geared towards specific type of devices, for example credential lists associated with common IoT devices or Raspberry Pi distributions.

6.4.3 Diverse credential lists

While both previously described clusters could also have been discovered using password-based grouping as adversaries shared significant credentials, we found a third behavior of brute forcing which would have remained undetected to established methods. Clusters 6 through 8 belonged to a new category characterized by a high number of credentials list and a low number of IPs using identical credential lists.

Consider for example the case of group 6, where 557 different password lists appear across 564 different hosts. This is due to the fact that attackers in this cluster use random or pseudo randomly generated passwords in combination with 22 fixed credential tuples. The generated passwords are all 10 characters long consisting of letters and numbers. Next to randomization, each host only performs a limited amount of login credentials, 80% (463 out of 564 IPs) use less than 60 unique login credentials. Another indicator of randomization is the low overlap between passwords, 6,592 of the 10,318

Type	High credential / tool correlation			Popular tool		Diverse credential lists		
Banner	SSH-2.0-OpenSSH_7.4p1-Raspbian-10+deb9u3	SSH-2.0-OpenSSH_7.4p1-Raspbian-10+deb9u4	SSH-2.0-libssh2_1.8.1_DEV	SSH-2.0-libssh2_1.7.0	SSH-2.0-libssh2_1.8.0	SSH-2.0-Go	SSH-2.0-Go	SSH-2.0-libssh-0.6.3
Kex	0df0d56bb50c6b2426d8d40234bf1826	0df0d56bb50c6b2426d8d40234bf1826	1616c6d18e845e7a01168a44591f7a35	a7a87fbe86774c2e40cc4a7ea2ab1b3c	a7a87fbe86774c2e40cc4a7ea2ab1b3c	c39f4cec145ee3d50fb590595143b9d5	72d744cee7c48197c1b56973e8600140	51cba57125523ce4b9db67714a90bf6e
Cluster	6	7	8	4	5	1	2	3
IP count	684	1138	85	6473	86805	564	208	4479
# cred. lists	9	5	7	2688	635	557	111	4438
Credential list								
Top 1 list	672	1127	79	3602	64140	3	65	18
Top 2 list	2	6	1	16	6024	2	11	3
Top 3 list	2	3	1	16	4488	2	7	3
Top 4 list	2	1	1	13	4347	2	5	3
Top 5 list	2	1	1	10	2832	2	3	2

login credentials are used by only one IP address. The password randomization is confirmed by a mean Jaccard index of 0.4 of the credential lists used in the cluster.

While randomization of credentials would make it challenging for a password-based clustering algorithm to detect similarly acting groups, the same will hold true for brute forcers that rely on a *very* long lists of candidate credentials from which username/password combinations are picked. This will dilute possible linkage from the perspective of common credentials, however a clustering based on the fingerprint will find these relationships.

In this section we have shown that clustering based on SSH banners and key exchange algorithms can find different types of clusters. The proposed method is able to find groups using extensive password lists or random password, whereas password-based solutions would struggle. However, password-based solutions can provide better clustering performance for popular tools having multiple credentials lists such as cluster 5, hence a combination of both a fingerprint-based and password-based approach promises to provide more, and complementary findings.

7 Conclusion

In this work, we have described a method for fingerprinting tools for SSH brute forcing based on version strings and advertised algorithms. As this method distinguishes tools based on the data exchanged during the key initialization, we are able to detect tools prior to even entering the session authentication, and can deploy the mechanism transparently without any changes necessary to an existing enterprise infrastructure.

We have deployed the fingerprinting method over 4,500 honeypots for a period of one month, and from 35 million login attempts been able to detect 49 different fingerprints. The results indicate that different tools are used in different

ways, indicating that attackers customize, or develop their own tools. Looking at the behavior of different tools, we were able to identify clear timing patterns originating from different tools, while based on timing patterns we can detect distributed brute forcing campaigns. By fingerprinting the tools used in a campaign, we are able to track and analyze the campaign over time. Additionally password analysis of detected clusters allowed us to identify different brute forcing methods. Both assessments contributed in providing insights into the tactics, techniques and procedures of the attackers.

Future Work

The work presented in this paper was conducted from the angle of cyber threat intelligence, with the aim of augmenting the portfolio of methods to fingerprint adversarial tooling and gather insights into their activities and behavior. This study led to a variety of different fingerprints, some of which could be traced back to specific brute-forcing tools. In principle, the contribution of this method could however be wider, and potentially be suitable as an additional detection rule within the context of intrusion detection systems. To evaluate its merit for such active threat detection and prevention, further research is however needed to evaluate its efficacy, which has not been done within the scope of this study.

References

- [1] Aris Adamantiadis, Andreas Schneider, Nick Zitzmann, Norbert Kiesel, and Jean-Philippe Garcia Ballester. libssh. <https://www.libssh.org/>.
- [2] Timothy Barron and Nick Nikiforakis. Picky attackers: Quantifying the role of system properties on intruder behavior. In *Annual Computer Security Applications Conference*, 2017.
- [3] David J. Bianco. The pyramid of pain, 2013.

- [4] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman, and Vern Paxson. The security impact of https interception. In *The Network and Distributed System Security Symposium*, 2017.
- [5] Vincent Ghiette, Norbert Blenn, and Christian Doerr. Remote identification of port scan toolchains. In *IFIP International Conference on New Technologies, Mobility and Security*, 2016.
- [6] Vincent Ghiette and Christian Doerr. How media reports trigger copycats: An analysis of the brewing of the largest packet storm to date. In *ACM SIGCOMM Workshop on Traffic Measurements for Cybersecurity (WTMC)*, 2018.
- [7] Laurens Hellemons, Luuk Hendriks, Rick Hofstede, Anna Sperotto, Ramin Sadre, and Aiko Pras. Sshcure: a flow-based ssh intrusion detection system. In *Conference on Autonomous Infrastructure, Management and Security*, 2012.
- [8] Marc Heuse, David Maciejak, and Jan Dlabal. Hydra. <https://github.com/vanhauser-thc/thc-hydra>.
- [9] Martin Husák, Milan Čermák, Tomáš Jirsík, and Pavel Čeleda. Https traffic analysis and client identification using passive ssl/tls fingerprinting. *The European Association for Signal Processing Journal on Information Security*, 2016.
- [10] Mobin Javed and Vern Paxson. Detecting stealthy, distributed ssh brute-forcing. In *ACM Special Interest Group on Security, Audit and Control Conference on Computer & Communications Security*, 2013.
- [11] Maciej Korczyński and Andrzej Duda. Markov chain fingerprinting to classify encrypted traffic. In *IEEE International Conference on Computer Communications*, 2014.
- [12] Gordon Lyon and Fotios Chantzis. Ncrack. <https://nmap.org/ncrack/>.
- [13] Joe Mondloch. Medusa. <http://foofus.net/goons/jmk/medusa/medusa.html>.
- [14] Maryam M Najafabadi, Taghi M Khoshgoftaar, Clifford Kemp, Naeem Seliya, and Richard Zuech. Machine learning for detecting brute force attacks at the network level. In *International Conference on Bioinformatics and Bioengineering*, 2014.
- [15] Vincent Nicomette, Mohamed Kaâniche, Eric Alata, and Matthieu Herrb. Set-up and deployment of a high-interaction honeypot: experiment and lessons learned. *Journal in Computer Virology*, 2011.
- [16] Daniel Ramsbrock, Robin Berthier, and Michel Cukier. Profiling attacker behavior following ssh compromises. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007.
- [17] Daniel Stenberg, Marc Hörsken, Viktor Szakats, and Will Cosgrove. libssh2. <https://www.libssh2.org/>.
- [18] Qixiang Sun, Daniel R Simon, Yi-Min Wang, Wilf Russell, Venkata N Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *IEEE Symposium on Security and Privacy*, 2002.
- [19] Alexander Vetterl and Richard Clayton. Bitter harvest: Systematically fingerprinting low- and medium-interaction honeypots at internet scale. In *USENIX Workshop on Offensive Technologies*, 2018.
- [20] T. Ylonen and C. Lonvick. The secure shell (ssh) transport layer protocol. Technical report, Internet Engineering Task Force, 2006.

Timing Patterns and Correlations in Spontaneous SCADA Traffic for Anomaly Detection

Chih-Yuan Lin
Linköping University, Sweden
chih-yuan.lin@liu.se

Simin Nadjm-Tehrani
Linköping University, Sweden
simin.nadjm-tehrani@liu.se

Abstract

Supervisory Control and Data Acquisition (SCADA) systems operate critical infrastructures in our modern society despite their vulnerability to attacks and misuse. There are several anomaly detection systems based on the cycles of polling mechanisms used in SCADA systems, but the feasibility of anomaly detection systems based on non-polling traffic, so called spontaneous events, is not well-studied. This paper presents a novel approach to modeling the timing characteristics of spontaneous events in an IEC-60870-5-104 network and exploits the model for anomaly detection. The system is tested with a dataset from a real power utility with injected timing effects from two attack scenarios. One attack causes timing anomalies due to persistent malfunctioning in the field devices, and the other generates intermittent anomalies caused by malware on the field devices, which is considered as stealthy. The detection accuracy and timing performance are promising for all the experiments with persistent anomalies. With intermittent anomalies, we found that our approach is effective for anomalies in low-volume traffic or attacks lasting over 1 hour.

1 Introduction

Supervisory Control and Data Acquisition (SCADA) systems are used for monitoring and controlling critical infrastructures such as waste water distribution facilities, gas production systems, and power stations. In the past decades, SCADA systems have increasingly adopted open protocols and connected to the Internet for improved flexibility and ease of use. These changes make SCADA systems vulnerable to cyber attacks, in particular with Advanced Persistent Threats (APT). Such attacks are usually slow and stealthy but ultimately lead to severe damage on physical devices.

One building block for protection of these systems is the implementation of anomaly detection. Most earlier works on the network-based anomaly detection exploit the cyclic traffic patterns found in the request-response communica-

tion mode [3, 14, 9]. That is, the SCADA master cyclically sends a request to field devices such as a Remote Terminate Unit (RTU) and a Programmable Logic Controller (PLC) and the field devices return monitored values after receiving a request. However, some SCADA protocols also allow non-requested communication whereby the field devices can report monitored values in spontaneous events without receiving any request. In a previously proposed anomaly detector [18], the authors observed low detection rates on an IEC-60870-5-104 (from now referred to as IEC-104) dataset because most of the communications were issued in non-requested mode. This motivates the search for modeling approaches that deal with spontaneous event sequences used for anomaly detection.

Anomaly detection is a technique that captures stable characteristics of spontaneous traffic and identifies unusual behaviours. In related research [17], Lin and Nadjm-Tehrani characterized the timing attributes of two emulated IEC-104 datasets. The datasets show a diverse set of possible timing behaviours and many of them exhibit event sequences varying in an irregular manner. However, process dynamics might imply two stable attributes over a longer period: (1) The probability distribution of event inter-arrival times shows clear peaks. That is, some inter-arrival times are more likely to be present than others, and (2) The probability distribution of inter-arrival times changes in groups of different data flows in the same system. The flows in the same group tend to change at approximately the same time. That is, there could exist positive correlation between the occurrence of spontaneous events in different flows.

In this work, we model the timing attributes of spontaneous traffic from a real power facility based on the above two hypotheses. We also propose an anomaly detector to detect anomalies within such traffic. The proposed detector is tested with two attack scenarios at the field device level. Our threat model is as follows. A field device such as RTU or PLC is an interface between the field sensors, actuators and SCADA master. Instead of directly breaking a field device, attackers may target disrupting the controlled process and/or

communication between the field device and SCADA master in a gradual manner. Our goal is to detect anomalies caused by the attacks before they cause irreparable damages on critical infrastructures. The contributions of this work are:

- Provide an empirical study on the timing attributes of spontaneous traffic from a real power facility. The study shows that inter-arrival times and correlation between flows are stable and there are persistent timing characteristics in spontaneous traffic.
- Present two attack scenarios producing timing anomalies in the spontaneous traffic and provide the corresponding anomaly generation approaches.
- Explore the potential of anomaly detection for IEC-104 spontaneous traffic by combining two methods for modeling event inter-arrival times and correlation between flows in the same network.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 provides the background knowledge needed for this work. Section 4 discusses the threat models and caused traffic anomalies. Section 5 describes the components of the proposed anomaly detection and evaluation system. Section 6 provides an overview of the datasets and the parameter settings. Section 7 reports the experimental methods and results. Section 8 concludes this study.

2 Related Work

Various anomaly detection models have been proposed for SCADA systems with different research methodologies. To better position our work, this section reviews related work in three orthogonal research domains: (1) network-based anomaly detection, (2) physic-based anomaly detection, and (3) content-based anomaly detection for SCADA systems.

2.1 Network-based Anomaly Detection

Network-based anomaly detection models target how the components in a SCADA system communicate to each other instead of looking into the payload. Many of the network-level detectors make use of the cycles of SCADA traffic with a focus on their cyclic sequences.

Timing-based anomaly detection usually employs statistical models and raises alarms when the monitored statistical parameters exceed certain thresholds. Valdes and Cheung [27] capture the attributes *bytes per packet* and *packet inter-arrival time* and compare the testing traffic against the historical records by T-test. Sayegh et al. [25] model the historical time intervals between signatures (i.e., a fixed sequence of packets) and Barbosa et al. [3] model the historical period of repeated messages in an orderless group. Lin et al. [18] focus on the inter-arrival time for periodic events and propose

a modeling approach based on the sampling distribution of mean and range.

Sequence-aware intrusion detection systems usually employ automata-based models. Goldenberg and Wool [9], Kleinmann et al. [15, 14], and Yang et al. [30] use Deterministic Finite Automata to model the message sequences of Modbus¹, Siemens S7², and IEC-104 traffic respectively. Yoon et al. [31] model the Modbus command sequences as Dynamic Bayesian Network.

Recently, more non-cyclic traffic has been found in studies with production data. Casselli et al. [5] propose a methodology to model the message sequences of Modbus, MMS³, and IEC-104 in Discrete-time Markov Chain. The authors observe no clear cyclic message patterns in the IEC-104 datasets. Faisal et al. [7] apply the Goldenberg and Wool's approach on a large dataset collected from a water facility in the U.S. and show that the model performs poorly because 72% of the flows did not exhibit clear cyclic patterns. Later, Markman et al. [21] propose a variant of the Goldenberg and Wool's model by considering the traffic on each flow as a series of bursts and create a burst-deterministic finite automata for each flow. This methodology is tested with the same data used in Faisal et al.'s study and 95% to 99% of the packets are successfully captured by this model.

Our work differentiates itself from the previous work by providing dedicated anomaly detection methodologies for spontaneous SCADA traffic while others focus on the cyclic attributes. In an earlier work by Lin and Nadjm-Tehrani [17], where they analyze the IEC-104 spontaneous traffic and show that the emulated traffic contains some stable timing attributes other than cyclic sequences and periodicity, no anomaly detection is proposed. In this work, we propose an anomaly detection system for IEC-104 spontaneous traffic and test the system with data collected from a real power facility.

2.2 Physics-based Anomaly Detection

Physics-based anomaly detection models the physical process managed by SCADA systems. Physics-based approaches model the process and detect the anomalies with high accuracy while network-based approaches model the network behaviour without knowing process semantics but aim to alarm the users before the attack has major impacts on the process. We consider these two approaches complementary not competing with each other. Giraldo et al. [8] presents a literature review on physics-based anomaly detection.

System identification techniques can model behaviours of a physical system by its inputs and outputs. Two models that

¹Modbus. <http://www.modbus.org/>

²A proprietary protocol used Siemens S7 series PLCs.

³Manufacturing Message Specification (MMS). <https://www.iso.org/standard/37079.html>

are widely used are the *Linear Dynamic State-space (LDS)* [26, 19] and the *Auto-Regressive (AR)* [11] models. These models may accurately predict the system behaviour, but they require a detailed description of the process and deep understanding of system that are not always available.

Machine learning approaches require less or no prior knowledge of the underlying process. Krotofil et al. [16] use the correlation entropy in clusters of related sensors to detect sensor signal manipulations. Kiss et al. [13] adopt the Gaussian mixture model to form sensor clusters. Aoudi et al. [2] propose a departure-based detection system that measures the distance between the normal signals and the signals under attack projected to a subspace. These works show that sensors in SCADA systems are intricately correlated. Since the sensor values and spontaneous events have a cause-effect relationship, these works explain and support our hypothesis that spontaneous traffic from different flows can be correlated.

2.3 Content-based Anomaly Detection

Content-based anomaly detection based on in-depth analysis of packet contents is an important research topic for general-purpose networks. Due to the use of proprietary protocols and lack of availability of specifications, content-based anomaly detection for SCADA systems is still in its infancy. Düssel et al. [6] present an anomaly detection system based on n -grams using distance metrics. Hadžiosmanović et al. [10] investigate four anomaly detection algorithms, POSEIDON, Anagram, PAYL, and McPAD, all using n -gram analysis for message payloads. The authors conclude that there is no absolute best algorithm among the tested methods in terms of detection rates and false positive rates. Wressneger et al. [29] propose an anomaly detection system based on n -grams with a special focus on proprietary binary protocols. The authors show that the content-based approach is applicable to binary protocols with high-entropy data.

3 Preliminaries

This section briefly presents how an IEC-104-compatible RTU works and generates spontaneous events, together with important fields of IEC-104 packets. The section also provides a short review on spontaneous traffic attributes found in a previous study.

3.1 IEC-60870-5-104

The IEC-104 protocol applies to modern SCADA systems for monitoring and controlling geographically widespread processes. It enables communication between a master (control station) and one or more slaves (substations) via a standard TCP/IP network. IEC-104-compatible RTUs or PLCs store inputs from the controlled process in its own storage

which is indexed by Information Object Addresses (IOA). In order to improve the communication efficiency, IEC-104-compatible RTUs scan monitored data in certain IOAs with a fixed rate and generate spontaneous events when the monitored data have changed.

Every spontaneous packet contains two important fields in addition to IOA. First, Cause Of Transmission (COT) specifies whether it is a spontaneous packet or other type of packet such as a reply on interrogation or a cyclic data report. Second, Type IDentification (TID) specifies the type of the monitored data. The most common data type is Monitored MEasured point, normalized value (M_ME_NA) or its variant (M_ME_TA and M_ME_TD). This datatype is 16-bit long and contains a measured value from a certain IOA. The system administrator needs to define an acceptable range for each measured point. The RTU sends out a spontaneous event when the measured value falls outside the range. Monitored Single Point (M_SP_NA) is 1-bit long and Monitored Double Point (M_DP_NA) is 2-bit long representing the state of a switch or circuit breakers⁴. A RTU sends out spontaneous events if the value of these two datatypes changes.

3.2 Known Spontaneous Traffic Attributes

This work is motivated by earlier results [17] which show that the IEC-104 spontaneous traffic contains some timing attributes that could last for a long time. Our notion of flow is based on a sequence of timed events with a given IEC-104 type, from a given RTU, based on values stored in a given IOA. There are two attributes explored this study. (1) Inter-arrival time attribute: assuming that the spontaneous traffic has limited groups of event inter-arrival times, and (2) correlation attribute: assuming that positive correlation exists between the changes on inter-arrival times in different flows.

To illustrate these attributes, we present the partial characterization results of two flows, named IOA_10091 and IOA_10092, which are initiated from the same RTU but different IOAs in a virtual SCADA testbed, RICS-el, developed in a national research project [1]. The flows are from a 12 day collection and separated into 2-hour long segments for characterization. We choose these flows because they show strong characteristics, but the other flows in the same collection also contain similar attributes.

Inter-arrival time attribute. Figure 1 (a) presents the histogram of event inter-arrival times within a segment of two hours from IOA_10091. There exists clear peaks with high frequency and several groups of inter-arrival times with low frequency in the collected data over two hours. The analysis of the collected data shows that few of inter-arrival intervals fall outside these groups in the subsequent 12 days.

⁴IEC 60870-5-104 summary.

<https://spinengenharria.com.br/en/biblioteca/protocolo-iec-60870-5-104-client/download/>

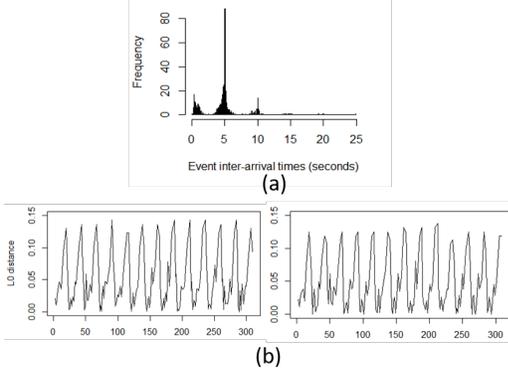


Figure 1: Traffic characteristics found in the collected data. (a) Histogram of spontaneous event inter-arrival times from IOA_10091. (b) L0 distances over time (hours) for two spontaneous traffic flows from the same RTU but different IOAs [17].

Correlation attribute. Continuing with the above example, we get several groups (i.e., peaks) of inter-arrival times in the first segment of IOA_10091 and IOA_10092. The empirical probability of each group can be easily calculated as number of observations in the group divided by number of observations in the segment. Figure 1 (b) shows the changes in the probability distribution of each segment over 12 days. The x axis gives the relative time in every two hours (a segment). The y axis gives the distance of probability distribution between the first segment and the later ones. The distance is defined as

$$D = \sum_{\sigma_i^1 = \sigma_i^2} (p(\sigma_i^1) - p(\sigma_i^2))^2 + \sum_{\sigma_i^1 \neq \sigma_i^2} p(\sigma_i^k)^2 \quad (1)$$

where σ_i is the i^{th} group of inter-arrival times in the set of identified groups, and σ_i^k means the i^{th} group in distribution k , $k = 1$ or 2 . $k = 1$ means the distribution of the first two hours and $k = 2$ means another distribution in comparison. $p(\sigma_i^k)$ thus means the empirical probability of the group. Previous work does not calculate any type of correlation coefficients for the datasets. It only categorizes the traffic *for each flow* by observing changes of L0 distances over time.

In Section 5.1 we will enrich the modeling of the spontaneous traffic by exploiting both attributes. For the sake of simplicity, we refer to the model based on the first attribute as *inter-arrival time model* and the second one as *correlation model* in the rest of the paper.

4 Threat Model

This section presents two types of attacks on the field device level and the possible traffic anomalies caused by the attacks.

This provides an intuitive view on the need for anomaly detection.

In order to disrupt the controlled process and communication, an attacker can (1) take control of other components and launch attacks such as Denial-of-Service (DoS) to the targeted field devices or (2) exploit code-integrity vulnerabilities in the targeted field devices and run malware on them.

Attack against field devices. Legacy SCADA field devices have limited computing capabilities so that they are fragile and sensitive to network traffic increment [4, 28]. Niedermaier et al. [24] showed that most of the PLCs from major vendors are vulnerable to packet flooding in different network levels and even scanning for benign purposes. The attacker can influence the PLC cycle times and the controlled processes. With regards to the network anomalies, this type of attacks causes network performance degradation due to competition on resources such as CPU and I/O port of the devices. Long et al. [20] model the network performance under DoS attack with two queuing models but no real attacks and devices are involved. Other authors [23, 12] simulated the impact of different DoS attacks on SCADA networks. Both simulations showed performance impacts including network delays, packet drops, and unavailability of targeted devices.

Malware on field devices. Many SCADA field devices were designed without security considerations, and potential vulnerabilities allowing remote code execution on RTUs have been documented officially (e.g., CVE-2017-12738, CVE-2018-10605). Therefore, an attacker would gain remote access to field devices and modify their software to launch an attack against the critical infrastructure. Malicious changes on the controlled process can possibly be observed by an operator from abnormal number of alarms, values of measurements, amount of traffic, etc. In order to hide the malicious activity, the malware usually suppresses the real outbound packets and sends altered packets to the SCADA master. Stuxnet⁵ and Irongate⁶ capture normal outbound values and replay it to mask anomalies produced while launching attack to the controlled process. In this work we propose a more stealthy scenario wherein the replayed traffic not only contains the historical measurements but also follows historical timings to send. With regards to the anomalies, the value of the payload, size of messages, and timings are totally legitimate. However, SCADA traffic usually contains phase transitions or seasonal changes. This makes it detectable (after a period of time) by comparing the similarities between field devices. A slow detection is acceptable because this kind of malware is usually a part of an APT, which is designed to be effective for an extended period of time.

The two proposed attack scenarios will be used to generate relevant test cases for evaluation in Section 7.

⁵https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

⁶https://www.fireeye.com/blog/threat-research/2016/06/irongate_ics_malware.html

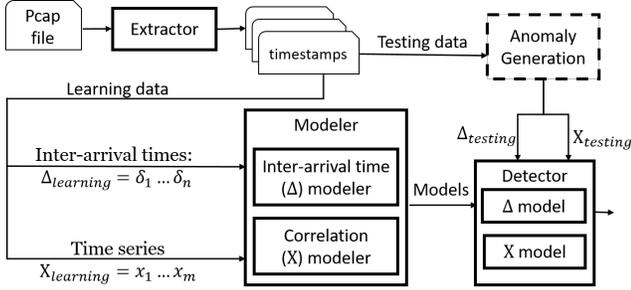


Figure 2: System components and workflow

5 Proposed Anomaly Detection System

The proposed system contains three main modules as illustrated in Figure 2. First, an *Extractor* software that can be run in a operation site, e.g., a utility company, for collecting the input traces to the anomaly detection process. The *Extractor* records the timestamps of packets containing spontaneous events (COT=spont) and separates the timestamps in flows which are defined by a three-tuple {RTU, TID, IOA}. The timestamp sequences in each flow are then used as learning data and later sequences collected in a similar way can be used as testing data.

Second, for each flow from the first step the learning data is transformed into two data types as the inputs for the *Modeler* to build models. For the *inter-arrival time model*, we form a sequence of inter-arrival times $\Delta_{learning}$, and denote each inter-arrival time appearing in the sequence by δ_i . For the *correlation model*, we form a time series $X_{learning}$, and x_i denotes the number of spontaneous events in i^{th} bin. The bin size is a configurable parameter, and will be 1 minute in this work.

Third, the models are sent to the *Detector*. The testing data in the same format as the learning data will be transformed into $\Delta_{testing}$ and $X_{testing}$ and used to test our *Detector*.

In our evaluation of the method, as we are creating synthetic attacks to test the detector, we also have a separate Anomaly Generation function (dashed box) which will inject suitable anomalies in the collected test data to create test sets which are a combination of real utility data and attack-induced variations of them.

The *Extractor* is written in Python and the other modules are developed and run in R.

5.1 Modeling Spontaneous Events

This section presents how we construct the *inter-arrival time model* and the *correlation model*.

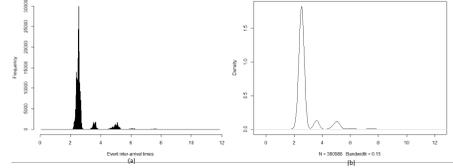


Figure 3: Distribution of inter-arrival times from a sequence within an event set in our data: (a) Histogram of $\delta_i \leq 12$ seconds. (b) The smoothed version of the sequence, bandwidth=0.15.

5.1.1 Inter-arrival time model

The process of building the *inter-arrival time model* contains three steps: smoothing, grouping, and finding boundaries. The first two steps are developed in an earlier work [17] with modifications on choice of parameters (bandwidth, size of *GroupList*). This section includes these two steps for the sake of completeness.

Smoothing. This step uses kernel density function to smooth out the discontinuities in the distribution of $\Delta_{learning}$. The smoothed distribution of inter-arrival times is called $\Delta_{smoothed}$. Figure 3 shows part of the frequency distribution of inter-arrival times which is less than 12 seconds and its corresponding smoothing results. The bandwidth parameter of the kernel decides the smoothness level of $\Delta_{smoothed}$. The parameter can be an arbitrary large number as long as the peaks are still higher than the bottoms in the resulting $\Delta_{smoothed}$.

Grouping. The next step starts with finding the relative low points in $\Delta_{smoothed}$ with Algorithm 1. The algorithm outputs a *GroupList* which contains a list of low points in pairs. The inter-arrival times within a pair of low points are considered as in the same group.

Finding boundaries. The last step finds the best fitting boundaries of each group. These boundaries are used for detection later. There are two methodologies used for finding the best fitting boundaries. (Section 6 compares the two methodologies.) (1) The best-fitting boundaries can be set as simply percentile of 99.99% and 0.01% of observations in each group without any assumptions on their distribution. (2) With a Gaussian distribution assumption, the module calculates the mean and standard deviation of the observations in each group and sets the boundaries as mean plus/minus three standard deviations.

In both of the methodologies, the module needs to estimate the boundaries of groups having too few observations in a specialized way. Without a specific assumption of distribution, the module takes the range between boundaries in the largest group of the same flow and centers the range with the median of the current group. With a Gaussian distribution assumption, the module calculates only the mean and reuses the standard deviations in the largest group of the same flow.

Algorithm 1: Finding low points

```
Input :  $\Delta_{smoothed}$ 
Output: GroupList
1 GroupList  $\leftarrow$  empty // list for output
2 while true do
3   peak  $\leftarrow$  HighestPoint( $\Delta_{smoothed}$ );
4    $L = R = \textit{peak}$ ;
5   while  $\Delta_{smoothed}[R + 1] < \Delta_{smoothed}[R]$  do
6      $R \leftarrow R + 1$ ;
7   end
8   while  $\Delta_{smoothed}[L - 1] < \Delta_{smoothed}[L]$  do
9      $L \leftarrow L - 1$ ;
10  end
11  if ( $L == R$ ) then
12    break;
13  end
14  GroupList[i]  $\leftarrow$  ( $L, R$ );
15   $\Delta_{smoothed} \leftarrow \Delta_{smoothed} - \textit{PointsBetween}(L, R)$ ;
16 end
```

The criteria for adopting this specialized estimation needs to be decided experimentally (See Appendix A).

5.1.2 Correlation model

The process of building the *correlation model* contains three steps: correlation pairing, phase separation, and finding boundaries. We aim to find a monotonic relation between the time-series and pair them. The two paired time-series change (increase or decrease) with the same direction but not always at the same rate. As observed in the earlier work [17, 22, 9], SCADA traffic exhibits phases, and the timing characteristics of the traffic may be different in each phase. We assume correlated time-series are in a monotonic relation instead of linear relation because of phase transitions.

Correlation pairing. This step starts with calculating the Spearman rank correlation matrix between event volume time-series of different flows. Spearman correlation measures the strength and direction of two variables belonging to a monotonic relation. For two time-series $X^p = x_1^p, \dots, x_m^p$ and $X^q = x_1^q, \dots, x_m^q$, the Spearman rank correlation is ⁷:

$$\rho_{pq} = \frac{COV(R(X^p), R(X^q))}{\sigma_{R(X^p)} \sigma_{R(X^q)}} \quad (2)$$

where $R(X^k)$ denotes the ranked time-series $R(x_1^k), \dots, R(x_m^k)$. In a ranked time-series the numerical value in each bin x_i^k is replaced by their rank in the sorting. In addition, $COV(R(X^p), R(X^q))$ is the covariance of ranked time series and $\sigma_{R(X^p)}$ and $\sigma_{R(X^q)}$ are the standard deviations.

⁷Spearman, C. The Proof and Measurement of Association Between Two Things. American Journal of Psychology (1904).

In the calculated correlation matrix, for any row representing a flow in the system, the column with the highest coefficient thereby gives the most correlated flow and pairs with it. Note that the pairs are not always symmetric. It can be the case that time-series X^p is most correlated with X^q and X^q is most correlated with another time-series.

Phase separation. This step separates the time-series into phases by assuming the relation between two paired time-series X^p and X^q at time point i and phase k follows:

$$x_{ki}^p = x_{ki}^q + \epsilon_k^{pq}, \quad \epsilon_k^{pq} \sim N(\mu_{pqk}, \sigma_{pqk}^2) \quad (3)$$

where ϵ_k^{pq} is a constant with added noise. ϵ_k^{pq} is independent from x_{ki}^p and follows Gaussian distribution. That is, the difference between x_{ki}^p and x_{ki}^q should fall within the range of ϵ_k^{pq} .

The process of separation starts by observation of the histogram of D^{pq} , the arithmetic difference between two paired time-series X^p and X^q . If the histogram is not a bell curve, we consider it as a Gaussian Mixture Model (GMM). Then, we use an Expectation–Maximization (EM) algorithm ⁸ implemented in the R library to separate the GMM into a few Gaussian models. Each Gaussian model is called a component and represents a phase. This algorithm gives each d_i^{pq} a posterior probability of belonging to one of these components. We can assign d_i^{pq} , x_i^p and x_i^q into the most likely component at every time point i . The set of datapoints in component k is called X_k^p and X_k^q .

Finding boundaries. This step removes the outliers in X_k^p and X_k^q , sets entering conditions for each phase, and finds the estimated boundaries of ϵ_k^{pq} for detection. Fluctuations in the volume of each bin from the same phase causes outliers in the probability distribution of X_k^p and X_k^q to be generated in the previous step (See Figure 4). Our model excludes the outliers by adjusting entering conditions (cut-off lines) of phase k (e.g., $x_i^p \in X_k^p$, $k = 1$, if $7 > x_i^p > 3$).

The module calculates the boundaries of ϵ_k^{pq} with the adjusted results X_k^p and X_k^q . Let D_k^{pq} as the arithmetic difference between X_k^p and X_k^q for each k . The estimated boundaries of ϵ_k^{pq} is set as the mean plus/minus three standard deviations of D_k^{pq} . The example calculation of boundary settings with the used datasets are illustrated in detail in section 6.

5.2 Detection Mechanism

We use a two-layer approach to generate alarms for unusual network behaviours. In the first layer, the input traffic is pre-processed into two data types $\Delta_{testing}$ for the inter-arrival time model and $X_{testing}$ for the correlation model. The detector generates an inter-arrival time warning if an inter-arrival time falls outside the boundaries of the learned groups and generates a correlation warning if the difference of two paired

⁸NormalmixEM in library "mixtools". <https://cran.r-project.org/web/packages/mixtools/mixtools.pdf>

Dataset	TID	# Flows	# Used Flows	AEF
RTU A	M_ME_NA	21	19	228780
	M_DP_TB	8	0	13
	M_SP_TB	3	0	2
RTU B	M_ME_NA	16	14	7837

Table 1: Overview of datasets during 30 days. Here AEF stands for Average Number of events per Flow.

time-series falls outside the boundaries of the identified current phase in the pair.

In the second layer, the detector calculates the number of inter-arrival time warnings for each RTU and correlation warnings in every minute to form a multivariate time-series. The system uses sliding windows $W_{\Delta RTU_i}$ and W_X on the time-series to filter out noise. The detector generates alarms when the number of warnings exceeds the predefined second-layer thresholds $T_{\Delta RTU_i}$ or T_X . Every RTU in the network has its own detection parameters $W_{\Delta RTU_i}$ and $T_{\Delta RTU_i}$, and we refer to it as a *RTU_i* component of the inter-arrival time model. As usual, the configurable parameters of a detection mechanism are a means to set a desired level of accuracy or false positive rate, as will be evident in Section 7.3. The alarms will be sent to the operator monitoring the system but the warnings are internal elements of the analysis used to set thresholds.

6 Datasets and Parameter settings

This section presents an overview of the datasets and the parameter settings for model construction. It elaborates which parts of our data are used for the further experiments and how they are used.

The experiments employ network traces collected from a real-world power facility. The data collection was performed by the utility personnel embedding the *Extractor* component in Figure 2 in their facility and providing the event inter-arrival times for spontaneous events for our evaluation. The network uses multiple protocols and we collect only the timings of communications between the SCADA master and 2 IEC-104 specific RTUs. The datasets have a duration of 30 days and we choose learning data to be 300-hour long (i.e., about 12 days). Table 1 presents a brief summary of the datasets we used in our experiments.

Our datasets contain different types of measurements (TID) and flows. The flows with small event quantities (less than 200 events) are outside our scope because these flows, such as M_DP_TB, M_SP_TB and unused M_ME_NA traffic, apparently have very different timing characteristics and should be monitored using alternative methods. Consequently, all of the used 33 flows are from M_ME_NA measurement traffic. Our datasets also show different event rates in each RTU. RTU A has a much higher event rate than RTU

B.

6.1 Parameter and design choices for the inter-arrival time model

The top 5 flows with the highest event frequencies from RTU A and the top 3 flows from RTU B are used for building the models. This is because the slow flows contain less observations for each group. Therefore, there are more groups in the slow flows requiring specialized estimation for the boundaries as mentioned in Section 5.1.1. We exclude these flows to focus the feasibility study of our approach on the more frequently populated flows.

Table 2 shows the learning results in terms of warning rates for the selected flows. It explains the methodology to find the best-fitting boundaries for detection with limited length of learning data. Warning rate is defined as the number of inter-arrival times which can not be categorized in any group divided by the amount of inter-arrival times in the whole learning data in percentage (%). Warning rate I is calculated directly after the *Grouping step* (the second step of the inter-arrival time model construction). The results show that the initial grouping covers most of the data points by the chosen bandwidth parameter for algorithm 1.

There are two alternative methodologies to find the best fitting boundaries for detection. Warning II and warning III are calculated after completing the *Finding boundaries step*. In warning II cases we find best fitting boundaries without any specific assumption on their distribution (i.e., the first methodology for finding boundaries in Section 5.1.1). We can observe that this methodology leads to high warning rates for the flows from the low-event-rate RTU (RTU B). In warning rate III cases we find boundaries with Gaussian distribution assumption (i.e., the second methodology). We note that the two methods are suitable in different cases. The warning II method works better when there are sufficient number of observations, and the second method (Gaussian, warning III) performs best with learning data with fewer observations. Therefore, the boundaries estimated with the first methodology are used for RTU A component and the boundaries estimated with the second methodology are used for RTU B. The detailed warning rates for the whole dataset will be presented in Table 4 of Appendix A.

6.2 Parameter and design choices for the correlation model

For the correlation model, we form 33 pairs from the used 33 flows through the correlation pairing process described in Section 5.1.2. 3 out of 33 pairs are a mixture of two distributions and require a phase separation process. These pairs contain phase transitions which is the desired feature as mentioned in Section 4. They are included for further experiments no matter how many warnings they have produced in

Flow	Warning rate I	Warning rate II	Warning rate III
A_3019	≈ 0	0.03	0.46
A_3014	0	0.02	0.59
A_3013	≈ 0	0.02	0.52
A_3012	≈ 0	0.02	0.46
A_3020	≈ 0	0.02	0.49
B_3019	0.10	14.88	0.54
B_3016	0.10	14.86	0.46
B_3006	0.08	0.22	1.48

Table 2: Overview of selected datasets and learning results. Warning rates are presented in %.

the learning period. 20 out of the remaining 30 pairs have warning rates below 1 % for the learning data and are included for further experiments as well. It is apparent that the pairs having high warning rates are not tightly correlated. These flows are thus useless to include in our correlation-based learning. Table 4 of Appendix A lists all the pairs and their warning rates for the learning data.

Figure 4 (a) presents an example of two paired time-series containing phases. The two time-series show clear high and low values with somewhat visible correlation. Figure 4 (c) is the histogram of difference between the two time-series and indicates it may be a mixture model of two components. The EM-algorithm results suggest that it is a mixture of component 1 $\sim N(-6.25, 8.20)$ and component 2 $\sim N(-0.31, 1.08)$, where $\sim N(\mu, \sigma)$ denotes a Gaussian distribution with mean μ and standard deviation σ . The proportion of these two components are 0.62 and 0.38 respectively. We separate the original time-series into two groups according to the posterior probabilities. Figure 4 (d) is the histogram of RTUA_3018 dataset that are in the component 2 (high-value phase). Since we model the distribution with two components, the boundary of the high-value phase is simply set at one-side as presented by the red line at 18. Figure 4 (e) is the histogram of the difference between RTUA_3018 and RTUB_3015 when the value of RTUA_3018 is higher than 18. The learned boundaries of differences for this pair in the high-value phase are -7 and 10 and the boundaries in the low-value phase are -27 and 6.

This example shows that the replay attacks may break the correlation between some paired time-series if one flow replays the historical data but another one has changed to another phase. It indicates our hypothesis for the approach being suitable for detecting replay attacks is likely to work.

7 Evaluation

We evaluate the proposed anomaly detector in two steps. First we study whether the testing data (with no attacks injected) exhibits signs of pre-existing anomalies. Then we create synthetic anomalies according to the threat models

presented in Section 4 and show the performance of the detector against the synthetic anomalies. The performance is evaluated with detection accuracy and time-to-detection.

7.1 Warnings and Actual Anomalies

Figure 5 shows the warnings generated by each model on the collected data from the utility for testing over time. The inter-arrival time model generates warnings at a relatively stable rate compared to the correlation model. The correlation model shows an anomalous burst of warnings around 40000 minutes and it lasts for thousand minutes till the end of the data. These anomalies are generated mostly from a correlation pair as shown in figure 6. As a result of discussions with the utility company we found that the correlation break is caused due to a manual intervention by the operator at the company, where a line breaker was used to turn off the current on a line at 8:53, perform a redirection, including connecting the ground, and restarted the transmission on the original line at 9:44. The normally correlated line was not touched, so the anomaly detected was grounded in reality. The detailed warning rates for each flow can be found in Table 5 of Appendix B.

Due to the existence of actual anomalies, we exclude the data after 39750 minutes for the following evaluation of the anomaly detector in the step where injected anomalies are inserted in the dataset. The authors have noticed that there are still some short bursts of warnings in the used segment of the dataset and these bursts may be caused by manual operations on the system or traffic noise. Since the log of operations is not available for our study, we decide to treat them as normality.

7.2 Synthetic Anomaly Generation

We create two types of synthetic anomalies according to the threat models presented in Section 4. We refer to the anomalies caused by the attack against field devices as performance downgrade anomalies and the one caused by the malware on field devices as replay anomalies. For each type of synthetic anomaly, we conduct several experiments on each RTU with different severity/scale. An experiment will be repeated 25 times with randomly generated anomalies as explained below.

7.2.1 Performance Downgrade Anomalies

In this scenario we emulate cases where attacks lead to performance downgrade. This is done through an anomaly generator that adds delays to and drops packets from the RTU traffic. The anomaly generator is implemented to emulate the impacts of such attacks with a queuing model as $g(L, \mu, \phi, \psi)$, where g represents the queue model with queue length L , μ is the mean service rate (events/second),

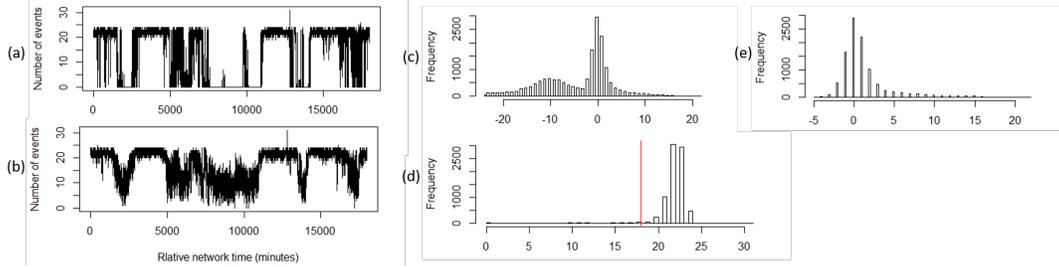


Figure 4: An example pair of time-series. (a) Time-series from flow {RTUA, M_ME_NA, 3018}, called RTUA_3018. (b) Time-series from flow {RTUA, M_ME_NA, 3015}, called RTUA_3015. (c) The histogram of difference between RTUA_3018 and RTUA_3015. (d) The result of running EM-algorithm: the histogram of RTUA_3018 dataset that are in the high-value phase. (e) The histogram of difference between RTUA_3018 and RTUA_3015 when the value of RTUA_3018 is higher than 18.

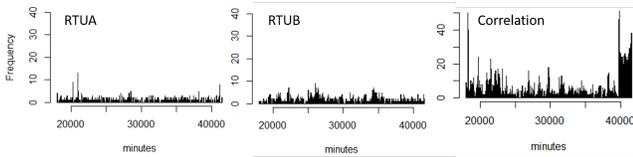


Figure 5: Number of warnings generated by component RTU A of the inter-arrival time model, component RTU B of the inter-arrival time model, and the correlation model in every 24 minutes

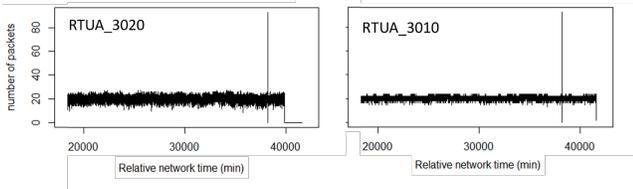


Figure 6: A pair of time-series with correlation breaks.

ϕ is the malicious traffic rate, and ψ is the original traffic randomly selected in our testing data.

To create the queuing model we first abstract the original network packets with their arrival times and number of events (a packet may contain multiple number of events). An event occupies one unit of queue length L . Then, we select a malicious packet rate ϕ which represents the severity of attacks. We can escalate the severity of attacks by increasing ϕ . The original and malicious traffic compete for some shared resources such as CPU time, network card bandwidth, etc. The queue and the service module model the shared resources. The synthetic arrival time of an event is the sum of its original time and waiting time. Service time is only used to model the normal congestion in the queue but not included in the calculation of synthetic arrival times. This is set to prevent generating anomalies when the malicious traffic rate is

0.

There are three configurable parameters in the model, ϕ, μ, L . The parameters for experiments are tuned to match the results of simulation of DoS attacks on real RTUs [12] where an RTU shows little delay when ϕ is 100 packets per second and can barely send traffic at the rate of 580 packets per second. Therefore, μ is set as 82.64 to create congestion in the queue when ϕ is 100. We assume the queue becomes full of malicious packets after a period of time $t = 180$ seconds (corresponding to the attack rate $\phi = 580$). The length of the queue is thus set as $L = (\phi - \mu) * t \cong 89524$.

We conduct an experiment on each RTU for ϕ equals to 100 and 200 respectively. This leads to four experiments. The experiments with ϕ equals to 100 are intended to model the case where resource claims only create delays. They will be used to show the effectiveness of delay detection. The experiments with ϕ equals to 200 model a case where the resource exhaustion creates both delays and packet loss. They will be used to show the timing efficiency of the delay detection (i.e., the detector can find delays before packet losses.).

The total data set was used as follows: The 18 days remaining after removing the 12 days learning time was further reduced to remove the 3 days that included the "benign" anomalies as described in Section 7.1 that appeared at the end of the period. This leaves us with a 15 day interval as testing data. From these 15 days we extract 40-minute long samples at random as the inputs (ψ) of the anomaly generator and then replace the 40-minute long samples in the testing data with the resulting traffic of the queuing model.

7.2.2 Replay Anomalies

Next, we create a threat scenario that the malware repeats certain daily activities and gradually makes damages on the controlled process. It records a 20-second⁹ traffic sequence before it starts the malicious activity and replaces the out-

⁹Stuxnet records process values for 21 seconds. Irongate records 5 seconds.

bound traffic with forged traffic while running the activities. The forged traffic replays the values in the recorded and follows the recorded inter-arrival times. We expect that an evasive attacker’s forged traffic can bypass the monitor of inter-arrival time component since it replays the regular timings. However, the traffic may cause correlation breaks for the correlated flows from different RTUs.

We conduct one experiment for each activity duration equal to 10 hours, 1 hour, 30 minutes and 5 minutes respectively. This will be done once for RTU A and once for RTU B, leading to 8 experiments. From the 15-day testing data we use the first day to extract the 20-second intervals for the replay action at random, and selected the 10 hour, 1 hour, 30 minutes, and 5 minutes intervals for the 8 replay experiments from the remaining 14 days at random.

7.3 Detection accuracy

This section summarizes the resulting detection accuracy for the experiments presented in Section 7.2.1 (performance downgrade anomalies) and 7.2.2 (replay anomalies). The section first discusses the impact of detection parameters using AUC (Area Under the Curve) and ROC (Receiver Operating Characteristics) curve. During the calculation of ROC, True Positive Rate (TPR) is defined as the number of true alarms divided by the number of windows with inserted events, and False Positive Rate (FPR) is defined as the number of false alarms divided by the number of windows without inserted events. Detection rate is defined as the number of detected event insertions divided by the number of inserted events. Our evaluation is based on the assumption that one type of attack is active at any interval of time. Thereby we evaluate reactions to each type of attack separately with a set of experiments.

There are two types of tunable detection parameters: window size ($W_{\Delta RTU_i}$, W_X) and detection threshold ($T_{\Delta RTU_i}$, T_X). Figure 7 presents the detection ability for each model and the corresponding experiments in terms of AUC ranging from 0.5 to 1, where 1 is the best value with highest detection and lowest false positives. We execute our experiments 25 times with a given window size from 10 to 100 minutes (shown on the x axis) and present the median of the 25 AUCs. The results show that the inter-arrival time model has AUCs close to 1 for detection on all types of downgrade anomalies and with all the window sizes. In contrast, the correlation model has a wide range of AUCs from slightly higher than 0.5 to 1, where the duration of malicious activity is varies from 5 to 600 (minutes).

For the correlation model and the replay experiments, there are three groups of results. The leading group includes the experiments on RTU B with activity duration of 600, 60, and 30 minutes and the experiment on RTU A with activity duration of 600 minutes and 60 minutes. The middle group contains only the experiment on RTU B with activity dura-

Anomaly Type	Scale	Location	Detection Rate	FPR
Performance Downgrade	100	RTU A	100	0.1
	100	RTU B	100	0.5
	200	RTU A	100	0.1
	200	RTU B	100	0.5
Replay	600	RTU A	61.3	1.2
	600	RTU B	69.4	1.2
	60	RTU A	76.5	1.1
	60	RTU B	80.0	1.0
	30	RTU A	61.4	1.1
	30	RTU B	69.3	1.0
	5	RTU A	44.5	1.1
	5	RTU B	52.9	1.0

Table 3: Detection accuracy (%) with FPR around 1 %

tion of 5 minutes. The remaining experiments are in the bottom group. The leading and middle group show higher AUCs with small window size, and the AUCs decrease as the window sizes grow. The bottom group has AUCs just above 0.5 with small window size. Though the AUCs grow as the window sizes increase, we speculate this is just because some false positives become true positives in an increased window size (i.e., some inserted events are included in the window). Since this would give an overly positive picture, we suggest a small window size for anomaly detection.

Figure 8 shows an example ROC curve of the inter-arrival time model detecting downgrade anomalies on RTU A with a fixed window size 100 (minutes). The detection thresholds decide the cut-off line. The configuration of detection thresholds depends on the acceptable FPR. If considering an acceptable FPR as 1%, which means 1 false alarm per hundred minutes, we can set the detection parameters as $W_{\Delta RTUA} = 30$, $W_{\Delta RTUB} = 30$, $W_X = 10$ and $T_{\Delta RTUA} = 11$, $T_{\Delta RTUB} = 7$, $T_X = 10$. Table 3 presents the average detection accuracy (%) with these settings.

The inter-arrival time model providing 100% detection rates and false positive rates below 0.5%. The correlation model has detection rates between 61% to 80% for the leading group of experiments. This is satisfactory because these attacks are expected to be hard to detect as mentioned in section 4. On the other hand, the correlation model shows lower detection rates for the remaining experiments which have low AUCs. The detection rates are between 44% to 61%. This implies that the correlation technique applied here would not be an effective technique for finding anomalies that occur in high volume traffic with short period of activities.

7.4 Timing Performance

This section presents the timing performance for the experiments presented in Section 7.2.1 (performance downgrade anomalies) and 7.2.2 (replay anomalies) in TTD (Time To Detection). The TTD measurement is based on time-series bins (minutes). It is defined as the time on which the IDS

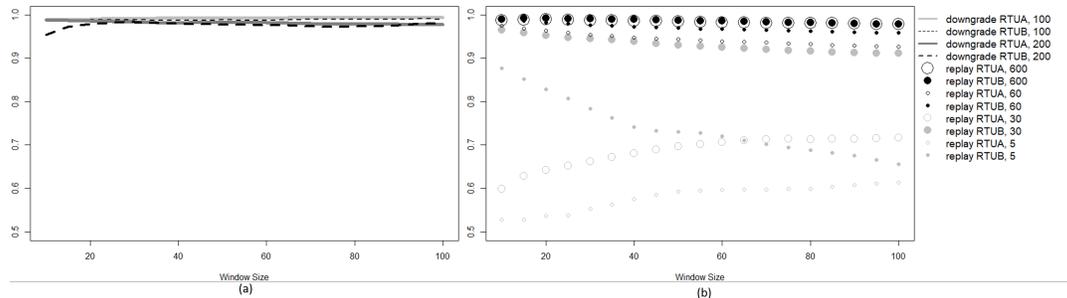


Figure 7: The median of AUCs for: (a) performance downgrade experiments, (b) replay experiments.

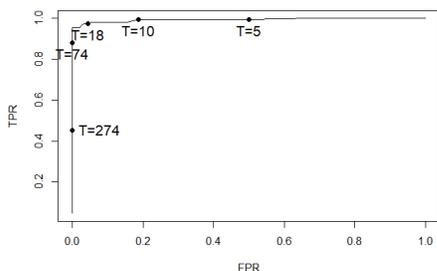


Figure 8: ROC curve of the inter-arrival time model on its first run of experiment performance downgrade RTU A, $\phi=100$. T means threshold T_{RTUA}

raises its first alarm minus the time of the first event insertion. The results of the measurements are shown in a boxplot with median, quantile 25% and 75 %.

7.4.1 Performance Downgrade Experiments

The TTD in each downgrade experiment is compared with the time to the first packet drop in the experiment with $\phi=200$. If our approach can detect anomalies before the first packet drop, it outperforms the general-purpose network monitoring tools which identifying and monitoring packet loss. As mentioned in Section 7.2.1, we abstract the original network packets with their arrival times and number of events and emulate the performance degradation with a queuing model. The experiments with ϕ equals to 200 model a case where the resource exhaustion creates both delays and packet loss. If the i^{th} packet in the original traffic Pkt^i is the first packet being dropped in the emulation results, the time to first packet drop is calculated as $T(Pkt^{i-1}) + \delta_{i,i-1}$, where $T(Pkt^{i-1})$ denotes the delayed time of Pkt^{i-1} and $\delta_{i,i-1}$ denotes the inter-arrival time between the Pkt^{i-1} and Pkt^i in the original traffic.

Figure 9 presents the timing performance for performance downgrade experiments. The time to the first packet drop measurements are centered around 30 minutes for both of the

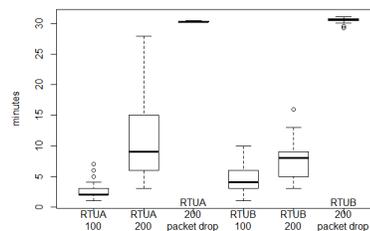


Figure 9: Comparison between time to detection and time to first drop

RTUs. The median of TTD measurements for each experiment from left to right is 2, 9, 4, and 8 minutes. Every iteration of all the experiments has TDD less than 30 minutes. It is worth noting that TTD measurements become longer with higher severity of attack. This is because the attacks with a higher input rate (ϕ) to the queuing model cause more delays between two events such that the number of events and alarms in one minute (bin) decreases.

7.4.2 Replay Experiments

Figure 10 presents the timing performance for replay experiments. In some of the iterations for some experiments there is no detection. Figure 10 (a) presents the TTD measurements for the iterations with detection. The median of TDD for each experiment from left to right is respectively 4, 5.5, 7, 9, 3.5, 4, 4 and 4 minutes. This shows, for most of the cases, our approach detects the anomalies in a short time if they are detectable.

Figure 10 (b) shows the number of iterations without detection for all the replay experiments. The number for each experiment from left to right is 13, 7, 4, 1, 9, 5, 0, and 0. This shows, in a given dataset, the recorded traffic characteristics, the traffic characteristics during the replay, and the length of replay period influence the probability of occurrences of detectable anomalies. Among them, the length of replay period

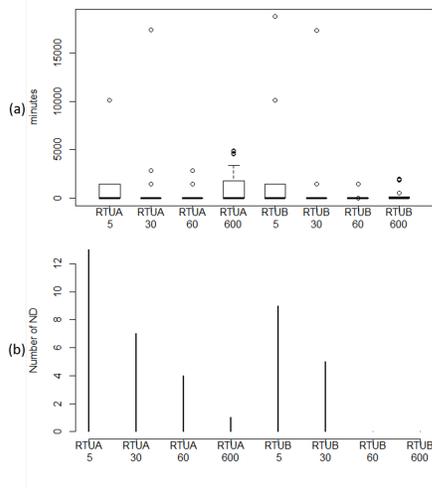


Figure 10: Timing performance for all replay experiments. (a) Time to detection measurements. (b) Number of iteration without detection, where ND denotes no detection.

plays a decisive role in our experiments. When the length of replay period is short, it could be pretty hard to detect replayed events if both of the recorded traffic and the traffic during replay do not deviate from the learned normality.

7.5 Discussions

This section discusses the proposed approach in terms of its generality and robustness to APT according to the analysis. We then go on to discuss the challenges regarding FPR and lack of semantics for network-based anomaly detection systems and how can them be addressed in practice.

This research inherited the findings of previous work which used a 12-day dataset generated from a virtualized testbed combining a SCADA network formed of virtual machines and a power grid simulator [17]. Though the previous work did not conclude the existence of the used attributes, inter-arrival time and correlation attributes, we can infer from its observations that these attributes may exist as shown in the section 3.2. Considering the results of empirical study in this work, we can expect the used attributes to appear in other power grid network traffic as well, but more work on new sites would be useful before generalizing the result.

Our method was shown promising for flows that have a pattern which is possible to model. It is still an open question whether smart grids with frequent re-configurations in the face of market changes or demand-response imbalances may exhibit new patterns of timing for spontaneous events.

The proposed approach is a composition of two components. The inter-arrival time component examines the pcap timestamp of the packets, which means the time when the

packet arrives at the data collection workstation, instead of the time tag in the payload. In practice, this kind of IDS is usually located in the kernel of the workstation and extracts the arrival time directly from the network driver. This information can not be simply altered for evasion or poisoning as long as the adversaries do not take control of the network driver. When the network driver has been compromised, it's too late for any network-based IDS. However, the adversaries may take control of a field device and send forged data that mimic the regular timings as shown in section 7.2.2. This kind of evasion can bypass our inter-arrival time component but it is still detectable by the correlation component as shown in Table 3. This is because the correlated traffic flows are located across the RTUs in the network. The only way to bypass the monitoring of the correlation component is to take control of all the monitored field devices and send the forged data simultaneously. Hence, our approach fits the distributed nature of SCADA systems.

Two main challenges for network-based anomaly detection are avoidance of high false positive rates and reaction to alarms lacking semantics. Our approach reported around 1% FPR, which means 1 alarm per 100 minutes in the configured setting and less than 15 alarms a day. The evaluation on whether the FPR is manageable depends on the settings of the facilities. Solely receiving the alarm with no connection to the system information puts the burden on the operator for further investigations. However, a growing number of critical infrastructure facilities are supported by a security operation center housing security expertise as well as domain knowledge about the operational setting. Combining capabilities of different technologies such as digital forensics, an anomaly detector with 15 alarms a day can be helpful.

8 Conclusions and Future Work

This work explores the potential of modeling IEC-104 spontaneous events using two timing attributes. One is based on the inter-arrival times, and the other is based on the correlation between flows. These two attributes have been shown very stable in the whole experiment period of this work on a dataset collected from a real power station. Our experiments generate consistent false alarm rates in the learning and testing period if the learning period contains enough number of observations.

We also propose methods for modeling two attack scenarios at the field device level and analyze their impact on timings. The performance downgrade anomalies are caused by attacks against a field device and their impact is persistent. The replay anomalies are caused by malware on a field device and the impact is intermittent.

The proposed anomaly detector successfully detects real anomalies in the dataset from a real power facility as well as some of the synthetic anomalies. Our approach shows different level of detection accuracy for different type of anom-

lies. For the performance downgrades (persistent) anomalies, our approach exhibits a 100% detection rate with at most 0.5% false positive rate. For the replay (intermittent) anomalies, the detection rates and timing performances are satisfactory for experiments under the following conditions: (1) the anomalies last for a longer period (over 1 hour), or (2) the original traffic has relatively low event rates.

The preliminary results show that the idea of modeling the timing characteristics of spontaneous events for anomaly detection is fruitful. The spontaneous traffic exhibits persistent timing characteristics with regards to its inter-arrival times and correlation between flows. This study can be used as a foundation of future research on anomaly detection in spontaneous traffic and our dataset is available for further trials. The obvious future work includes studying a more advanced machine learning model to enhance the detection ability of replayed events. The fact that the timing performance of the inter-arrival time model decreases when the severity of attacks increases also indicates a potential need of a more complicated model to monitor the change to alarm distributions, such as a model using range value or moving average.

To our knowledge this is the first study of anomaly detection focusing on IEC-104 spontaneous traffic. We do not compare our approach with methodologies designed for other types of traffic. Nevertheless, we plan to generate more attack scenarios and study their impact on timing in the context of a national emulated testbed for SCADA systems (RICS-el). Emulated testbeds allow comparison of multiple methods in the same repeatable scenario, but a functioning shared research infrastructure for this purpose is still missing in the research community.

Acknowledgments

This research is supported by the Swedish Civil Contingencies Agency (MSB) through the RICS (www.rics.se) project. The authors would also like to thank the industrial partners who participated in the data collection as well as the anonymous reviewers who helped improving our paper.

Availability

The used dataset, which is formed of CSV files containing timings extracted from a real power station, is available via the following URL.

https://gitlab.liu.se/ida-rtslab/public-code/2019_iec-104.ad

References

[1] ALMGREN, M., ANDERSSON, P., BJÖRKMAN, G., EKSTEDT, M., HALLBERG, J., NADJM-TEHRANI, S., AND WESTRING, E. RICS-el: Building a national

testbed for research and training on SCADA security. In *2018 Critical Information Infrastructures Security (CRITIS)* (2019), LNCS, Springer.

- [2] AOUDI, W., ITURBE, M., AND ALMGREN, M. Truth will out: Departure-based process-level detection of stealthy attacks on control systems. In *Proc. of the Conference on Computer and Communications Security (CCS)* (2018), ACM.
- [3] BARBOSA, R. R. R., SADRE, R., AND PRAS, A. Exploiting traffic periodicity in industrial control networks. *International Journal of Critical Infrastructure Protection* 13, C (June 2016).
- [4] CÁRDENAS, A. A., AMIN, S., AND SASTRY, S. Research challenges for the security of control systems. In *Proc. of the 3rd conference on Hot topics in security (HotSec)* (2008), ACM.
- [5] CASELLI, M., ZAMBON, E., AND KARGL, F. Sequence-aware intrusion detection in industrial control systems. In *1st Workshop on Cyber-Physical System Security (CPSS)* (2015), ACM.
- [6] DÜSSEL, P., GEHL, C., LASKOV, P., BUSSE, J.-U., STÖRMANN, C., AND KÄSTNER, J. Cyber-critical infrastructure protection using real-time payload-based anomaly detection. In *2009 Critical Information Infrastructures Security (CRITIS)* (2010), LNCS, Springer.
- [7] FAISAL, M., CÁRDENAS, A. A., AND WOOL, A. Modeling modbus TCP for intrusion detection. In *2016 Conference on Communications and Network Security (CNS)* (2016), IEEE.
- [8] GIRALDO, J., URBINA, D., CÁRDENAS, A., VALENTE, J., FAISAL, M., RUTHS, J., TIPPENHAUER, N. O., SANDBERG, H., AND CANDELL, R. A survey of physics-based attack detection in cyber-physical systems. *ACM Computing Surveys* 51, 4 (September 2018).
- [9] GOLDENBERG, N., AND WOOL, A. Accurate modeling of Modbus/TCP for intrusion detection in SCADA systems. *International Journal of Critical Infrastructure Protection* 6, 2 (June 2013).
- [10] HADŽIOSMANOVIĆ, D., SIMIONATO, L., BOLZONI, D., ZAMBON, E., AND ETALLE, S. N-gram against the machine: On the feasibility of the n-gram network analysis for binary protocols. In *Research in Attacks, Intrusions, and Defenses (RAID)* (2012), vol. 7462 of *RAID 2012*, LNCS, Springer.

- [11] HADŽIOSMANOVIĆ, D., SOMMER, R., ZAMBON, E., AND H.HARTEL, P. Through the eye of the plc: Semantic security monitoring for industrial processes. In *Proc. of the 30th Annual Computer Security Applications Conference (ACSAC)* (2014).
- [12] KALLURI, R., MAHENDRA, L., KUMAR, R. S., AND PRASAD, G. G. Simulation and impact analysis of denial-of-service attacks on power SCADA. In *2016 National Power Systems Conference (NPSC)* (2016), IEEE.
- [13] KISS, I., GENGE, B., AND HALLER, P. A clustering-based approach to detect cyber attacks in process control systems. In *13th International Conference on Industrial Informatics (INDIN)* (2015), IEEE.
- [14] KLEINMANN, A., AND WOOL, A. Automatic construction of statechart-based anomaly detection models for multi-threaded SCADA via spectral analysis. In *2nd Workshop on Cyber-Physical Systems Security and Privacy (CPSS)* (2016), ACM.
- [15] KLEINMANN, A., AND WOOLKW, A. Accurate modeling of the Siemens S7 SCADA protocol for intrusion detection and digital forensic. *The Journal of Digital Forensics, Security and Law* 9, 2 (2014).
- [16] KROTOFIL, M., LARSEN, J., AND GOLLMANN, D. The process matters: Ensuring data veracity in cyber-physical systems. In *Proc. of the 10th Symposium on Information, Computer and Communications Security (AsiaCCS)* (2015), ACM.
- [17] LIN, C.-Y., AND NADJM-TEHRANI, S. Understanding IEC-60870-5-104 traffic patterns in SCADA networks. In *Proc. of the 4th Cyber-Physical System Security Workshop (CPSS)*. (2018), ACM.
- [18] LIN, C.-Y., NADJM-TEHRANI, S., AND ASPLUND, M. Timing-based anomaly detection in SCADA networks. In *2017 Critical Information Infrastructures Security (CRITIS)* (2018), LNCS, Springer.
- [19] LIU, Y., NING, P., AND REITER, M. K. False data injection attacks against state estimation in electric power grids. In *Proc. of the 16th conference on Computer and Communications Security (CCS)* (2009), ACM.
- [20] LONG, M., WU, C.-H., AND HUNG, J. Y. Denial of service attacks on network-based control systems: Impact and mitigation. *IEEE Transactions on Industrial Informatics* 1, 2 (MAY 2005 2005).
- [21] MARKMAN, C., WOOL, A., AND CARDENAS, A. A. A new burst-DFA model for SCADA anomaly detection. In *Proc. of the 2017 Workshop on Cyber-Physical Systems Security and Privacy (CPS-SPC)* (2017), ACM.
- [22] MARKMAN, C., WOOL, A., AND CARDENAS, A. A. Temporal phase shifts in SCADA networks. In *Proc. of the 2018 Workshop on Cyber-Physical Systems Security and Privacy (CPS-SPC)* (2018), ACM.
- [23] MARKOVIC-PETROVIC, J. D., AND STOJANOVIC, M. D. Analysis of SCADA system vulnerabilities to DDoS attacks. In *2013 11th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services (TELSIKS)* (2014), IEEE.
- [24] NIEDERMAIER, M., MALCHOW, J.-O., FISCHER, F., MARZIN, D., MERLI, D., ROTH, V., AND VON BODISCO, A. You snooze, you lose: Measuring PLC cycle times under attacks. In *12th Workshop on Offensive Technologies (WOOT)* (2018), USENIX Association.
- [25] SAYEGH, N., ELHAJJ, I. H., KAYSSI, A., AND CHEHAB, A. SCADA intrusion detection system based on temporal behavior of frequent patterns. In *17th Mediterranean Electrotechnical Conference (MELECON)* (2014), IEEE.
- [26] SHOUKRY, Y., MARTIN, P., YONA, Y., DIGGAVI, S., AND SRIVASTAVA, M. Pycra: Physical challenge-response authentication for active sensors under spoofing attacks. In *Proc. of the 22nd Conference on Computer and Communications Security (CCS)* (2015), ACM.
- [27] VALDES, A., AND CHEUNG, S. Communication pattern anomaly detection in process control systems. In *Conference on Technologies for Homeland Security (HST)* (2009), IEEE.
- [28] WEDGBURY, A., AND JONES, K. Automated asset discovery in industrial control systems - exploring the problem. In *Proc. of the 3rd International Symposium for ICS & SCADA Cyber Security Research (ICS-CSR)* (2015), ACM.
- [29] WRESSNEGGER, C., KELLNER, A., AND RIECK, K. Zoe: Content-based anomaly detection for industrial control systems. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2018), IEEE.
- [30] YANG, Y., MCLAUGHLIN, K., SEZER, S., YUAN, Y., AND HUANG, W. Stateful intrusion detection for IEC-60870-5-104 SCADA security. In *PES General Meeting (PES GM)* (2014), IEEE.
- [31] YOON, M.-K., AND CIOCARLIE, G. Communication pattern monitoring: Improving the utility of anomaly detection for industrial control systems. In *Workshop on Security of Emerging Networking Technologies (SENT)* (2014), Internet Society.

Appendix

This appendix section lists additional measurement results for the proposed modeling approach. The notion of warning means the first-level anomalies generated by the proposed detection methodology as described in 5.2. They form the basis of further analysis to derive the alarms that will be sent to the operators.

A warning rates over the learning period

Table 4 shows the overview of the datasets and the warning rates generated by the inter-arrival time model and the correlation model for the learning data. For the inter-arrival time model, warning rate I is calculated directly after the *Grouping step* (the second step of the inter-arrival time model construction). Warning II is calculated after completing the *Finding boundaries step*. In these cases we find best fitting boundaries without any specific assumption on their distribution. Warning rate III is calculated after completing the *Finding boundaries step* with Gaussian distribution assumption. For the correlation model, warning rate (Corr.) is calculated after finishing all steps of (correlation) modeling and used to select useful pairs.

The flows are listed in a descending order of its size (number of events). With this specific dataset, we choose to adopt the specialized estimation of inter-arrival time group boundaries when the number of observations is less 50 in the first methodology (percentile) and when the number of observations is less 5 in the second methodology (Gaussian).

It is worth noticing that the inter-arrival time model produces high amount of (false) warnings in either methodologies when the number of events is too small (less than 8330). The specialized estimation of boundaries for groups with too few observations is no longer effective when most of the groups in the same flow are too small.

B warning rates over the testing period

Table 5 presents the warning rates generated by the inter-arrival time model and the correlation model for the testing period from the original traffic. Corr. Warning Rate I is calculated in the original testing period till 41591 minutes. Corr. Warning Rate II is calculated in the testing period before 39500 minutes. Δ Warning Rate is calculated in the original testing period as well. Most of the flows have warning rates that are consistent with the results in Table 4. This shows the stability of the used timing characteristics.

Table 4: Overview of datasets and learning results.

RTU	IOA	# Events	Used in Δ	Warning Rate I	Warning Rate II	Warning Rate III	Paired IOA	Used in Corr.	Warning Rate (Corr.)
RTU A	3019	381038	Y	0.0003	0.026	0.46	A3010	Y	0.422
	3014	347000	Y	0	0.020	0.59	A3013	N	2.578
	3013	346959	Y	0.0003	0.022	0.52	A3014	N	2.578
	3012	341235	Y	0.0003	0.023	0.46	A3013	N	2.439
	3020	328329	Y	0.0003	0.021	0.49	A3010	Y	0.003
	3011	311823	N	0.0003	0.025	0.52	A3015	N	1.761
	3015	311087	N	0	0.024	0.55	A3011	N	1.761
	3017	252486	N	0.0003	0.025	0.41	A3016	Y	0.211
	3018	243580	N	0.0057	0.026	0.85	A3015	Y	1.772
	3010	204059	N	0.0044	0.026	0.28	A3019	Y	0.422
	3021	202539	N	0	0.027	0.50	B3006	Y	0.156
	3016	198924	N	0	0.025	0.40	A3017	Y	0.211
	3005	192750	N	0.0026	0.025	0.54	A3002	N	1.728
	3004	128184	N	0.0062	0.033	0.38	A3009	Y	0.350
	3007	128166	N	0	0.020	0.25	A3010	Y	0.617
	3002	122920	N	0.0016	0.024	0.57	A3005	N	1.728
	3008	111891	N	0	0.017	0.23	A3003	Y	0.594
	3003	111562	N	0.0009	0.024	0.25	A3008	Y	0.594
	3009	82339	N	0.0073	0.035	0.38	A3004	Y	0.350
	RTU B	3019	16448	Y	0.10	14.88	0.54	B3016	Y
3016		16276	Y	0.10	14.86	0.46	B3019	Y	0.850
3006		15706	Y	0.08	0.22	1.48	B3019	N	1.144
3009		8330	N	0.16	0.16	0.23	B3002	N	1.622
3002		8278	N	0.13	1.96	4.42	B3009	N	1.622
3004		6421	N	0.20	2.85	6.17	B3011	Y	0.067
3012		6011	N	0.13	2.23	7.32	B3005	Y	0.228
3011		5858	N	0.15	2.36	8.04	B3005	Y	0.356
3018		5657	N	0.16	11.47	0.69	B3009	Y	0.156
3014		4936	N	0.12	2.90	9.70	B3013	Y	0.111
3008		4923	N	0.08	3.64	10.97	B3014	Y	0.172
3005		4157	N	0.07	7.22	11.47	B3011	Y	0.356
3013		4138	N	0.05	4.18	13.03	B3014	Y	0.111
3015		2576	N	0.70	1.86	5.09	A3021	Y	0.033

Table 5: The plain warning rates for the inter-arrival time model and the correlation model in the test phase.

RTU	IOA	Used in Δ	Δ Warning Rate	Paired IOA	Used in Corr.	Corr. Warning Rate I	Corr. Warning Rate II
RTU A	3019	Y	0.002	A3010	Y	0.725	0.363
	3014	Y	0.012	A3013	N	4.849	4.910
	3013	Y	0.029	A3014	N	4.849	4.910
	3012	Y	0.043	A3013	N	4.565	4.519
	3020	Y	0.001	A3010	Y	7.503	0.345
	3011	N	0.025	A3015	N	0.886	0.924
	3015	N	0.006	A3011	N	0.886	0.924
	3017	N	0.007	A3016	Y	0.682	0.657
	3018	N	0.154	A3015	Y	0.026	3.434
	3010	N	0.003	A3019	Y	0.752	0.363
	3021	N	0.001	B3006	Y	0.144	0.156
	3016	N	0.000	A3017	Y	0.682	0.657
	3005	N	0.015	A3002	N	3.090	3.117
	3004	N	0.021	A3009	Y	0.271	0.262
	3007	N	0.003	A3010	Y	0.674	0.708
3002	N	0.009	A3005	N	3.090	3.117	
3008	N	0.004	A3003	Y	0.606	0.625	
3003	N	0.002	A3008	Y	0.606	0.625	
3009	N	0.027	A3004	Y	0.271	0.262	
RTU B	3019	Y	0.682	B3016	Y	1.441	1.324
	3016	Y	0.597	B3019	Y	1.441	1.324
	3006	Y	1.252	B3019	N	2.039	1.779
	3009	N	1.816	B3002	N	1.246	1.256
	3002	N	7.473	B3009	N	1.246	1.256
	3004	N	11.398	B3011	Y	0.013	0.014
	3012	N	11.909	B3005	Y	0.322	0.271
	3011	N	12.716	B3005	Y	0.568	0.520
	3018	N	5.463	B3009	Y	0.102	0.106
	3014	N	16.947	B3013	Y	0.424	0.400
	3008	N	16.188	B3014	Y	0.411	0.400
	3005	N	23.503	B3011	Y	0.568	0.520
	3013	N	22.965	B3014	Y	0.424	0.400
	3015	N	21.467	A3021	Y	0.008	0.009

USBESAFE: An End-Point Solution to Protect Against USB-Based Attacks

Amin Kharraz^{†‡} Brandon L. Daley^{◇‡} Graham Z. Baker[◇] William Robertson[‡] Engin Kirda[‡]

[◇]MIT Lincoln Laboratory [†]University of Illinois at Urbana-Champaign [‡]Northeastern University

Abstract

Targeted attacks via transient devices are not new. However, the introduction of BadUSB attacks has shifted the attack paradigm tremendously. Such attacks embed malicious code in device firmware and exploit the lack of access control in the USB protocol. In this paper, we propose USBESAFE as a mediator of the USB communication mechanism. By leveraging the insights from millions of USB packets, we propose techniques to generate a protection model that can identify covert USB attacks by distinguishing BadUSB devices as a set of *novel* observations. Our results show that USBESAFE works well in practice by achieving a true positive [TP] rate of 95.7% with 0.21% false positives [FP] with latency as low as *three* malicious USB packets on USB traffic. We tested USBESAFE by deploying the model at several end-points for 20 days and running multiple types of BadUSB-style attacks with different levels of sophistication. Our analysis shows that USBESAFE can detect a large number of mimicry attacks without introducing any significant changes to the standard USB protocol or the underlying systems. The performance evaluation also shows that USBESAFE is transparent to the operating system, and imposes no discernible performance overhead during the enumeration phase or USB communication compared to the unmodified Linux USB subsystem.

1 Introduction

Transient devices such as USB devices have long been used as an attack vector. Most of these attacks rely on users who unwittingly open their organizations to an internal attack. Instances of security breaches in recent years illustrate that adversaries employ such devices to spread malware, take control of systems, and exfiltrate information.

Most recently, researchers have shown that despite several warnings that underscore the risk of malicious peripherals, users are still vulnerable to USB attacks [27, 28]. To tackle this issue, antivirus software is becoming increasingly adept at scanning USB storage for malware. The software

automatically scans removable devices including USB sticks, memory cards, external hard drives, and even cameras after being plugged into a machine. Unfortunately, bypassing such checks is often not very difficult as the firmware of USB devices cannot be scanned by the host. In fact, the introduction of BadUSB attacks has shifted the attack paradigm tremendously as adversaries can easily hide their malicious code in the firmware, allowing the device to take covert actions on the host [9]. A USB flash drive could register itself as both a storage device and a Human Interface Device (HID) such as a keyboard, enabling the ability to inject surreptitious keystrokes to carry out malice.

Existing defenses against malicious USB devices have resulted in improvements in protecting end-users, but these solutions often require major changes in the current USB protocol by introducing an access control mechanism [26], modifying the certificate management [20], or changing the user experience (i.e., a user-defined policy infrastructure) [3, 24]. Our goal is different in a sense that we seek to improve the security of USB devices while keeping the corresponding protection mechanism completely in the background. The immediate benefit of such a solution is flexibility, allowing: (1) organizations to use standard devices, (2) manufacturers to avoid changing how their hardware operates, and (3) users to continue using their current USB devices.

In this paper, we propose USBESAFE, a system to identifying BadUSB-style attacks, which are probably the most prominent attack that exploits the USB protocol. Our approach relies upon analyzing how benign devices interact with the host and the operating system. By leveraging the insights from millions of USB Request Blocks (URBs) collected over 14 months from a variety of USB devices such as keyboards, mice, headsets, mass storage devices, and cameras, we propose classification techniques that can capture how a benign USB device interacts with a host by monitoring URBs as they traverse the bus. Starting with a wide range of classification features, we carefully analyze the labeled data and narrow down to three feature categories: content-based, timing-based, and type-based features. We train several

different machine learning techniques including SVM [14], Nearest Neighbor [13], and Cluster-based Techniques [12] to find the most accurate algorithm for building our detection model. Our analysis showed that One-Class SVM achieved the highest detection results with a low false positive rate (a true positive [TP] rate of 95.7% with 0.21% false positives [FPs]) on the labeled dataset. The constructed model allows us to identify covert USB attacks by distinguishing BadUSB devices as *novel* observations for the trained dataset.

To test USBESAFE, we deployed the constructed model as a service on end-user machines for 20 days. Our analysis shows that USBESAFE is successful in identifying several forms of BadUSB attacks with a low false positive rate on live, unknown USB traffic. For a real-world deployment, we also performed a training/re-training analysis to determine how USBESAFE should be deployed on new machines to keep the detection rate constantly high with under a 1% false positive rate. We show that training USBESAFE with as low as two training days and re-training it every 16 days for 82 seconds are sufficient to maintain the detection rate over 93% across all the machines.

The most important finding in this paper is practical evidence that shows it is possible to develop models that can explain the benign data in a very precise fashion. This makes anomaly detection a promising direction to defend against BadUSB-style attacks without performing any changes to the standard USB protocol or underlying systems. We ran multiple forms of adversarial scenarios to test USBESAFE's resilience to evasion with the assumption that adversaries have significant freedom to generate new forms of BadUSB-style attacks to evade detection. Our analysis shows that USBESAFE can successfully detect mimicry attacks with different levels of sophistication without imposing a discernible performance impact or changing the way users interact with the operating system. We envision multiple potential deployment models for USBESAFE. Our detection approach can be incorporated as a light-weight operating system service to identify BadUSB attacks and disable the offending port or an early-warning solution to automatically identify the attacks and notify system administrators.

2 Background, Threat Model, and Related Work

A Universal Serial Bus (USB) device can be a peripheral device such as a Human Interface Device (HID), printer, storage, or a USB transceiver. An attached USB device can have multiple functionalities where each functionality is determined by its interfaces. The host controller interacts independently with these interfaces by loading a device driver for each interface. When a USB device is attached, the USB controller in the host issues a set of control requests to obtain the configuration parameters of the device in order to activate the supported configuration. The host parses the configuration, and reads

the device descriptor which contains the information about the functionality of device. This information allows the host to load a driver based on the configuration information. This procedure is called *enumeration* phase. In the enumeration phase of the USB protocol, the endpoints are addressed as `IN` and `OUT` to manage the USB traffic. The `IN` endpoint stores the data coming to the host, and the `OUT` endpoint receives the data from the host. After the enumeration phase, the host loads the USB interfaces which allow a device to operate.

2.1 Threat Model

In our threat model, we assume that a connecting device can report any capabilities to the bus, and the host machine trusts the information that it receives from the device. Similar to BadUSB attacks [9], an adversary can use this capability by rewriting the firmware of an existing device to hide malware in the code that communicates with a host. More specifically, upon insertion into a host USB port, a mass storage device – i.e., a USB flash drive (with capabilities for Windows and Linux) – covertly performs keyboard actions to open a command prompt, issue a shell command to download malicious code from the Internet, and execute the downloaded malware.

We should mention that classic USB attacks, for example using the autorun capabilities of USB devices to distribute malware, are out of the scope of the paper as these attacks can be detected by most of malware scanners. Similar to prior work [24], we try to address the advanced persistent threat (APT) scenario where an adversary is attempting to expand its presence in a network by distributing USB devices with malicious firmware as described above. We assume that the malicious USB device is capable of generating new device identities during the enumeration phase by providing varying responses in each enumeration to evade potential device identification mechanisms. We also assume that there exists no USB-level authentication mechanism between the device and the target host. The OS simply acts on information provided by the device and will load a driver to accept the USB drive as, e.g., an HID device. We assume that once the device has been connected, the adversary can use any technique to expand her presence. For example, the malicious firmware can open a command prompt to perform privilege escalation, exfiltrate files, or copy itself for further propagation. Finally, in this work, we also assume that the trusted computing base includes the display module, OS kernel, and underlying software and hardware stack. Therefore, we consider these components of the system free of malicious code, and that normal user-based access control prevents attackers from running malicious code with superuser privileges.

2.2 Related Work

A wide range of attacks have been introduced via USB including malware, data exfiltration on removable storage [8, 16, 17, 22], and tampered device firmware [5, 9]. These cases

show that defending against USB attacks is often not straightforward as these attacks can be tailored for many scenarios. In the remainder of this section, we explore existing solutions for this class of attack vector and their limitations.

One approach to defend against attacks involving subverted firmware is to hardwire USB microcontrollers to only allow firmware updates that are digitally signed by the manufacturer. Currently, the de facto technology to protect against malicious data residing on and executing from a device exists in IEEE Standard 1667 [20]. The standard seeks to create a means for bidirectional authentication via an X.509 certificate infrastructure between hosts and devices. Unfortunately, the adoption of IEEE 1667 has been slow, and USB devices do not possess any entity authentication mechanism as a means of vouching for the safety of data residing on the device.

One of the first research efforts to secure the USB protocol was conducted by Bates et al. [4, 15] where they measured the timing characteristics during USB enumeration to infer characteristics of host machines. Another class of work focuses on proposing access control mechanisms on USB storage devices [6, 19, 23, 30]. While these approaches can lead to better defense mechanisms, recent studies [21, 24] have shown that these approaches are coarse and cannot distinguish between desired and undesired usage of a particular interface. Very recently, Hernandez et al. [7] introduced FirmUSB, a firmware analysis framework, to examine firmware images using symbolic analysis techniques. By incorporating the tool, the authors identified the malicious activity without any source code analysis while decreasing the analysis time. In fact, the proposed technique is very effective in addressing some of the increasing concerns on the trustworthiness and integrity of USB device firmwares.

One other approach to mitigating such attacks is to minimize the attack surface without changing the fundamentals of USB communication or patching major operating systems. Recently, Tian et al. [24] have proposed GoodUSB which has similar goals to ours. Their approach is based on constructing a policy engine that relies on virtualization and a database that consists of already seen USB devices and reporting unknown USB devices to the user. The proposed solution mediates the enumeration phase, and verifies what the device claims as its functionality by consulting to a policy engine. GoodUSB shifts the burden of responsibility to the user to decide whether a USB device is malicious or benign.

In another work, Tian et al. [26] proposed USBFilter, a packet-driven access control mechanism for USB, which can prevent unauthorized interfaces from connecting to the host operating system. USBFilter traces individual USB packet, and blocks unauthorized access to the device. Tian et al. [25] complemented their previous work by introducing ProvUSB which incorporated provenance-based data forensics and integrity assurance to defend against USB-based threats. Angel et al. [3] uses a different approach and leverages virtualization to achieve the same goal. We posit that a solution such

as the one described in this paper that introduces as little change as possible to the user operational status quo is more likely to prevent exploitation in practice, given that the underlying detection mechanism is reliable. That being said, these approaches are fundamentally orthogonal and could be composed to obtain the benefits of both.

3 Overview of The Approach

In this section, we provide more details on USBESAFE components and the model we use to detect BadUSB attacks. Figure 1 shows the pipeline used by USBESAFE to identify BadUSB-style attacks.

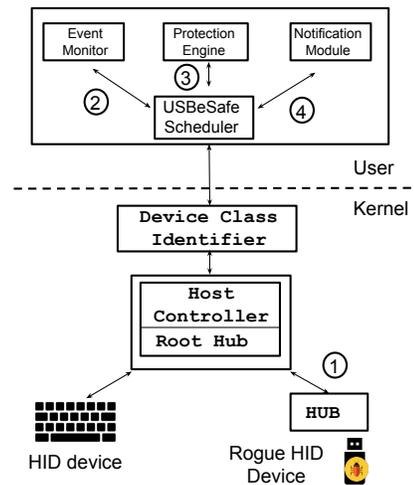


Figure 1: A high level view of a USBESAFE-enabled machine.

3.1 System Design

The architecture of a USBESAFE-enhanced system requires interactions among multiple components of the operating system. In this section, we describe the abstract design of USBESAFE, independent of the underlying OS. Later, we will demonstrate how our design can be realized in a prototype running on Linux. USBESAFE’s components are mostly managed by a user space daemon. The daemon includes three main subsystems as shown in Figure 1: First, a lightweight user space module that processes transaction flows between the host and the connected device; second, a detection module that implements the USB mediator logic; and third, a user interface that generates alerts and notifies the user. When a USB device is connected to the host (1), USBESAFE collects and preprocesses the URBs (2). The protection engine utilizes the preprocessed data to construct the feature vector, and test whether the incoming USB packets are in fact new observations (3). In cases where the system detects a novel sequence of USB packets, it creates a notification, and sends an alert

to the user (4). In the following, we provide more details on each proposed module.

3.1.1 USB Event Monitor

The ultimate goal of the event monitor is to analyze URBs and transform them to an appropriate format that can be used in the protection engine. To this end, the USB event monitor detects a connected device, and processes the transaction flows in the form of URBs which contain USB packets during a USB connection lifecycle from the enumeration, to configured communication, to termination. To store and analyze USB packets, we implemented a set of data objects. The module parses each URB, extracts the USB packet, and generates a *TraceEvent* containing the USB header information and payload. In fact, each *TraceEvent* is a tuple that contains the host bus ID as well as the assigned device ID on the bus. Each *TraceEvent*, representing a single USB packet, is appended to a *Trace* – a list of *TraceEvents*. USBESAFE generates a single *Trace* file for each USB device from the enumeration to disconnection phase of the connected device. *TraceEvents* in each *Trace* are sorted according to their timestamp, from earliest to most recent. For each *Trace*, we identify the device and configuration descriptor responses, storing them as auxiliary information for the *Trace*.

The root of the data structure is called the *TraceLibrary* which contains all *Traces* in the dataset. For each *trace* in the *TraceLibrary*, we sort existing *traces* into *TraceLists* which contain all the *traces* with the same (*busID*, *deviceID*) tuple. USBESAFE uses the class codes field in the device and interface descriptors to determine the device type and expected functionality. While USBESAFE has an extensible design, we focus specifically on USB HIDs including USB keyboard traffic and the features that characterize such traffic as benign. This focus stems from our goal which is to determine whether a covert HID configuration is present and active on a device.

3.1.2 Protection Engine

The protection engine is central to the security model proposed in USBESAFE which decides whether a new, previously unseen set of USB packets is potentially malicious. In the following, we explain the features we employed to characterize the USB packets, and train the detection model in USBESAFE.

Packet Interarrival Times Packet interarrival times characterize the USB keyboard traffic across a bus and, specifically, the timing information of packets. The timing information can help reveal user’s typing patterns by serving as a proxy for inter-keystroke times, or how the bus manages the URBs of different kinds. Interarrival time values are measured in milliseconds, between one packet and the next for all the *TraceEvents*. Note that a user may enter a keystroke after a longer pause, ranging from a few seconds to hours. To tackle the problem of potentially unbounded interarrival time values

in these situations, we explicitly define an upper bound for the interarrival time value between two interrupt packets. We explain this procedure and the selection of specific threshold values in more details in Section 5.

Event Type USBESAFE monitors the value that defines the type of each USB packet. More precisely, a URB event type can take two values which indicate whether there is an ongoing transaction (`URB_SUBMIT (0x53)`) or if a transaction is complete (`URB_COMPLETE (0x43)`).

Transfer Type USBESAFE also monitors the value of URB transfer type between the host and the USB device. The transfer type can take four possible values which are `URB_INTERRUPT`, `URB_CONTROL`, `URB_BULK`, and `URB_ISOCHRONOUS`. This value is selected for a USB device according to the requirements of the device and the software which is determined in the endpoint descriptor.

Post-enumeration Time USBESAFE monitors when the post-enumeration activity starts. For example, in a normal scenario, a user connects a USB keyboard to the host, and starts interacting with it. Along with other features, looking at millions of URBs allows USBESAFE to observe the normal start of post-enumeration activity of a HID device after attaching the device to the host. The system incorporates this feature as a numerical value by calculating the time period between a successful enumeration and the start of data packet transfer.

Packet Payload USBESAFE monitors the payload of individual USB packets. USBESAFE examines the payload to determine patterns in data by using a byte histogram to measure value frequencies within each *TraceEvent*. The histogram represents a space of 256 values ($[0, 255]$) by bucketizing values into 16 equal intervals or bins. For each *TraceEvent*, the system generates a feature vector which contains *interarrival_time*, *event_type*, *transfer_type*, *post_enumeration*, *data_histogram*. The feature vector is then used to construct a detection model which will be used as an augmented service in the operating system.

To analyze the URB payloads, we extracted all the *n*-grams of *EventTraces* that appeared in a sliding window of the length *n* where the value of *n* varied from 2 to 4. Each unique sequence of length *n* is added to the detection model for the USB HID class. The intuition here is that those *n*-grams are characteristics of benign USB packets, and any traffic that does not follow similar patterns compared to the extracted model for a given user is a novel observation, and with high likelihood correspond to a new typing pattern. In Section 5.3, we provide more details on our model searching process since we should take into account several configuration parameters to achieve the highest detection rate and the lowest false positive rate.

4 Implementation

In this section, we provide more details on the implementation of USBESAFE's prototype which relies on the Linux USB stack. The implementation of USBESAFE consists of three independent modules which were discussed in Section 3.1: (1) a USB event monitor which interposes the bus transactions, (2) a protection module which constructs the feature vector and validates whether the incoming USB packets comply with the generated model, and (3) a notification module which produces an alert and notifies the user if a novel traffic pattern is detected. In the following, we provide more details on the implementation details of each module.

4.1 USB Event Monitor

We used the `usbmon` Linux kernel module, as a general USB layer monitor, to capture all the URBs transmitted across the monitored USB bus. In user space, USBESAFE implements the transaction flow introspection module by extracting device information using `sysfs`, `lsusb` and device activities using `usbmon` and `tcpdump`. Monitoring the USB devices starts at the boot time. USBESAFE collects self-reported device information as well as actions taken by associated drivers during the normal usage. The USB event monitor module is a user space program which is developed in Python. This module is loaded prior to the device's enumeration phase by updating the `udev` database. We defined a set of datastructure to collect interface information (e.g., Descriptor Type, Interface Number, Interface Class and protocol), configuration information (e.g., max power), and device information (e.g., manufacturer) for each connected device.

4.2 Protection Engine

As mentioned earlier in Section 3.1, the protection engine in USBESAFE is responsible for determining whether a set of USB packets from a connected device to the host are, in fact, new observations and whether the USB device should be disabled or not. In cases where the USBESAFE identifies a set of USB packets as new observations, it notifies the user as well as kernel space components to block the corresponding interface. As the protection engine is the core part of the USBESAFE, we want to make sure that the model is constructed based on a suitable algorithm. To this end, we evaluated multiple machine learning algorithms by measuring their detection accuracy on the labeled dataset. In Section 5, we provide more details on the detection accuracy of the algorithms as well as the parameter configurations. The results of our analyses revealed that one-class SVM [14] achieved the highest detection rate with a very low false positive rate. In our detection model, the one-class SVM can be viewed as a regular two-class SVM where all the training data is benign and lies in the first class, and the unseen data by a large margin from the hyperplane is taken as the second class. In

fact, the constructed model in USBESAFE solves an optimization problem to find a term with maximal geometric margin. Therefore, if the geometric margin is less than zero, the test sample is reported as a novel observation. As USBESAFE has a high privilege, it automatically unbinds the offending USB port by calling `/sys/bus/usb/drivers/usb/unbind` without involving the user.

4.3 Notification Module

The notification module is deployed as a user space daemon which produces alerts whenever the protection module identifies a novel observation. We should mention that there are several design choices for implementing the notification module. However, the core requirement of the module is that the notification should be always stacked on top of the screen contents, and cannot be obscured, interrupted, or interfered with by other processes. We achieve this by leveraging `libnotify-bin` module which is usually used to send desktop notifications to users. To prevent the notification module from being killed programmatically by a potentially malicious process with the same user ID, we recommend creating another user ID to run the process. Consequently, only root could kill the process. As the notification module is not the core part of our contributions, we do not explore the notification module further in this paper.

5 Evaluation

To test USBESAFE we conducted two experiments. In the first experiment, we train and test USBESAFE with a labeled dataset, and in the second experiment, we test the derived model on a previously unseen dataset to evaluate the detection capability of the system in a real-world deployment. Although our design is sufficiently general to be applied to different operating systems, we built our prototype for Ubuntu 14.04 LTS with the Linux kernel 3.19. In the following, we first describe how we created our dataset, and then provide the details of evaluation and benchmarks.

5.1 Data Collection

In order to create the training dataset, we monitored USB packet exchanged between a set of devices and five machines. Each time a USB device was connected, USBESAFE generated a new trace, named it based on the bus and device ID of the USB device, and created logs for real-time USB packets across the monitored USB bus. On the system shutdown, the module saved the generated trace file to the disk. We sorted each `TraceEvent` based on USB device class code which was extracted from the interface descriptor. The HID class, which keyboards, mice, headsets, and game joysticks fall under, is defined by the class code 3. During 14 months of data collection, several types of USB devices such as different keyboards, storage devices, cameras, and headsets were connected to the machines. We considered a connected USB device as a HID

Machine	URBs	No. of Traces
Machine1	3,385,445	124
Machine2	2,394,345	90
Machine3	2,884,345	101
Machine4	943,984	50
Machine5	1,620,265	58
Total	11,228,384	423

Table 1: The collected USB Packets over 14 months. We collected 423 HID trace files which contained more than 11 million USB Packets. The trace files were collected from several types of keyboards, mice, headsets. The dataset also includes traces of other USB devices such as cameras, storage devices which are not HID devices, but registered themselves as a HID device in addition to their main driver.

device, if it requested HID driver from the operating system. For example, we found a benign USB printer that registered itself both as a printer and a HID device to enable the touchscreen. Although standard USB printers are not considered as a HID device, we considered the corresponding traces in the training phase as the device potentially had the capability to run commands.

In total, 423 trace files were collected from HID devices, consisting of 11,228,384 URBs. Note that the actual number of USB packets that crossed the bus was greater than this value as any usbmon-based packet actually represented two or three USB packets on the bus, depending on whether the interaction included a payload or not. Table 1 illustrates a summary of the data collected over 14 months from five different machines.

Note that our approach is not an outlier detection method, but a novelty detection technique. This means that we need to have a clean training dataset representing the population of regular observations for building a model and detecting anomalies in new observations. Therefore, the malicious data used in our experiments was solely collected for testing purposes. To generate this malicious dataset, we used a *Rubber Ducky* USB drive [2], updated the firmware, and generated a set of scripts that establish covert channels, inject code for data exfiltration, and connect to a remote server. Such attacks have several forms, and an adversary has significant freedom to generate such attacks. For example, it is quite feasible to write a malicious script that checks an active session and verifies whether a user is logged in or not before launching an attack. In Section B, we provide some case studies, and show how USBESAFE identifies these attacks as a set of novel observations.

For our experiments, we created eight realistic attack scenarios. In each experiment, we connected the device to the measurement machine and made sure the attack executed while logging the USB packets from the device enumeration to device termination. The malicious dataset contains 202,394 USB packets, which was significantly smaller than the benign dataset, reflecting the expected low base rate of BadUSB-style attacks.

5.1.1 Preprocessing the Dataset

Over the course of the data collection, we found several un-predicted situations during the device enumeration phase. For example, a subset of USB keyboards used in our experiments were not reported as the USB class code 3, but were instead reported as the USB class code 0. Though this occurred, each observed instance of these event sequences yielded a successfully enumerated device, and the host accepted the keyboard input immediately after receiving the device descriptor. For this reason, we worked around the issue during class bucketization in the feature extraction phase, moving all the class code 0 traffic into the class code 3 bucket.

Another issue we had to account for in processing the trace files was to determine what action to take when encountering malformed packets. In some instances, when the host requested a device descriptor, the device would respond with a malformed descriptor packet, forcing the host to make the request again. For the purposes of prototype evaluation, we chose to ignore these request/response pairs when they occurred.

5.2 Model Selection

One of the first questions that arises is which machine learning algorithm achieves the highest detection results if it is trained with the labeled dataset. To this end, we used five different algorithms that are known to model anomaly detection problems. We also considered the *local* and *global* features of the anomaly detection approaches in order to determine whether the novelty score of an incoming URB should be determined with respect to the entire training data or solely based on a subset of previous URBs. To run the experiment, we used one-class SVM as a classifier-based approach [14], *k*-NN as a *global Nearest-neighbor* [13], and *Local Outlier Factor (LOF)* as a *local Nearest-neighbor*-based approach [13]. We also incorporated *Cluster-based Local Outlier Factor (CBLOF)* [12] as a *global Clustering*-based approach and *Local Density Cluster-based Outlier Factor (LDCOF)* [12] as a *local Clustering*-based approach.

To identify the best detection algorithm, we performed an analysis on the 423 traces we collected from five machines (see Section 5.1). More specifically, we split the USB traces of each machine to a training and a testing set using 4-fold cross-validation, and averaged the value of the detection rate and false positive rate for each algorithm. As shown in Table 2, the analyses reveal that LDCOF and LOF, which use local data points, produce lower false positive cases. However, the empirical evidence suggests that the one-class SVM achieves the best detection results among the selected algorithms on the same dataset. Based on our analysis, one likely reason is that the one-class SVM classifier maps the USB traffic to a high dimensional feature space more accurately. This results in producing less false positive cases by identifying the maximal margin hyperplane that best separates the new

Metric	OCSVM	k-NN	LOF	CBLOF	LDCOF
TPR	94.2%	90.6%	91.2	92.7%	92.3%
FPR	0.71%	11.3%	5.3%	3.2%	1.9%

Table 2: The detection results of different machine learning algorithms on the labeled dataset. The analyses show that one-class SVM achieves the highest TPs with a very low FP rate on the same dataset.

observations. Based on this empirical analysis, we used the one-class SVM as our default machine learning algorithm for the rest of the paper.

5.2.1 Determining the Novelty Score

In the model testing process, USBESAFE applies the trained decision function to determine whether an input observation falls within the trained class or outside the trained class. We consider a URB as a *novel* observation, if the decision function assigns the input to the -1 class. In fact, the assigned value -1 implies that the input observation is outside the trained region. The novelty score is calculated as the ratio of inputs classified as novel observations over the total number of input observations. We ran an experiment by incorporating four different kernel functions to train the one-class SVM (see Appendix A), and understand what novelty score should be selected as the threshold at which we decide whether the observation is novel or not. Our analysis shows that the system produced less than 1% false positives when the threshold value = 13.2% was selected for all the four different kernel functions in the one-class SVM algorithm. In Section 5.3, we describe how we enhanced the detection model by empirically identifying a specific set of parameters for the kernel functions.

5.3 Optimizing the Model

Another question that we wanted to answer was whether we can improve the detection model by changing the required configuration parameters of the model on the same labeled dataset. After constructing possible n-grams (see Section 3.1.2), we performed a grid search [31] over the parameter space which consists of: (1) the one-class SVM model parameters (e.g., the polynomial degree), (2) the n-gram window size, and (3) the combinations of detection features.

Based on the resulting parameter space with 105 model parameter settings, 5 features and n-grams with window size 2 (see Table 6 in Appendix A), we generated 6,510 unique one-class SVM model instances. To test the accuracy of the models against BadUSB attacks, we created a set of attacks using a Rubber Ducky USB drive [2]. The attacks were designed to perform covert HID attacks which open a command prompt and execute a malicious code, or connect to a remote server. We elaborate on the malicious dataset later in Section 5. For each individual one-class SVM test, we logged the parameter setting used to generate the model, calculated the average accuracy across all the 4-fold cross validations, and

Machine	No. of Traces	TPs	FPs
Machine1	124	97.4%	0.16%
Machine2	90	95.6%	0.23%
Machine3	101	96.7%	0.15%
Machine4	50	94.0%	0.31%
Machine5	58	94.3%	0.28%
Per User Model (avg)	423	95.7%	0.21%
General Model	423	94.9%	0.93%

Table 3: The detection results of USBESAFE on different machines. In this experiment, we used one-class SVM with the polynomial kernel with degree 3, $\gamma = 0.1$ and $\nu = 0.75$ using all the features. Our analysis shows that per user model is more effective in terms of producing lower false positive cases.

their corresponding standard deviations. We removed a model instance from our search space if the false positive rate of the model was more than 4.0%. Our assumption was that it is very unlikely that an end-point solution with a false positive rate more than 4.0% would be useful to be deployed on user machines. Our analysis shows that USBESAFE achieves the highest TP and FP rates (TP rate 95.7% at 0.21% FPs) when one-class SVM uses the polynomial kernel with degree 3, $\gamma = 0.1$ and $\nu = 0.75$ using all the features defined in Section 3.1. Table 3 shows the True Positives (TPs) and False Positives (FPs) for each user machine based on the derived model. The results show that it is possible to achieve even a higher detection rate at a lower false positive rate on the same dataset by tuning the detection model and incorporating appropriate configuration parameters.

A question that arises is whether a general multi-user model can achieve the same level of detection accuracy when being used on several machines. To test this, we incorporated all the 423 traces in the learning process and tested the detection results of USBESAFE. We observed that the general model, unsurprisingly, produced a higher false positive rate on the labeled dataset. Table 3 summarizes the detection accuracy of USBESAFE in the per user and multi-user scenarios. While the general model achieved a lower detection rate with a higher false positive rate, we observed that it can be deployed temporarily on new machines while the per user model is in the training phase. We provide more details on the real-world deployment of USBESAFE in Section 6.

5.3.1 Feature Set Analysis

We also performed an experiment to measure the contribution of the proposed features by testing the model with the labeled datasets collected from all the five machines, and calculating the average of TPs and FPs. To this end, we used a *recursive feature elimination* (RFE) approach on the labeled dataset. We divided the feature set into three different categories: Type-based features which are transfer and event type of packets (F_1), Time-based features which are interarrival and post-enumeration time of the packets (F_2), and Content-based feature which is the payload of the packets (F_3). The

procedure started by incorporating all the feature categories while measuring the FP and TP rates. Then, in each step, a feature set with the minimum weight was removed, and the FP and TP rates were calculated by performing 4-fold cross-validation to quantify the contribution of each feature in the proposed feature set. Table 4 provides the details of feature set evaluation using One-class SVM with the configuration parameters we tested in Section 5.3.

Our experiments show that the highest false positive rate is 43.4% and is produced when USBESAFE only incorporates type-based features. When time-based and content-based features were used together, USBESAFE achieved (1.8% FP with 94% TP). F_{23} resulted in higher detection rate as USBESAFE was able to detect evasive scenarios where we intentionally imposed an artificial delay, similar to stalling code in malware attacks, before launching a command injection attack. When all the features were combined, USBESAFE achieved (0.21% FP with 95.7% TPs) on labeled dataset. Note that if USBESAFE uses a larger window size ($n = 3$), it is possible to achieve 100% TPs. However, it results in higher false positive cases as the number of suspicious sequence of USB packets also increases. Therefore, as a design decision, we decided to use the window size ($n = 2$). We provide more details in Section 5. The results clearly imply that USBESAFE achieves the highest accuracy by incorporating all the features.

Feature Sets	FPs	Tps
F_1	43.4%	54.7%
F_2	14%	78%
F_3	16%	69%
F_{12}	2.2%	86.3%
F_{13}	5.6%	65%
F_{23}	1.8%	94%
All Features	0.21%	95.7%

Table 4: The true positive and false positive rate for different combinations of features. The analysis shows that USBESAFE achieves the best results by incorporating all the features.

We performed another experiment to rank the relative contribution of each feature. We first incorporated all the features, and measured the FP and TP rates. Then, in each step, we removed the feature with the minimum weight, and calculated the FP and TP rates to quantify the contribution of each feature. Table 5 shows the results by ranking all the features with the most significant one at the top. For easier interpretation, we calculated the score ratio by dividing the score value of each feature with the most significant score value. The ratio of each feature simply tells how much the corresponding feature can contribute to identify novel observations.

5.3.2 Modeling the USB Traffic Pattern

A question that arises here is how URB arrivals can be modeled. This is an important question as we want to test the possibility of developing mimicry attacks where an adversary can bypass the proposed detection mechanism. For example, an attacker can create BadUSB attacks that generate URBs

Rank	Category	Feature	Type	Score Ratio
1	Time	Packet Interarrival Times	Continuous	100%
2	Content	Packet Payload	Ordinal	83.2%
3	Time	Post-enumeration Time	Continuous	35.6%
4	Type	Event Type	Categorical	14.4%
5	Type	Transfer Type	Categorical	12.1%

Table 5: The rank of each feature in USBESAFE to detect novel observations.

which are similar to a normal user typing pattern. Prior work revealed that user-generated traffic arrivals such as Telnet can be well modeled as Poisson distribution [18]. To test whether the URB arrivals follow Poisson distribution, we ran a simple statistical methodology where we tested whether the URB arrivals follow *exponentially distributed* and *independent* interarrivals – the two requirements for Poisson distribution.

To this end, we randomly selected 100 traces from the labeled dataset. Figure 2 represents the results of the analysis. The x-axis represents the percentage of the intervals in the traces that follow exponentially distributed interarrivals and the y-axis represents the percentage of the intervals that follow independent interarrivals. We used Anderson – Darling test [1] to verify whether the interarrivals follow an exponential distribution. To test the interarrivals for independence, one simple way is to check whether there is significant autocorrelation among URB arrivals in a given time lag. To this end, we used Durbin – Watson statistics [29] to test the autocorrelation among URBs. As shown in the figure, more than 95% of the intervals pass the test showing that the URB arrivals are truly Poisson. We use this finding to generate mimicry attacks and test whether the system can detect attacks that follow Poisson arrivals (see Appendix B).

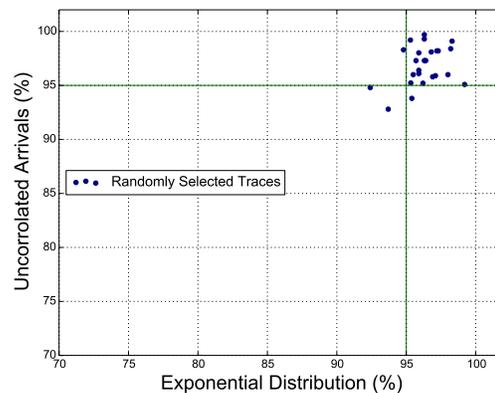
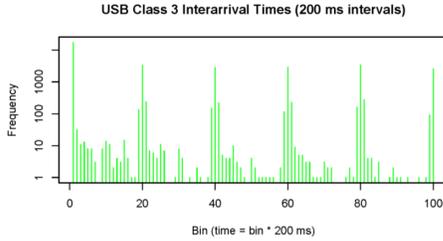


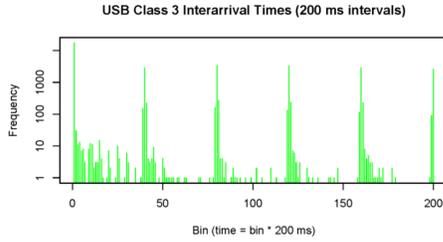
Figure 2: The result of statistical analysis on 100 randomly selected traces. Our analysis shows that the URB arrivals can be well modeled by Poisson arrivals.

5.3.3 Determining the Effect of Pause Time

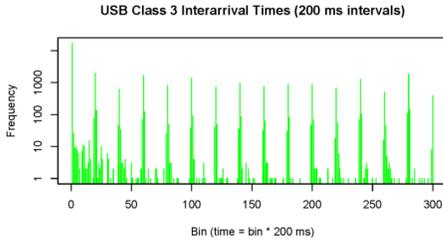
As mentioned in Section 3.1, to tackle the issue of the unbounded interarrival time value between two consequent USB



(a) Payload histogram on a 20,000 ms pause with bin intervals of 200 ms.



(b) Payload histogram on a 40,000 ms pause with bin intervals of 200 ms.



(c) Payload histogram on a 60,000 ms pause with bin intervals of 200 ms.

Figure 3: The effect of pause time value on payload histogram. The figures show localized modality occurs approximately every 4,000 ms with large spikes.

packets, we defined two configuration parameters: pause time and session. A session is a series of USB packets where the interarrival time value within the series does not exceed a specified pause length. To determine the impact of pause time value, we performed a set of experiments. The experiments had multiple goals: (1) to empirically characterize the distribution of the benign interarrival time values; (2) to find out whether varying the pause time value has any impact on the volume of information in each session as well as n-gram construction; and (3) to define the maximum interarrival time value between two TraceEvents before we consider the user to be starting a new typing session. To determine an optimal pause time, we examined three pause time candidates (in milliseconds): 20,000, 40,000, and 60,000 milliseconds, with a sampling period of 200 and 500 milliseconds. For each pause time value, we normalized the values for class codes 0 and 3.

More specifically, when a raw interarrival time value i was greater than the pause time, we reset i to 0, thereby starting a new session.

We observed some common patterns by generating payload histograms while varying pause values and interval lengths. An interesting observation for the HID traffic was that, regardless of the pause time value, a localized modality occurs approximately every 4000 ms, or 4 seconds, with large spikes in the number of packets transmitted during these times. Ultimately, the results of this experiment revealed that there was minimal information payload differences among the pause time values used, indicating that the value we chose is not consequential to overall model performance. For this reason, we set the pause to our lowest value of 20,000 ms. Figure 3 shows the details of this experiment.

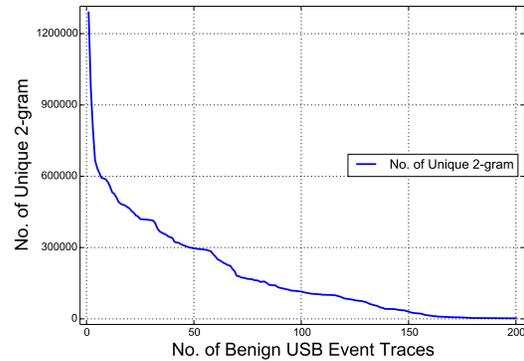


Figure 4: Unique 2-grams for the first 200 USB packet traces in our dataset. The number of unique sequences significantly decreases as USBESAFE observes more USB packets.

5.3.4 Determining the Effect of N-Grams

To understand the diversity of the collected USB packets for the USB HID class, we performed an experiment on constructing n-grams by varying the value of n from 2 to 3. First, we examined the number of unique 2-grams that can be found in the first 200 USB trace files which contained 5,938,492 USB packets. The number of unique 2-grams on labeled dataset is shown in Figure 4. As depicted, the number of unique sequences significantly decreases as USBESAFE observes more USB packets. The finding suggests that n-grams can closely capture the characteristics of the benign dataset. That is, if the model is deployed, it is unlikely that benign keyboard activities will not have been observed in our training phase, resulting in low false alarms.

To verify this, we performed an experiment that incorporated the entire labeled dataset that is a representative mix of possible BadUSB attacks as well as benign USB HIDs. We varied n and the threshold k of malicious n-grams that need to be observed before a USB device is flagged as malicious. The results for $n = 2$ and $n = 3$ and k ranging over an interval

from 1 to 50 are evaluated. Figure 5 shows the results of the analysis. As depicted, the detection rates are very high, especially for small values of k . The false positive rate is 0.21% for $k = 3$.

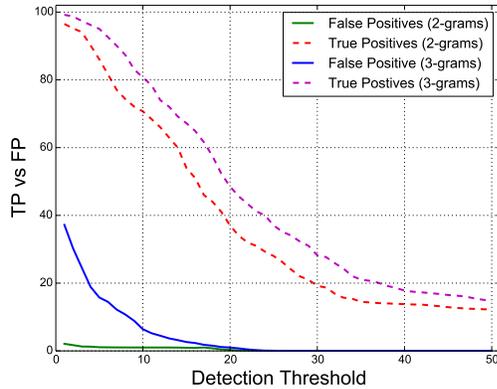


Figure 5: Detection results for 2-grams and 3-grams. The detection threshold k is on the X-axis (e.g., $k = 2$ and $n = 3$ means that a USB trace must match two 3-grams to generate an alert).

6 Real-world Deployment

The main goal of this experiment is to evaluate the detection accuracy of USBESAFE by incorporating an unlabeled dataset which has not been observed during the training phase. We incorporated the results of our previous measurement on the labeled datasets by setting the window size and number of n -grams to established values ($n = 2$ and $k = 3$). For a real-world deployment, we first need to determine how much data is required for initial model training if a new user decides to use USBESAFE as a service. To answer this question, we ran an experiment on seven new machines for 20 days. Depending on the type of machines and their usage, multiple HID and non-HID devices were connected to the machines. This resulted in generating different numbers of trace files per machine. Therefore, for easier interpretation, we performed the tests by varying the number of days, referred to as the training window size, instead of the number of trace files as five out of seven machines had more than one trace file per day. We also generated five new BadUSB attacks (e.g., establishing covert channels, logging keyboard activity) for testing.

To run the test, we varied the training window size from 1 to 20 days for all the machines and computed TP and FP rates to determine the optimal training window size. The result of this experiment showed that USBESAFE requires two to four training days to keep the TP rate over 93% with a 0.9% FP rate in all the machines. Our analysis on the training window size also showed that the machines with two connected HID devices (a keyboard and a mouse) do not usually need

more than three training days to reach that level of detection accuracy.

We ran another experiment to test whether the general model that was constructed based on our labeled 423 trace files would work well in the new machines. We observed that USBESAFE achieved on average 90% TP rate with a 2.2% FP rate across all the machines. In fact, per user deployment model achieved a higher detection rate with a significantly lower false positive rate (FP = 0.9%) at the cost of two to four training days. However, since the general model does not require any initial training for a large-scale deployment, we recommend temporarily activating the general model on a new machine while USBESAFE is in the training phase.

We also deployed the general model on three multi-user machines for five consecutive days. We did not receive any complaint from users during the test period. However, we cannot provide any strong security guarantees to protect multi-user machines or produce low false positive rate as we do not have enough data to make any statistically significant claim on the accuracy of USBESAFE for this deployment option. Furthermore, recall that one of the main design goals of USBESAFE was to reduce the risk of BadUSB attacks – a form of targeted attacks on end-users. Consequently, the architecture, feature selection, and implementation details make USBESAFE a more effective solution for single user machines. In fact, protecting multi-user machines has a different set of security and privacy requirements and is out of the scope of this paper.

6.1 Re-training the Detection Model

USBESAFE should counter the problem of *model drift*, in which the constructed model makes an assumption that the incoming USB packets will exhibit *new normal patterns* that have not been observed during the training phase. For example, users' typing patterns can change for various reasons (e.g., completing a specific task) or URBs interarrival rate might change across different devices which might affect the detection accuracy. Therefore, a practical deployment of USBESAFE requires periodically re-training the system. To simulate a practical deployment, we started an experiment by training USBESAFE on all the new machines and tested with the attack payloads we developed. Our analysis shows that, based on the labeled dataset and subsequent data collection, training USBESAFE with an initial dataset similar to ours and re-training every 16 days were sufficient to maintain the detection rate over 93% with less than 1% false positive cases across all the machines.

The re-training process, including the false positive and false negative analysis, usually took on average 2.1 hours each time during the course of experiment. More specifically, the time needed to re-train the model and address model drift was a function of the size of input data which took approximately 82 seconds on average every 16 days on normal PCs and laptops. However, the manual intervention for evaluating the results was almost inevitable. We had to verify *how* and *why*

false positive or false negative cases occurred, and whether they were produced as a result of model drift or an evasive attack. In a real-world deployment, USBESAFE requires only the re-training schedule which is less than two minutes. In Section 7, we provide more details on the risk of adversaries' malicious influence during the re-training process.

6.2 Evaluating False Positives

During 20 days of experiment, the system processed 3,434,452 USB packets across seven machines. To speed up the false positive analysis, we asked the users to log the number of times, the exact time and date they received the system's alert. We received false positive reports on two machines. A more in-depth analysis revealed that all the false positive cases in one of the machines were produced in two consecutive days when the user was filling out a set of web forms with random data for research purposes. USBESAFE detected these USB packets as new observations because the payload histograms as well as the interarrival time values among the URBs were following a significantly different pattern with the novelty score 32%.

We observed that the false positive cases on the other machine was because of running a user study experiment in which several users were asked to run a test on the perceived functionality of websites by interacting with them while triggering their event listeners. A few users in that experiment were typing random characters in multiple fields of web forms in those websites. In fact, the false positive cases in both machines were very similar in a sense that they were flagged by USBESAFE when the users performed a set of activities that did not match with their normal interaction with the machines. We did not encounter any other cases of legitimate USB packets being incorrectly reported. These results are in fact quite encouraging as the experiment was performed on a set of new machines with relatively small training window size compared to our first experiment without imposing a discernible impact on the detection accuracy of USBESAFE.

7 Discussions and Limitations

Note that a fundamental design goal of USBESAFE is to keep the protection mechanism completely in the background. We assume that adversaries have significant freedom in providing varying responses for device identities to evade potential defense mechanisms. Furthermore, adversaries can convince users to connect seemingly benign devices to hosts for various reasons. Consequently, shifting the burden of responsibility to users to verify the reported identity, and decide on *unknown* devices is less likely to be a very reliable defense mechanism. In this section, we discuss the limitations of USBESAFE, and the implications of these limitations on the detection results.

First, recall that USBESAFE is an anomaly-based detection system where the detection results depend on the quality and volume of the trained dataset. If an attack occurs during the learning phase, USBESAFE accepts data or behavior that

would otherwise be considered malicious. Therefore, an additional analysis should be performed on the authenticity of the new data for re-training purposes to prevent such malicious influences. This may increase the cost of the data collection as the proposed model is a per user solution. Furthermore, as mentioned earlier, an attacker can try to imitate benign USB traffic patterns and evade the detection mechanism. An attacker can be successful in running these attacks, if she is able to accurately learn the typing behavior of the target user. Our analysis shows that automatically injecting artificial delays (See Appendix B) can decrease the novelty score of USB traffic. However, it cannot entirely change the traffic patterns, or possibly adapt to each user typing pattern.

Second, recall that one of the primary design decisions of USBESAFE is to treat existing operating systems in a black box fashion, and build a central security model of USBESAFE independent of the user's perception of malice. However, USBESAFE cannot provide strong protection guarantees against scenarios where an adversary attempts to trick users into voluntarily disabling USBESAFE, for instance, by mimicking the output of USBESAFE, and forcing the user to disable protection. We stress that these issues are fundamental to any host-based protection tool.

Third, USBESAFE cannot provide any security guarantees in scenarios where an adversary has a privilege to run code in the kernel. In fact, if an adversary can successfully run malicious code in the kernel, she can also disable all the possible defense mechanisms, including USBESAFE. For this reason, we explicitly consider kernel-level attacks outside the scope of USBESAFE's threat model. Despite all the limitations, USBESAFE provides important practical security benefits that complement the standard USB protocol employed in operating systems without any significant detriments to performance.

8 Conclusion

In this paper, we empirically show that it is possible to develop models that can accurately explain the nature of USB traffic. We presented the design and implementation of USBESAFE, and demonstrated that it can successfully block modern BadUSB-style attacks without relying on end-user security decisions or requiring changes in the current USB protocol or the operating system. We hope that the concepts we propose will be useful for end-point protection providers and facilitate creating similar services on other platforms to enhance defense mechanisms against future malicious devices.

Acknowledgements

This work was partially supported by the Office of Naval Research (ONR) under grant N00014-19-1-2364 award and the United States Air Force under Air Force Contract No. FA8702-

15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

References

- [1] *Anderson–Darling Test*. Springer New York, New York, NY, 2008, pp. 12–14.
- [2] USB Rubber Ducky. <https://hakshop.com/products/usb-rubber-ducky-deluxe>, 2017.
- [3] ANGEL, S., WAHBY, R. S., HOWALD, M., LENERS, J. B., SPILO, M., SUN, Z., BLUMBERG, A. J., AND WALFISH, M. Defending against malicious peripherals with cinch. In *USENIX Security Symposium* (2016).
- [4] BATES, A. M., LEONARD, R., PRUSE, H., LOWD, D., AND BUTLER, K. R. B. Leveraging USB to establish host identity using commodity devices. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23–26, 2014* (2014).
- [5] BROCKER, M., AND CHECKOWAY, S. iseeyou: disabling the macbook webcam indicator led. In *Proceedings of the 23rd USENIX conference on Security Symposium* (2014), USENIX Association, pp. 337–352.
- [6] DIWAN, S., PERUMAL, S., AND FATAH, A. Complete security package for usb thumb drive. *Computer Engineering and Intelligent Systems* 5, 8 (2014), 30–37.
- [7] HERNANDEZ, G., FOWZE, F., YAVUZ, T., BUTLER, K. R., ET AL. Firmusb: Vetting usb device firmware using domain informed symbolic execution. *ACM Conference on Computer and Communications Security* (2017).
- [8] JIM WALTER. “Flame Attacks”: Briefing and Indicators of Compromise. http://downloadcenter.mcafee.com/products/mcafee-avert/sw/old_mfe_skywiper_brief_v.1.pdf.zzz, 2012.
- [9] KARSTEN NOHL, SACHA KRIBLER, JAKOB LELL. BadUSB—On accessories that turn evil. *BlackHat*, 2014.
- [10] KHARAZ, A., ARSHAD, S., MULLINER, C., ROBERTSON, W., AND KIRDA, E. UNVEIL: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 757–772.
- [11] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS ’11, ACM, pp. 285–296.
- [12] LEARN, S. Anomaly detection with Local Outlier Factor (LOF). http://scikit-learn.org/stable/auto_examples/neighbors/plot_lof.html.
- [13] LEARN, S. Nearest Neighbors. <http://scikit-learn.org/stable/modules/neighbors.html>.
- [14] LEARN, S. One Class SVM. <http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html>.
- [15] LETAW, L., PLETCHER, J., AND BUTLER, K. Host identification via usb fingerprinting. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on* (2011), IEEE, pp. 1–9.
- [16] NEUGSCHWANDTNER, M., BEITLER, A., AND KURMUS, A. A transparent defense against usb eavesdropping attacks. In *Proceedings of the 9th European Workshop on System Security* (2016), ACM, p. 6.
- [17] NICOLAS FALLIERE, LIAM O MURCHU, ERIC CHIEN. W32. Stuxnet Dossier. <http://www.bbc.com/news/technology-36478650>, 2011.
- [18] PAXSON, V., AND FLOYD, S. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on Networking (ToN)* 3, 3 (1995), 226–244.
- [19] PHAM, D. V., HALGAMUGE, M. N., SYED, A., AND MENDIS, P. Optimizing windows security features to block malware and hack tools on usb storage devices. In *Progress in electromagnetics research symposium* (2010), pp. 350–355.
- [20] RICH, D. Authentication in transient storage device attachments. *Computer* 40, 4 (2007).
- [21] SCHUMILO, S., AND SPENNEBERG, R. Don’t trust your usb! how to find bugs in usb device drivers.
- [22] SHIN, S., AND GU, G. Conficker and beyond: a large-scale empirical study. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010), ACM, pp. 151–160.
- [23] TETMEYER, A., AND SAIEDIAN, H. Security threats and mitigating risk for usb devices. *IEEE Technology and Society Magazine* 29, 4 (2010), 44–49.

- [24] TIAN, D. J., BATES, A., AND BUTLER, K. Defending against malicious usb firmware with goodusb. In *Proceedings of the 31st Annual Computer Security Applications Conference* (2015), ACM, pp. 261–270.
- [25] TIAN, D. J., BATES, A., BUTLER, K. R., AND RANGASWAMI, R. Provsb: Block-level provenance-based data protection for usb storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16, pp. 242–253.
- [26] TIAN, D. J., SCAIFE, N., BATES, A., BUTLER, K., AND TRAYNOR, P. Making usb great again with usbfilter. In *Proceedings of the USENIX Security Symposium* (2016).
- [27] TIAN, J., SCAIFE, N., KUMAR, D., BAILEY, M., BATES, A., AND BUTLER, K. Sok: "plug & pray" today - understanding usb insecurity in versions 1 through c. In *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, pp. 613–628.
- [28] TISCHER, M., DURUMERIC, Z., FOSTER, S., DUAN, S., MORI, A., BURSZEIN, E., AND BAILEY, M. Users Really Do Plug in USB Drives They Find. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P '16)* (San Jose, California, USA, May 2016).
- [29] WATSON, G. S., AND DURBIN, J. Exact tests of serial correlation using noncircular statistics. *The Annals of Mathematical Statistics* (1951), 446–451.
- [30] YANG, B., QIN, Y., ZHANG, Y., WANG, W., AND FENG, D. Tmsui: A trust management scheme of usb storage devices for industrial control systems. In *International Conference on Information and Communications Security* (2015), Springer, pp. 152–168.
- [31] ZHUANG, L., AND DAI, H. Parameter optimization of kernel-based one-class classifier on imbalance learning. *Journal of Computers* 1, 7 (2006), 32–40.

A Model Search Configurations

Table 6 represents all the possible configurations for training the detection model.

Configuration Setting	Values	#
Error Upper Bound (v)	[0.01, 0.25, 0.5, 0.75, 1]	–
Kernel Coefficient (γ)	[0.1, 0.01, 0.001, 0.0001]	–
Degree of Polynomial	[1, 2, 3]	–
Kernel Functions	RBF, v, γ	20
	Sigmoid, v, γ	20
	Linear, v	5
	Polynomial, v, γ, degree	60
Total		105

Table 6: All combinations of parameters for any applicable v , γ , and degree settings for each kernel option. Defining this parameter space results in 105 parameter settings to apply to SVM instances.

B Case Studies

As mentioned earlier, an attacker has significant freedom in developing malicious code that can potentially bypass USB-SAFE. Therefore, as an end-point solution, it is quite useful to study how the system responds to different levels of attack sophistication. To this end, we ran each of the following attacks, collected the corresponding USB traces, and measured the percentage of USB packets in each attack that was novel to the system based on the model learned on each machine.

Attack No. 1: Running a Malicious Payload By running a malicious payload, we specifically focus on executing commands to call a binary that downloads code from the Internet, and installs malware. Note that this attack can be designed to be as stealthy as possible. For example, the malicious code can start when the user is logged off with the assumption that the user is very likely not physically present.

Our analysis showed that this attack had an average novelty score of 47.9% when tested with different learned models. Our further investigation revealed that the USB packets received a relatively high novelty score compared to the learned model because the interarrival time values among URBs was

Machine	Attack1	Attack2	Attack3
Machine6	63.2%	58.2%	37.3%
Machine7	54.5%	52.5%	42.4%
Machine8	49.8%	40.8%	19.2%
Machine9	31.5%	17.4%	30.8%
Machine10	41.2%	42.8%	30.6%
Machine11	46.5%	47.6%	29.8%
Machine12	49.1%	49.3%	33.6%
Average	47.9%	44.1%	27.1%

Table 7: the novelty score of the evasion tests in the real-world deployment. The novelty score of all the attacks are significantly higher than the threshold value ($t = 13.2\%$).

significantly smaller than most of real user typing behaviors. Furthermore, the 2-gram analysis shows that the average content histogram of the first 103 request packets were more than 195 during the command injection which was significantly higher than the content of the USB packets in the benign dataset.

Attack No. 2: Adding Artificial Delays A question that arises is that in the previous attack, the malicious code launched a list of commands immediately after the enumeration phase. We updated the code to wait for a random period of time similar to the stalling code in malware attacks [10, 11], and then open a terminal to run the commands. Our analysis revealed that this attack could bypass the post-enumeration feature by waiting for a random period of time before running the commands. As shown in Table 7, compared to attack No. 1, the novelty score of the malicious code decreased in all the machines. However, USBESAFE reported this attack as a new observation as the interarrival of the packets was still too small.

Attack No. 3: Manipulating the Interarrival Times We enhanced the attack payload to be more stealthy by adding delays among the injected commands in order to simulate human typing patterns. The delays were injected such that the arrivals of URBs followed Poisson distribution. We used Poisson distribution as we observed in Section 5 that the URBs' interarrivals in our labeled dataset can be well-modeled using Poisson. While the novelty score of the USB traffic in attack 3 (see Table 7) is relatively lower than the novelty score of the other attacks, the attack is still detected since the novelty scores of the USB traffic in all the traces are significantly higher than the pre-defined threshold ($t = 13.2\%$). Further analysis suggests that injecting artificial delays with Poisson distribution during the command injection phase is not sufficient to automatically generate very serious mimicry attacks that perfectly resemble users' typing patterns. In fact, we empirically found that to successfully run such attacks, the adversary needs a more precise mechanism to learn the normal typing behavior of individual users. This makes crafting mimicry attacks more complicated as the adversary has to incorporate other techniques to reliably hook certain OS functions in order to learn the typing pattern of each user. This particular area has been studied extensively in malware detection, i.e., spyware detection, and is out of the scope of this paper.

C Benchmarks

Since USBESAFE is intended as an online monitoring system, it may impact the performance of other applications or the operating system. We expect USBESAFE's performance overhead to be overshadowed by I/O processing delays, but

in order to obtain measurable performance indicators and characterize the overhead of USBESAFE, we ran experiments that exercised the critical performance paths of USBESAFE. Note that designing custom test cases and benchmarks require careful consideration of factors that might influence our runtime measurements. In these tests, we mainly focused on the core parts of the USB device communication which were the *USB device enumeration* and *data transfer* mechanisms. We explain each of these benchmarks in more details below.

Device Enumeration In the first experiment, we tested whether USBESAFE introduces any noticeable performance impact during the USB enumeration phase. The testing USB device was a headset which had a HID interface. We manually plugged the headset into the host 20 times and compared the results between USBESAFE-enhanced host and the standard machine. The average USB enumeration time was 37.4 ms for the standard system and 39.1 ms for the USBESAFE-enhanced host respectively. Comparing to the standard host, USBESAFE only introduced 4.5% or less than 2 ms. We created the same benchmark using a mouse, and repeated it for 20 times. The system imposed 4.1% or 1.4 ms for device enumeration. The measurement results imply that USBESAFE does not have a significant impact on the enumeration of USB devices. More details are provided in Table 8.

USB Packet Inspection In the second experiment, we created a benchmark to characterize the performance overhead of our system during a normal device use. To measure the overhead of the detection model, we plugged in a USB optical mouse and moved it around to generate USB traffic. We then measured the time used by USBESAFE to determine whether the incoming USB packets should be filtered or not. The required time is calculated from the time a URB is delivered to the packet inspection subsystem to the time the packet is analyzed by the protection engine. We tested the experiment on the first 2,000 URBs, and repeated the experiment 10 times as shown in Table 8. As shown, the average cost per URB is 12.7 μ s, including the time used by the benchmark to get the timing and print the results.

File System We also created a benchmark to measure the latency of file operations under the baseline and USBESAFE-enabled machines. The goal of the experiment is to measure the performance overhead of the system during normal usage of a USB storage device, where users plug in flash drives to copy or edit files. We ran the experiments using a 16 GB USB flash drive and varied file sizes from 1 KB to 1 GB. Each test was done 10 times and the average was calculated. As shown, the throughput of USBESAFE is close to baseline when the file size is less than 100 MB (approximately 3.9%). When the mean file size becomes greater than 100 MB, USBESAFE shows lower throughput compared to the standard machine

Experiment	Device	Standard	USBESAFE	Overhead
Enumeration	<i>Head Set</i>	37.4 ms	39.1 ms	4.5%
	<i>Mouse</i>	33.5 ms	34.9 ms	4.1%
	<i>Keyboard</i>	34.2 ms	35.6 ms	4.2%
	<i>Mass Storage</i>	36.6 ms	38 ms	3.9%
Overhead Mean		35.4 ms	36.8 ms	4.2%
Event Inspection	<i>Mouse</i>	-	12.3 μ s	-
	<i>Keyboard</i>	-	13.1 μ s	-
Overhead Mean (per packet)		-	12.7 μ s	-

Table 8: USBESAFE’s overhead on the USB communication protocol. USBESAFE imposes on average 4.2% overhead during the enumeration phase and 12.7 μ s per packet during USB packet inspection.

as a result of pattern monitoring on the bus. The results show that USBESAFE imposes 7.2% and 11.4% overhead when the mean file sizes are 100 MB and 1 GB respectively. For example, if a user wants to copy 10 100 MB files, throughput would drop from 8.9 MB/s to 8.26 MB/s when USBESAFE is enabled on the user’s machine.

Minimal Kernel: An Operating System Architecture for TEE to Resist Board Level Physical Attacks

Shijun Zhao¹, Qianying Zhang^{2,*}, Yu Qin¹, Wei Feng¹, Zhining Lv³, and Dengguo Feng¹

¹*Institute of Software, Chinese Academy of Sciences, Beijing, China*

²*College of Information Engineering, Capital Normal University, Beijing, China*

³*Shenzhen Power Supply Bureau Co., Ltd., Shenzhen, China*

Abstract

ARM specifications recommend that software residing in TEE's (Trusted Execution Environment) secure world should be located in the on-chip memory to prevent board level physical attacks. However, the on-chip memory is very limited, placing significant limits on TEE's functionality. The *minimal kernel* operating system architecture addresses this problem by building a small kernel which executes the whole TEE system only on the on-chip memory on demand and cryptographically protects all the data/code stored outside of SoC. In the architecture, a small kernel is built inside the TEE OS kernel space and achieves the minimal size by only including the very essential components used to execute and protect the TEE system. The minimal kernel consists of a minimal demand-paging system, which sets the on-chip memory as the only working memory for the TEE system and the off-chip memory as a backing store, and a memory protection component, which provides confidentiality and integrity protection on the backing store. A Merkle tree based memory protection scheme, reducing the requirement for on-chip memory, allows the minimal kernel to protect large trusted applications (TAs). This OS organization makes it possible to achieve the goal of physical security without losing any TEE's functionality. We have incorporated a prototype of minimal kernel into OP-TEE, a popular open source TEE OS. Our implementation only requires a runtime footprint of 100 KB on-chip memory but can protect the entire OP-TEE kernel and TAs, which are dozens of megabytes.

1 Introduction

As the rapid development of mobile commerce and mobile applications for enterprises, such as BYOD (Bring Your Own Device), mobile devices store and process more and more security-sensitive data. To improve the security of mobile devices, ARM proposes the TrustZone security extension to its CPU architecture. By now, TrustZone technology has

been widely studied [4, 30, 31, 33, 47, 50–52], and deployed in various of commercial products: almost all mainstream smartphone OEMs have integrated TrustZone in their products and leverage it to provide security services, such as Samsung pay, Huawei pay, and KNOX [46].

As TrustZone is becoming a popular hardware security architecture for mobile devices, it is important to ensure the security of TrustZone itself. TrustZone is designed to only resist software attacks and cannot resist physical attacks. Unfortunately, the feature that mobile devices can easily be stolen or lost puts sensitive data stored in TrustZone at the risk of physical attacks. What's worse, there exists a class of physical attacks which are low-cost and easy to set up: board level physical attacks.

Board level physical attacks target sensitive data in DRAM, such as cryptographic keys and passwords. This class of attacks usually is launched by tampering DRAM or snooping memory buses between CPU and DRAM. Several types of board level physical attacks have been developed: cold boot attacks [24], bus snooping attacks, and DMA attacks. There are some well-known successful attacks on commercial products: cold boot attacks on Galaxy Nexus smartphones [40], bus snooping attacks on Xbox [29] and PlayStation 3 [28], and there even exist attacks on security chips such as D-S5002FP [32] and TPM [41]. These attacks are inexpensive because they only requires some cheap tools, such as memory bus probes. Even worse, mature attack tools [14, 16, 40, 42] have been released publicly, with which hackers can reproduce attacks easily.

The way to prevent board level physical attacks is memory encryption. Most CPU manufactures have added this feature in their products, such as Apple's secure enclave technology [2], Intel's SGX [1, 37], IBM's SecureBlue [45] and SecureBlue++ [5]. Take Intel SGX for example, it addresses board level physical attacks by constructing an isolated memory enclave, which is transparently encrypted by a separate memory encryption engine (MEE). However, ARM CPUs, which dominate mobile devices, are not equipped with memory encryption technologies, so TrustZone cannot pre-

*Corresponding author

vent physical attacks. Fortunately, software-based approaches called *SoC-bound execution environments* are proposed, which run applications in CPU registers [17, 38, 39, 48], CPU cache [20, 21, 58, 59], GPU registers and cache [54], or on-chip memory (OCM) [8, 19, 25, 43].

Although the state-of-the-art software-based approaches can prevent some specific board level physical attacks, they cannot protect mature TEE OSes from board level physical attacks for the following two reasons. First, *limited space*: most SoC-bound execution environments can only protect a small piece of code because they require the protected application to be loaded into the execution environment entirely, while the size of CPU registers, CPU cache or on-chip memory is quite limited, usually about a few hundreds kilobytes. However, the size of a mature TEE OS is much larger. Take OP-TEE [35] for example, its kernel address space is about 1 megabyte, and its user address space can reach up to a few megabytes. Therefore, it is impossible to load the whole TEE software into such space-limited execution environments. Second, *insecurity under full power board level physical attacks*: board level physical attackers have abilities of intercepting information and injecting code via the buses between CPU and DRAM [12, 13], so they can inject malicious code into the address space where the SoC-bound execution environment is contained. Unfortunately, this security problem is not considered by most approaches. In most approaches, the address space containing the SoC-bound execution environment is the kernel space. Only the SoC-bound execution environment is located in the memory of SoC, and the other code and data of the kernel address space is located in DRAM in plaintext. An attacker can launch his attack as follows (Figure 1): when CPU runs code outside the SoC-bound execution environment, it will fetch the code from DRAM, and the attacker replaces the code transmitted on the bus with his malicious code. The injected malicious code is able to access sensitive data in the SoC-bound execution environment because they are in the same address space (kernel space).

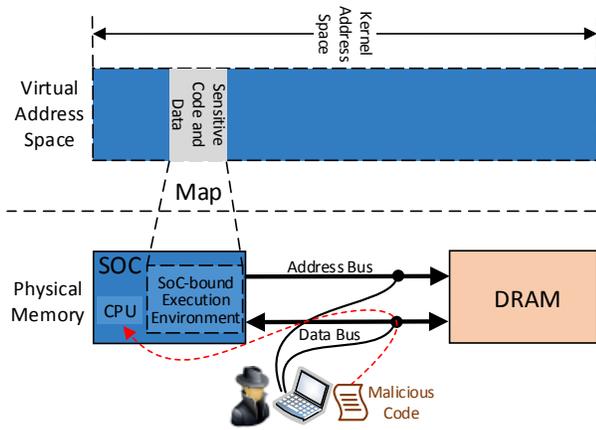


Figure 1: An Injection Attack against SoC-bound Execution Environment

The bus injection attack (Figure 1) shows that, under full power board level physical attacks, if some part of the address space where the SoC-bound execution environment resides is mapped to DRAM without memory protection, attackers can inject malicious code into the data bus when CPU fetches code from the unprotected address space. Thus, it is necessary to provide memory protection on the address space that is mapped to DRAM. Many approaches [17, 20, 21, 38, 39, 48] implement SoC-bound execution environments as kernel modules. Therefore, the rest kernel address space, which is located in the DRAM, should be protected. However, most approaches do not provide memory protection on the rest kernel address space. Some approaches [25, 59] can prevent board level physical attacks by creating an address space only including the SoC-bound execution environment, which would put a significant limit on the environment's functionality.

To overcome the above restriction, we have designed a new TEE OS architecture, the *minimal kernel*, which leverages the size-limited OCM to provide memory protection on the whole TEE system. The key design feature of the architecture is to extract a *minimal kernel*, consisting of the very essential components required to maintain the execution of the whole TEE system and cryptographically protect the whole TEE system, from the TEE OS kernel; the rest of the TEE OS kernel is called the *main kernel*. In the architecture, the OCM is set as the only working memory and DRAM as a backing store for the TEE system. The minimal kernel (permanently residing in the OCM) leverages the demand-paging mechanism to maintain the execution of the main kernel and TAs (initially stored in the DRAM), and protects the confidentiality and integrity of the main kernel and TAs.

One crucial requirement for the minimal kernel is that it should be as small as possible to reduce its demand for OCM. The minimal kernel also should be self-contained because its execution should not rely any components of the main kernel. To meet the above requirements, we propose a principle for the design of the minimal kernel, whose main idea is that a kernel component is tolerated inside the minimal kernel only if moving it outside the minimal kernel would prevent the CPU from running software normally.

In addition to the demand-paging functionality, the minimal kernel protects the confidentiality and integrity of the main kernel and TAs stored in DRAM. Two integrity protection schemes are designed: a trivial scheme requiring that all the integrity values should be stored in the OCM, and a Merkle tree based scheme that only requires secure storage of the root node but requires more computations. The two schemes offer system designers a trade-off between the size and the performance of the minimal kernel.

Leveraging the above demand-paging and memory protection mechanisms, the minimal kernel architecture protects the whole TEE system against physical attacks but only requires the minimal kernel to reside in the OCM, making it possible to protect mature TEE systems whose sizes are large.

We have implemented a prototype by retrofitting the minimal kernel architecture into a mature TEE OS: OP-TEE [35]. Although OP-TEE’s kernel space reaches up to a few megabytes, the minimal kernel only requires a runtime footprint of 100 KB OCM, proving that our principle is very effective. We deploy the following security policy to protect the main kernel: for code sections, we guarantee their integrity; for data sections, we guarantee both of their confidentiality and integrity. Our prototype also provides memory protection for TAs. One benefit of enabling memory protection in the kernel level is enforcing TA encryption transparently, which reduces programmers’ burden. Our evaluation shows that compared with OP-TEE’s original memory protection solution, *Pager* [36], the minimal kernel architecture not only reduces 44% of OCM requirement but also improves performance greatly. The main contributions of this paper are:

- The minimal kernel architecture that provides a software-based approach to full system encryption for TEE systems. It enables a TEE system with rich functionality to obtain a high level of physical security without requiring specialized memory encryption hardware.
- A principle of building the minimal kernel, by which we can identify the very essential components that are required to maintain the execution of a TEE system and implement the minimal kernel with the minimum amount of software.
- A minimal kernel prototype based on a mature TEE OS: OP-TEE. We implement a minimal kernel for OP-TEE, called mTEE. mTEE resides in the OCM, restricts the execution of OP-TEE within the OCM by swapping pages between OCM and DRAM on demand, and protects the confidentiality and integrity of the whole OP-TEE system.
- Performance evaluations of our prototype under the following memory security policy: protecting the integrity of the code sections and the confidentiality and integrity of the data sections.

The rest of the paper is organized as follows. Section 2 gives the background information related with this paper. Section 3 describes the threat model. Section 4 illustrates the design of the minimal kernel architecture. Section 5 depicts the details of the implementation. Section 6 evaluates the performance overhead. Section 7 discusses how the minimal kernel architecture can be improved by hardware enhancement. Section 8 surveys related work. Section 9 concludes this paper.

2 Background

2.1 The Security of TrustZone

Attacks on computer systems can be classified into software attacks, physical attacks, and side channel attacks, and we do not consider side channel attacks in this paper. Physical attacks refer to board level attacks and chip level attacks. Board level attacks are launched at PCB (printed circuit board)

level and leverage chip wired interfaces, such as the buses between DRAM and CPU. Chip level attacks target at on-chip secrets, and require depackaging of the chip to direct access to its internal components. We do not consider chip level attacks because they require high qualified specialists and specialized equipment such as microprobing and FIB workstations.

The TrustZone technology is able to resist software attacks by isolation. It partitions all resources of the platform (CPU, memory, and peripherals) into two worlds: the secure world and the normal world. TrustZone guarantees that no secure world resources can be accessed by normal world components, so software attacks even compromising the OS in the normal world cannot access resources in the secure world.

As for the physical security, the TrustZone technology does not have any countermeasures against chip level attacks, but it can resist board level attacks if system designers follow the recommendations of ARM TrustZone white paper [3]: storing all the code and data in the secure world in OCM, which is not subject to board level attacks. However, this approach will place significant limits on the functionality of TEE systems because the OCM is quite limited.

2.2 On-chip Memory

The OCM is an important building block in SoC. It has the following two advantages. First, it exhibits better access performance because it connects to the CPU via fast internal connection buses (AXI). Second, OCM is more secure against physical intrusions: it does not expose any physical pins or wires, which could be potentially tapped by board level physical attackers. Although OCM has the above attractive properties, it is prohibitively expensive to support a large OCM in SoC, so the OCM in commodity SoCs is limited.

We perform a survey on the OCM size of some popular platforms that support TrustZone (Table 1). Our survey shows that OCM is a general building block in an SoC, and the OCM of most platforms is about a few hundred kilobytes.

3 Assumptions and Threat model

In this section we first describe the hardware assumptions and threat model, then proves the OCM is secure under board level physical attacks, and at last show that most real-world examples of board level physical attacks are covered by our threat model.

3.1 Hardware Assumptions

We assume the existence of a device-unique symmetric key in each device. The key is generated at manufacture time, and should be stored in the secure storage of SoC, such as processor’s eFuses. This key is common on mobile SoCs, such as the Device-Unique Hardware Key in Samsung’s KNOX.

Platform	CPU (ARM)	OCM	Platform	CPU (ARM)	OCM	Platform	CPU (ARM)	OCM
NXP QorIQ-LS1021A	Cortex A5	128 KB	NXP WaRP7	Cortex A7	256 KB	Zynq 7000 ZC702	Cortex A9	256 KB
NXP i.MX6Q-SDB	Cortex A9	272 KB	Hikey	Cortex A53	72 KB	Zynq UltraScale+	Cortex A53	256 KB
NXP i.MX6UL-EVK	Cortex A7	128 KB	TI DRA7xx	Cortex A15	512 KB	Atmel SAMA5D2	Cortex A5	128 KB
NXP i.MX7-SABRE	Cortex A7	256 KB	TI AM57xx	Cortex A15	512+ ¹ KB	ARM Juno	Cortex A72	128 KB

¹ The '+' symbol means that some SoCs' OCMs are bigger than the size listed in the table.

Table 1: A Survey on the Size of OCM

3.2 Threat Model

We assume that mobile devices are exposed to a hostile environment where board level physical attacks are feasible. The main assumption of the threat model is that only the SoC is resistant to board level physical attacks and thus is trusted, and all components outside of the SoC are vulnerable, including DRAM, address/data buses between CPU and DRAM, I/O devices, and so on. We also assume that the software running in the secure world is trusted because we do not aim to increase the security of TEE regarding software attacks.

Board level physical attacks involve DRAM tampering and CPU-DRAM bus probing, which allow observation of the DRAM contents and injection of arbitrary data/code into buses or directly into the DRAM. In particular, attackers can perform *passive attacks* and *active attacks*. In passive attacks, attackers can intercept traffic between CPU and DRAM by bus probing or directly obtain data in DRAM, breaking memory confidentiality. In active attacks, attackers can actively modify or inject data/code into the bus, breaking memory integrity. Based on how an attacker chooses the injected data, we define three classes of active attacks:

1. *Spoofing attacks*: the attacker exchanges a memory block transmitted on the bus with an arbitrary fake one.
2. *Splicing or relocation attacks*: the attacker swaps a memory block transmitted on the bus with another memory block in DRAM. Such an attack can be viewed as a spatial permutation of memory blocks.
3. *Replay attacks*: a memory block located at a given address is recorded and injected at the same address later. This class of attacks replaces a memory block's value with an older one.

Our model defines a high level of physical security, the same as Intel SGX's. However, most current approaches [8, 17, 20, 21, 38, 39, 43, 48, 54] cannot achieve this security level. The reason is that they only focus on resisting one kind of board level physical attacks, instead of all-sided and comprehensive protection. For example, attackers in most of their models do not have the ability of injecting malicious code into buses. Under the attacker defined in our threat model, their security can be broken by the following spoofing attack: the attacker exchanges a memory block of the kernel – the address space where the SoC-bound execution environment is included – with a memory block containing malicious code designed to steal security data in the secure environment.

3.3 The Physical Security of OCM

We leverage OCM as the working memory for TEE systems, and it is the only memory where TEE software resides in plaintext. So we need to show that OCM is secure against board level attacks. As OCM has no address or data lines at physical pins, board level attackers, which launch attacks through interfaces outside of the SoC, such as off-chip buses and DRAM interfaces, cannot directly attack OCM through the passive and active attacks defined in our threat model. However, there are two types of attacks that can indirectly access OCM by inserting malicious components in the device: cold boot attacks and DMA attacks. Cold boot attacks can access OCM by rebooting the device and launching a malicious memory-dumping kernel that can retrieve memory contents. However, Sentry [8] and paper [60] show that OCM is cleared by BootROM upon boot up. BootROM is burned into the hardware of the device at manufacturing and is immutable. Since BootROM is the first piece of code running on the device, software running after it can only obtain the cleared content. Besides, real-world cold boot attacks usually load the malicious kernel by a non-secure bootloader (e.g., u-boot) [40], which can only access non-secure resources, so the malicious kernel cannot access OCM, which has already been assigned to the secure world by the secure bootloader. DMA attacks can access OCM by sending a DMA request to the DMA controller, but ARM TrustZone is able to prevent DMA attacks by denying all DMA requests from malicious devices of the normal world. Therefore, OCM is secure under DMA attacks.

3.4 Real-world Board Level Physical Attacks

Here we show that our threat model covers the most popular board level physical attacks in the real-world: cold boot attacks, bus probing attacks and DMA attacks.

3.4.1 Cold Boot Attacks

Cold boot attacks exploit the data remanence effect [23] of DRAM to recover sensitive data stored in it. The remanence effect says that memory contents fade away gradually over time. The fading speed slows significantly at low temperatures, so the attacker can retrieve the remaining data by cold-booting the device and re-flashing malicious code which steals sensitive data.

Cold boot attacks are covered in our threat model: the attacker obtains all the data stored in DRAM by performing passive attacks, which allow the attacker to directly read data from DRAM.

3.4.2 Bus Probing Attacks

By attaching a bus probing tool [14, 16, 42] to the bus, attackers can intercept and modify data transmitted on the bus, and even can inject malicious code into the bus. To the best of our knowledge, all current software-based approaches, which only protect a piece of code of a monolithic kernel, are susceptible to bus probing attacks. Although Sentry [8] claims to resist this kind of attacks, actually it only protects applications and the kernel is not protected. Thus, the attacker can inject malicious code into the kernel space, gain kernel privileges, and further compromise the whole OS and applications.

Bus probing attacks are covered in our threat model for the following reason: intercepting and modifying data on the bus, and bus-injecting are exactly the abilities that the passive and active attackers are given in our model.

3.4.3 DMA Attacks

By connecting DMA-capable peripherals to devices, attackers can gain access to physical memory without exploiting vulnerabilities present in software. Many peripherals have been exploited to launch DMA attacks, such as network interface cards [10] and video cards [53].

TrustZone can deny all DMA requests from the normal world, so DMA attacks are unable to access any memory in the secure world.

4 Minimal Kernel Design

In this section, we enumerate the main design goals that the minimal kernel architecture should achieve. Then, we discuss in detail the techniques that we use to achieve our goals.

4.1 Design Goals

Minimal size. The size of the minimal kernel should be minimized for the following reasons. First, OCM is quite limited (Table 1), and most platforms' OCMs are less than 256 KB, so the minimal kernel should be as small as possible to fit into such a small memory. Second, the less memory occupied by the minimal kernel, the more OCM remained as the working memory for TEE software stored in DRAM, which improves the performance of TEE OS and TAs.

Memory protection on the whole kernel address space. Figure 1 has shown that it is not enough to only protect the SoC-bound execution environment: if the whole address space where the environment resides is not protected, attackers

are able to launch board level physical attacks. The minimal kernel resides in the kernel space, so the whole TEE OS kernel space should be protected.

Memory protection on TAs. Usually the TEE OS kernel only provides fundamental functionality, such as memory management and TA session management, while TAs provide rich functionality for users, such as secure storage and fingerprint authentication. So TAs contain much sensitive information such as cryptographic keys and fingerprints, and these valuable data should be protected.

Memory protection in the kernel level. Performing memory protection in the kernel level makes protection transparent to TAs, so TA developers can get rid of entangling with designing security policies on how to protect their data and code, reducing programming burden on developers. Besides, previous TAs can be reused after the TEE OS is retrofitted by the minimal kernel architecture.

4.2 System Overview

4.2.1 Overall Architecture

The minimal kernel architecture splits the kernel space into two parts (Figure 2.a): a minimal kernel mapped to the OCM, and a main kernel initially stored in the DRAM. The minimal kernel, which maintains the execution of the whole TEE system on the OCM and cryptographically protects the code and data stored in DRAM, consists of a *minimal demand-paging system* and a *memory protection component* (Figure 2.b).

The minimal demand-paging system sets the OCM as the only working memory for the TEE system, and sets the DRAM as a backing store for the TEE system. It is composed of a *core of minimal kernel* and an *OCM-DRAM Channel*. Unlike ordinary demand-paging systems, which are components of the kernel and included in the kernel space, the minimal demand-paging system is separated from the main kernel and cannot use any functions of the main kernel, so it should be self-contained and contain all components required by the CPU to maintain execution of the TEE system. Another crucial design requirement for the minimal demand-paging system is that it should contain minimum amounts of components to fit into the size-limited OCM. All the required components comprise the *core of minimal kernel*.

When the CPU accesses code/data of the main kernel or TAs, the *OCM-DRAM channel* component of the minimal demand-paging system automatically intercepts the access and loads the demanded page into the OCM. The reason for explicitly building a software-based channel between OCM and DRAM is as follows: memory protection (encryption/decryption and integrity computation) should be triggered when data is transferred in or out of the trust boundary (i.e., the SoC), but unlike hardware-based memory protection approaches [22, 49], which provide memory protection by interposing a hardware protection engine on the boundary

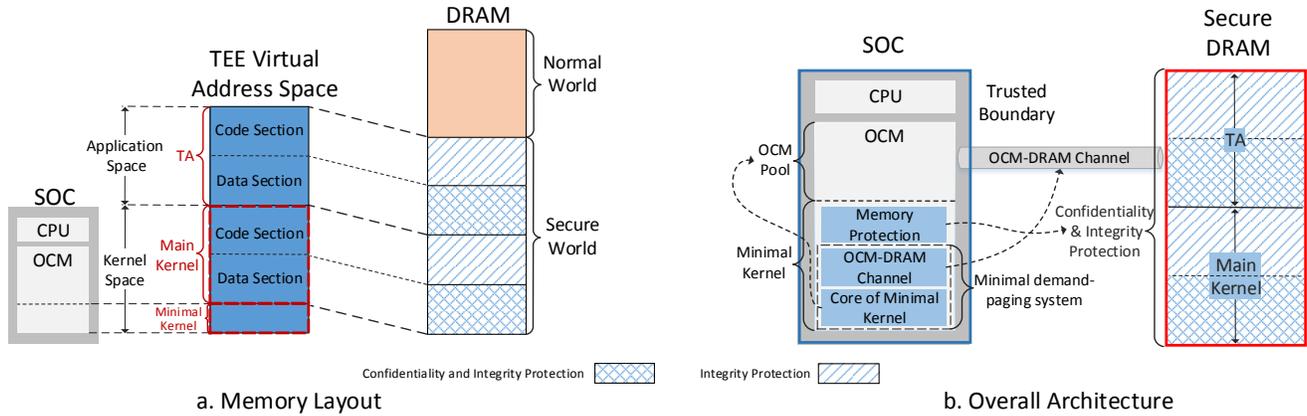


Figure 2: The Memory Layout and Overall Architecture of Minimal Kernel

between cache and DRAM, i.e., the cache-DRAM transmission channel, software-based memory protection approaches cannot leverage the cache-DRAM channel because they are unable to modify hardware, so they have to build a channel between OCM and DRAM in software, based on which memory protection can be performed.

The memory protection component is responsible for protecting the part of the TEE system stored in the backing store. Specifically, it guarantees the integrity of the code to prevent attackers from tampering the software, and it guarantees both the confidentiality and integrity of the data to prevent attackers from obtaining and modifying sensitive information.

4.2.2 Workflow

The workflow of the minimal kernel is as follows (Figure 3).

1. When the CPU accesses data/code of the main kernel or TAs, if the data/code is not in the OCM, the *minimal demand-paging system* automatically intercepts this access. The details of how to intercept the access automatically are described in Section 4.3.2.
2. The *minimal demand-paging system*, which keeps a pool of free OCM frames, allocates an OCM frame from the pool, loads the DRAM page storing required data/code into the OCM frame. During loading, if the virtual address of the required data/code belongs to a code section, the *memory protection component* performs integrity check of the DRAM page; if the virtual address belongs to a data section, the *memory protection component* decrypts the loaded page and performs integrity check.
3. After loading the DRAM page into OCM, the *minimal demand-paging system* changes the page table entry (PTE) which maps the virtual address of the required data/code page, and re-maps the page to the allocated OCM frame. At this time, the page is loaded into OCM thoroughly.
4. The CPU re-accesses the data/code from OCM.
5. When the OCM pool is empty, the *minimal demand-paging*

system picks one page from OCM, frees it, and adds it to the pool. If the selected page is a code page, it is freed directly; if it is a data page, before freeing it, the *memory protection component* encrypts the page, stores the encrypted page to DRAM, computes and stores the hash value of the page in a reserved integrity area of the OCM.

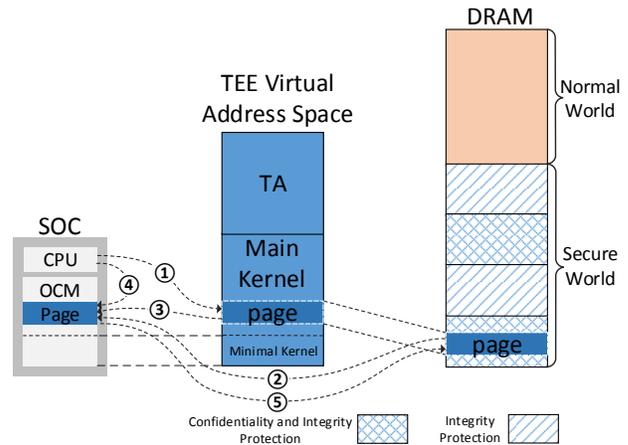


Figure 3: The Workflow of Minimal Kernel

4.3 The Minimal Demand-paging System

The minimal demand-paging system consists of the core of minimal kernel and the OCM-DRAM channel.

4.3.1 The Core of Minimal Kernel

As described in Section 4.2.1, the core of minimal kernel should only contain the very essential components required by the CPU to maintain execution of the TEE system. The difficulty of designing such a kernel lies in how to distinguish all the required components from unnecessary ones. Inspired by the minimality principle for the microkernel architecture

[34], we propose a principle for designing the core of minimal kernel: *one kernel component is tolerated inside the core of minimal kernel only if moving it out would prevent the CPU from running software normally.*

For a component not in the working memory and therefore not accessible to the CPU, if the CPU can access the component later by leveraging the demand-paging mechanism and the delayed-access does not cause software crash, according to the above principle, this component should not be included in the core of minimal kernel. Thus, the core of minimal kernel should only contain the components that must be directly accessed by the CPU to run software. From the above observation, we give the definition of essential components:

Definition 1 (Essential Components) Essential components are components that the CPU has to access directly when running the kernel, and if the CPU fails to directly access these components, the kernel would crash immediately.

From Definition 1, we identify all the essential components as follows, which comprise the core of minimal kernel.

- Page tables. Programs use virtual addresses, and the CPU obtains physical addresses by a virtual-physical address translation. The mappings between virtual addresses and physical addresses are stored in page tables, and the MMU (Memory Management Unit) uses page tables to perform address translations. If page tables are not directly accessible, the CPU will be unable to access any physical memory. So the page tables of the TEE OS are essential components.
- The exception vector table. When an exception occurs, the CPU immediately jumps to the exception handler entry defined in the exception vector table. If the CPU cannot access the exception vector table directly, the kernel will not address any exceptions. So the exception vector table for the TEE OS is an essential component.
- Low level exception handlers. Upon entering into the exception handler, the CPU has to save the context of the non-banked registers immediately. Otherwise, the CPU cannot return back to the point where the interrupt occurs. So the low level exception handlers which save contexts are essential components.
- The kernel stack. One usage of the stack is to store calling conventions. The ARM procedure call standard [11] defines that subroutines should use the stack to receive parameters and return results when there are insufficient argument registers available. As subroutine/function calls are trivial in the kernel, the stack is an essential component.

4.3.2 The OCM-DRAM Channel

The OCM-DRAM channel needs to intercept the data transmission between the CPU and DRAM using software, so it requires a software-controllable mechanism to interrupt the CPU's access to the DRAM data. Based on the mechanism, the OCM-DRAM channel can be implemented by the following steps: first, interrupt the access to DRAM; second,

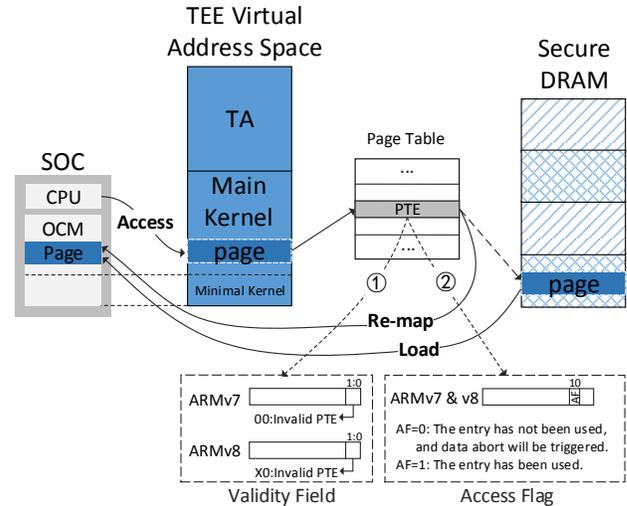


Figure 4: The OCM-DRAM Channel

load the required data into the OCM in the interrupt handler. Fortunately, we find two software-controllable interrupt mechanisms based on the virtual memory technology. When the CPU accesses the DRAM, the MMU first translates the virtual address of the data into a physical address by a hardware page table walk. The translation procedure can be interrupted by configuring any one of the following two fields in the PTE:

- ① Validity field. This field indicates whether the PTE is valid. If the field is set to invalid, when CPU accesses the virtual address mapped by this PTE, the access will cause the CPU to generate a translation fault which will be handled by the data abort exception handler, so the access is interrupted. We describe how to set the field invalid for ARMv7 and ARMv8 architectures in Figure 4.
- ② Access flag (AF). This flag indicates whether the PTE is used for the first time. If this flag is cleared, when the corresponding page is accessed, the CPU will generate an access flag fault and transfer control to the data abort exception handler. In the exception handler, the AF is set, and then the CPU accesses the page normally. Therefore, this flag can be used to interrupt the access to DRAM. One disadvantage of the access flag mechanism is that ARM specifications allow the access flag being managed by hardware [26, 27], in which case the translation procedure cannot be interrupted by software. The details of AF are depicted in Figure 4.

Based on the above interrupt mechanisms, we design the OCM-DRAM channel as follows (Figure 4):

1. When the CPU accesses some data stored in DRAM, the channel interrupts this access by invalidating the validity field or clearing the AF of the PTE mapping the virtual address of the accessed data, and then the CPU jumps to the data abort exception handler.
2. In the data abort exception handler, the channel copies the accessed page into OCM, re-maps the accessed virtual

address to the page in OCM, and makes the page accessible to the CPU by validating the validity field or setting the AF of the PTE.

- When the data abort exception handler completes, the CPU re-accesses the data from the OCM.

4.4 Memory Protection

The memory protection component is triggered when code/data in the main kernel or TAs is transferred into OCM or stored back into DRAM through the OCM-DRAM channel, and it provides memory protection on the main kernel and TAs (Figure 5): for code sections, it protects their integrity to prevent memory tampering; for data sections, it protects their confidentiality and integrity to prevent information leakage and data tampering.

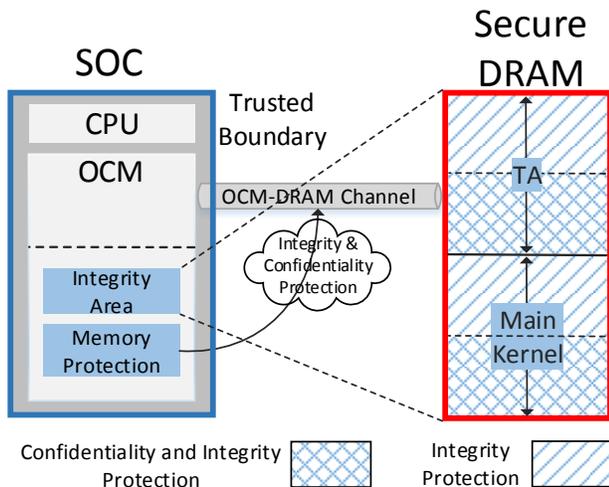


Figure 5: The Memory Protection Component

The memory protection component needs to store the legal integrity values of the main kernel and TAs to perform integrity verification. A direct method is to store all the integrity values of the DRAM pages in the OCM. This method gains high performance because it only needs to compute the integrity value of the loaded page for each integrity verification. Its disadvantage is the large storage for the integrity values. Take a 1M size TA for example, if SHA-256 is used to compute the integrity value, each page requires 32B OCM, and the TA requires 8KB OCM to store its integrity values. If multiple TAs are supported, more OCM will be occupied. The method of Merkle trees [18] can address this disadvantage because it only needs to store the root node of the Merkle tree, but it introduces high performance overhead: it needs to re-compute all the nodes in the path from the leaf to the root of the tree for each integrity verification.

Based on the above two integrity verification methods, we design two memory protection schemes: a trivial memory protection scheme which stores all integrity values in the OCM,

and a Merkle tree based memory protection scheme which leverages the Merkle tree technique to reduce the requirement for OCM and makes it possible to protect large TAs. The two schemes can be used to trade-off between the performance of the minimal kernel and its requirement for OCM.

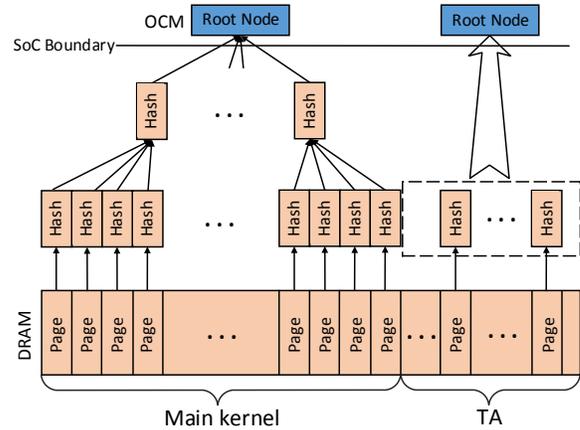


Figure 6: The Merkle Tree Based Integrity Protection

4.4.1 Trivial Memory Protection

The memory protection component allocates an area in the OCM to store integrity values, and uses an encryption key k to protect the confidentiality of the pages in DRAM. The encryption key is derived from the device-unique device key (denoted by k_d): $k = HKDF(k_d, \text{"memory encryption"}, klen)$ where $HKDF$ is an $HMAC$ -based key derivation function whose output length is $klen$.

The memory protection component works as follows. When an OCM page is going to be transferred to DRAM by the OCM-DRAM channel, if the page belongs to data sections, the memory protection component encrypts the page using k , computes its integrity tag and stores the integrity tag in the integrity area, and if the page belongs to code sections, the memory protection component does nothing. When a page is going to be loaded into the OCM, if the page belongs to data sections, the memory protection component decrypts the page using k , computes its integrity tag, and compares it with the legal integrity value, and if the page belongs to code sections, the memory protection component only computes and checks its integrity tag.

4.4.2 Merkle Tree Based Memory Protection

The Merkle tree based memory protection scheme works similar to the trivial memory protection scheme except for the integrity protection procedure (Figure 6). We list the differences between them as follows.

- The Merkle tree based scheme generates a Merkle tree for the main kernel and one tree for each TA. The integrity tag

of a page is a leaf node. Only root nodes are stored in the OCM, and other nodes are stored in the DRAM.

- When a data page is going to be transferred to DRAM, the Merkle tree based scheme needs to update the corresponding tree: all the nodes in the path from the leaf to the root node (including the leaf and root nodes) are updated.
- When a page is going to be loaded into the OCM, the Merkle tree based scheme verifies its integrity as follows. First, compute the integrity tag of the page. Then compute all the nodes in the path from the leaf to the root. Finally, verify the integrity of the page by comparing the computed root node with the root node stored in the OCM.

5 mTEE: A Minimal Kernel Prototype

We present a concrete implementation of the minimal kernel architecture, named mTEE, on a popular open source TEE OS: OP-TEE. In this section, we first introduce the architecture of OP-TEE OS, and then give our implementation of mTEE.

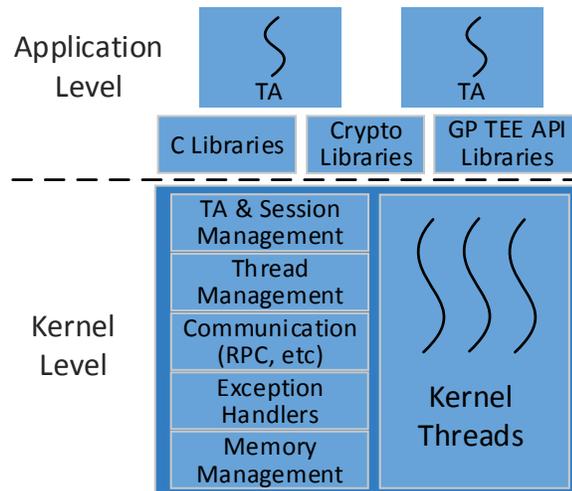


Figure 7: The Overall Architecture of OP-TEE

5.1 Architecture of the OP-TEE OS

The OP-TEE project publishes some documents introducing the OS components briefly but lacks the description of the whole kernel architecture, implementation details of kernel components, and interactions between the kernel components, which are needed to implement the minimal kernel architecture. So we go deep into the details of the OP-TEE source code to obtain the above information. However, the OP-TEE kernel is composed of more than 46,000 lines of C code and 7,000 lines of assembly code, which makes our task difficult.

The overall architecture of OP-TEE is illustrated in Figure 7. When the normal world sends a TA request, the thread management component captures the request and allocates a thread from the thread pool. The new thread first runs TA &

session management routines to open the demanded TA, and then initializes a session for this request, and finally executes the TA. OP-TEE supports two kinds of TAs: static TAs, which are statically linked in the OP-TEE kernel image and run in kernel threads, and user TAs, which are loaded from normal world and run in user threads. During execution, the TA can communicate with the normal world through the communication component which supports RPC (Remote Procedure Call) and shared memory mechanisms. The exception handlers deal with software and hardware exceptions. The memory management component manages memory pools, and allows dynamical memory allocation for threads. OP-TEE also provides a collection of libraries for user TAs, such as C libraries and cryptography libraries.

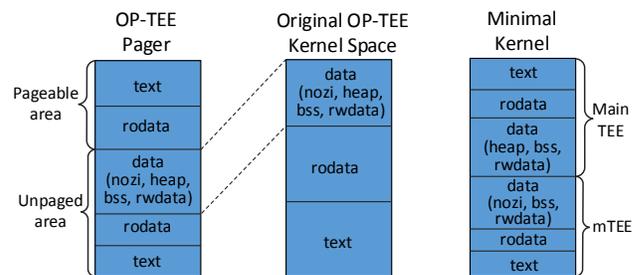


Figure 8: The Memory Layout of OP-TEE, Pager, and Minimal Kernel

Figure 8 (middle) depicts the memory layout of the original OP-TEE kernel, which consists of text, rodata and data sections. The data section is composed of the rwdata, bss, heap, and nozi sections, and the nozi section consists of page tables and stacks. Figure 8 (left) illustrates the memory layout of the OP-TEE’s memory protection solution (Pager): it builds an unpagged area residing in the OCM, which plays a role similar to the minimal kernel, and the remaining code and data form the pageable area, which resides in DRAM. As Pager lacks the design principle described in Section 4.3.1, it contains many unnecessary components. For example, Pager puts all the rwdata, heap and bss sections into OCM, which makes it very large and unsuitable for devices with small OCM.

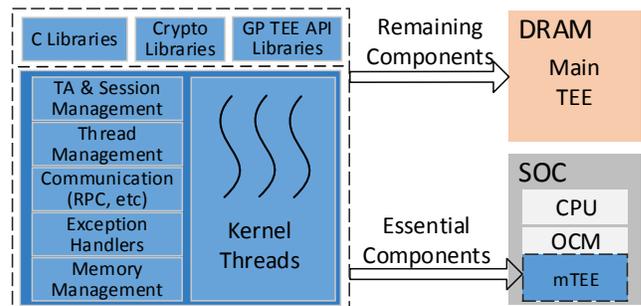


Figure 9: The Implementation of mTEE

Our implementation (Figure 9) divides OP-TEE into two parts: *mTEE* and *main TEE*. The *mTEE* is built by following

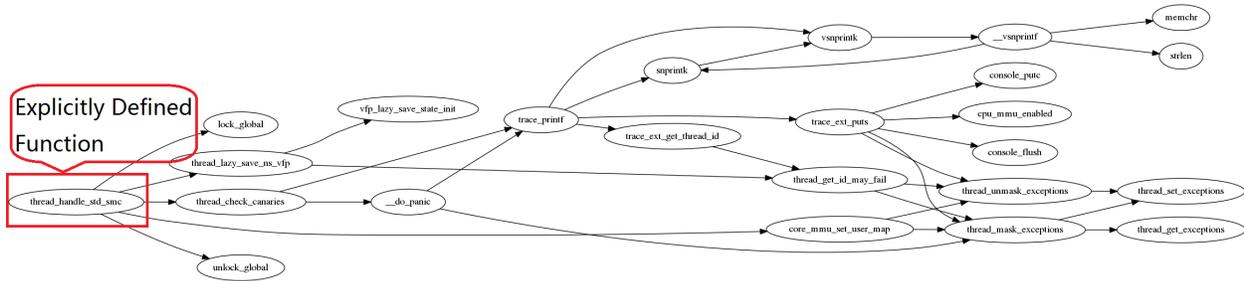


Figure 11: Call Dependency Graph of an Explicitly Defined Function

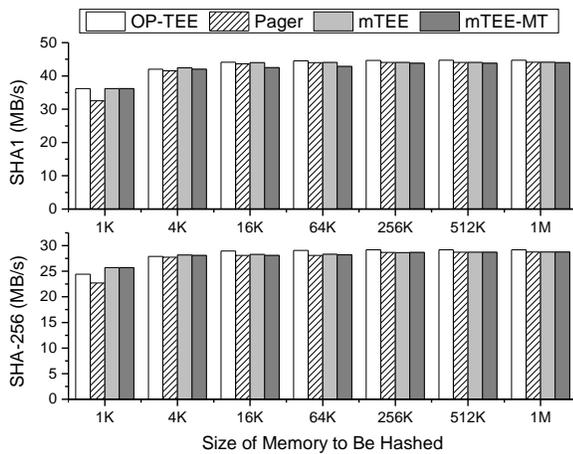


Figure 12: Throughput Comparison of SHA

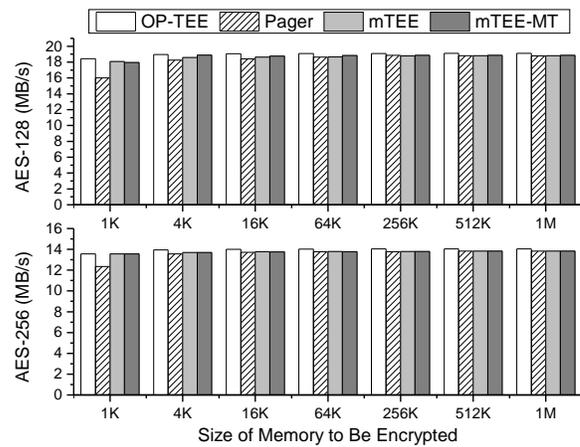


Figure 13: Throughput Comparison of AES

than 400 KB. The evaluation results also show that mTEE only needs about half (56%) of OCM required by Pager.

	text	rodata	rwdata	bss	heap	nozi	others	total
OP-TEE	147	81	8	60	76	56	12	440
Pager	35	19	8	7	65	36	10	180
mTEE	26	16	0	12	0	36	10	100

Table 2: Sizes of OP-TEE, Pager and mTEE (KB)

6.2 Crypto Evaluation

To evaluate the performance overhead of mTEE on cryptographic computations, we measure the performance of SHA1, SHA-256, AES-128, AES-256, and RSA on four systems: original OP-TEE, OP-TEE Pager, mTEE, and mTEE-MT. Figures 12, 13, and 14 depict the evaluation results.

The four systems achieve similar performance on AES and SHA, except that when the memory to be hashed or encrypted is small (less than 1KB), Pager is slightly slower than the other three systems. This is because Pager has smaller working memory, which makes the overhead of demand-paging be non-negligible compared to the cryptographic computations when the hashed/encrypted memory is small.

For RSA operations, mTEE and mTEE-MT perform much better than Pager. Especially in RSA verification, mTEE and

mTEE-MT is 7 times faster than Pager. This is due to that RSA verification requires less cryptographic computations than signing, so the overhead of demand-paging takes up a large proportion of the whole overhead.

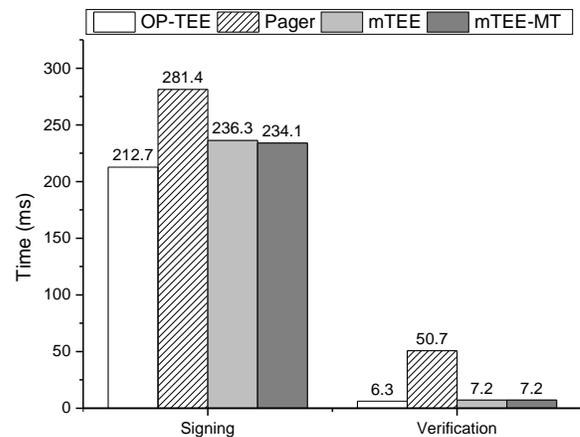


Figure 14: Performance Comparison of RSA

6.3 TA Evaluation

To evaluate the impact that mTEE has on TAs, we run the following three TAs that perform security-related computations on the five systems.

- *Random TA*: generate random numbers for applications in the normal world.
- *Data Protection TA*: use AES to encrypt the provided data, and return the ciphertext to the normal world.
- *One Time Password (OTP) TA*: receive a shared key from the normal world, and compute HMAC-based OTPs.

We measure both the entire execution time of TAs and the execution time of TA services (Figure 15). The entire execution time includes the time of loading a TA into the memory, mapping it to the TA address space, creating a session for the TA, running the TA service in the session, and destroying the session. The execution time of TA services only includes the time of running the TA service. Compared to OP-TEE, the performance of running the whole TA on Pager is 7.9 ~ 21.4 times slower, mTEE is 4.0 ~ 4.4 times slower, and mTEE-MT is 4.4 ~ 5.0 times slower; the performance of TA service on Pager is 20 ~ 200 times slower, mTEE is 3.3 ~ 7.5 times slower, and mTEE-MT is 3.5 ~ 7.7 times slower.

The evaluation results show that both mTEE and mTEE-MT achieve better performance than Pager. In the aspect of running whole TAs, mTEE is 1.8 ~ 5.3 times faster than Pager, and mTEE-MT is 1.6 ~ 4.7 times faster. In the aspect of TA service runtime, mTEE is 6.0 ~ 60 times faster than Pager, and mTEE-MT is 5.7 ~ 54 times faster. This is because Pager only offers 76KB (256KB - 180KB = 76KB) OCM to run TAs, while mTEE and mTEE-MT offer more OCM, about 156KB (256KB - 100KB = 156KB).

We also observe that, for TA services, the performance advantage of mTEE and mTEE-MT over Pager is very significant: mTEE can reach up to 60 times faster than Pager. This is because that Pager's free OCM (76KB) is smaller than the size of a TA (about 100KB) and therefore the execution of a TA service requires a lot of page-swappings, whose overhead is much more than the execution time of the pure TA service.

Another interesting observation beyond our expectation is that mTEE-MT gains almost the same performance with mTEE. This is because the major performance overhead of the memory protection mechanism is introduced by cryptographic computations (decryption, encryption or hash) on the 4KB pages swapped between OCM and DRAM, and mTEE-MT only adds a few hash computations on tree nodes (only dozens of bytes) for each page swapping, which are negligible compared to the cryptographic computation on the 4KB page.

From the above observations, we come to the following conclusions: for a software-based memory protection scheme, providing a large working memory for the protected software is critical to improve its performance, and the Merkle tree is a good choice for integrity protection because it reduces the

required physical secure memory while only introducing a slight increase on performance overhead.

6.4 Evaluation of the Impact of TA Size on Performance

Since the available OCM is quite limited, the performance of mTEE is sensitive to the size of TAs, especially when the working set size is larger than the available OCM because in this case paging is required. To evaluate the impact of TA size on the performance of mTEE, we leverage a random code generator tool Csmith [55] to generate a series of test TAs whose sizes range from 100KB to 1MB. We measure the entire execution time and service runtime of these test TAs on the five systems. Figures 16 and 17 show the results for the Pager, mTEE, mTEE-MT, and mTEE-plain systems relative to the OP-TEE system.

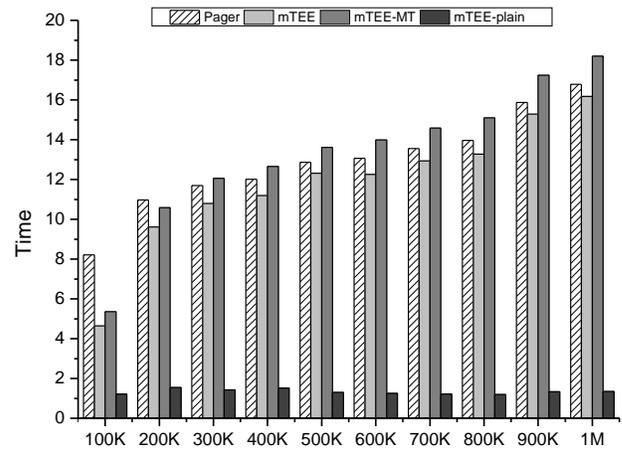


Figure 16: Performance of Test TAs

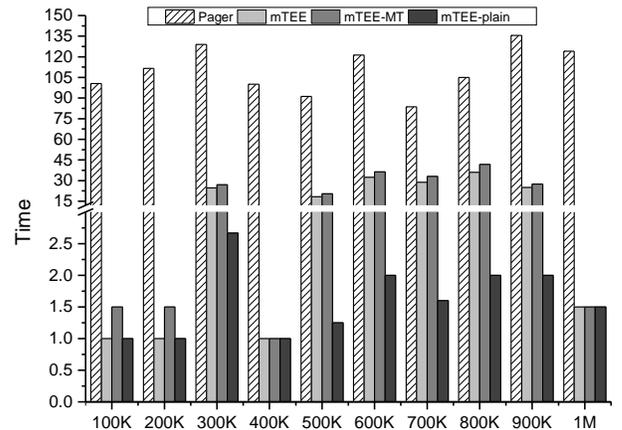


Figure 17: Performance of Test TA Services

Figure 16 shows that, in the aspect of executing whole TAs, Pager, mTEE, and mTEE-MT are about 10X times slower relative to the baseline OP-TEE, and as the size of TA increases, the performance overhead of these three systems

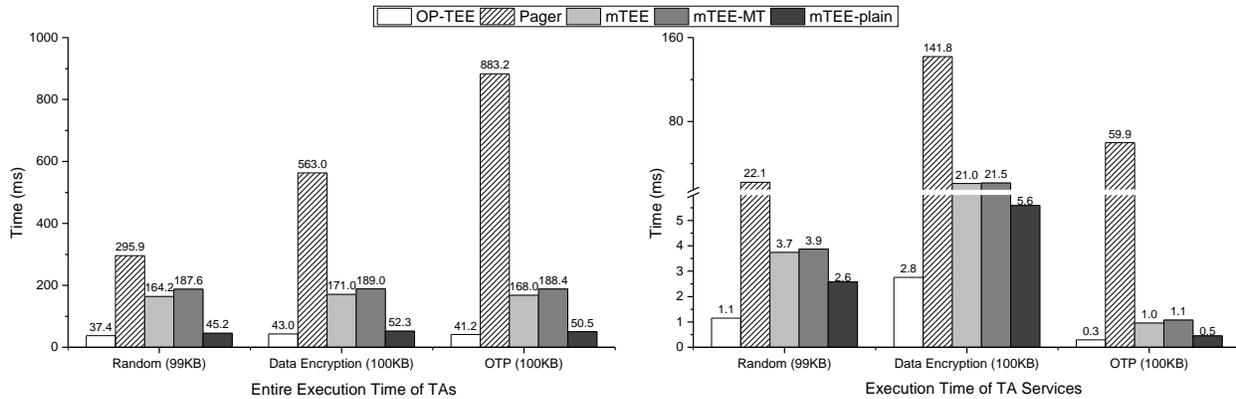


Figure 15: Performance Comparison of TA

grows. We also observe that, in most cases, mTEE performs better than Pager and mTEE-MT; the reason is that mTEE has more available OCM than Pager and needs less cryptographic computations than mTEE-MT.

Figure 17 shows that, in the aspect of TA services, Pager performs worst, and its performance is about 100X times slower than OP-TEE; we infer that this is because Pager itself occupies too much OCM, and the left OCM is far less than the working set size of all test TAs. For mTEE and mTEE-MT, they achieve almost the same performance in all test cases; in the 100KB, 200KB, 400KB and 1MB test cases, they achieve similar performance to OP-TEE; this is because Csmith programs often contain some dead code, so the active code of these four test cases can fit into the free OCM left by mTEE and mTEE-MT; in the rest six test cases, mTEE and mTEE-MT are, at most, 42 times slower than OP-TEE, and are 4 times faster than Pager on average.

From the above observations, we get the same conclusions as Section 6.3: first, mTEE and mTEE-MT achieve much higher performance than Pager, especially for TA services, because they reserve much more working memory; second, for software-based memory protection schemes, the Merkle tree is a good choice for integrity protection because it saves much secure memory while introducing slight overhead.

6.5 Evaluation of the Overhead of Paging and Cryptographic Computations

The performance overhead comes from two aspects: the memory demand-paging between OCM and DRAM, and the cryptographic computations (encryption, decryption, and integrity check) during the memory swapping. Here we measure the overhead of each aspect.

Figures 15, 16 and 17 compare the performance of mTEE-plain (which only enables demand-paging) with the other four systems. The evaluation results show that the overhead introduced by the demand-paging mechanism takes up a small portion of the whole overhead. Take the Random, Data Protec-

tion, and OTP TAs for example, mTEE-plain introduces 61% overhead on average, while mTEE introduces about 344% overhead on average. So we conclude that most performance overhead comes from the cryptographic computations during memory protection.

7 Discussion

Just like other software-based memory protection solutions, the most challenge for deploying the minimal kernel architecture is its big performance overhead. Leveraging customized cryptographic hardware accelerators is a good way to reduce the performance overhead. For example, Intel SGX integrates a specialized memory encryption engine [22] in the CPU, which only imposes 5.5% performance degradation on average. Besides the performance overhead, the size of the minimal kernel is another key feature that concerns the application of the minimal kernel architecture: the minimal kernel should be small enough to fit into the OCM. From the experience we have learned from the design of the minimal kernel architecture and implementation of the mTEE prototype, we discuss and give suggestions on how CPU designers can revise or extend their designs to provide more efficient memory protection solutions.

Performance Improvement. The evaluation results in Section 6.5 show that the overhead mainly comes from cryptographic computations. One direct solution to reduce the overhead is leveraging hardware cryptographic accelerators, which have been integrated into most mobile SoCs. For example, the evaluation board used in this paper, NXP i.MX6Q, is integrated with a cryptographic accelerator, namely Cryptographic Acceleration and Assurance Module (CAAM), which provides basic cryptographic primitives, such as hash algorithms and symmetric block ciphers. CAAM is implemented as a separate co-processor, and data is transferred to it through DMA. Paper [6] evaluates the performance of CAAM, but the results show that CAAM is only efficient when data sent to the accelerator is larger than 100 KB each time, and if the

input length is smaller than 100 KB, CAAM is slower than software implementations of cryptographic algorithms. The reason is that the data communication overhead with DMA disadvantages the acceleration of CAAM. So we recommend that CPU designers build a cryptographic accelerator into the primary CPU and provide interfaces to software in the form of ISA extensions (such as AES-NI). One advantage of this approach is that, unlike SGX's MEE, which is dedicated to memory protection, it can serve to all software.

Reducing the Minimal Kernel. The ARM CPU architecture uses two sets of translation tables: TTBR0 and TTBR1. The N field of the register TTBCR determines the virtual address range translated by each set of translation tables. Usually, TTBR0 translation tables are used to map user space, and TTBR1 translation tables are used to map kernel space. No matter what TTBCR.N is set, the first level translation table of TTBR1 must contain a fixed number of PTEs to map the whole address space. Take the ARMv7 CPU architecture for example, the first level translation table of TTBR1 contains 4096 PTEs and takes up 16 KB. The minimal kernel architecture requires that the page tables mapping the kernel space should be contained in the OCM. However, as the kernel space of TEE OS is small, most of the 16 KB memory is wasted. Take OP-TEE for example, the kernel space is a few megabytes. Including the aliased mapping of the secure DRAM, which is about dozens of megabytes, only dozens of the 4096 PTEs of the first level page table are used, which means that no more than 1 KB of the 16 KB is utilized. We recommend that the CPU support small page tables like Sanctum [9], or SGX's Enclave Page Cache Map (EPCM) mechanism which stores the virtual-physical address mapping in a specialized data structure, and then the address mapping for the protected address space can be stored using smaller memory. As a result, the minimal kernel's size can be reduced.

8 Related Work

To protect computer systems from physical attacks without the support of specialized hardware memory encryption engines, some research works using software-based memory encryption mechanism to protect the whole address space have been proposed.

Memory protection for applications. Cryptkeeper [44] presents a software-encrypted virtual memory manager, which divides DRAM into a plaintext working set and an encrypted segment, and swaps pages between the two segments on demand. Cryptkeeper only mitigates but cannot fully prevent physical attacks because the working set is always in plaintext. Another weakness of Cryptkeeper is that the whole kernel is exposed to physical attacks because the kernel resides in DRAM permanently. Paper [43] implements main memory encryption for applications by static/dynamic instrumentation of load and store instructions with encryption and decryption instructions. However, under physical attacks, just protect-

ing applications is not enough: attackers can attack and control the kernel, which is more severe. TrustShadow [19] and CryptMe [7] protect applications in the normal world from physical attacks using a lightweight runtime system, whose main task is to maintain execution environments and provide memory protection for applications. Ginseng [56] protects small sensitive data of applications by static protection extended to the compiler and runtime protection in the secure world, which are used to keep sensitive data in CPU's registers when they are used and encrypting them when switching context.

Memory protection for whole computer systems. Bear OS [25] is a microkernel which requires only about 35 KB secure memory and can be entirely loaded into the OCM. It encrypts all code and data outside of the chip boundary at section granularity. SoftME [57] protects tasks on embedded platforms by locating the embedded OS in the OCM and extending a task scheduler and a memory protection engine in the embedded OS. Both Bear OS and SoftME only work for small embedded OSes, and they are not applicable to mature OSes whose sizes are bigger than OCM. OP-TEE Pager is the most closely related system to our work. It implements a demand-paging system, which runs in the OCM and protects all code and data stored in DRAM. However, Pager is implemented without a design principle that determines which components are necessary and which components are unnecessary, causing it to include some unnecessary components, so the size of Pager exceeds OCM of many platforms. Komodo [15] presents an approach to implementing a formally verified enclave architecture in software, which achieves the security equivalent to Intel SGX. However, Komodo assumes a memory encryption engine to prevent physical attacks, which does not exist in ARM CPUs. So our work can be seen as the solution to the assumption.

9 Conclusion

In this paper, we present the minimal kernel architecture, which protects the whole TEE OS kernel and TAs against board level physical attacks. As the whole kernel address is protected, the minimal kernel achieves the same security level as SGX. Our principle of designing the minimal kernel makes it feasible to build a small kernel running inside the chip in the minimum amount of software, and our prototype fits into most commodity SoCs' OCM. By reserving OCM for TA's working memory as much as possible, mTEE achieves better performance than OP-TEE's original memory protection solution Pager: mTEE is several times faster, and for TA services, mTEE can reach up to 60 times faster. We give suggestions on how to modify or extend CPU hardware to improve the performance of software-based memory protection schemes. We expect that the minimal kernel architecture can help system designers to revise their CPUs in an economical way and propose efficient SGX-like solutions by hardware-software co-designs.

Acknowledgement

This research was supported by the National Key R & D Program of China (2018YFB0904900, 2018YFB0904903) and the National Natural Science Foundation of China (61802375, 61872343, 61602455, 61602325).

References

- [1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13. ACM New York, NY, USA, 2013.
- [2] Apple. iOS Security. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf, 2018.
- [3] ARM. Security Technology - Building a Secure System using Trustzone Technology. *ARM Technical White Paper*, 2009.
- [4] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.
- [5] Rick Boivie and Peter Williams. SecureBlue++: CPU support for Secure Execution. *Technical report*, 2012.
- [6] Aymen Boudguiga, Witold Klaudel, and Jimmy Durand Wesolowski. On the Performance of Freescale i.MX6 Cryptographic Acceleration and Assurance Module. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, page 8. ACM, 2015.
- [7] Chen Cao, Le Guan, Ning Zhang, Neng Gao, Jingqiang Lin, Bo Luo, Peng Liu, Ji Xiang, and Wenjing Lou. CryptMe: Data Leakage Prevention for Unmodified Programs on ARM Devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 380–400. Springer, 2018.
- [8] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal De Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. *ACM SIGARCH Computer Architecture News*, 43(1):177–189, 2015.
- [9] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, pages 857–874, 2016.
- [10] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can You Still Trust Your Network Card. *CanSecWest/core10*, pages 24–26, 2010.
- [11] Richard Earnshaw. Procedure Call Standard for the ARM Architecture. *ARM Limited, October*, 2003.
- [12] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B Lee, Nachiketh Potlapally, and Lionel Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In *Transactions on Computational Science IV*, pages 1–22. Springer, 2009.
- [13] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Michel Bardouillet, and Albert Martinez. A Parallelized Way to Provide Data Encryption and Integrity Checking on a Processor-Memory Bus. In *Proceedings of the 43rd annual Design Automation Conference*, pages 506–509. ACM, 2006.
- [14] EPN Solutions. Analysis Tools for DDR1, DDR2, DDR3, Embedded DDR and Fully Buffered DIMM Modules. <http://www.epnsolutions.net/ddr.html>, 2014.
- [15] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305. ACM, 2017.
- [16] FuturePlus System. DDR2 800 Bus Analysis Probe. http://www.futureplus.com/download/datasheet/fs2334_ds.pdf, 2006.
- [17] Behrad Garmany and Tilo Müller. PRIME: Private RSA Infrastructure for Memory-less Encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 149–158. ACM, 2013.
- [18] Blaise Gassend, G Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 295–306. IEEE, 2003.
- [19] Le Guan, Chen Cao, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Building a Trustworthy Execution Environment to Defeat Exploits from both Cyber Space and Physical Space for ARM. *IEEE Transactions on Dependable and Secure Computing*, 2018.
- [20] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. Copker: Computing with Private Keys without RAM. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, pages 23–26, 2014.
- [21] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting Private Keys against Memory Disclosure Attacks using Hardware Transactional Memory. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*, pages 3–19. IEEE, 2015.
- [22] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.
- [23] Peter Gutmann. Data Remanence in Semiconductor Devices. In *Proceedings of the 10th USENIX Security Symposium*, page 4. USENIX Association, 2001.
- [24] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.
- [25] Michael Henson and Stephen Taylor. Beyond Full Disk Encryption: Protection on Security-Enhanced Commodity Processors. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, pages 307–321. Springer, 2013.
- [26] ARM Holdings. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition. <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/>, 2014.
- [27] ARM Holdings. ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.k_10775/index.html, 2016.
- [28] George Hotz. PS3 Glitch Hack. http://www.eurasia.nu/wiki/index.php/PS3_Glitch_Hack, 2010.
- [29] Andrew Huang. Keeping Secrets in Hardware: The Microsoft Xbox™ Case Study. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 213–227. Springer, 2009.
- [30] Dongxu Ji, Qianying Zhang, Shijun Zhao, Zhiping Shi, and Yong Guan. MicroTEE: Designing TEE OS Based on the Microkernel Architecture. In *The 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2019.
- [31] Kari Kostiaainen, Jan-Erik Ekberg, N Asokan, and Aarne Rantala. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 104–115. ACM, 2009.

- [32] Markus G Kuhn. Cipher Instruction Search Attack on the Bus-encryption Security Microcontroller DS5002FP. *IEEE Transactions on Computers*, 47(10):1153–1157, 1998.
- [33] Wenhao Li, Yubin Xia, Long Lu, Haibo Chen, and Binyu Zang. TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 2–16. ACM, 2019.
- [34] Jochen Liedtke. On u-Kernel Construction. In *Proceedings of 15th ACM Symposium on Operating System Principles*, pages 237–250, 1995.
- [35] Linaro. OP-TEE: Open Portable Trusted Execution Environment. <https://www.op-tee.org/>, 2014.
- [36] Linaro. OP-TEE Pager. https://github.com/OP-TEE/optee_os/blob/master/documentation/optee_design.md, 2015.
- [37] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanhogue, and Uday R Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 10, 2013.
- [38] Tilo Müller, Andreas Dewald, and Felix C Freiling. AESSE: A Cold-boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security*, pages 42–47. ACM, 2010.
- [39] Tilo Müller, Felix C Freiling, and Andreas Dewald. TRESOR Runs Encryption Securely Outside RAM. In *USENIX Security Symposium*, volume 17, 2011.
- [40] Tilo Müller and Michael Spreitzenbarth. Frost: Forensic Recovery of Scrambled Telephones. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.
- [41] NCC Group. TPM Genie: Interposer Attacks Against the Trusted Platform Module Serial Bus. <https://www.nccgroup.trust/us/our-research/tpm-genie-interposer-attacks-against-the-trusted-platform-module-serial-bus>, 2018.
- [42] NCC Group. TPM Genie Tool. <https://github.com/nccgroup/TPMGenie>, 2018.
- [43] Panagiotis Papadopoulos, Giorgos Vasiliadis, Giorgos Christou, Evangelos Markatos, and Sotiris Ioannidis. No Sugar but all the Taste! Memory Encryption without Architectural Support. In *European Symposium on Research in Computer Security*, pages 362–380. Springer, 2017.
- [44] Peter AH Peterson. Cryptkeeper: Improving Security with Encrypted RAM. In *Proceedings of the 22nd IEEE International Conference on Technologies for Homeland Security (HST)*, pages 120–126. IEEE, 2010.
- [45] Rick Boivie, Eric Hall, Charanjit Jutla, Mimi Zohar. Secure Blue - Secure CPU Technology. https://researcher.watson.ibm.com/researcher/view_page.php?id=6904, 2006.
- [46] Samsung. Whitepaper: Samsung KNOX Security Solution. 2017.
- [47] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–80. ACM, 2014.
- [48] Patrick Simmons. Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 73–82. ACM, 2011.
- [49] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368. ACM, 2014.
- [50] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. TrustOTP: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 976–988. ACM, 2015.
- [51] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. Trust-dump: Reliable Memory Acquisition on Smartphones. In *European Symposium on Research in Computer Security*, pages 202–218. Springer, 2014.
- [52] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 367–378. IEEE, 2015.
- [53] Arrigo Triulzi. The Jedi Packet Trick Takes over the Deathstar. *Central Area Networking and Security (CANSEC 2010)*, 2010.
- [54] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Pixelvault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142. ACM, 2014.
- [55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.
- [56] Min Hong Yun and Lin Zhong. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. *NDSS*, 2019.
- [57] Meiyu Zhang, Qianying Zhang, Shijun Zhao, Zhiping Shi, and Yong Guan. SoftME: A Software-Based Memory Protection Approach for TEE System to Resist Physical Attacks. *Security and Communication Networks*, 2019, 2019.
- [58] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y Thomas Hou. CacheKit: Evading Memory Introspection Using Cache Incoherence. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 337–352. IEEE, 2016.
- [59] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. Case: Cache-assisted secure execution on arm processors. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, pages 72–90. IEEE, 2016.
- [60] Shijun Zhao, Qianying Zhang, Guangyao Hu, Yu Qin, and Dengguo Feng. Providing Root of Trust for ARM Trustzone Using On-chip SRAM. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, pages 25–36. ACM, 2014.

ScaRR: Scalable Runtime Remote Attestation for Complex Systems

Flavio Toffalini
SUTD

Eleonora Losiouk
University of Padua

Andrea Biondo
University of Padua

Jianying Zhou
SUTD

Mauro Conti
University of Padua

flavio_toffalini@mymail.sutd.edu.sg, jianying_zhou@sutd.edu.sg
elosiouk@math.unipd.it, andrea.biondo.1@studenti.unipd.it, conti@math.unipd.it

Abstract

The introduction of remote attestation (RA) schemes has allowed academia and industry to enhance the security of their systems. The commercial products currently available enable only the validation of static properties, such as applications fingerprint, and do not handle runtime properties, such as control-flow correctness. This limitation pushed researchers towards the identification of new approaches, called runtime RA. However, those mainly work on embedded devices, which share very few common features with complex systems, such as virtual machines in a cloud. A naive deployment of runtime RA schemes for embedded devices on complex systems faces scalability problems, such as the representation of complex control-flows or slow verification phase.

In this work, we present ScaRR: the first Scalable Runtime Remote attestation schema for complex systems. Thanks to its novel control-flow model, ScaRR enables the deployment of runtime RA on any application regardless of its complexity, by also achieving good performance. We implemented ScaRR and tested it on the benchmark suite SPEC CPU 2017. We show that ScaRR can validate on average 2M control-flow events per second, definitely outperforming existing solutions that support runtime RA on complex systems.

1 Introduction

RA is a procedure that allows an entity (*i.e.*, the *Verifier*) to verify the status of a device (*i.e.*, the *Prover*) from a remote location. This is achieved by having first the *Verifier* sending a challenge to the *Prover*, which replies with a report. Then, the *Verifier* analyzes the report to identify whether the *Prover* has been compromised [10]. In standard RA, usually defined as static, the *Prover* verification involves the integrity of specific hardware and software properties (*e.g.*, the *Prover* has loaded the correct software). On the market, there are already several available products implementing static RA, such as Software Guard Extensions (SGX) [18] or Trusted Platform Module (TPM) [42]. However, these do not provide a defence

against runtime attacks (*e.g.*, the control-flow ones) that aim to modify the program runtime behaviour. Therefore, to identify *Prover* runtime modifications, researchers proposed runtime RA. Among the different solutions belonging to this category, there are also the control-flow attestation approaches, which encode the information about the executed control-flow of a process [8, 9].

In comparison to static RA, the runtime one is relatively new, and today there are no reliable products available on the market since researchers have mainly investigated runtime RA for embedded devices [8, 9, 21, 22, 50]: most of them encode the complete execution path of a *Prover* in a single hash [8, 22, 50]; some [9] compress it in a simpler representation and rely on a policy-based verification schema; other ones [21] adopt symbolic execution to verify the control-flow information continuously sent by the *Prover*. Even if they have different performances, none of the previous solutions can be applied to a complex system (*e.g.*, virtual machines in a cloud) due to the following reasons: (i) representing all the valid execution paths through hash values is unfeasible (*e.g.*, the number of execution paths tends to grow exponentially with the size of the program), (ii) the policy-based approaches might not cover all the possible attacks, (iii) symbolic execution slows down the verification phase.

The purpose of our work is to fill this gap by providing ScaRR, the first runtime RA schema for complex systems. In particular, we focus on environments such as Amazon Web Services [2] or Microsoft Azure [3]. Since we target such systems, we require support for features such as multi-threading. Thus, ScaRR provides the following achievements with respect to the current solutions supporting runtime RA: (i) it makes the runtime RA feasible for any software, (ii) it enables the *Verifier* to verify intermediate states of the *Prover* without interrupting its execution, (iii) it supports a more fine-grained analysis of the execution path where the attack has been performed. We achieve these goals thanks to a novel model for representing the execution paths of a program, which is based on the fragmentation of the whole path into meaningful sub-paths. As a consequence, the *Prover* can send

a series of intermediate partial reports, which are immediately validated by the *Verifier* thanks to the lightweight verification procedures performed.

ScaRR is designed to defend a *Prover*, equipped with a trusted anchor and with a set of the standard solutions (e.g., $W \oplus X/DEP$ [34], Address Space Layout Randomization - ASLR [29], and Stack Canaries [14]), from attacks performed in the user-space and aimed at modifying the *Prover* runtime behaviour. The current implementation of ScaRR requires the program source code to be properly instrumented through a compiler based on LLVM [31]. However, it is possible to use lifting techniques [5], as well. Once deployed, ScaRR allows to verify on average $2M$ control-flow events per second, which is significantly more than the few hundred per second [21] or the thousands per second [9] verifiable through the existing solutions.

Contribution. The contributions of this work are the following ones:

- We designed a new model for representing the execution path for applications of any complexity.
- We designed and developed ScaRR, the first schema that supports runtime RA for complex systems.
- We evaluated the ScaRR performances in terms of: (i) attestation speed (i.e., the time required by the *Prover* to generate a partial report), (ii) verification speed (i.e., the time required by the *Verifier* to evaluate a partial report), (iii) overall generated network traffic (i.e., the network traffic generated during the communication between *Prover* and *Verifier*).

Organization. The paper is organized as follow. First, we provide a background on standard RA and control-flow exploitation (Section 2), and define the threat model (Section 3). Then, we describe the ScaRR control-flow model (Section 4) and its design (Section 5). We discuss ScaRR implementation details (Section 6) and evaluate its performance and security guarantees (Section 7). Finally, we discuss ScaRR limitations (Section 8), related works (Section 9), and conclude with final remarks (Section 10).

2 Background

The purpose of this section is to provide background knowledge about standard RA procedures and control-flow attacks.

Remote Attestation. RA always involves a *Prover* and a *Verifier*, with the latter responsible for verifying the current status of the former. Usually, the *Verifier* sends a challenge to the *Prover* asking to measure specific properties. The *Prover*, then, calculates the required measurement (e.g., a hash of the application loaded) and sends back a report R , which

contains the measurement M along with a digital fingerprint F , for instance, $R = (M, F)$. Finally, the *Verifier* evaluates the report, considering its freshness (i.e., the report has not been generated through a replay attack) and correctness (i.e., the *Prover* measurement is valid). It is a standard assumption that the *Verifier* is trusted, while the *Prover* might be compromised. However, the *Prover* is able to generate a correct and fresh report due to its trusted anchor (e.g., a dedicated hardware module).

Control-Flow Attacks. To introduce control-flow attacks, we first discuss the concepts of control-flow graph (CFG), execution-path, and basic-block (BBL) by using the simple program shown in Figure 1a as a reference example. The program starts with the acquisition of an input from the user (line 1). This is evaluated (line 2) in order to redirect the execution towards the retrieval of a privileged information (line 3) or an unprivileged one (line 4). Then, the retrieved information is stored in a variable (y), which is returned as an output (line 5), before the program properly concludes its execution (line 6).

A CFG represents all the paths that a program may traverse during its execution and it is statically computed. On the contrary, an execution path is a single path of the CFG traversed by the program at runtime. The CFG associated to the program in Figure 1a is depicted in Figure 1b and it encompasses two components: nodes and edges. The former are the BBLs of the program, while the latter represent the standard flow traversed by the program to move from a BBL towards the next one. A BBL is a linear sequence of instructions with a single entry point (i.e., no incoming branches to the set of instructions other than the first), and a single exit point (i.e., no outgoing branches from the set of instructions other than the last). Therefore, a BBL can be considered an atomic unit with respect to the control-flow, as it will either be fully executed, or not executed at all on a given execution path. A BBL might end with a control-flow event, which could be one of

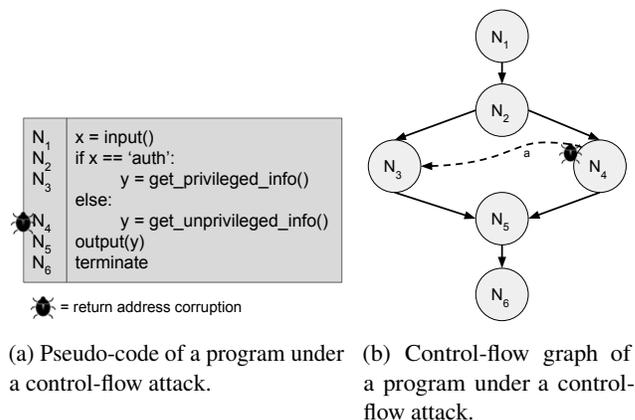


Figure 1: Illustrative example of a control-flow attack.

the following in a x86_64 architecture: procedure calls (e.g., `call`), jumps (e.g., `jmp`), procedure returns (e.g., `ret`), and system calls (e.g., `syscall`). During its execution, a process traverses several BBLs, which completely define the process execution path.

Runtime attacks, and more specifically the control-flow ones, aim at modifying the CFG of a program by tampering with its execution path. Considering Figure 1, we assume that an attacker is able to run the program (from the node N_1), but that he is not authorized to retrieve the privileged information. However, the attacker can, anyway, violate those controls through a memory corruption error performed on the node N_4 . As soon as the attacker provides an input to the program and starts its execution, he will be redirected to the node N_4 . At this point, the attacker can exploit a memory corruption error (e.g., a stack overflow) to introduce a new edge from N_4 to N_3 (edge labeled as a) and retrieve the privileged information. As a result, the program traverses an unexpected execution path not belonging to its original CFG. Even though several solutions have been proposed to mitigate such attacks (e.g., ASLR [29]), attackers still manage to perform them [43].

This illustrative example about how to manipulate the execution path of a program is usually the basic step to perform more sophisticated attacks like exploiting a vulnerability to take control of a process [49] or installing a persistent data-only malware without injecting new code, once the control over a process is taken by the attacker [44].

Runtime RA provides a reliable mechanism which allows the *Verifier* to trace and validate the execution path undertaken by the *Prover*.

3 Threat Model and Requirements

In this section, we describe the features of the *Attacker* and the *Prover* involved in our threat model. Our assumptions are in line with other RA schemes [8, 9, 18, 21, 47].

Attacker. We assume to have an attacker that aims to control a remote service, such as a Web Server or a Database Management System (DBMS), and that has already bypassed the default protections, such as Control Flow Integrity (CFI). To achieve his aim, the attacker can adopt different techniques, among which: Return-Oriented Programming (ROP)/Jump-Oriented Programming (JOP) attacks [16, 17], function hooks [35], injection of a malware into the victim process, installation of a data-only malware in user-space [44], or manipulation of other user-space processes, such as security monitors. In our threat model, we do not consider physical attacks (our complex systems are supposed to be virtual machines), pure data-oriented attacks (e.g., attacks that do not alter the original program CFG), self-modifying code, and dynamic loading of code at runtime (e.g., just-in-time compil-

ers [41]). We refer to Section 7.4 for a comprehensive attacker analysis.

Prover. The *Prover* is assumed to be equipped with: (i) a trusted anchor that guarantees a static RA, (ii) standard defence mitigation techniques, such as $W \oplus X/DEP$, ASLR. In our implementation, we use the kernel as a trusted anchor, which is a reasonable assumption if the machines have trusted modules such as a TPM [42]. However, we can also use a dedicated hardware, as discussed in Section 8. The *Prover* maintains sensitive information (i.e., shared keys and cryptographic functions) in the trusted anchor and uses it to generate fresh reports, that cannot be tampered by the attacker.

4 ScaRR Control-Flow Model

ScaRR is the first schema that allows to apply runtime RA on complex systems. To achieve this goal, it relies on a new model for representing the CFG/execution path of a program. In this section, we illustrate first the main components of our control-flow model (Section 4.1) and, then, the challenges we faced during its design (Section 4.2).

4.1 Basic Concepts

The ScaRR control-flow model handles BBLs at assembly level and involves two components: *checkpoints* and *List of Actions (LoA)*.

A *checkpoint* is a special BBL used as a delimiter for identifying the start or the end of a sub-path within the CGF/execution path of a program. A *checkpoint* can be: *thread beginning/end*, if it identifies the beginning/end of a thread; *exit-point*, if it represents an exit-point from an application module (e.g., a system call or a library function invocation); *virtual-checkpoint*, if it is used for managing special cases such as *loops* and *recursions*.

A *LoA* is the series of significant edges that a process traverses to move from a *checkpoint* to the next one. Each edge is represented through its source and destination BBL and, comprehensively, a *LoA* is defined through the following notation:

$$[(BBL_{s1}, BBL_{d1}), \dots, (BBL_{sn}, BBL_{dn})].$$

Among all the edges involved in the complete representation of a CFG, we consider only a subset of them. In particular, we look only at those edges that identify a unique execution path: procedure call, procedure return and branch (i.e., conditional and indirect jumps).

To better illustrate the ScaRR control-flow model, we now recall the example introduced in Section 2. Among the six nodes belonging to the CFG of the example, only the following four ones are *checkpoints*: N_1 , since it is a *thread beginning*; N_3 and N_4 , because they are *exit-points*, and N_6 ,

since it is a *thread end*. In addition, the *LoAs* associated to the example are the following ones:

$$\begin{aligned} N_1 - N_3 &\Rightarrow [(N_2, N_3)] \\ N_1 - N_4 &\Rightarrow [(N_2, N_4)] \\ N_3 - N_6 &\Rightarrow [] \\ N_4 - N_6 &\Rightarrow [] \end{aligned}$$

On the left we indicate a pair of *checkpoints* (e.g., $N_1 - N_3$), while on the right the associated *LoA* (empty *LoAs* are considered valid).

4.2 Challenges

Loops, *recursions*, *signals*, and *exceptions* involved in the execution of a program introduce new challenges in the representation of a CFG since they can generate uncountable executions paths. For example, *loops* and *recursions* can generate an indefinite number of possible combinations of *LoA*, while *signals*, as well as *exceptions*, can introduce an unpredictable execution path at any time.

Loops. In Figure 2, we illustrate the approach used to handle *loops*. Since it is not always possible to count the number of iterations of a loop, we consider the conditional node of the loop (N_1) as a *virtual-checkpoint*. Thus, the *LoAs* associated to the example shown in Figure 2 are as follows:

$$\begin{aligned} S_A - N_1 &\Rightarrow [] \\ N_1 - N_1 &\Rightarrow [(N_1, N_2)] \\ N_1 - S_B &\Rightarrow [(N_1, N_3)] \end{aligned}$$

Recursions. In Figure 3, we illustrate our approach to handle *recursions*, i.e., a function that invokes itself. Intuitively, the *LoAs* connecting P_B and P_E should contain all the possible invocations made by $a()$ towards itself, but the number of invocations is indefinite. Thus, we consider the node performing the recursion as a *virtual-checkpoint* and model only the path that could be chosen, without referring to the number of times it is really undertaken. The resulting *LoAs* for the

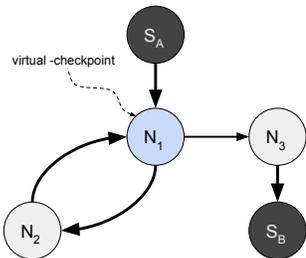


Figure 2: Loop example in the ScaRR control-flow model.

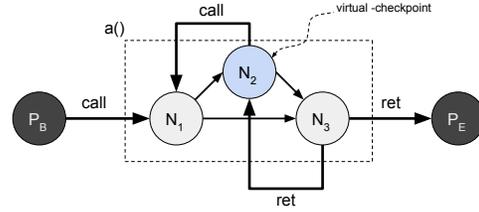


Figure 3: Recursion example in the ScaRR control-flow model.

example in Figure 3 are the following ones:

$$\begin{aligned} P_B - N_2 &\Rightarrow [(P_B, N_1), (N_1, N_2)] \\ N_2 - N_2 &\Rightarrow [(N_2, N_1), (N_1, N_2)] \\ N_2 - N_2 &\Rightarrow [(N_2, N_1), (N_1, N_3), (N_3, N_2)] \\ N_2 - P_E &\Rightarrow [(N_2, N_1), (N_1, N_3), (N_3, P_E)] \\ P_B - P_E &\Rightarrow [(P_B, N_1), (N_1, N_3), (N_3, P_E)] \end{aligned}$$

Finally, the *virtual-checkpoint* can be used as a general approach to solve every situation in which an indirect jump targets a node already present in the *LoA*.

Signals. When a thread receives a *signal*, its execution is stopped and, after a context-switch, it is diverted to a dedicated handler (e.g., a function). This scenario makes the control-flow unpredictable, since an interruption can occur at any point during the execution. To manage this case, ScaRR models the signal handler as a separate thread (adding *beginning/end thread checkpoints*) and computes the relative *LoAs*. If no handler is available for the *signal* that interrupted the program, the entire process ends immediately, producing a wrong *LoA*.

Exception Handler. Similar to *signals*, when a thread rises an *exception*, the execution path is stopped and control is transferred to a catch block. Since ScaRR has been implemented for Linux, we model the catch blocks as a separate thread (adding *beginning/end thread checkpoints*), but it is also possible to adapt ScaRR to fulfill different exception handling mechanisms (e.g., in Windows). In case no catch block is suitable for the *exception* that was thrown, the process gets interrupted and the generated *LoA* is wrong.

5 System Design

To apply runtime RA on a complex system, there are two fundamental requirements: (i) handling the representation of a complex CFG or execution path, (ii) having a fast verification process. Previous works have tried to achieve the first requirement through different approaches. A first solution [8, 22, 50] is based on the association of all the valid execution paths of the *Prover* with a single hash value. Intuitively, this is not a scalable approach because it does not allow to handle complex

CFG/execution paths. On the contrary, a second approach [21] relies on the transmission of all the control-flow events to the *Verifier*, which then applies a symbolic execution to validate their correctness. While addressing the first requirement, this solution suffers from a slow verification phase, which leads toward a failure in satisfying the second requirement.

Thanks to its novel control-flow model, ScaRR enables runtime RA for complex systems, since its design specifically considers the above-mentioned requirements with the purpose of addressing both of them. In this section, we provide an overview of the ScaRR schema (Section 5.1) together with the details of its workflow (Section 5.2), explicitly motivating how we address both the requirements needed to apply runtime RA on complex systems.

5.1 Overview

Even if the ScaRR control-flow model is composed of *checkpoints* and *LoAs*, the ScaRR schema relies on a different type of elements, which are the *measurements*. Those are a combination of *checkpoints* and *LoAs* and contain the necessary information to perform runtime RA. Figure 4 shows an overview of ScaRR, which encompasses the following four components: a *Measurements Generator*, for identifying all the program valid *measurements*; a *Measurements DB*, for saving all the program valid *measurements*; a *Prover*, which is the machine running the monitored program; a *Verifier*, which is the machine performing the program runtime verification.

As a whole, the workflow of ScaRR involves two separate phases: an *Offline Program Analysis* and an *Online Program Verification*. During the first phase, the *Measurements Generator* calculates the CFG of the monitored *Application A* (Step 1 in Figure 4) and, after generating all the *Application A* valid *measurements*, it saves them in the *Measurements DB* (Step 2 in Figure 4). During the second phase, the *Verifier* sends a challenge to the *Prover* (Step 3 in Figure 4). Thus, the *Prover* starts executing the *Application A* and sending partial reports to the *Verifier* (Step 4 in Figure 4). The *Verifier* validates the freshness and correctness of the partial reports by comparing the received new *measurements* with the previous ones stored in the *Measurements DB*. Finally, as soon as the *Prover* finishes the processing of the input received from the *Verifier*, it sends back the associated output.

5.2 Details

As shown in Figure 4, the workflow of ScaRR goes through five different steps. Here, we provide details for each of those.

(1) Application CFG. The *Measurements Generator* executes the *Application A*(), or a subset of it (e.g., a function), and extracts the associated CFG G .

(2) Offline Measurements. After generating the CFG, the *Measurements Generator* computes all the program *offline measurements* during the *Offline Program Analysis*. Each

offline measurement is represented as a key-value pair as follows:

$$(cp_A, cp_B, H(LoA)) \Rightarrow [(BBL_{s1}, BBL_{d1}), \dots, (BBL_{sn}, BBL_{dn})]$$

The key refers to a triplet, which contains two *checkpoints* (i.e., cp_A and cp_B) and the hash of the *LoA* (i.e., $H(LoA)$) associated to the significant BBLs that are traversed when moving from the source *checkpoint* to the destination one. The value refers only to a subset of the BBLs pairs used to generate the hash of the *LoAs* and, in particular, only to procedure calls and procedure returns. Those are the control-flow events required to mount the shadow stack during the verification phase.

(3) Request for a Challenge. The *Verifier* starts a challenge with the *Prover* by sending it an input and a nonce, which prevents replay attacks.

(4) Online Measurements. While the *Application A* processes the input received from the *Verifier*, the *Prover* starts generating the *online measurements* which keep trace of the *Application A* executed paths. Each *online measurement* is represented through the same notation used for the keys in the *offline measurements*, i.e., the triplet $(cp_A, cp_B, H(LoA))$.

When the number of *online measurements* reaches a pre-configured limit, the *Prover* encloses all of them in a partial report and sends it to the *Verifier*. The partial report is defined as follows:

$$P_i = (R, F_K(R||N||i))$$

$$R = (T, M).$$

In the current notation, P_i is the i -th partial report, R the payload and $F_K(R||N||i)$ the digital fingerprint (e.g., a message authentication code [15]). This is generated by using: (i) the secret key K , shared between *Prover* and *Verifier*, (ii) the nonce N , sent at the beginning of the protocol, and (iii) the index i , which is a counter of the number of partial reports. Finally, the payload R contains the *online measurements* M along with the associated thread T .

The novel communication paradigm between *Prover* and *Verifier*, based on the transmission and consequent verification of several partial reports, satisfies the first requirement for applying runtime RA on complex systems (i.e., handling the representation of a complex CFG/execution path). This is achieved thanks to the ScaRR control-flow model, which allows to fragment the whole CFG/execution path into sub-paths. Consequently, the *Prover* can send intermediate reports even before the *Application A* finishes to process the received input. In addition, the fragmentation of the whole execution path into sub-paths allows to have a more fine-grained analysis of the program runtime behaviour since it is possible to identify the specific edge on which the attack has been performed.

(5) Report Verification. In runtime RA, the *Verifier* has two different purposes: verifying whether the running application is still the original one and whether the execution paths

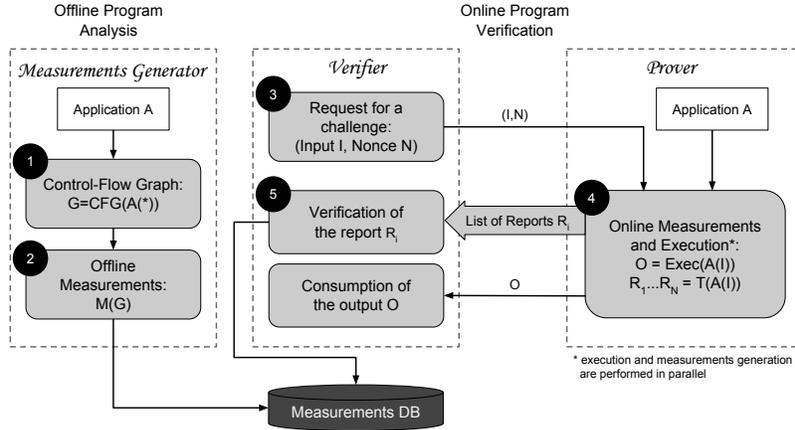


Figure 4: ScaRR system overview.

traversed by it are the expected ones. The first purpose, which we assume to be already implemented in the system [18, 47], can be achieved through a static RA applied on the *Prover* software stack. On the contrary, the second purpose is the main focus in our design of the ScaRR schema.

As soon as the *Verifier* receives a partial report P_i , it first performs a formal integrity check by considering its fingerprint $F_K(R||N||i)$. Then, it considers the *online measurements* sent within the report and performs the following checks: (C1) whether the *online measurements* are the expected ones (i.e., it compares the received *online measurements* with the offline ones stored in the *Measurements DB*), (C2) whether the destination *checkpoint* of each *measurement* is equal to the source *checkpoint* of the following one, and (C3) whether the *LoAs* are coherent with the stack status by mounting a shadow stack. If one of the previous checks fails, the *Verifier* notifies an anomaly and it will reject the output generated by the *Prover*.

All the above-mentioned checks performed by the *Verifier* are lightweight procedures (i.e., a lookup in a hash map data structure and a shadow stack update). The speed of the second verification mechanism depends on the number of procedure calls and procedure returns found for each *measurement*. Thus, also the second requirement for applying runtime RA on complex systems is satisfied (i.e., keeping a fast verification phase). Once again, this is a consequence of the ScaRR control-flow model since the fragmentation of the execution paths allows both *Prover* and *Verifier* to work on a small amount of data. Moreover, since the *Verifier* immediately validates a report as soon as it receives a new one, it can also detect an attack even before the *Application A* has completed the processing of the input.

5.3 Shadow Stack

To improve the defences provided by ScaRR, we introduce a shadow stack mechanism on the *Verifier* side. To illustrate

S	int main(int argc, char ** argv) {
M ₁	a(10);
M ₂	/* irrelevant code */
M ₃	a(6);
M ₄	return 0;
E	}
A ₁	void a(int x) {
C	/* irrelevant code */
A ₂	printf("%d\n", x);
	return;
	}

Figure 5: Illustrative example to explain the shadow stack on the ScaRR *Verifier*.

it, we refer to the program shown in Figure 5, which contains only two functions: `main()` and `a()`. Each line of the program is a BBL and, in particular: the first BBL (i.e., S) and the last BBL (i.e., E) of the `main()` function are a *beginning thread* and *end thread checkpoints*, respectively; the function `a()` contains a function call to `printf()`, which is an *exit-point*. According to the ScaRR control-flow model, the *offline measurements* are the following ones:

$$\begin{aligned} (S, C, H_1) &\Rightarrow [(M_1, A_1)], \\ (C, C, H_2) &\Rightarrow [(A_2, M_2), (M_3, A_1)], \\ (C, E, H_3) &\Rightarrow [(A_2, M_4)]. \end{aligned}$$

The significant BBLs we consider for generating the *LoAs* are: (i) the ones connecting the BBL S to the *checkpoint* C , (ii) the ones connecting two *checkpoints* C , and (iii) the ones to move from the *checkpoint* C to the last BBL E .

In this scenario, an attacker may hijack the return address of the function `a()` in order to jump to the BBL M_3 . If this happens, the *Prover* produces the following *online measure-*

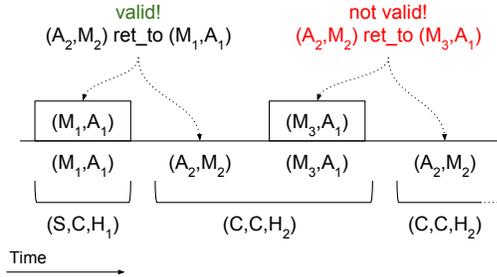


Figure 6: Implementation of the shadow stack on the ScaRR Verifier.

ments:

$$(S, C, H_1) \rightarrow (C, C, H_2) \rightarrow (C, C, H_2) \rightarrow \dots$$

Although generated after an attack, those measurements are still compliant with the checks (C1) and (C2) of the Verifier. Thus, to detect this attack, we introduce a new relation (i.e., `ret_to`) to illustrate the link between two edges. The Measurements Generator computes all the `ret_to` relations during the Offline Program Analysis and saves them in the Measurements DB using the following notation:

$$\begin{aligned} (A_2, M_2) \text{ ret_to } (M_1, A_1), \\ (A_2, M_2) \text{ ret_to } (M_3, A_1). \end{aligned}$$

Figure 6 shows how the Verifier combines all these information to build a remote shadow stack. At the beginning, the shadow stack is empty (i.e., no function has been invoked yet). Then, according to the online measurement (S, C, H_1) , the Prover has invoked the `main()` function passing through the edge (M_1, A_1) , which is pushed on the top of the stack by the Verifier. Then, the online measurement (C, C, H_2) indicates that the execution path exited from the function $a()$ through the edge (A_2, M_2) , which is in relation with the edge on the top of the stack and therefore is valid. Moving forward, the Verifier pops from the stack and pushes the edge (M_3, A_1) , which corresponds to the second invocation of the function $a()$. At this point, the third measurement (C, C, H_2) indicates that the Prover exited from the function $a()$ through the edge (A_2, M_2) , which is not in relation with (M_3, A_1) . Thus, the Verifier detects the attack and triggers an alarm.

6 Implementation

Here, we provide the technical details of the ScaRR schema and, in particular, of the Measurements Generator (Section 6.1) and of the Prover (Section 6.2).

6.1 Measurements Generator

The Measurements Generator is implemented as a compiler, based on LLVM [31] and on the CRAB framework [24].

Despite this approach, it is also possible to use frameworks to lift the binary code to LLVM intermediate-representation (IR) [5].

The Measurements Generator requires the program source code to perform the following operations: (i) generating the offline measurements, and (ii) detecting and instrumenting the control-flow events. During the compilation, the Measurements Generator analyzes the LLVM IR to identify the control-flow events and generate the offline measurements, while it uses the CRAB LLVM framework to generate the CFG, since it provides a heap abstract domain that resolves indirect forward jumps. Again during the compilation, the Measurements Generator instruments each control-flow event to invoke a tracing function which is contained in the trusted anchor. To map LLVM IR BBLs to assembly BBLs, we remove the optimization flags and we include dummy code, which is removed after the compilation through a binary-rewriting tool. To provide the above-mentioned functionalities, we add around 3.5K lines of code on top of CRAB and LLVM 5.0.

6.2 Prover

The Prover is responsible for running the monitored application, generating the application online measurements and sending the partial reports to the Verifier. To achieve the second aim, the Prover relies on the architecture depicted in Figure 7, which encompasses several components belonging either to the user-space (i.e., Application Process and ScaRR Libraries) or to the kernel-space (i.e., ScaRR sys_addaction, ScaRR Module, and ScaRR sys_measure).

Each component works as follows:

- *Application Process* - the process running the monitored application, which is equipped with the required instrumentation for detecting control-flow events at runtime.
- *ScaRR Libraries* - the libraries added to the original application to trace control-flow events and checkpoints.
- *ScaRR sys_addaction* - a custom kernel syscall used to trace control-flow events.
- *ScaRR Module* - a module that keeps trace of the online measurements and of the partial reports. It also extracts the BBL labels from their runtime addresses, since the ASLR protection changes the BBLs location at each run.
- *ScaRR sys_measure* - a custom kernel syscall used to generate the online measurements.

When the Prover receives a challenge, it starts the execution of the application and creates a new online measurement. During the execution, the application can encounter checkpoints or control-flow events, both hooked by the instrumentation. Every time the application crosses a control-flow event, the ScaRR Libraries invoke the ScaRR sys_addaction syscall to

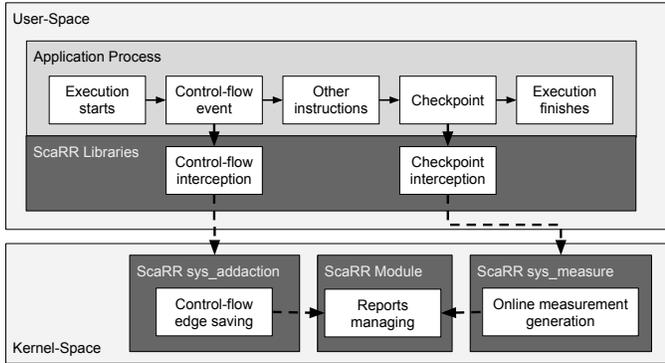


Figure 7: Internal architecture of the *Prover*.

save the new edge in a buffer inside the kernel-space. While, every time the application crosses a *checkpoint*, the *ScaRR Libraries* invoke the *ScaRR sys_measure* syscall to save the *checkpoint* in the current *online measurement*, calculate the hash of the edges saved so far, and, finally, store the *online measurement* in a buffer located in the kernel-space. When the predefined number of *online measurements* is reached, the *Prover* sends a partial report to the *Verifier* and starts collecting new *online measurements*. The *Prover* sends the partial report by using a dedicated kernel thread. The whole procedure is repeated until the application finishes processing the input of the *Verifier*.

The whole architecture of the *Prover* relies on the kernel as a trusted anchor, since we find it more efficient in comparison to other commercial trusted platforms, such as SGX and TrustZone, but other approaches can be also considered (Section 8). To develop the kernel side of the architecture, we add around 200 lines of code to a Kernel version v4.17-rc3. We also include the Blake2 source [4, 11], which is faster and provides high cryptographic security guarantees for calculating the hash of the *LoAs*.

7 Evaluation

We evaluate ScaRR from two perspectives. First, we measure its performance focusing on: attestation speed (Section 7.1), verification speed (Section 7.2) and network impact (Section 7.3). Then, we discuss ScaRR security guarantees (Section 7.4).

We obtained the results described in this section by running the bench-marking suite SPEC CPU 2017 over a Linux machine equipped with an Intel i7 processor and 16GB of memory¹. We instrumented each tool to detect all the necessary control-flow events, we then extracted the *offline measurements* and we ran each experiment to analyze a specific performance metrics.

¹We did not manage to map assembly BBL addresses to LLVM IR for 519.lbm_r and 520.omnetpp_r.

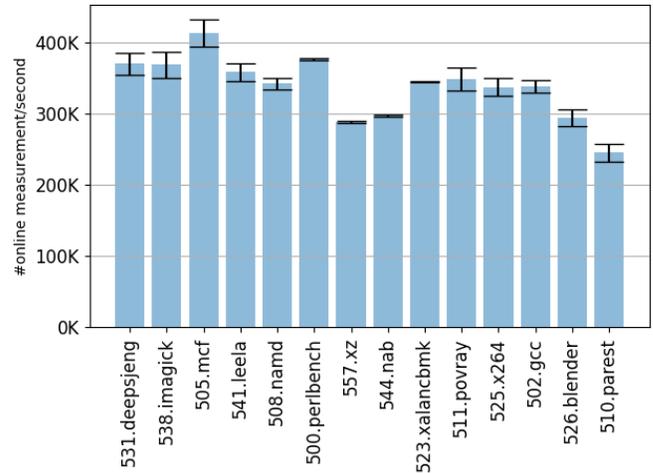


Figure 8: Average attestation speed measured as number of online measurements per second.

7.1 Attestation Speed

We measure the attestation speed as the number of *online measurements* per second generated by the *Prover*. Figure 8 shows the average attestation speed and the standard deviation for each experiment of the SPEC CPU 2017. More specifically, we run each experiment 10 times, calculate the number of *online measurements* generated per second in each run, and we compute the final average and standard deviation. Our results show that ScaRR has a range of attestation speed which goes from 250K (510.parest) to over 400K (505.mcf) of *online measurements* per second. This variability in performance depends on the complexity of the single experiment and on other issues, such as the file loading. Previous works prove to have an attestation speed around 20K/ 30K of control-flow events per second [8, 9]. Since each *online measurement* contains at least a control-flow event, we can claim that ScaRR has an attestation speed at least 10 times faster than the one offered by the existing solutions.

7.2 Verification Speed

During the validation of the partial reports, the *Verifier* performs a lookup against the *Measurements DB* and an update of the shadow stack. To evaluate the overall performance of the *Verifier*, we consider the verification speed as the maximum number of *online measurements* verified per second. To measure this metrics, we perform the following experiment for each SPEC tool: first, we use the *Prover* to generate and save the *online measurements* of a SPEC tool; then, the *Verifier* verifies all of them without involving any element that might introduce delay (e.g., network). In addition, we also introduce a digital fingerprint based on AES [39] to simulate an ideal scenario in which the *Prover* is fast. We perform

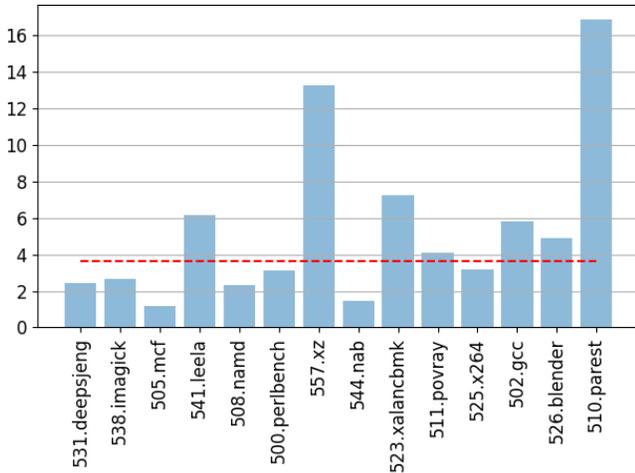


Figure 9: Average number of procedure calls and procedure returns found during the *Online Program Analysis* of the SPEC CPU 2017 tools.

the verification by loading the *offline measurements* in an in-memory hash map and performing the shadow stack. Finally, we compute the average verification speed of all tools.

According to our experiments, the average verification speed is 2M of *online measurements* per second, with a range that goes from 1.4M to 2.7M of *online measurements* per second. This result outperforms previous works in which the authors reported a verification speed that goes from 110 [21] to 30K [9] of control-flow events per second. As for the attestation speed, we recall that each *online measurement* contains at least one control-flow event.

The performance of the shadow stack depends on the number of procedure calls and procedure returns found during the generation of *online measurements* in the *Online Program Analysis* phase. To estimate the impact on the shadow stack, we run each experiment of the SPEC CPU 2017 tool and count the number of procedure calls and procedure returns. Figure 9 shows the average number of the above-mentioned variables found for each experiment. For some experiments (*i.e.*, 505.mcf and 544.nab), the average number is almost one since they include some recursive algorithms that correspond to small *LoAs*. If the average length of the *LoAs* tends to one, ScaRR behaves similarly to other remote RA solutions that are based on cumulative hashes [8,9]. Overall, Figure 9 shows that a median of push/pop operations is less than 4, which implies a fast update. Combining an in-memory hash map and a shadow stack allows ScaRR to perform a fast verification phase.

7.3 Network Impact and Mitigation

A high sending rate of partial reports from the *Prover* might generate a network congestion and therefore affect the ver-

ification phase. To reduce network congestion and improve verification speed, we perform an empirical measurement of the amount of data (*i.e.*, MB) sent on a local network with respect to the verification speed by applying different settings. The experiment setup is similar to Section 7.2, but the *Prover* and the *Verifier* are connected through an Ethernet network with a bandwidth of 10Mbit/s. At first, we record 1M of *online measurements* for each SPEC CPU 2017 tool. Then, we send the partial reports to the *Verifier* over a TCP connection, each time adopting a different approach among the following ones: *Single*, *Batch*, *Zip* [7], *Lzma* [32], *Bz2* [1] and *ZStandard* [6]. The results of this experiment are shown in Figure 10. In the first two modes (*i.e.*, *Single* and *Batch*), we send a single *online measurement* and 50K *online measurements* in each partial report, respectively. As shown in the graph, both approaches generate a high amount of network traffic (around 80MB), introducing a network delay which slows down the verification speed. For the other four approaches, each partial report still contains 50K *online measurements*, but it is generated through different compression algorithms. All the four algorithms provide a high compression rate (on average over 95%) with a consequent reduction in the network overload. However, the algorithms have also different compression and decompression delays, which affect the verification speed. The *Zip* and *ZStandard* show the best performances with 1.2M of *online measurements/s* and 1.6M of *online measurements/s*, respectively, while *Bz2* (30K of *online measurements/s*) and *Lzma* (0.4M of *online measurements/s*) are the worst ones. The number of *online measurements* per partial report might introduce a delay in detecting attacks and its value depends on the monitored application. We opted for 50K because the SPEC CPU tools generate a high number of *online measurements* overall. However, this parameter strictly depends on the monitored application. This experiment shows that we can use compression algorithms to mitigate the network congestion and keep a high verification speed.

7.4 Attack Detection

Here, we describe the security guarantees introduced by ScaRR.

Code Injection. In this scenario, an attacker loads malicious code, *e.g.*, *Shellcode*, into memory and executes it by exploiting a memory corruption error [38]. A typical approach is to inject code into a buffer which is under the attacker control. The adversary can, then, exploit vulnerabilities (*e.g.*, buffer overflows) to hijack the program control-flow towards the shellcode (*e.g.*, by corrupting a function return address).

When a $W \oplus X$ protection is in place, this attempt will generate a memory protection error, since the injected code is placed in a writable memory area and it is not executable. In case there is no $W \oplus X$ enabled, the attack will generate a wrong *LoA* detected by the *Verifier*.

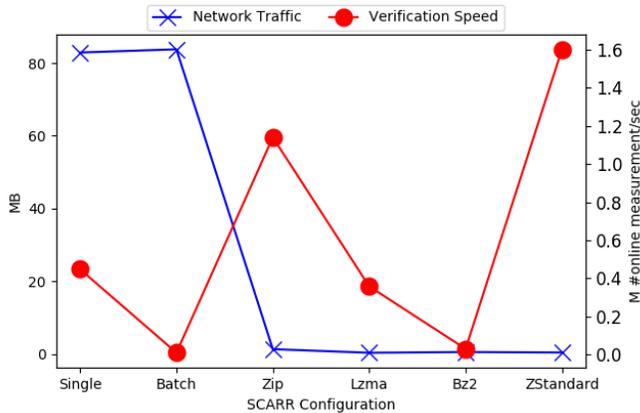


Figure 10: Comparison of different approaches for generating partial reports in terms of network traffic and verification speed.

Another strategy might be to overwrite a node (*i.e.*, a BBL) already present in memory. Even though this attempt is mitigated by $W \oplus X$, as executable memory regions are not writable, it is still possible to perform the attack by changing the memory protection attributes through the operating system interface (*e.g.*, the `mprotect` system call in Linux), which makes the memory area writable. The final result would be an override of the application code. Thus, the static RA of ScaRR can spot the attack.

Return-oriented Programming. Compared to previous attacks, the code-reuse ones are more challenging since they do not inject new nodes, but they simply reorder legitimate BBLs. Among those, the most popular attack [37] is ROP [17], which exploits small sequences of code (gadgets) that end with a `ret` instruction. Those gadgets already exist in the programs or libraries code, therefore, no code is injected. The ROP attacks are Turing-complete in nontrivial programs [17], and common defence mechanisms are still not strong enough to definitely stop this threat.

To perform a ROP attack, an adversary has to link together a set of gadgets through the so-called ROP chain, which is a list of gadget addresses. A ROP chain is typically injected through a stack overflow vulnerability, by writing the chain so that the first gadget address overlaps a function return address. Once the function returns, the ROP chain will be triggered and will execute the gadget in sequence. Through more advanced techniques such as stack pivoting [19], ROP can also be applied to other classes of vulnerabilities, *e.g.*, heap corruption. Intuitively, a ROP attack produces a lot of new edges to concatenate all the gadgets, which means invalid *online measurements* that will be detected by ScaRR at the first *checkpoint*.

Jump-oriented Programming. An alternative to ROP attacks are the JOP ones [16, 48], which exploit special gadgets based on indirect `jump` and `call` instructions. ScaRR

can detect those attacks since they deviate from the original control-flow.

Function Reuse Attacks. Those attacks rely on a sequence of subroutines, that are called in an unexpected order, *e.g.*, through virtual functions calls in C++ objects. ScaRR can detect these attacks, since the ScaRR control-flow model considers both the calling and the target addresses for each procedure call. Thus, an unexpected invocation will result in a wrong *LoA*. For instance, in Counterfeit Object-Oriented Programming (COOP) attacks [36], an attacker uses a loop to invoke a set of functions by overwriting a *vtable* and invoking functions from different calling addresses generates unexpected *LoAs*.

8 Discussion

In this section we discuss limitations and possible solutions for ScaRR.

Control-flow graph. Extracting a complete and correct CFG through static analysis is challenging. While using CRAB as abstract domain framework, we experienced some problems to infer the correct forward destinations in case of virtual functions. Thus, we will investigate new techniques to mitigate this limitation.

Reducing context-switch overhead. ScaRR relies on a continuous context-switch between user-space and kernel-space. As a first attempt, we evaluated SGX as a trusted platform, but we found out that the overhead was even higher due to SGX clearing the Translation-Lookaside Buffer (TLB) [40] at each enclave exit. This caused frequent page walks after each enclave call. A similar problem was related to the Page-Table Isolation (PTI) [46] mechanism in the Linux kernel, which protects against the Meltdown vulnerability. With PTI enabled, TLB is partially flushed at every context switch, significantly increasing the overhead of syscalls. New trusted platforms have been designed to overcome this problem, but, since they mainly address embedded software, they are not suitable for our purpose. We also investigated technologies such as Intel PT [25] to trace control-flow events at hardware level, but this would have bound ScaRR to a specific proprietary technology and we also found that previous works [25, 27] experienced information loss.

Physical attacks. Physical attacks are aimed at diverting normal control-flow such that the program is compromised, but the computed measurements are still valid. Trusted computing and RA usually provide protection against physical attacks. In our work, we mainly focus on runtime exploitation, considering that ScaRR is designed for a deployment on virtual machines. Therefore, we assume to have an adversary performing an attack from a remote location or from the user-space and the hosts not being able to be physically compromised. As a future work, we will investigate new solutions to prevent physical attacks.

Data-flow attestation. ScaRR is designed to perform runtime RA over a program CFG. Pure data-oriented attacks might force the program to execute valid, but undesired paths without injecting new edges. To improve our solution, we will investigate possible strategies to mitigate this type of attacks, considering the availability of recent tools able to automatically run this kind of exploit [28].

Toward a full semantic RA. We will investigate new approaches to validate series of *online measurements* by using runtime abstract interpretation [25, 27, 33].

9 Related Work

Runtime RA shares properties with classic CFI techniques. Thus, we discuss current state-of-the-art of both research areas.

Remote Attestation. Existing RA schemes are based on a cryptographic signature of a piece of software (*e.g.*, software modules, BIOS, operating system). Commercial solutions that implement such mechanisms are already available: TPM [42], SGX [18], and AMD TrustZone [47]. Academic approaches, which focus on cloud systems, are proposed by Liangmin et al. [45] and Haihe et al. [12]. More specifically, their solutions involve a static attestation schema for infrastructures as a service and JVM cloud computing, respectively. Even though these technologies can provide high-security guarantees, they focus on static properties (*i.e.*, signatures of components) and cannot offer any defence against runtime attacks.

To overcome design limitations of static RA, researchers propose runtime RA. Kil et al. [30] analyze base pointers of software components, such as stack and heap, and compare them with the measurements acquired offline. Bailey et al. [13] propose a coarse-grained level that attests the order in which applications modules are executed. Davi et al. [20] propose a runtime attestation based on policies, such as the number of instructions executed between two consecutive returns. Previous works suggest first to acquire a runtime measurement of software properties, but do not provide a fine-grained control-flow analysis.

A modern fine-grained control-flow RA is represented by C-FLAT, which is proposed by Abera et al. [8]. This schema measures the valid execution paths undertaken by embedded systems and generates a hash, which length depends on the number of control-flow events encountered at runtime. Then, the hash is compared with a list of offline measurements. The main differences between ScaRR and C-FLAT are the following ones: (i) C-FLAT control-flow representation grows along with software complexity, while ScaRR manages complex control-flow paths by using partial reports, and (ii) ScaRR is designed to use features of modern computer architectures (*e.g.*, multi-threading, bigger buffers). Dessouky et al. propose LO-FAT [22], which is a C-FLAT hardware implementation aimed at improving runtime performances for embedded systems. However, LO-FAT inherits all of C-FLAT design

limitations in terms of control-flow representation. Zeitouni et al. designed ATRIUM [50], that strengthens runtime RA schemes against physical attacks for embedded devices. Even though the authors address different use cases, this solution might be combined with ScaRR.

Dessouky et al. propose LiteHax [21], that deals with data-only attacks. Their approach shares some similarities with ScaRR: they send detailed control-flow events information to a *Verifier*. However, they target data-oriented attacks (instead of control-flow). Moreover, LiteHax uses symbolic execution to validate the reports, which slows down the verification phase. Abera et al. discuss DIAT [9], which is a scalable RA for collaborative autonomous system. They model a runtime control-flow as a multi-set. This allows DIAT to represent complex control-flow graphs by using a relatively short hash. However, its model loses information about the execution order of the branches. This makes their approach prone to attacks like COOP [36]. ScaRR, instead, combines a strong static analysis and a shadow execution at the *Verifier* side that provides a sound approach by design. Overall, our experiments show that ScaRR can handle a higher number of branches per second compared to all the state-of-the-art runtime RA schemes.

Haldar et al. [26] propose a semantic RA, which leverages a virtual machine to validate semantic properties (*e.g.*, subclass inherited). However, the authors focus on run-time languages, while ScaRR works at a binary level.

Control-Flow Integrity. In the last few years, some authors have proposed architectures that share some similarities with RA [23, 27, 33]. These works are composed by two concurrent processes: a target process (that might be under attack), and a monitor process (that validate some target property). However, ScaRR considers a different attacker model since we consider a fully compromised user-space, *i.e.*, an attacker may tamper with the target software code or attack the monitor process itself. Moreover, unlike ScaRR, these solutions are not designed to provide any report about the execution path of the target process.

10 Conclusion

In this work, we propose ScaRR, the first schema that enables runtime RA for complex systems to detect control-flow attacks generated in user-space. ScaRR relies on a novel control-flow model that allows to: (i) apply runtime RA on any software regardless of its complexity, (ii) have intermediate verification of the monitored program, and (iii) obtain a more fine-grained report of an incoming attack.

We developed ScaRR and evaluated its performance against the set of tools of the SPEC CPU 2017 suite. As a result, ScaRR outperforms existing solutions for runtime RA on complex systems in terms of attestation and verification speed, while guaranteeing a limited network traffic.

Future works include: investigating techniques to extract more precise CFG, facing compromised operating systems, and studying new verification methods for partial reports.

Acknowledgments

This work was partly supported by the SUTD start-up research grant SRG-ISTD-2017-124 and by the European Commission under the Horizon 2020 Programme (H2020), as part of the LOCARD project (Grant Agreement no. 832735).

References

- [1] Bzip2, 2002. Last access March 2019.
- [2] Amazon web services (aws), 2006. Last access March 2019.
- [3] Microsoft azure, 2010. Last access March 2019.
- [4] Blake2, 2013. Last access March 2019.
- [5] Mcsema, 2014. Last access Feb 2019.
- [6] Zstandard, 2016. Last access March 2019.
- [7] Zlib, 2017. Last access March 2019.
- [8] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.
- [9] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous systems.
- [10] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013.
- [11] Jean-Philippe Aumasson, Willi Meier, Raphael C.-W. Phan, and Luca Henzen. *BLAKE2*, pages 165–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
- [12] Haihe Ba, Huaizhe Zhou, Shuai Bai, Jiangchun Ren, Zhiying Wang, and Linlin Ci. jmonatt: Integrity monitoring and attestation of jvm-based applications in cloud computing. In *Information Science and Control Engineering (ICISCE), 2017 4th International Conference on*, pages 419–423. IEEE, 2017.
- [13] Katelin A Bailey and Sean W Smith. Trusted virtual containers on demand. In *Proceedings of the fifth ACM workshop on Scalable trusted computing*, pages 63–72. ACM, 2010.
- [14] Arash Baratloo, Navjot Singh, Timothy K Tsai, et al. Transparent run-time defense against stack-smashing attacks. In *USENIX Annual Technical Conference, General Track*, pages 251–262, 2000.
- [15] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, 2000.
- [16] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.
- [17] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, pages 385–399, 2014.
- [18] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.
- [19] Dino Dai Zovi. Practical return-oriented programming. In *SOURCE Boston*, 2010.
- [20] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54. ACM, 2009.
- [21] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: Lightweight hardware-assisted attestation of program execution. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD '18*, pages 106:1–106:8, New York, NY, USA, 2018. ACM.
- [22] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2017.
- [23] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Conference on Security Symposium, SEC'17*, pages 131–148, Berkeley, CA, USA, 2017. USENIX Association.

- [24] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. An abstract domain of uninterpreted functions. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 85–103. Springer, 2016.
- [25] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17*, pages 585–598, New York, NY, USA, 2017. ACM.
- [26] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, volume 2004, 2004.
- [27] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 1470–1486, New York, NY, USA, 2018. ACM.
- [28] 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 *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.
- [29] Chongkyung Kil, Jinsuk Jun, 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.
- [30] Chongkyung Kil, Emre C Sezer, Ahmed M Azab, Peng Ning, and Xiaolan Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 115–124. IEEE, 2009.
- [31] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.
- [32] E Jebamalar Leavline and DAAG Singh. Hardware implementation of lzma data compression algorithm. *International Journal of Applied Information Systems (IJ AIS)*, 5(4):51–56, 2013.
- [33] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 1635–1648, New York, NY, USA, 2018. ACM.
- [34] Gian Filippo Pinzari. Introduction to nx technology, 2003.
- [35] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult. A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys Tutorials*, 19(2):1145–1172, Secondquarter 2017.
- [36] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.
- [37] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.
- [38] Nathan P Smith. Stack smashing vulnerabilities in the unix operating system, 1997.
- [39] William Stallings. The advanced encryption standard. *Cryptologia*, 26(3):165–188, July 2002.
- [40] Paulus Stravers and Jan-Willem van de Waerdt. Translation lookaside buffer, December 10 2013. US Patent 8,607,026.
- [41] Toshio Sukanuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the ibm java just-in-time compiler. *IBM systems Journal*, 39(1):175–193, 2000.
- [42] Allan Tomlinson. Introduction to the tpm. In *Smart Cards, Tokens, Security and Applications*, pages 173–191. Springer, 2017.
- [43] Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. Memory errors: The past, the present, and the future. In *International Workshop on Recent Advances in Intrusion Detection*, pages 86–106. Springer, 2012.
- [44] Sebastian Vogl, Jonas Pföh, Thomas Kittel, and Claudia Eckert. Persistent data-only malware: Function hooks without code. In *NDSS*, 2014.

- [45] Liangming Wang and Fagui Liu. A trusted measurement model based on dynamic policy and privacy protection in iaas security domain. *EURASIP Journal on Information Security*, 2018(1):1, 2018.
- [46] Robert NM Watson, Jonathan Woodruff, Michael Roe, Simon W Moore, and Peter G Neumann. Capability hardware enhanced risc instructions (cheri): Notes on the meltdown and spectre attacks. Technical report, University of Cambridge, Computer Laboratory, 2018.
- [47] Johannes Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.
- [48] Fan Yao, Jie Chen, and Guru Venkataramani. Jop-alarm: Detecting jump-oriented programming-based anomalies in applications. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 467–470. IEEE, 2013.
- [49] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-assisted fine-grained code-reuse attack detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 66–85. Springer, 2015.
- [50] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 384–391. IEEE Press, 2017.

Toward the Analysis of Embedded Firmware through Automated Re-hosting

Eric Gustafson^{1,2}, Marius Muench³, Chad Spensky¹, Nilo Redini¹, Aravind Machiry¹, Yanick Fratantonio³
Aurélien Francillon³, Davide Balzarotti³, Yung Ryn Choe², Christopher Kruegel¹, and Giovanni Vigna¹

¹University of California, Santa Barbara

{edg, cspensky, nredini, machiry, chris, vigna}@cs.ucsb.edu

²Sandia National Laboratories

{edgusta, yrchoe}@sandia.gov

³EURECOM

{marius.muench, francill, yanick.fratantonio, balzarot}@eurecom.fr

Abstract

The recent paradigm shift introduced by the Internet of Things (IoT) has brought embedded systems into focus as a target for both security analysts and malicious adversaries. Typified by their lack of standardized hardware, diverse software, and opaque functionality, IoT devices present unique challenges to security analysts due to the tight coupling between their firmware and the hardware for which it was designed. In order to take advantage of modern program analysis techniques, such as fuzzing or symbolic execution, with any kind of scale or depth, analysts must have the ability to execute firmware code in emulated (or virtualized) environments. However, these emulation environments are rarely available and are cumbersome to create through manual reverse engineering, greatly limiting the analysis of binary firmware.

In this work, we explore the problem of *firmware re-hosting*, the process by which firmware is migrated from its original hardware environment into a virtualized one. We show that an approach capable of creating virtual, interactive environments in an automated manner is a necessity to enable firmware analysis at scale. We present the first proof-of-concept system aiming to achieve this goal, called PRETENDER, which uses observations of the interactions between the original hardware and the firmware to automatically create models of peripherals, and allows for the execution of the firmware in a fully-emulated environment. Unlike previous approaches, these models are interactive, stateful, and transferable, meaning they are designed to allow the program to receive and process new input, a requirement of many analyses. We demonstrate our approach on multiple hardware platforms and firmware samples, and show that the models are flexible enough to allow for virtualized code execution, the exploration of new code paths, and the identification of security vulnerabilities.

1 Introduction

The new wave of commercialized embedded systems, brought about by trends such as the IoT, has resulted in their use for an

increasing number of security and safety-critical applications. The most unusual feature of this new computing paradigm is its extreme diversity, in terms of both hardware and software. At the software level, each new device comes with its unique firmware, which is purpose-built for its specific function, and may not include a conventional operating system. At the hardware level, each device includes its own unique selection of hardware, both on the board (sensors, actuators, etc.) and on the chip (bus controllers, timers, and other I/O peripherals), which combine to form the unique execution environment of the firmware.

Unfortunately for security researchers, in stark contrast to the desktop and mobile ecosystems, market forces have not created any *de facto* standard for components, protocols, or software, hampering existing program analysis approaches, and making the understanding of each new device an independent, mostly manual, time-consuming effort.

Emulators for these systems are a key component in enabling dynamic analysis of the firmware at scale, as transparent on-device analysis is rarely possible, and it is impractical to acquire hundreds of identical physical devices to parallelize the analysis process. However, appropriate emulators are typically unavailable, particularly due to the impracticality of properly supporting the thousands of incompatible embedded CPUs, and an enormous selection of external peripherals. Worse yet, the physicality of these devices means that analyzing their firmware without the sensors, actuators, and other components may not be useful, or even possible at all.

Previous efforts have avoided the problem through the use of an operating system abstraction [3, 8], or with a hardware-in-the-loop scheme [15, 16, 26]. However, these techniques impose severe limits on the scale and scope of analyzable targets, such as requiring that a general-purpose OS is present, or a significant amount of potentially costly original hardware to be tractable. Without these approaches, analysts must manually implement models of all the on-chip and off-chip peripherals for a device. This requires that the analyst can obtain complete documentation or thorough understanding for every component of the system, and spends the time

to manually develop components usable by the emulator. Manufacturers can also use completely custom components, for which no documentation can be obtained, rendering emulation by any existing method extremely difficult.

We explore the possibility of automated *firmware re-hosting*. The key idea behind firmware re-hosting consists of analyzing a given firmware/hardware combination (possibly through multiple execution rounds), understanding what the firmware expects from the surrounding hardware, and then attempting to *replace the hardware altogether, so that the firmware analysis can be carried out with software-only components*. In essence, firmware re-hosting would allow analysts to decouple the execution of firmware from the hardware on which it expects to be executed. This allows for the scaling of popular dynamic analysis techniques, outperforming hardware-in-the-loop or device-only approaches [20].

We identified four key aspects that are necessary for building a re-hosting solution to deal with today’s embedded firmware: A re-hosting scheme must be *virtual* to allow for scale and reduce costs; should also be *interactive*, to allow the firmware to process new input and actually withstand program analysis; should be *abstraction-less* (i.e., it should not rely on high-level concepts, such as operating systems and hardware abstraction layers) to allow the system to handle the widest possible variety of firmware. Finally, re-hosting should be *automated*, so that the system can overcome the extreme diversity that is impractical for humans to handle. Although previous approaches to the problem are numerous, all are missing at least one of these aspects.

In this work, we develop an approach to re-hosting that achieves all of them, and propose a proof-of-concept system, called PRETENDER, which is able to observe hardware-firmware interactions and create models of hardware peripherals automatically. Our system first creates a recording of real interactions between the firmware and its hardware, and uses machine learning and pattern recognition techniques to create models for each peripheral on the CPU. The generated models can then be leveraged by popular full-system emulators (e.g., QEMU [2]) or program analysis engines (e.g., angr [23]) to enable precise, scalable, interactive analyses of the accompanying firmware.

While automated re-hosting may seem conceptually straightforward, the challenges in modeling even simple hardware-firmware interactions are numerous. We may think of a peripheral, such as a serial port, as a simple object that sends and receives data, but the firmware’s view of this hardware is much more complex, consisting of dozens of individual configuration, status, or data registers, which, from the point-of-view of the firmware, appear as only opaque memory accesses, without any indication of their layout or behavior. Two peripherals performing the same function on two different CPUs, even from the same vendor, vary wildly in terms of memory layout and implementation details. On top of this, accesses to these peripherals occur within the

CPU itself, and obtaining these interactions for modeling is its own challenge. Interrupts are also a common feature of embedded peripherals, and must occur exactly as expected, or the hardware or firmware may fail.

To evaluate our approach, we demonstrate our recording and modeling techniques on a set of six unique “blob” firmware samples, each on three different hardware platforms, with associated external peripheral devices. Our experiments show that PRETENDER is able to successfully extract the peripheral models and execute the firmware in a fully emulated environment. The models offer enough interactivity to allow for the exploration of parts of the program not seen during recording or training. We further show the potential for direct applications to dynamic analysis, by using these modeled environments to trigger synthetic security vulnerabilities in the firmware samples. The hardware modeled in these experiments represents CPUs and other components common to low-power IoT and embedded devices. However, many challenges remain before typical commercial devices can be modeled in full. We nevertheless believe that the goal of automated firmware re-hosting is both achievable and necessary. Therefore, we conclude with a discussion of limitations, open problems, and next steps toward tackling the complexity of commercial devices.

In summary, our contributions are as follows:

- We explore the problem of *firmware re-hosting*, and show that virtual, interactive, automatic, and abstraction-less approaches are needed to handle today’s diverse firmware.
- We present PRETENDER, a proof-of-concept system able to automatically build hardware models, through a mix of novel hardware and interrupt recording techniques, machine learning, and peripheral state approximation.¹
- We apply PRETENDER to multiple firmware samples across multiple hardware platforms and show that the generated peripheral models are accurate, automatic, and interactive enough to enable program analysis and vulnerability discovery.

2 The Re-hosting Problem

To deal with the plethora of software applications that need to be analyzed on desktop and mobile platforms, the security community has developed many techniques for enabling the scalable analysis of programs to find bugs and detect malice. In this section, we examine what makes embedded systems different and much less tractable to these techniques, as well as propose qualities that a system capable of analyzing arbitrary firmware must have.

Today’s state-of-the-art program analysis techniques, including dynamic analysis tools such as AFL [27] or symbolic execution engines such as angr [23] or S2E [4], rely on some

¹To allow the reproducibility of this work, the source code to this work is available at <https://github.com/ucsb-seclab/pretender>

form of abstraction to be tractable. Dynamic approaches typically rely on virtualization to enable parallel, scalable analyses, while symbolic approaches rely on function summarization of the underlying operating system to minimize the code that they need to execute. In order to use any of these tools, the analyst must take the program out of its original execution environment, and provide a suitable analysis environment able to execute it. This is a process referred to as *re-hosting*.

For desktop and mobile programs, the standardization of the execution environments (e.g., commodity hardware, which consists of a relatively small number of OSes and architectures) has made this re-hosting process simpler. However, with embedded firmware, many well-established assumptions fail. For example, there may not be a general-purpose operating system designed to run arbitrary code on the device, leaving the analyst to deal with the hardware directly. This is especially true for low-power IoT devices, which are typically based on microcontroller-class CPUs that lack the ability to run such OSes. Firmware for these devices is typically obtained in the form of a *binary blob*, an opaque code object containing no metadata about its contents. How this blob is handled is entirely dependent on the CPU hardware, and will vary widely from chip to chip. This also makes distinguishing between library code and device-specific code challenging. With no visible abstractions to use, the execution environment for embedded firmware is the hardware itself. We can break this hardware down into three distinct categories:

- **CPU Core.** The CPU core itself must, of course, be emulated. This includes the instruction set, but also any function able to directly alter code execution, such as the chip’s primary interrupt controller.
- **On-Chip Peripherals.** These peripherals include timers, bus controllers, serial ports, General Purpose Input and Output (GPIO), and other features typically included on the die of the CPU itself. Most CPUs expose these peripherals to the program as Memory-Mapped Input/Output (MMIO), where they are organized as a group of contiguous memory locations, that do not behave like normal memory. Each group may contain multiple locations, used for configuring, checking the status of, and exchanging data with the peripheral. An example of a typical MMIO peripheral mapping is shown in Figure 1. On-chip peripherals are also responsible for issuing *interrupts*, events that trigger asynchronous changes in control flow in response to a hardware event. More precisely, a peripheral is associated with one or more numbered interrupt “channels” or “lines”; when an interrupt occurs, the code in the firmware associated with that interrupt (known as an Interrupt Service Routine, or ISR) is executed. When, how, and why a peripheral issue interrupts are all properties of the peripheral’s hardware on a particular chip, but typically includes the arrival of data, the expiration of timers, and error conditions.
- **External Peripherals.** These peripherals are the sensors, actuators, and other circuitry on the device’s circuit

Table 1: Excerpt of tools tackling the re-hosting problem

Tool	Virtual	Interactive	Abstraction-less	Automatic
Simics [17]	✓	✓	✓	-
FIE [9]	✓	✓	✓	-
Avatar [26]	-	✓	✓	-
PROSPECT [14, 15]	-	✓	-	✓
Surrogates [16]	-	✓	✓	-
Firmadyne [3]	✓	✓	-	✓
Avatar ² [19]	✓	✓	✓	-
PRETENDER	✓	✓	✓	✓

board(s). They are exposed to the program only through one of the on-chip peripherals, including GPIO, or a bus such as Inter-Integrated Circuit (I2C) or Serial Peripheral Interface (SPI). While from the programmer’s perspective, communicating with these peripherals is as easy as sending and receiving messages thanks to software libraries, the resulting compiled firmware does so through a complex series of accesses to the MMIO regions of on-chip peripherals, making the direct flow of data in and out of each peripheral difficult to observe. This is also the source of the most variety in embedded systems, as these devices typically contain entirely-custom circuit boards, with whatever array of components the designers felt were necessary.

2.1 Re-hosting Aspects and Related Work

Many solutions have been proposed to enable firmware re-hosting, each with their own qualities and drawbacks. To showcase their differences, we identify four salient properties that an ideal analysis system, capable of handling arbitrary firmware, should possess. Table 1 shows prevalent tools that tackled the re-hosting problem in the past, and classifies them according to the aspects, which are described as follows.

Virtual. A re-hosting solution should not depend on the presence of hardware during analysis. Many proposed approaches to firmware analysis [7, 15, 16, 26] require *hardware-in-the-loop* execution. However, such approaches inherently limit the scale of the analyses. In a dynamic context, only one thread of execution is possible per-device, and re-starting execution, which happens very often in modern fuzzers, can incur a significant time penalty [20]. Symbolic execution is even more impacted by such approaches; analyses using hardware-in-the-loop must be careful to only execute portions of code that do not contain hardware interactions, to avoid corrupting the hardware’s state visible by all parallel code paths being explored. Cost also becomes a factor, as each analyst wishing to explore a set of devices must purchase and instrument the devices, which raises the barrier to entry for firmware analysis. While hardware-in-the-loop techniques do allow for interactive, relatively low-effort analyses, they are by no means adequate for thorough program analyses of arbitrary firmware.

Interactive. A re-hosting solution should be responsive to new program input. While defining the notion of input on an embedded firmware is itself a nuanced problem, the

remaining hardware (not used as the source of input) should react accordingly. Trace replay-based solutions, such as PANDA [10], while quite flexible and useful for certain analyses, are not interactive and cannot be used to implement fuzzing or symbolic execution, which rely on this primitive.

Abstraction-less. An ideal re-hosting solution should not rely on a software abstraction that greatly limits the kinds of firmware on which it can be used. Recently, advances have been made in re-hosting firmware based on the abstractions provided by the Linux OS [3, 8]. Using such an abstraction, when it exists, is advantageous, but it naturally limits the scope of firmware to those that do not have a significant coupling between their primary function and the underlying hardware. Relying on an OS precludes the analysis of, for example, the *blob* firmware we explore in this work.

Automatic. An ideal re-hosting solution should not require a significant effort per-device to use. The diversity in on-chip and external peripherals is so severe, that it is highly unlikely that any firmware can be emulated out-of-the-box with a commercial or open-source emulation package. While some commercial systems provide the ability to rehost completely custom hardware architectures (e.g., Simics [17]), these systems still require the hardware models to be programmed manually. This is made worse by customizable CPU cores, and the diverse array of electronics components that the electronics industry continues to support. Even static and symbolic analysis tools [9, 12, 22] heavily rely on the manual specification of hardware behavior, particularly around IO and interrupts.

While there is little useful data able to quantify embedded CPU diversity, and documentation from vendors is not in a comparable form, we managed to locate a dataset of 555 CMSIS System View Description (SVD) files [21], which are XML files describing chipsets based on Cortex-M microcontrollers. They detail the on-chip peripheral locations and layouts of 463 distinct chips across 13 different chip vendors. This collection is by no means complete (it does not even include all of the chips used in our experiments in Section 4), but it shows the complexity and the scale of this problem. In this dataset alone, we could identify 1592 unique implementations of peripherals demonstrating the immense variety of peripheral and chip designs.

This complexity increases even more when considering external peripherals connected to the chip via on-chip buses and interrupt controllers. Hence, emulators such as QEMU [2] have to include carefully and—up to now—manually crafted implementations of peripherals and align them at the right location. In fact, the upstream version of QEMU only exposes implementations for three different Cortex-M chips, none of them present in the above dataset. As a result, analysts end up creating their own peripheral and board implementations and maintaining them in separate forks of the project, such as *QEMU STM32* [1] or *GNU MCU Eclipse* [13]. A different approach is taken by *LuaQEMU* [5] and *avatar*² [19], which provide an interface for the analyst to define the peripheral

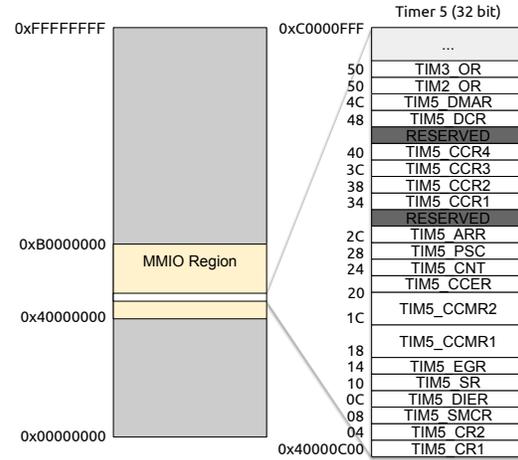


Figure 1: The memory layout for a simple 32 bit memory-mapped timer on the STM32 embedded processor.

layout. While these may be preferable to languages such as C used by QEMU itself, the analyst is still required to obtain and understand the full documentation for the particular CPU model used, and this effort may not transfer entirely to other similar CPUs, even from the same vendor. Therefore, it is very clear that an automated solution is needed to be able to make firmware analysis tractable.

While we, of course, do not claim to have achieved the goal of ideal re-hosting in this work, in the following sections, we will showcase a proof-of-concept approach that has all of the above properties, with limitations discussed in Section 5.

3 Methodology

In this section, we present PRETENDER, a step toward automating the modeling of MMIO and interrupt-driven hardware peripherals to enable re-hosting. The goal is to gather data on, and build models of, these peripherals, such that the firmware under analysis can later be independently executed in a CPU emulator. We present our solution in the context of its use to support dynamic analysis of firmware, although the generated models have other possible uses, which we will discuss in more detail in Section 5.

The success metric we adopt to evaluate the completeness of the extracted models is what we call *survivable execution*, which we define as the ability for the firmware to execute the same regions of code as it would if the original hardware were present, without faulting, stalling, or otherwise impeding this process. We include in this definition the need for our program to be interactive, as this is a requirement for many analyses. That is, the firmware and our hardware models need to be able to *operate on inputs and execute code paths that were not observed during the recording and model-generation phase*.

Assumptions and Prerequisites. We make a few basic

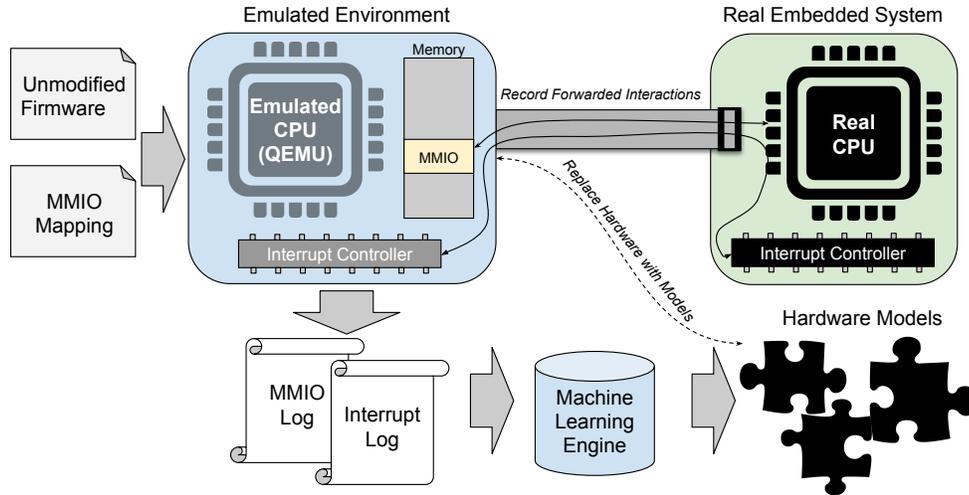


Figure 2: Overview of the functionality of PRETENDER

assumptions in the implementation of PRETENDER.

- We assume that a CPU emulator is available for the target device, and that this emulator supports all CPU features that can impact control flow, including the interrupt controller.
- We assume the analyst has the ability to observe memory accesses and the occurrence of interrupts in the device in real-time. We will present a method for accomplishing this on any device with a basic debugging interface, lowering the requirement to the ability to read and write the device’s memory.
- We assume that the basic memory layout of the target device is known, particularly the location of code and data in the memory space. More generally, we need to know where these areas are *not* located, as we can assume that the remaining areas are interesting locations we wish to model, including the MMIO regions.
- We assume that a human or automated process is able to interact with the hardware and that it achieves sufficient code coverage during the recording phase to reveal enough hardware interactions to generate a model. The more complete the code coverage is, the more detailed the extracted model will be.

A discussion of these assumptions can be found in Section 5. PRETENDER works in the following phases:

1. **Recording.** We instrument the device to obtain a trace of accesses to the MMIO regions, and any interrupt that occurs during the execution.
2. **Peripheral Clustering.** We locate the boundaries of each distinct peripheral within the device’s memory space, and divide the recording into sub-recordings for each peripheral.
3. **Interrupt Inference.** Based on the interleaving of interrupts with MMIO, we assign each numbered interrupt event to a peripheral group. We then infer which bits in

which memory location in the peripheral control interrupts, and create timing patterns to be used during emulation.

4. **Memory Model Training.** In this step, we attempt to select and train known models for each memory location within the identified peripheral regions. Any unidentified memory locations will be modeled using State Approximation.
5. **Test Harness Creation.** Finally, the analyst must decide how input should be introduced into the system, through the creation of a simple test harness. This is the only manual step in the process, as the decision depends on the analyst’s needs.

A complete overview of PRETENDER and the interplay between its different parts can be seen in Figure 2. In the remainder of this section, we will discuss the individual phases of the system in detail.

3.1 Recording

On ARM-based platforms, MMIO accesses occur through normal load or store instructions from the CPU, and take place across the CPU’s internal memory buses. Since we cannot observe this activity directly, or either via a debugger or through physical access, we can instead effectively extend the memory bus outside the chip where the data required for modeling can be recorded. To this end, we leverage a hardware-in-the-loop execution approach, where the firmware is deployed in an emulator, and the MMIO requests are forwarded to the original hardware, which allows recording in-transit. We built upon the *avatar*² framework [19], which allows for the simultaneous control and orchestration of emulators and hardware. *Avatar*² supports an event-based callback infrastructure, which allowed us to implement the recording of memory events. All extensions and modifications to *avatar*² developed during this

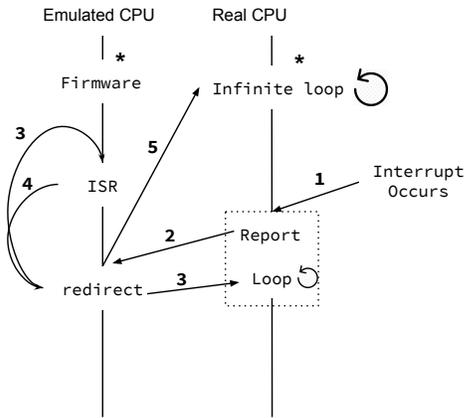


Figure 3: State diagram of interrupt recording in PRETENDER. * indicates the initial state.

work will be released as open-source alongside with the code of PRETENDER upon the publication of this paper.

Recording Interrupts In order to fully model on-chip hardware peripherals, we must observe the interrupts that they generate, *in the context of the MMIO activity of the firmware.*² Figure 3 shows how interrupts are recorded in PRETENDER. As interrupts are generated on the real device, we should have the Real CPU running. Hence, we always have the Real CPU execute an infinite loop. Furthermore, we replace the ISR of all the interrupts with a recording stub (shown in dotted box in the Figure 3).

When an interrupt occurs (Step 1), the recording stub is triggered, which immediately reports the interrupt number to PRETENDER (i.e., the Emulated CPU), and halts the Real CPU (Step 2). The emulated CPU then starts executing the actual ISR for the corresponding interrupt, and directs the real CPU to run a loop in the interrupt’s context to mimic the execution of the interrupt (Step 3). Once the ISR completes execution on the emulated CPU (Step 4), PRETENDER redirects the execution of the Real CPU to the default infinite loop, and the Emulated CPU to continue executing the firmware (Step 5). This ensures that both the hardware and emulated interrupt controllers are synchronized.

3.2 Peripheral Clustering

With the combined MMIO and interrupt recording collected, we can now proceed to reason about and model the peripherals themselves. In the end, we need to construct a model, such that each MMIO location that the firmware accesses returns a reasonable value. However, these locations are not independent; multiple locations represent one logical device in the silicon of

²Recording interrupts is a particularly complex matter, requiring precise synchronization of the emulator and hardware to avoid incorrect behavior. We detail the problem and the rationale behind our approach in Appendix A.

the chip, which has its own concept of state, control interrupts, and so on. For example, writing a byte to the data register of a serial port may cause the “transfer in progress” or “busy” flag to become active in the same peripheral’s status register. Therefore, a major prerequisite to the future modeling steps is to group all memory accesses by their associated peripherals.

To do this, we rely on the intuition that each MMIO peripheral is typically associated with a block of contiguous memory addresses (e.g., 0xC00–0xCFF in Figure 1). While we cannot be sure exactly what the boundaries between the peripherals are, we assume there is some fixed alignment for—and the minimal gap between—them, likely due to the underlying details of the peripheral buses that serve MMIO peripherals. These details are supported by the SVD data explored in Section 2, as well as the manuals for all of the devices explored in Section 4. We can, therefore, find our peripheral boundaries through clustering techniques. For this work, we take the set of accessed addresses and employ the Density-based Spatial Clustering of Applications with Noise (DBSCAN) algorithm [11] to recover the peripheral groupings.

The intuition behind this choice is that each peripheral will appear as a small cluster of accesses in a relatively sparse memory space. For example, in Figure 1, while an entire page of memory (0x1000) is allocated to the timer, only a small portion (0x00–0x50) of that memory space is actually used, meaning that subsequent peripherals in memory will likely have large gaps between their relative clusters. DBSCAN is able to quickly discern these clusters, providing us with the capability to efficiently group the various accesses. In our work, we set our maximum gap between any of the addresses in a cluster (i.e., *epsilon*) to be 0x100 and the minimum cluster size to be *one*. Almost any reasonable value for *epsilon* (e.g., 0x8–0x100) would likely produce identical and useful clusters, and our minimum cluster size of one ensures that we will not exclude simple or infrequently-used peripherals from our models.

3.3 Interrupt Inference

In order to model interrupts correctly, we need to establish a reasonable approximation for when to fire each interrupt and which MMIO event triggered it. First, we find the association between the interrupt number and the peripheral firing the interrupt, which is a property of the hardware that varies widely between chip models. Then, we discern which MMIO register is used to enable and disable each interrupt, so that we do not fire it too soon or too late in the execution. Finally, we determine how often to fire interrupts when they are eventually enabled.

To associate an interrupt with a peripheral, we examine the interleaved interrupt and MMIO traces and locate all of the MMIO operations that occur during an Interrupt Service Routine (ISR) (e.g., between an interrupt event and the emulator returning from the ISR). We leverage the

intuition that the purpose of most interrupts is to trigger the firmware to communicate with the interrupting peripheral, by executing the code in the ISR. Therefore, we associate an interrupt number with a peripheral if that peripheral's MMIO addresses were accessed the most during the ISR's execution.

We then locate the memory location containing the interrupt's *trigger*, which is a location in the peripheral which, when a certain bit pattern is written, causes interrupts to be enabled. The location can be determined by finding the very first interrupt for a given interrupt number, and seeking backward in the MMIO/interrupt trace until a write to the associated peripheral is found. This is intuitively the configuration, or interrupt-enable register, as it is best practice to enable interrupts as the final step during peripheral configuration, as, after this point, any operation could be interrupted. However, this memory location may be shared with other functions, and many bit patterns may be written to it during an execution which have no effect on interrupts. The next step is therefore to refine the bit pattern which can enable interrupts in the model, based on which writes appear to control interrupt behavior in the hardware. We start with the assumption that all bits in the trigger location control the interrupts. For each write to the detected trigger location, if a bit is set to 0 when interrupts occur, it is unlikely to be the interrupt trigger bit, and is removed from consideration. The remaining bits are considered the final interrupt trigger; during emulation, when these bits are set in the trigger location, interrupt events will be fired by the model.

Finally, we must determine how often to fire interrupts when they are enabled. There are various kinds of interrupts: *pulse* interrupts occur once for every event they represent, and *level* interrupts occur repeatedly until some MMIO action disables them. While level interrupts would be easy to model based on the state of the peripheral, we cannot reliably distinguish these two types in the recording data. As a result, the most general, flexible approach is to use interrupt timings. Interrupts can also be very frequent. Since these are the timings seen during PRETENDER's recording, we can be sure that the emulator can at least support interrupts at this speed. We collect the timings between an interrupt return and the beginning of the next interrupt (as well as between the trigger and the first interrupt) and create a repeating sequence. As long as interrupts are enabled via the correct bits in the interrupt trigger location, they will be fired repeatedly until they are disabled.

The result is a peripheral model for which interrupts can be enabled and disabled by the program in a realistic manner, and with timing intervals that the emulator can support. We find that these intuitive heuristics both align well with the design of peripherals, and also work well in practice, as we show in Section 4.

3.4 Memory Model Training

In this step, we select a model for each memory location in a peripheral. We first look for common memory access patterns, which allow us to train accurate models for these common types of interactions. For some memory locations, where more complex, stateful, functionality is implemented, we employ a *state approximation* mechanism, able to provide known-valid sequences of observed values for that specific memory location, based on what state we infer the peripheral to be in.

There are a few basic types of MMIO registers common to many peripherals (e.g., configuration registers, status registers, and counters). By using simplified models for these, we can allow this part of our model to maintain flexibility, and operate as independently as possible from the circumstances of the recording. We identify and model a number of classes of MMIO:

- the *Simple Storage Model* is used for memory locations that were observed to *always* act like normal memory. That is, the value returned for a read from a location was always identical to the most recent value written to that location;
- the *Pattern Model* is used for memory locations whose read values appear to follow some repeating pattern (e.g., 0, 1, 1, 0, 1, 1, ...), including locations that always return a static value;
- the *Increasing Model* is used for values that are *eventually* monotonically increasing (i.e., the last half of the observations were increasing), which is typically indicative of a timer or counter;
- and the *Write-only Model* is used for memory locations that were only ever observed to be written to, which are effectively ignored from a modeling perspective, but interesting for our state approximation, as they are likely configuration registers that directly affect the state of the peripheral.

While these models are relatively straightforward, our Increasing Model requires multiple iterations of linear regression modeling to find the best fit line. This is because these incrementing values are typically configured during the boot process, which means that their initially read values are unlikely to be indicative of the actual rate of increase. For example, a counter may start on boot at a certain rate, then the firmware will configure a new rate and reset the timer, resulting in two distinct functions represented by the same memory value. To handle this, we iteratively remove outliers (i.e., values that have a correct p-value greater than 0.0001) from our regression model until we have a good-fitting function for the steady-state increase. When we are replaying this model, we first replay the initial outlier values verbatim, and only switch our projection function once initial values are exhausted and the long-term behavior is expected.

State Approximation. The remainder of locations within a peripheral represent those locations that do not follow any easily identifiable pattern. These locations can represent external sources of input or external physical phenomena,

reflect large amounts of state invisible to the CPU (e.g., the internals of on-chip peripherals), and be related to the behavior of interrupts. Therefore, methods relying on function-fitting or direct recovery of a state machine involving these memory locations simply will not suffice.

As a first step toward addressing these challenges, we instead make an approximation of the device’s state, using only the observed trace’s data and ordering, by inferring state transitions we know must exist. We observe that writes to MMIO addresses are typically used to cause a change in state (e.g., the transmission of data to external hardware or a change in the internal configuration of a peripheral), and approximate that the activity between two writes found in an MMIO recording may roughly represent the same state of the overall peripheral. Interrupts also represent a change in state, although we cannot know concretely what change in state they represent. Reading data can also change the state of a peripheral, but in a more subtle way (e.g., reading a byte from a serial port causes it to be removed from an internal hardware buffer, and a subsequent read to the same address will return a different value).

With these intuitions in mind, our State Approximation model consists of the trace of MMIO and interrupt activity for a given peripheral, and a *state pointer* consisting of where in the trace we believe best approximates the state of both the program and the peripheral. At the beginning of execution, the state points to the beginning of the trace. We update this state based on the following rules: When an MMIO address for this peripheral is read, we look ahead in the trace to find the next time this location was read. If it is found, we return this value, and update the state pointer to this location. If we encounter a write, an interrupt, or the end of the trace before we find one, we instead return the most recent value for that location, and do not update the state pointer. This encodes the behavior that values read from MMIO may be sequential (as in the serial port buffer mentioned earlier) and that they respect the boundaries of state caused by writes and interrupts.

When a write to the peripheral’s MMIO occurs, or the associated interrupt event is triggered, we look forward in the trace for the next location where the same event occurred, and update the state pointer. If we do not find it before the end of the trace, we instead seek backward through the trace. If the value written is entirely new, we do not update the state pointer. These rules allow our model to respond intelligently to changes in its mode, or new commands, regardless of the order they occur during execution, particularly when new input causes deviation from the trace.³

Test Harness Creation. Finally, in order for this system to be fully *interactive*, as we discuss in Section 2, the analyst must decide how input is to be introduced into the emulated environment. No standards exist for input and output in embedded firmware and hardware; exactly where an input is introduced is both a function of the target device’s hardware,

³For a walk-through of the state approximation model in action and the challenges faced by it, see Appendix B.

and the analyst’s goals. For example, a serial port, in one device, could be connected to a human-controlled terminal (the obvious source of input), while in another, it could be wired across the circuit board to a simple sensor with a serial interface (a model-able device). PRETENDER, therefore, requires the analyst to provide their own means of input, in the form of a test harness. We leverage *avatar*²’s Python scripting interface to allow any MMIO location to be easily replaced by custom logic. As an example, for the firmware presented in Section 4, we created a harness consisting of feeding input data via the device’s serial port.

4 Evaluation

To demonstrate the efficacy of PRETENDER, we use it to create models of the hardware in the context of multiple firmware images. We then use these models, together with freshly generated inputs, to uncover code paths and orderings *not seen during recording and modeling*. The newly covered parts of the firmware include synthetic security vulnerabilities, which the system is able to trigger and detect within the modeled environment.

Targets. We applied our system to firmware running on three different embedded CPUs on development hardware, the ST Nucleo L152RE, the Maxim MAX32600MBED [18] and the STM Nucleo F072RB [24]. The targets represent ARM-based microcontrollers common to embedded applications; the first two represent Cortex-M3-based designs, while the latter is based on a Cortex-M0. The layout of the peripherals, and the function of each MMIO register varies widely, even between the two targets from the same vendor. It is worth noting that QEMU has no official support for any of these chips, or any of their contained peripherals. Third-party forks contain partial support for related chips but would have to be heavily adapted and extended to work on these firmware samples. Access to all devices was obtained using a commodity CMSIS-DAP debugger. We showcase the function of our models in-depth in the context of the STM Nucleo L152RE, but provide results from all three.

We evaluated our technique on six example firmware: four of these were directly obtained from the ARM `mbed` [25] development suite’s library of examples. These were designed to exercise interesting features of the hardware, and we chose them to demonstrate the challenges PRETENDER has to overcome for successful hardware modeling. We extended three of these examples with additional functionality, which we do not trigger during the recording and modeling phases. Besides additional hardware interactions, our additions also include synthetic security vulnerabilities, similar to the kind that an analyst may wish to locate in a binary firmware. The other two examples, not taken from the `mbed` examples, are more complex and mimic real-world firmware found on a door lock controller and a thermostat. All of our examples

Table 2: Approximate basic block coverage for firmware samples with PRETENDER, as measured by QEMU

Firmware Name	Peripherals	Blocks Executed			
		Rec.	Null Model	SA	Fuzzing
Nucleo L152RE					
blink_led	Timer, GPIO	218	86	218	n/a
read_hypercentinal	Timer, GPIO, UART	545	85	545	636
i2c_master	Timer, I2C, AM3215	1185	61	1185	n/a
button_interrupt	Timer, GPIO, Button	344	68	314	n/a
thermostat (<i>custom</i>)	Timer, I2C, AM3215	1263	62	1261	1276
rf_door_lock (<i>custom</i>)	Timer, GPIO, Radio,	665	87	665	758
Nucleo F072RB					
blink_led	Timer, GPIO	405	117	405	n/a
read_hypercentinal	Timer, GPIO, UART	828	102	828	999
i2c_master	Timer, I2C, AM3215	1572	103	1572	n/a
button_interrupt	Timer, GPIO, Button	362	103	362	n/a
thermostat (<i>custom</i>)	Timer, I2C, AM3215	1662	103	1662	1918
rf_door_lock (<i>custom</i>)	Timer, GPIO, Radio,	960	102	960	972
MAX32600MBED					
blink_led	Timer, GPIO	280	9	280	n/a
read_hypercentinal	Timer, GPIO, UART	514	8	514	668
i2c_master	Timer, I2C, AM3215	941	8	942	n/a
button_interrupt	Timer, GPIO, Button	188	8	188	n/a
thermostat (<i>custom</i>)	Timer, I2C, AM3215	1009	8	1009	1066
rf_door_lock (<i>custom</i>)	Timer, GPIO, Radio,	692	8	692	712

were compiled using GCC 5.0, and ARM’s `mbed` hardware abstraction layer. While we had the source code available during our analysis, it should be noted that no part of PRETENDER leverages this information; PRETENDER operates solely on binary firmware and the hardware itself. While this may seem like a small number of samples in comparison to previous approaches [3, 8], the need to obtain and instrument original hardware necessarily limits the number of firmware samples.

We evaluated our system’s effectiveness in terms of its achieved code coverage on each example, as measured through execution traces from QEMU. We note that good code coverage during our recording phase is an important factor in our modeling, as we want to explore as much of the hardware’s functionality as possible. Table 2 summarizes the used peripherals and execution behavior of each firmware. We note that the reported block counts are approximate, particularly for those examples with interrupts, as QEMU re-defines basic blocks based on where an interrupt occurs and returns, leading to imprecision. The table shows vastly different amounts of covered basic blocks for the same firmware across different devices, although the exact same compiler, source code, and system library was used for all of the examples. This hints toward the many subtle differences in the hardware abstraction layer, which are required to deal with the diverse hardware platforms. The block count in the “Rec.” column serves for baseline comparison and shows the coverage reached during the initial recording phase. The “Null Model” column represents the coverage obtained when all MMIO is replaced with a model that simply returns a zero value for every location (this is in contrast to not having a model at all, where all of the firmware would cause QEMU to crash). The “SA” column shows the coverage with complete modeling, including the State Approximation of the firmware’s source of input. A

Table 3: Snippets from a capture of all memory-mapped input/output (MMIO) accesses from an STM32 firmware.

(a) Increasing read-only (Timer 5 @ 0x4000C24)			(b) Read/write storage (Flash controller configuration @ 0x40023C00)		
Op. #	Operation	Value	Op. #	Operation	Value
...
524	READ	3690781	14	READ	0
...	15	WRITE	4
595	READ	3731433	16	READ	4
...	17	WRITE	6
658	READ	3534604
662	READ	5549086	77	READ	6
663	READ	6053877	78	WRITE	7
665	READ	7060952	79	READ	7

firmware that is entirely input-driven will have finite behavior when the source of input is modeled, but unlike previous approaches, the firmware will continue to execute after the input ends, but with no additional input-triggered behavior. We manually verified that all of the firmware samples performed the same overall behavior as was present during recording. That is, even when no hardware was present, the firmware used our generated models to function similarly to when it was running on the actual hardware. In the last column, *Fuzzing*, we feed automatically generated random data to the three firmware examples whose execution is data-dependent, which is equivalent to a naïve fuzzing approach. We accomplished this by attaching a test harness in place of a serial port controller to the system, which, instead of supplying modeled data, provides IO from the host system. This allows new input to be supplied to the firmware program for exploring new functionality, while letting the rest of the PRETENDER-created models function normally. As the table shows, PRETENDER successfully discovered new blocks, and, subsequently, revealed new functionality of the firmware. In all cases, this extra functionality actively interacted with the *other* peripherals models, such as timers and system configuration, not just the serial port. While we discuss details of the hardware peripherals when commenting on PRETENDER’s behavior, our system is not aware of the specific layout, names, or functionality of any of the peripherals, aside from the test harness, and basic details of the standardized interrupt controller coupled to the CPU.

Our evaluation demonstrates that PRETENDER is able to successfully allow re-hosting, while enabling *survivable execution* at the same time. As a result, analysis techniques such as fuzzing could be parallelized and scaled. Rather than simple random data, smarter fuzzing techniques [6] could be used; however, we would like to emphasize that the goal in this work is not specifically to find new bugs in firmware via fuzzing, but to enable dynamic analysis, which is necessary to achieve this, and other security goals going forward.

In the remainder of this section, we will describe the hardware platform and each example more in-depth, together with the detailed re-hosting capabilities enabled by PRETENDER.

blink_led. This simple example blinks a Light Emitting Diode (LED) every 0.5 seconds. While this example may seem overly trivial, we use it to illustrate the basic level of complexity inherent in any firmware compiled with ARM `mbed`, and the basic behavior of timers. When booting even the simplest firmware, the board performs a number of initialization tasks, including using the Reset and Clock Control (RCC) to enable various clock devices, the management of the on-board flash controller, and the configuration of GPIO pins. The firmware performs various self-checks on these peripherals during boot, and if they fail to report correct status information, the firmware will hang in an infinite loop. While this can also be solved with simple replay, the ability to execute this firmware indefinitely can only be achieved using modeling. Table 3 shows a memory trace acquired by PRETENDER, and shows interactions with the timer (Table 3a) and the flash memory controller (Table 3b). PRETENDER correctly identified the timer as an `Increasing Model`, and our linear regression approach correctly resolved the rate at which the timer increases. Whenever `wait()` is called, the value of the timer is periodically checked and the firmware continues execution only when it exceeds an ever increasing amount. PRETENDER’s model can correctly produce the required values indefinitely. Furthermore, the various RCC and other system configuration registers checked by the timer and GPIO code continue to produce the correct values, as we correctly deduced their simplified storage, pattern, and state-approximated values.

read_hyperterminal. This firmware receives external input from a user or other device over a serial port, and turns an LED on or off (“1” or “0”) based on the input. This example shows diverging firmware execution based on different inputs, as a user can send various possible inputs, in any order. We stimulated the program by sending random “on” and “off” commands over the serial port for the duration of the recording. During our State Approximation-based execution, we were able to identically reproduce the execution. After the recorded input ends, the firmware continued to execute, waiting for more data from the serial port. To make things more interesting, we added a special backdoor to the firmware code. More precisely, if a “2” is sent, the firmware will prompt for a password, a common behavior for a hidden backdoor functionality. This functionality is also vulnerable to a buffer overflow when reading the password. In order to explore code-paths of the program not seen during recording, we use the serial port test harness described above, and provide random bytes as input. Even though this backdoor was not exercised during our recording, PRETENDER was able to successfully rehost the firmware accurately enough so that our emulated version can handle this input, including the various timer and RCC interactions present in this section of code. When fuzzing the rehosted firmware, we were also able to trigger the implanted buffer overflow, leading to corruption of the program counter, and crashing the emulator.

button_interrupt. This example makes use of interrupts that are triggered by an external event (i.e., a physical button). When the physical button is pressed, it causes an interrupt to execute a callback that blinks an LED. During our recording, we pressed this button at random intervals over a period of two minutes. Our recording functionality receives the interrupt events and forwards them to the emulator, which in turn executed a callback that manipulated the GPIO peripheral. We located the trigger for the GPIO interrupt automatically (`0x40010408` with value `0x002000`). However, as the timings for the individual button presses were random, PRETENDER falls back to State Approximation for this peripheral, still allowing indefinite execution.

i2c_master. This example is modified from the original ARM `mbed` example to support an AM2315 I2C temperature sensor, and reports both the temperature and humidity in the room. Unlike the previous examples, this one contains multiple sources of interrupts; both the primary system timer (TIM5) and the I2C bus produce interrupts, which causes a conflict during recording. For this reason, we utilize the iterative modeling approach described in Section 3. On the first execution, we obtain a recording of the timer’s overflow-related interrupts, and convert this into a model. On the second execution, PRETENDER identifies that we have an interrupt-enabled model of the timer already, and uses it instead of the hardware. With this source of interrupts removed from the hardware, we are able to clearly observe the I2C bus’s interrupt patterns. This peripheral has multiple bits that control interrupts, and through observing the peripheral, we are able to locate the correct bit mask for the configuration register (`0x720`), such that these bits being enabled will cause our timing-based interrupts to occur. While this bus is a source of external input like our serial port, the input is only generated in response to an action by the firmware. Therefore, when the firmware writes the configuration and data registers for the I2C bus with the appropriate values to read from the temperature sensor, the state of the peripheral will advance or rewind to the appropriate time that this action occurred during recording and the events will occur as expected.

Thermostat. In this example, we present a firmware that would drive a typical thermostat, indicative of popular smart thermostats (e.g., Google’s Nest). The firmware reads the temperature and humidity from the AM2315 sensor used above, but now it also accepts commands that poll for the temperature and humidity. If the temperature is too far from a preset temperature, it will enable a GPIO to trigger a hypothetical air conditioning unit. However, in order to showcase that peripheral models generated with PRETENDER are not firmware-specific and can easily be transferred and reused, we did not actually leverage a recorded peripheral trace to build the models for this firmware.⁴ Instead, we reuse

⁴Note that we obtained a recording of the firmware’s execution nevertheless to provide coverage information for comparison.

the models from the `i2c_master` example above, together with our test harness to uncover new functionality offered by the firmware. However, when we fuzzed the firmware using our test harness, we were able to discover this previously un-reached functionality, which directly results into an increased coverage as shown in Table 2.

Rf_door_lock. This firmware uses a Grove Serial RF Pro radio module connected to an Universal Asynchronous Receiver/Transmitter (UART) peripheral, which accepts multiple commands. Among others, those commands include “ping” and “unlock,” which accept a password. If the password is correct, the firmware activates a GPIO, which unlocks a hypothetical mechanical lock. The functionality of this firmware is indicative of those on popular IoT smart locks. The radio module operates over a standard serial port. It can be configured using various commands, and once this is complete, it will simply transmit data received on the configured channel to nearby radios. Similar to many small embedded systems, this firmware provides a binary protocol we can use to send commands via our hypothetical smart lock client, including `unlock (0xbb)` and `ping (0xdd)`. To interact with this firmware during recording, we used another radio device to send random valid and invalid lock codes and pings to the firmware. This firmware has an additional functionality, implemented as a backdoor that allows any radio user to overwrite the lock code, by sending command `0xff`, followed by the desired code; this feature is also vulnerable to a buffer overflow. As our radio uses a normal serial port, State Approximation works as expected here, but we cannot directly apply our serial port model and feed it with random data to reach additional block coverage. Instead, we need to correctly format our inputs according to the format observed by the radio’s responses during recording; it checks that the radio responds correctly with “OK” to configuration commands, and will halt execution if it does not. This would be an excellent starting point for a mutational fuzzer, but for the sake of simplicity, we simply “mutate” by appending random data to the end of the data held in our model, and replaying it into our serial port. With this rudimentary fuzzer, we were able to automatically discover the hidden functionality, and even trigger the bug, causing QEMU to halt the execution.

5 Discussion and Future Work

We have shown that a virtual, interactive, and automatic re-hosting solution is necessary to tackle the diversity in IoT and embedded devices, and demonstrated the possibility of such a system through PRETENDER. However, we fully acknowledge that the problem of automated re-hosting is still challenging to be completely solved. This section discusses the assumptions and prerequisites laid out in Section 3, and explores a number of the open problems and challenges that must be overcome in order to apply re-hosting in any context

to production embedded devices.

Beyond ARM and MMIO. Currently, PRETENDER supports ARM devices, for which an emulator for the instruction set and any core peripherals (those which control code execution directly) are available. This is a reasonable requirement, as newer ARM designs, particularly the Cortex series, have provided more rigid standards to manufacturers governing memory layout and core components, such as the interrupt controller. This still leaves vendors ample room to customize every aspect of the remaining peripherals, however. While we focus on the ARM architecture, additional architectures can be added by providing a basic instruction set emulator, creating the short interrupt recording stub, and providing the needed physical memory access to the device to enable recording. Additionally, other architectures use “port-mapped IO” (PMIO) to perform their IO operations. While we do not support this today, PRETENDER could be trivially extended to record these operations instead. All other features of PRETENDER are completely device and architecture-agnostic.

Performance. As PRETENDER involves sending peripheral data and interrupts back and forth between the device and an emulator, this adds some overhead to the firmware’s execution. This is particularly noticeable with interrupts, as they tend to be performance- and timing-critical, which could cause issues during recording. This could be overcome through optimization of the implementation, or through the use of purpose-built hardware to interface with the device, as demonstrated in [7].

Obtaining Traces. The principal limitation on the applicability of PRETENDER is not the models or modeling techniques, but in fact the ability to obtain the data to generate them. First, we must be able to obtain a memory data trace for MMIO. In our case study, this is provided via the chip’s debug interface, which simply provides access to read and write to any memory address or CPU register. Any interface that also provides this functionality, whether it is an intended debugging interface or one adversarially obtained through an exploit, is sufficient, and could be used to also extract interrupt traces using only this basic requirement. Second, we must be able to observe enough hardware functionality to generate a useful model. This means that we require sufficient code coverage of those code paths that interface with the hardware. We can explore new program behavior using PRETENDER models, but will logically encounter incorrect behavior if these new code paths exercise dramatically different functionality than what has been recorded. For example, we can re-use our timer model on a completely new firmware that also configures the timer in the same way (e.g., to count up), but not one with a different configuration with vastly different behavior. In our case studies, we utilize human and automated stimulation to achieve maximal coverage during recording, but of course, in the general case, this is an open problem.

Additionally, there are a few aspects of many chips that we simply cannot model correctly with this visibility, particularly

Direct Memory Access (DMA) controllers, whose accesses to memory are initiated by the hardware itself, and therefore not visible externally by any conventional means. These are particularly common in higher-speed peripherals, including USB, networking, storage buses, and those common to modern CPUs designed for general-purpose computing. We are unaware of any CPU that allows introspection into DMA activity; however, insight into this problem may be gained by instead observing the firmware's code to locate DMA operations.

External Peripherals. External peripherals remain one of the most complex parts of re-hosting firmware. PRETENDER handles external peripherals, such as the I2C temperature sensor, and RF hardware examples, but does so by modeling the on-chip peripheral and its associated external device as a composite. This makes our models specific to a given physical hardware configuration. Ideally, this would not be the case; for example, a common serial port can be thought of a simple bi-directional channel over which the CPU and the external device communicate, and we could develop models for each external serial-based peripheral using this channel alone, and reuse these on different host CPUs. However, these ports and bus controllers have their own internal hardware, which follows its own state machine, that responds to the data transferred to and from the peripheral. A particular complication is that, from the point-of-view of MMIO, it is impossible to reliably distinguish values read from control or configuration registers from data coming from outside the CPU. Separating these two intertwined systems remains an important, open problem.

Heavily-stateful Peripherals. Not all peripherals, particularly external ones, are well-modeled by a state machine. As we discussed in Section 3, we make some assumptions to build a state machine approximation of devices which require it, but this is by no means guaranteed to be correct. One notable case where this will fail is external storage devices, such as SPI-based flash or EEPROM chips. While we could reconstruct much of the traffic to and from these chips seen during recording, reading and writing arbitrary data, as could be possible through a modeled serial port used to provide arbitrary input, will of course not succeed. Fortunately, this problem may be dramatically simplified through high-level modeling, or through the separation of external peripherals from their corresponding internal peripherals, as the behavior of a device as storage may become more apparent.

Adding Abstractions. While a system that is abstraction-less is the most ideal solution to the re-hosting problem, modeling using a higher-level abstraction, such as libraries or an OS, remains an important way to make re-hosting more robust. Many firmware images, including the ones used in this work, were written with such libraries, which perform most hardware interactions on behalf of the author's code. If located, these would also provide a convenient means of

dealing with the above problem of external peripherals and DMA, as they provide the firmware author a high-level way of communicating with peripherals, which can then be exploited for modeling. However, for firmware without an operating system, which is typically distributed as a binary blob, this reduces to the problem of identifying library functions in statically-compiled, stripped binary programs, a well-studied but yet-unsolved problem. Furthermore, any code which violates the abstraction by controlling hardware directly still requires the use of a technique like PRETENDER. This is found even in our simple examples, where all accesses to the GPIO peripheral were aggressively in-lined by the compiler, such that no library call or other abstraction remained.

6 Conclusion

In this work, we explored the area of firmware re-hosting, and showed that an entirely new class of approaches can enable scalable, thorough program analysis of firmware. As a first step toward achieving this goal, we presented PRETENDER, which generates models of peripherals automatically from recordings of the original hardware. We demonstrated the accuracy and interactivity of these models, by evaluating PRETENDER on multiple firmware samples across different hardware platforms. While there are many open problems remaining before this technique can be generally applicable, we believe this work shows that automated re-hosting is both possible and necessary to ensure that increasingly-important firmware does not go un-analyzed.

Acknowledgements

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

This material is based upon work supported by the National Science Foundation under Award No. CNS-1704253, and by the Office of Naval Research under Award # N00014-17-1-2011. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

Additionally, this work was in part funded by a research contract with Siemens AG.

References

- [1] A. Beckus, "Qemu with an stm32 microcontroller implementation," 2012, http://beckus.github.io/qemu_stm32/.

- [2] F. Bellard, “Qemu, a fast and portable dynamic translator.” in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.
- [3] D. D. Chen, M. Egele, M. Woo, and D. Brumley, “Towards automated dynamic analysis for linux-based embedded firmware,” in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.
- [4] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.
- [5] Comsecuris, “Luaqemu,” <https://github.com/comsecuris/luqemu>.
- [6] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, “Difuze: Interface aware fuzzing for kernel drivers,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. New York, NY, USA: ACM, 2017, pp. 2123–2138. [Online]. Available: <http://doi.acm.org/10.1145/3133956.3134069>
- [7] N. Corteggiani, G. Camurati, and A. Francillon, “Inception: System-wide security testing of real-world embedded systems software,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani>
- [8] A. Costin, A. Zarras, and A. Francillon, “Automated dynamic firmware analysis at scale: a case study on embedded web interfaces,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 437–448.
- [9] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, “Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution.” in *USENIX Security*, 2013, pp. 463–478.
- [10] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, “Repeatable reverse engineering with panda,” in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, 2015, p. 4.
- [11] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, “A density-based algorithm for discovering clusters in large spatial databases with noise.” in *Conference on Knowledge Discovery and Data Mining*, 1996.
- [12] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. Butler, “Firmusb: Vetting usb device firmware using domain informed symbolic execution,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2245–2262.
- [13] L. Ionescu, “Gnu mcu eclipse. a family of eclipse cdt extensions and tools for gnu arm & risc-v development,” 2015, <https://gnu-mcu-eclipse.github.io/>.
- [14] M. Kammerstetter, D. Burian, and W. Kastner, “Embedded security testing with peripheral device caching and runtime program state approximation,” in *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.
- [15] M. Kammerstetter, C. Platzer, and W. Kastner, “Prospect: peripheral proxying supported embedded code testing,” in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014, pp. 329–340.
- [16] K. Koscher, T. Kohno, and D. Molnar, “Surrogates: Enabling near-real-time dynamic analyses of embedded systems.” in *WOOT*, 2015.
- [17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 50–58, 2002.
- [18] Maxim Integrated, “MAX32600MBED ARM mbed Enabled Development Platform for MAX32600,” 2018, <https://www.maximintegrated.com/en/products/microcontrollers/MAX32600MBED.html>.
- [19] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, “Avatar²: A Multi-target Orchestration Platform,” in *Workshop on Binary Analysis Research (colocated with NDSS Symposium)*, ser. BAR 18, February 2018.
- [20] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, “What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices,” in *Network and Distributed System Security (NDSS) Symposium*, ser. NDSS 18, February 2018.
- [21] Osbourne, Paul, “Cmsis-svd repository and parsers,” <https://github.com/posborne/cmsis-svd>.
- [22] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmallice-automatic detection of authentication bypass vulnerabilities in binary firmware.” in *NDSS*, 2015.
- [23] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *IEEE Symposium on Security and Privacy*, 2016.
- [24] STMicroelectronics, “STM32F072RB,” 2018, <https://www.st.com/en/microcontrollers/stm32f072rb.html>.

- [25] R. Toulson and T. Wilmshurst, *Fast and effective embedded systems design: applying the ARM mbed*. Newnes, 2016.
- [26] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, “Avatar: A framework to support dynamic security analysis of embedded systems’ firmwares.” in *NDSS*, 2014.
- [27] M. Zalewski., “American fuzzy lop,” 2017, http://lcamtuf.coredump.cx/afl/technical_details.txt.

Appendices

A Recording Rationale

While we describe our means of recording in Section 3.1, our approach may seem overly complicated. In the following, we point out the rationale behind the design decisions for the recording subcomponent of PRETENDER.

Recording MMIO. The natural first step in building models of hardware is recording a trace of the IO activity that occurred during execution. As we outline in Section 2, the firmware depends on both internal “on-chip” peripherals, and external “off-chip” peripherals, both of which are needed for the firmware to operate as expected. However, the firmware only communicates with off-chip peripherals through its interactions with on-chip peripherals, so in order to have a complete recording, we must capture all memory accesses that constitute MMIO.

Peripherals are considered “memory-mapped” because they are attached to, and addressed via, one of the CPU’s internal memory buses. Unlike external buses, which can be physically probed and monitored, these interactions only occur within the CPU’s die, and cannot be directly monitored. While some debugging facilities used in the development of new chips offer a data trace of the memory bus, such as ARM’s ETM/HTM Data Trace, these features are seldom available on production chips, and are entirely absent in the low-cost, low-pin-count chips of commercial embedded devices. Typical CPUs found in the wild include, at best, a debugger capable of simple execution control, and memory/register access.

On top of this, MMIO behaves differently from a normal region of memory; instead of just storing data, these locations instead control or represent aspects of on-chip peripherals. Their value or function may change based on external factors, without any interaction with the firmware.

One possible alternative approach to MMIO recording would be to instrument the firmware to record IO interactions. This requires us to understand, from the binary firmware itself, where this IO takes place. This could be done on architectures where explicit `in` and `out` instructions are used for peripherals. On ARM, however, this is not a straightforward operation, as peripherals are accessed via normal memory handling instructions (`LDR/STR`), and it is often

difficult to tell statically whether an instruction is addressing a peripheral or normal memory. Inserting this instrumentation code non-destructively, and collecting the cumbersome volumes of data it generates, are both hard problems, and may even be impossible if the code is present on a Read-Only Memory (ROM). As a result of these complications, our approach involves virtually extending the internal memory bus of the device, by emulating the firmware, and forwarding and recording only the hardware-related accesses to the original physical device (as detailed in Section 3).

Recording Interrupts. Interrupts play an important role in most peripherals, and are a particularly difficult aspect to record and model correctly. Interrupts are triggered by some event, whether it is an explicit MMIO operation, or an event in the physical world, and cause the execution of Interrupt Service Routines (ISRs) as a result. These ISRs typically contain MMIO operations associated with the peripheral that triggered the interrupt (e.g., reading data that arrives at a serial port or counting the number of times a counter overflows). Without the peripherals’ ISRs executing at the correct times, the peripherals may not function, or the system may crash. This behavior is a property of the hardware itself; the internal logic of the peripheral decides when and how often to trigger its associated interrupts. Many peripherals allow this behavior to be adjusted at runtime, through their configuration registers. For example, many peripherals have a single bit in their configuration register controlling whether interrupt events are generated at all.

Hardware features exist on many chips for providing a log of the interrupts, such as ARM’s Instrumentation Trace Macrocell (ITM), but these features are not universal, and are difficult to coordinate with simultaneous peripheral recording or even basic hardware-in-the-loop emulation. Hence, previous solutions, such as the first version of the Avatar framework [26] or SURROGATES [16] tried to tackle interrupt forwarding with custom stubs injected onto the device under analysis. However, both of these solutions forward interrupts in a “fire-and-forget” manner. This results in inconsistencies between hardware and emulated firmware, as incoming interrupts on the hardware could easily be missed when the emulator serves a previous interrupt. Although those inconsistencies are a negligible problem for manual analysis, they dramatically complicate automated modeling, and must be avoided. A more recent approach, presented by Corteggiani et al. [7], uses a custom tailored protocol to keep hardware and emulator *synchronized* during interrupt forwarding. Unfortunately, this method requires custom debugging hardware that would greatly reduce the generality of PRETENDER.

Hence, we heavily extended *avatar*² to support the notion of forwarding and recording interrupts, while carefully keeping the two systems synchronized without the need of specialized debugging hardware. The current published version of *avatar*² retains the hardware in a “debug-halt” state while forwarding memory accesses, in order to avoid side-effects from the resident code. Unfortunately, this debug-halt state

inhibits all interrupts, and thus cannot be used as-is. However, we cannot simply keep the CPU running and forward all of the generated interrupts into the emulator; if too many un-handled interrupts arrive, or spurious, unwanted interrupts occur, the hardware or emulator can experience an unrecoverable fault. The current version of *avatar*² also does not support writing to memory while the CPU is running. To make matters worse, halting the CPU during interrupt routines is problematic, as we noticed that some peripherals, particularly those that control future interrupts, will not work properly in this halted state because they are bound to the CPU's instruction pipeline. As a final complication, we must ensure that we return from these interrupts properly, both in the emulator and on the hardware to ensure that the hardware continues to function, even though it is not executing any code.

B State Approximation Details

Our state approximation model is used when a MMIO location does not fit any other model. According to our observations, these tend to be the locations in a peripheral directly affected by external events, such as the data register of a serial port, a bus controller, or a status and event flag register.

These locations are the most challenging to model and emulate. For example, in the case of an I2C bus controller, there are many sources of state, and numerous causes for the state to change, many of which are not observable. From the software's perspective, the I2C bus controller presents an MMIO interface, which specifies how the bus protocol is spoken (baud rate, master/slave), whether queuing is enabled or interrupt are fired, and so on. At another layer, the hardware between the MMIO and the pins has a state, containing the data queue, bus-related timers, and other condition flags not visible directly through MMIO. Both of these portions also occur in the device on the other side of the bus. Finally, the two devices share a protocol spoken on the I2C bus itself, which specifies an ordering of events (start symbol, address, data with acknowledgment, etc.). The result of this is a series of composed, inter-related state machines, which also rely somewhat on the physical world's events, and can only be observed through the rather limited window of MMIO memory accesses.

Unfortunately, this means that we fail the requirements of state machine recovery techniques, which are typically used to infer states and transitions from an activity trace. We do not know the number of possible states, we cannot tell when two states are equivalent, and it is challenging to know concretely if we have even changed the state of the peripheral. We also cannot easily distinguish data registers, which may contain data respecting some protocol, from others containing status flags, error codes, and configuration data. However, it is also not sufficient to simply replay values verbatim from the recorded trace. This is because our models need to be able to function even when we observe deviation from the recording caused by new input, timing-related deviations caused by

differences between the hardware and emulator, as well as to tolerate the asynchronous and non-deterministic occurrence of interrupts. In avoiding these limitations, we created the State Approximation algorithm we describe in Section 3.

State Approximation Example. As an example, consider a hypothetical device that uses a serial port to act as a client for the thermostat we model in Section 4. This device's firmware will query the thermostat, with 't' and 'h', and expect a properly formatted temperature or humidity in return. Furthermore, the firmware reacts to this data, for instance by sending the information across a network, or raising an alarm. The device firmware must receive a response from the thermostat when expected, and the response must make sense for the given command, for the firmware to behave correctly.

An illustration of what this model might look like can be seen in Figure B.1. Note that, in a real-world scenario, there will be many peripherals needed to operate the firmware, but here we focus on just one to better explain its behavior. The client device's serial controller contains many registers, including a configuration register, a status register, a data register, as well as assorted registers governing physical hardware details, like baud rate. Each of these is addressed by its own MMIO location, in a contiguous memory region we identified during clustering. We notice, from our traces and previous Memory Model Training, that the configuration register is a simple storage location, and the baud rate control register is only ever written to. The contents of the status register follow a pattern, alternating between the values 0x1 and 0x3, which we will interpret as whether data is ready to receive or not. The data register, on the other hand, will change without respecting any pattern or direct stimulation from the firmware. Therefore, this location is handled by State Approximation.

When emulation begins, we start in the peripheral's initial state; during boot-up, the firmware configures the serial port, writing to the configuration register to enable the serial port, and set the baud rate to 9600, advancing the peripheral's state pointer to the point at which these actions occurred. The firmware then begins its main loop, and requests a temperature, by writing a 't' into the data register. Naturally, the next thing that happens chronologically is for the status register to indicate that bytes are ready to read, and the firmware will read a temperature value out of the data register one byte at a time (e.g., "24.24C"). Similar actions occur if an 'h' is written to the data register by the firmware; the status register indicates new data, and the firmware reads it back (e.g., "50.35%"). However, when emulating with new input, interrupts, or after the duration of the original peripheral's chronologically observed states, we must make a decision about what state the peripheral is in. In these cases, following the simple rules in Section 3, we will enter the state where a 't' or an 'h' was written to the data register, and subsequent reads will return a temperature or a humidity. In this simple example, the serial port will, after some time, return only the last valid temperature and humidity values, but it will continue

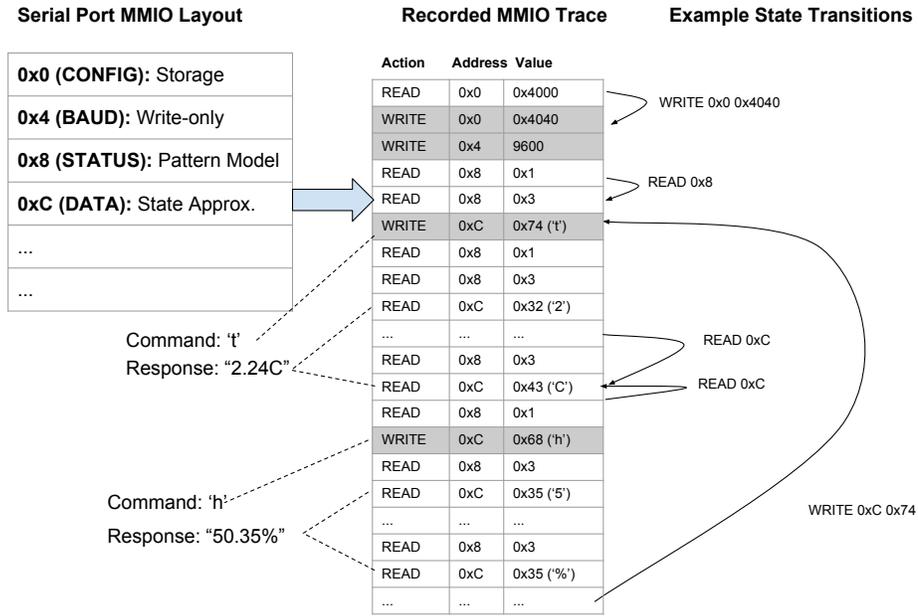


Figure B.1: Illustration of State Approximation in action, on a simplified serial port peripheral

to return only temperatures or humidities when asked for, and respect whatever formatting or encoding for these responses

the thermostat uses, which may be checked by the firmware.

CRYPTOREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices

Li Zhang*, Jiongyi Chen[†], Wenrui Diao^{‡§}(✉), Shanqing Guo^{‡§}, Jian Weng*, and Kehuan Zhang[†]

*Jinan University, {zhanglikernel, cryptjweng}@gmail.com

[†]The Chinese University of Hong Kong, {cj015, khzhang}@ie.cuhk.edu.hk

[‡]Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, {diaowenrui, guoshanqing}@sdu.edu.cn

[§]School of Cyber Science and Technology, Shandong University

Abstract

Cryptographic functions play a critical role in the secure transmission and storage of application data. Although most crypto functions are well-defined and carefully-implemented in standard libraries, in practice, they could be easily misused or incorrectly encapsulated due to its error-prone nature and inexperience of developers. This situation is even worse in the IoT domain, given that developers tend to sacrifice security for performance in order to suit resource-constrained IoT devices. Given the severity and the pervasiveness of such bad practice, it is crucial to raise public awareness about this issue, find the misuses and shed light on best practices.

In this paper, we design and implement CRYPTOREX, a framework to identify crypto misuse of IoT devices under diverse architectures and in a scalable manner. In particular, CRYPTOREX lifts binary code to a unified IR and performs static taint analysis across multiple executables. To aggressively capture and identify misuses of self-defined crypto APIs, CRYPTOREX dynamically updates the API list during taint analysis and automatically tracks the function arguments. Running on 521 firmware images with 165 pre-defined crypto APIs, it successfully discovered 679 crypto misuse issues in total, which on average costs only 1120 seconds per firmware. Our study shows 24.2% of firmware images violate at least one misuse rule, and most of the discovered misuses are unknown before. The misuses could result in sensitive data leakage, authentication bypass, password brute-force, etc. Our findings highlight the poor implementation and weak protection in today's IoT development.

1 Introduction

As an emerging field, the Internet of Things (IoT) system is on the track of rapid development. IoT devices have been deployed in several scenarios, such as smart home, smart transportation, and so forth. A recent marketing research report forecasts that the amount of IoT devices will grow to around 19 billion worldwide in 2022 [24].

Different from the traditional embedded devices, IoT devices (such as smart-home devices) are usually equipped with multiple sensors and connected to the network. In this context, the security of IoT devices becomes crucial as it involves not only the data privacy of users [13] but also the risk of dangerous incidents [28]. Due to the requirements of usage scenarios and the limitation of manufacturing costs, IoT devices often have customized hardware and software configurations. Such a closed hardware-software environment gives people an illusion of safety. However, that is not the case, and IoT related vulnerabilities emerge endlessly [10, 31, 38, 43, 50]. Given the prevalence of vulnerable devices, we believe these disclosed incidents are just the tip of the iceberg. What is worse, until now, there has not been a well-established set of technical standards for IoT security.

Previous studies of identifying vulnerabilities in IoT devices have been traditionally focused on memory corruptions [16–18, 21, 39], authentication bypass [46], and domain specific vulnerabilities such as BadUSB [30]. However, currently no tool can automatically identify crypto misuses in IoT devices. As a consequence, large-scale security analysis of crypto misuse in IoT devices has never been conducted before. On the one hand, existing solutions target specific platforms such as Android and iOS [23, 26, 40]. They are less suitable for IoT devices that often involve different architectures. For example, Android applications are provided in a reversible bytecode format, whereas IoT applications are compiled to machine code that could run on various CPU architectures (MIPS, ARM, PowerPC, and so forth).

On the other hand, prior works rely on specific crypto libraries and do not handle self-defined crypto functions. Given the lack of security standards in IoT development, developers tend to use self-defined crypto functions that wrap standard crypto functions. Although several well-established crypto libraries provide well-designed and carefully-implemented crypto APIs to facilitate the deployment of secure modules, there is no guarantee whether these crypto APIs are used and wrapped correctly. For example, inexperienced developers may use a non-random initialization vector for block cipher

mode encryption or use static seeds for random number generation functions. Such a problem usually results in confidential data leakage and even system intrusion.

In this paper, we introduce CRYPTOREX, a framework that achieves automated and large-scale analysis of crypto misuse in IoT firmware. On a high level, we first lift the binary code with different architectures to the unified intermediate representation (IR) format. Then we recover the stack layouts to precisely identify the arguments of low-level crypto API arguments and track the definition of the arguments with taint analysis. In order to further capture the self-defined crypto functions and the misuses, CRYPTOREX maintains a list of crypto APIs that can be dynamically updated during taint analysis.

To demonstrate the feasibility of CRYPTOREX, we implemented a prototype of CRYPTOREX and carried out a large-scale experiment based on 1327 firmware images (from 12 vendors, in 7 different architectures) crawled from the Internet. The device types include IP camera, network attached storage, router, smart plug, smart bulb, and so forth. CRYPTOREX successfully unpacked 521 firmware images and identified 679 crypto misuses. Surprisingly, the experiment also demonstrates remarkable performance: on average, it takes only 1120 seconds for CRYPTOREX to complete one firmware analysis. Further investigation shows that 126 firmware images violate at least one crypto misuse rule. The misuses could result in the compromise of secrecy, authentication bypass, password brute-force, and etc.

With a full implementation and a comprehensive evaluation, CRYPTOREX makes the first step towards scalable and quantitative measurement for (in)secure crypto usage in IoT devices. We make this tool publicly available for continuous research on IoT firmware analysis.¹

Contributions. The main contributions of this paper are:

- We designed a new analysis framework – CRYPTOREX, which can automatically identify crypto misuses in IoT devices. With new techniques such as stack layout recovery and dynamic update of crypto APIs, it can achieve reliable cross-architectural analysis in a large-scale manner.
- We performed the first large-scale measurement study on (in)secure crypto usage over a large number of firmware images. Our study has brought to light the worrisome situation (questionable practice and weak protection) in IoT development.

Roadmap. The rest of this paper is organized as follows. Section §2 gives the background knowledge of IoT firmware analysis and crypto misuse. The detailed design of CRYPTOREX is elaborated in Section §3. The evaluation results

¹<https://github.com/zhanglikernel/CRYPTOREX>

are summarized in Section §4. Section §5 discusses some limitations of our framework and experiments. Section §6 reviews the related work, and Section §7 concludes this paper.

2 Background

In this section, we provide the necessary background about firmware analysis and cryptography misuse.

2.1 Security Analysis of IoT Firmware

Firmware is a specific class of computer software that provides the low-level control for the device’s specific hardware. Unlike PCs, for which software engineers develop multi-purpose applications, firmware is usually designed for special purposes and runs on embedded devices (e.g., IoT devices) with limited resources and diverse architectures. Unfortunately, the firmware-specific features have also introduced several challenges to the security analysis. Here we summarize the challenges from the aspects of dynamic analysis and static analysis.

- **Dynamic analysis:** In dynamic analysis, the firmware is executed in a controlled environment. A bare-metal analysis (based on real devices) could output the most accurate result, but it needs the support of an exposed debug port on the device. Unfortunately, many manufacturers disable the debug port for security concerns. An alternative solution is to run the entire firmware or embedded programs in an isolated emulator. The challenge is the lack of non-volatile memory (NVRAM) parameters, which causes runtime failures during dynamic analysis. In previous work, Chen et al. [15] simulated the NVRAM parameters using userspace libraries. However, it is not suitable for a large-scale analysis due to the diversity of architectures.
- **Static analysis:** Compared with dynamic analysis, static analysis scrutinizes the binary code of firmware, instead of relying on emulation environments. In most cases, it achieves a balance of efficiency and accuracy. However, developing a unified static analysis framework for various IoT devices with different underlying architectures (MIPS, ARM, PowerPC, and so forth) is not a simple task. The disassembled binary files may contain different order sets with different operations and side-effects, which leads to various calling conventions and different stack layouts. Consequently, this could cause difficulty to further analysis such as recovering function arguments.

Our approach: IR-based analysis. Through the above discussion, for large-scale security analysis, the difficulty mainly comes from the non-unified underlying architectures of IoT firmware. To bridge the gap, we lift diverse binary code to

a unified intermediate representation (IR). In the process of compiling, the source is transformed into IR and then binary code. Vice versa, we can lift the binary code of different architectures to the same IR, and the subsequent analysis could be based on the IR. Previous work [20,46] has also demonstrated the feasibility of firmware analysis.

2.2 Cryptography Misuse

Though the standard cryptographic libraries provide well-implemented and well-defined APIs, developers may not fully understand the API documentation and misuse the APIs by delivering improper arguments, which could result in the compromise of confidentiality in network communication and data storage. In this paper, we focus on the inappropriate use of crypto functions and assume that the involved crypto algorithms are secure. Based on the study of Egele et al. [23] and Lazar et al. [33], we use the following six rules in cryptography that should be followed by IoT developers. As indicated in the OWASP guideline [44], these time-tested rules cover common misuses in symmetric key encryption, password-based encryption, and random number generation.

- **Rule 1.** *Do not use electronic code book (ECB) mode for encryption.* The ECB mode cannot provide strong enough security guarantee. For example, the `AES_ecb_encrypt` function of the `libcrypto` library should not be used for security-related modules.
- **Rule 2.** *Do not use a non-random initialization vector (IV) for ciphertext block chaining (CBC) encryption.* If the IV is static, the encryption scheme is considered insecure. For example, developers could use the `gcry_set_iv` function provided by `libgcrypt` to initialize the IV for the subsequent encryption operations.
- **Rule 3.** *Do not use constant encryption keys.* Constant encryption keys would bring the direct risk of cracking the encryption schemes. For example, when developers invoke the AES encryption function of the `wolfcrypt` library, they could use `wc_AesSetKey` to specify a secure key.
- **Rule 4.** *Do not use constant salts for password-based encryption (PBE).* In Linux, the most frequently used API for password encryption is `char *crypt(const char *key, const char *salt)` of `libcrypto`. Almost all Linux systems use it to encrypt users' passwords and save the outputs to `/etc/shadow`. The parameter `salt` could not be assigned as a constant value.
- **Rule 5.** *Do not use fewer than 1000 iterations for PBE.* For example, in the function `Evp_BytesToKey` of `libcrypto`, the argument `count` specifies the round of iterations. Some developers may set small values for performance consideration, which would result in the risk of brute-force attacks.

- **Rule 6.** *Do not use static seeds for random number generation (RNG) functions.* When a program needs to generate secure random numbers, it should not use `rand()` or `srand()` with a constant seed.

To provide an intuition, we list some crypto misuse examples in Table 1.

2.3 Intermediate Representation

There are several available IRs designed for different purposes, such as REIL [22], LLVM [32], and BAP [12]. Since our analysis focuses on the function arguments, intending to track variable types, the IR should support static data-flow analysis and represent registers and memory locations in a unified format, even the executables have different architectures.

In our work, we employ Valgrind's VEX IR [41] as the representation format. The VEX IR and its Python bindings PyVEX [46] provides some features which fit our requirements ideally.

- *Static program slicing:* PyVEX supports static program slicing. PyVEX translates binary code to IR code and divides it into basic blocks that are in the form of IRSBs (IR Super-Block). With IRSBs, we can conveniently construct the control flow graph (CFG) and data flow graph (DFG).
- *Base operations:* After disassembly, an assembly instruction is transformed into multiple statements in VEX. A statement is an "atomic action" which contains an operand and an IR expression. Besides, VEX classifies all operations into four base statements: Write Temp, Put Register, Store Memory, and Exit.

PyVEX transforms binary code into statements, which are "atomic actions" defined by VEX.

3 Design of CRYPTOREX

Here we describe the detailed design of CRYPTOREX. At a high level, CRYPTOREX takes a raw firmware image as the input and outputs a report indicating the misuse of crypto functions in the firmware image. Mainly, the analysis procedure consists of five steps (as shown in Figure 1):

- **Firmware Acquisition and Pre-processing.** First, we develop a crawler to automatically download firmware images from IoT vendors' websites and then extract executable files from them.
- **Lifting to VEX IR.** Next, we lift the binary executables in various architectures to VEX IR. The subsequent analysis is conducted upon the unified representation.

Table 1: Examples of Crypto Misuse

Rule #	[Library]: Function	Parameter	Misuse Condition
Rule 1	[libgcrypt]: gcry_cipher_open(gcry_cipher_hd_t *hd, int algo, int mode, unsigned int flag)	mode	== 1
Rule 2	[wolfcrypt]: int wc_AesSetIv(Aes *aes, const byte *iv)	iv	static string
Rule 3	[Nettle]: void aes192_set_encrypt_key (struct aes192_ctx *ctx, const uint8_t *key)	key	static string
Rule 4	[libcrypt]: char *crypt(const char *key, const char *salt)	salt	static string
Rule 5	[libcrypto]: int EVP_BytesToKey(const EVP_CIPHER *type, const EVP_CIPHER *type, const unsigned char *salt, const unsigned char *data, int datal, int count, unsigned char *key, unsigned char *iv)	count	< 1000
Rule 6	[C standard library]: void srand(int seed)	seed	static integer

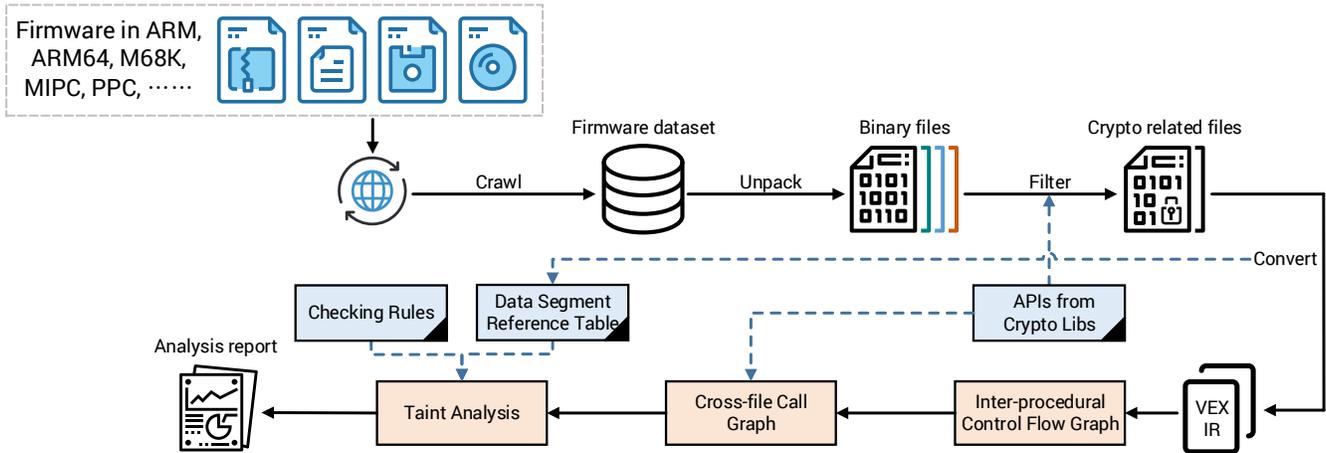


Figure 1: Framework overview.

- **Inter-procedural Control Flow Graph Construction.** Then, we construct inter-procedural control flow graphs (ICFG) for the executable files. Such graphs capture data flows across function calls.
- **Cross-file Call Graph Construction.** After that, a cross-file call graph for each crypto function is built to facilitate the data-flow tracking (in the next step) which crosses multiple executables.
- **Taint Analysis.** In the last step, we perform a backward taint analysis to track how each crypto function argument is defined. If the definition triggers the misuse rules, CRYPTOREX outputs a report listing the discovered crypto misuse cases.

3.1 Firmware Acquisition and Pre-processing

Since there is no well-established IoT firmware dataset, we developed a web crawler based on the prototype tool of Chen et al. [15] to download firmware from the IoT vendors' websites

automatically. Also, the existing tool only supports downloading from FTP servers. Therefore we added a module to support firmware downloads from dynamically generated websites.

For a timely firmware update, developers tend to pack firmware images to save transmission time and storage space. As a result, most of the firmware images that we collected are compressed with various compression algorithms and cannot be directly analyzed. To tackle this problem, we use the state-of-the-art firmware unpacking tool Binwalk [2]. It integrates several file system signatures and decompression algorithms to identify file systems and extract executables (binary files) from the firmware.

3.2 Lifting to VEX IR

File filter. To reduce the time consumption of IR conversion, CRYPTOREX first filters out the binary files that do not invoke the crypto APIs. Specifically, we use Buildroot [3], a cross-compilation tool, to analyze the header information of

each file to check whether cryptography libraries are included. If the cryptography libraries are not included, we then ignore the binary file. In practice, the filter utilizes the API data extracted from seven widely used open-source C/C++ cryptography libraries: `libcrypto` [42], `libcrypt` [29], `cryptlib` [4], `LibTomCrypt` [7], `libgcrypt` [6], `wolfcrypt` [9], and `Nettle` [8]. These libraries have covered nearly 100% usage cases in our firmware dataset.

Enhanced conversion. After that, built on top of the binary analysis framework Angr [1], CRYPTOREX invokes Angr APIs to disassemble binary files and lift different low-level instruction sets to the unified VEX IR. However, the direct binary-IR conversion is not sufficient to meet our requirements: (1) The call relations of Angr is incomplete because it only considers explicit invocation addresses. If the address is put into a register or memory, Angr cannot locate it. (2) Type information of variables is lost, which affects the data-flow tracking (especially the function parameters). (3) The function arguments are often passed via the register, stack, or both, and follow specific conventions. If the binary code is lifted to the IR, architecture-specific calling convention will be lost.

To solve the first two shortcomings, based on the functionalities of IDA Pro [5], we develop a recovery script to (1) locate the actual addresses of jump instructions to complete the function call relations and (2) infer data types (and save them as a *data segment reference table*) to facilitate the subsequent data-flow tracking. For the third shortcoming, we extract the arguments passing rules of different architectures in advance. During the testing, we identify the architecture types (also obtained through IDA Pro) of firmware images and apply the corresponding arguments passing rules.

3.3 Inter-procedural Control Flow Graph Construction

After that, we construct the *inter-procedural control flow graph* (ICFG), which is the foundation for the subsequent inter- and intra-procedural data-flow analysis. In particular, our ICFG construction starts with the entry point (i.e., `start()`) of each executable, and traverses functions and basic blocks through depth-first searching.

We consider function call relations in the graph, in order to support inter-procedural analysis. Except for the functions that can be reached from the entry point, we also discover isolated functions and their call relations, by scanning code segments with pre-defined function signatures [5]. In this way, function call relations and API call sites can be discovered as many as possible. This step is hugely beneficial for library files because they only contain isolated functions for external use.

Also, though Angr is able to resolve explicit target addresses, it cannot resolve implicit call relations. Therefore, to improve the precision of data-flow analysis (i.e., implicit

call relations), we utilize the data segment reference table obtained from Section 3.2 and perform value-set analysis [11] to recover the addresses of indirectly invoked functions. In other words, for indirect jumps (e.g., `jr $t0` in MIPS), we simulate previous instructions and compute the actual value (or the value range) of the register or memory location. If the destination is identified as a function, we then add it to the function call relations.

Furthermore, given that loop structures often contain computation-intensive instructions (like increment operations) and have less variable definitions and uses, we flatten the loop structure so that each loop is executed only a few times during data-flow analysis. This operation can significantly reduce the time consumption of loop processing.

3.4 Cross-file Call Graph Construction

We notice that considerable IoT solution providers defined their own crypto APIs wrapping low-level crypto APIs (from other libraries), which could be treated as some kind of optimization to facilitate the internal development process. Executable files could either invoke the original low-level APIs or the self-defined APIs. This phenomenon requires us to capture the function call relations between them during the data-flow analysis. Therefore, in this step, we construct the *cross-file call graph* (CFCG for short) that involves multiple executable files and represents the chains of function calls for each low-level crypto API. With such a representation, we can dynamically update the crypto API list and further detect the misuse of self-defined crypto APIs.

In the beginning, each crypto API (from `libcrypto`, `libcrypt`, `cryptlib`, `Nettle`, `libgcrypt`, `wolfcrypt`, and `LibTomCrypt`) acts as the starting point of a call graph. Next, we construct the chains of function calls for the crypto API. To this end, we first scan the call sites of the crypto API in the extracted executables and then recursively chain the callers of the functions that invoke the crypto API. For the non-library executables, we only need to consider internal functions and imported functions. For library files, exported functions should also be considered because other executables can further invoke them.

As an example, Listing 1 is a piece of code snippet from our dataset. The `smm` program is extracted from D-Link DSR-150 (VPN router) which supports up to 65 VPN tunnels. After code review, we find its traffic encryption module is implemented through invoking the self-defined crypto API `DES_ProcessFile()` provided by library `libSys.so`. In the definition of `DES_ProcessFile()`, low-level crypto API `DES_string_to_key()` is invoked to specify the key used by the following encryption. Except for that, both `DES_string_to_key()` and `DES_ProcessFile()` can also be invoked by other executables. We show the result of CFCG (partial) in Figure 2.

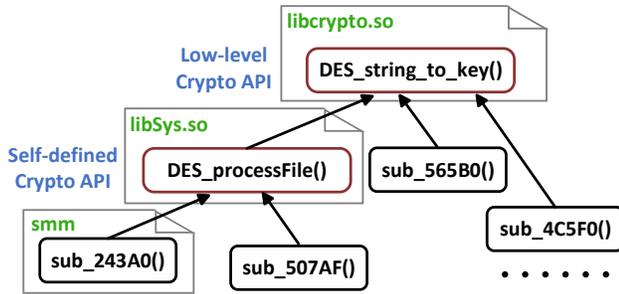


Figure 2: Example: cross-file call graph of Listing 1.

```

1  /***** libSys.so *****/
2  signed int __fastcall DES_processFile(int
3     a1, const char *a2, const char *a3,
4     int a4) {
5     int v7;
6     v7 = a4;
7     ...
8     // low-level crypto API
9     DES_string_to_key(v7, &v18);
10    ...
11    while(fread(&ptr, &v15, &v14, v5)) {
12        DES_ecb_encrypt(&ptr, &v15, v14, v5)
13    }
14    /***** smm *****/
15    signed int __fastcall sub_243A0(char *a1,
16        int a2) {
17        v2 = a2;
18        v3 = a1;
19        if(v3) {
20            v8 = v3;
21        } else {
22            v8 = "/pfrm2.0/sslvpn/var/conf/smm.
23                conf";
24        }
25        if(v2)
26            // defined in libSys.so
27            DES_processFile(1, "tmp/.smm.clr", v8, "
28                root123");
29    }

```

Listing 1: Self-defined crypto API: D-Link DSR-150.

3.5 Taint Analysis

Our final purpose is to detect whether the crypto APIs are misused in IoT firmware. We perform a static taint analysis (backward tracking) to tag the function arguments at the call sites and determine their actual values for checking. To this end, the first step is to precisely identify the sources, which are the arguments of crypto APIs. Then, we identify the wrappers

of crypto APIs and dynamically update the API list based on how the function arguments propagate. In the end, the misuse rules are applied at the sinks.

Taint sources. Although we have the prototypes of crypto APIs (as shown in Table 1), the arguments in binary code are not matched with the defined parameters. Therefore, to tag the taint sources (i.e., the arguments of crypto APIs), we need to identify which function argument corresponds to the function parameter that may lead to crypto misuse. For that, we utilize the calling conventions: the arguments passing rule of the firmware under testing has been matched and recorded in the step of lifting to VEX IR (Section §3.2).

In the procedure, a register is tagged as a taint source when a function argument is passed with it. For stack-based passing, we need to recover the stack layout at the call sites statically, in order to determine which stack variable serves as a function argument. For this reason, we first compute the values of the stack pointer and the base pointer to determine the range of the caller’s stack frame. Then we locate the stack-based arguments at the memory locations whose addresses are specified by offsetting the caller’s stack pointer.

Taint propagation. To build data dependence, we employ the use-define chain algorithm of Angr. However, it incurs false negatives at array operation APIs of C libraries. For example, the function strcpy(dest, src) copies the string from the source address to the destination address. However, the data dependency is not built between these two variables. To solve the problem, we implemented a module to simulate the functionality of array operation APIs (e.g., memset() and memcpy()) and build data dependency between related variables. In the meantime, we also dynamically update the list of crypto APIs during backward taint analysis. On the CF CGs that we construct, we add a function (i.e., a self-defined crypto API) to the list of crypto APIs if one of its function parameters is passed to the crypto APIs as an argument. As an example, the function DES_ProcessFile() is a self-defined crypto API (shown in Figure 2).

Taint sinks. We define taint sinks at constants. A constant can be interpreted as either a pointer or an immediate value, based on the types of function arguments given in the function prototypes. If it is a pointer, we get its referenced data in the data segment reference table generated in Section §3.2. If the constant is an immediate value, no further step is required. After the interpretation, we analyze whether the data is matched with the specifications defined in misuse rules. If a misuse condition is triggered, such a case is considered as violation and recorded in the analysis report.

4 Implementations and Evaluation

In this section, we give the implementation details of CRYPTOREX and the evaluation results based on a large-scale real-world IoT firmware dataset.

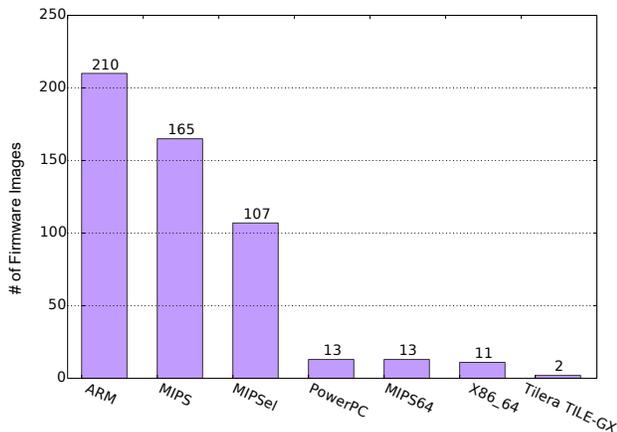


Figure 3: Architecture distribution.

4.1 Implementation

We implemented CRYPTOREX with 3310 lines of Python code. Apart from that, we integrated the APIs of existing open-source projects to avoid reinventing the wheel. Especially, Binwalk [2] is utilized to unpack firmware images. Angr [1] is used to convert low-level binary code into VEX IR, and Buildroot [3] is used in building cross-file call graphs.

4.2 Experiment Setup

Dataset. During September and October of 2018, we crawled 1327 firmware images in total, covering 12 different IoT vendors, including WD, TP-Link, Linksys, AT&T, Buffalo, and so forth. The device types include IP camera, network attached storage, router, smart plug, smart bulb, and so forth. The detailed data is provided in Table 3.

Among the firmware images we collected, CRYPTOREX successfully unpacked 521 of them (39.3%). Our unpacking implementation relies on Binwalk, which is the de facto tool for firmware unpacking and is able to handle common compression algorithms. However, inevitably, some IoT vendors use non-standard packing techniques such as proprietary compression algorithms and encryption algorithms. In this paper, we did not handle the unsuccessful cases and leave it as future work.

Our further analysis shows that the successfully unpacked firmware samples have seven different architectures – ARM, MIPS, MIPSel, Tiler TILE-GX, PowerPC, MIPS64, and X86_64. It should be diverse enough to demonstrate the capability of cross-architecture analysis of CRYPTOREX. The detailed distribution data is plotted in Figure 3.

Regarding crypto APIs, we extracted all 165 crypto APIs from 7 well-known crypto libraries, as listed in Table 2. In total, we tracked 190 crypto-related arguments.

Table 2: Statistics of used crypto APIs

Library	# of Crypto APIs	# of Tracked Arguments
cryptlib [4]	3	4
libcrypto [6]	4	4
libcrypto [42]	80	87
libcrypto [29]	3	4
wolfcrypt [9]	20	30
Nettle [8]	45	46
LibTomCrypt [7]	10	15

Execution environment. Also, in our experiment, CRYPTOREX run on an Ubuntu 16.04 PC equipped with Intel Core i7 quad-core 2.50 GHz CPU and 8G RAM.

4.3 Findings

At the firmware level (Table 3), CRYPTOREX discovered 679 crypto misuse bugs in 126 vulnerable firmware images from 8 vendors, indicating that 24.2% (126/521) of firmware images contain at least one misuse issue. Notably, we found that even 88.7% of Tomato device firmware images are vulnerable. Only the firmware images from Buffalo, Zyxel, TENNIS passed our checking, and no misuse issues were identified.

At the rule level (Table 4), the most common misuse is using ECB mode for encryption (Rule 1, 20.5%), which could result in several security risks. For instance, attackers can determine whether two ECB-encrypted messages are identical. Also, the misuse cases of using constant encryption parameters include constant IV (Rule 2, 4.6%), constant keys (Rule 3, 11.3%), and constant salts (Rule 4, 10.8%). Once they are leaked, the secrecy of stored data and transmitted data could be compromised. Moreover, we found that 4.4% of firmware images use less than 1000 iterations² for password-based encryption (Rule 5), indicating that developers tend to sacrifice security to achieve better performance. As a result, attackers can perform brute-force attacks using a large number of candidate passwords.

In our evaluation, we did not find the case of violating Rule 6. Our further investigation shows that the random number generation modules of most IoT firmware images rely on /dev/urandom or /dev/random directly, without using a random seed. Such implementations are deemed secure.

4.4 Accuracy

False positives. Given the tremendous efforts to manually confirm all 679 identified misuse bugs, we randomly sampled 30 cases from the reported misuses and manually examined them to make inferences about the statistical population. However, although 29 of them (96.7%) were confirmed misusing

²In fact, the iteration round of all firmware images violating Rule 5 is 1.

Table 3: Results of crypto misuse detection (by vendors)

Vendor	# of Firmware	# of Unpacked Firmware	% of Unpacked Firmware	# of Vulnerable Firmware	% of Vulnerable Firmware	# of R1 issues	# of R2 issues	# of R3 issues	# of R4 issues	# of R5 issues	# of R6 issues
D-Link	496	201	40.5%	22	11.0%	106	7	30	27	0	0
Linksys	121	70	57.9%	31	44.3%	48	19	41	20	20	0
WD	10	10	100%	4	40.0%	80	36	0	0	3	0
360	5	4	80%	1	25.0%	2	0	0	0	0	0
AT&T	12	0	0%	0	0%	0	0	0	0	0	0
Buffalo	7	4	57.1%	0	0%	0	0	0	0	0	0
Netgear	66	29	43.9%	1	3.5%	11	0	0	0	0	0
TP-Link	47	47	100%	1	2.1%	5	0	0	0	0	0
Tomato	71	71	100%	63	88.7%	102	0	58	58	0	0
MikroTik	23	23	100%	3	13.0%	0	0	6	0	0	0
Zyxel	450	50	11.1%	0	0%	0	0	0	0	0	0
TENVIS	19	12	63.2%	0	0%	0	0	0	0	0	0
Total	1327	521	39.3%	126	24.2%	354	62	135	105	23	0

Table 4: Results of crypto misuse detection (by rules)

Violated Rule	# of Firmware	% of Firmware
Rule 1	107	20.5%
Rule 2	24	4.6%
Rule 3	59	11.3%
Rule 4	56	10.8%
Rule 5	23	4.4%
Rule 6	0	0%
No violation	395	75.8%

crypto functions, there was a false positive for self-defined crypto functions (see Listing 2 extracted from D-Link NAS device DNS-326), which is caused by dead code (i.e., the constant argument is not used in crypto operations). Specifically, in the definition of the self-defined crypto function `sub_155F0()`, if it is invoked with its third parameter being 1 (at Line 19), the low-level crypto function will have its salt parameter being the constant string "\$1\$". However, when the self-defined crypto function `sub_155F0()` is invoked at line 5, a "0" is passed as its third parameter. It means that Line 19 is not executed and function `sub_15570()` is being called, and it generates a random string combine with time and PID, the low-level crypto function `crypt()` is not misused.

```

1 signed int __fastcall sub_15270(const char
    *a1, const char *a2, const char *a3,
    int a4){
2     int v6,v8;
3     v6 = a4;
4     v8 = (int)getpwnam(a1);
5     if(sub_155F0(v8,v6,0)){
6         ...
7     }
8     ...
9 }
10
11 // self-defined crypto function

```

```

12 signed int sub_155F0(int a1, int a2, int
    a3){
13     ...
14     int v3;
15     char *v5;
16     v3 == a3;
17     ...
18     if(v3 == 1){
19         v5 = "$1$"; //not executed if v3 is 0
20     }else{
21         v5 = (const char *)sub_15570(); //use
            time() and getpid() to generate
            salt
22     }
23     v6 = (const char *)sub_14E74(&v11,v5);
24 }
25
26 char* __fastcall sub_14E74(const char *a1,
    const char *a2){
27     char *v2;
28     v2 = crypt(a1,a2); //low-level crypto
        function
29     ...
30 }

```

Listing 2: Dead code: D-Link DNS-326.

False negatives. Previous crypto misuse detection approaches [23,26,36,47] focus on mobile platforms, rather than IoT devices. As a result, there is no immediate and labelled dataset as ground truth to quantify the false negative. Despite that, we manually checked the parameters of all crypto API invocations (based on our crypto API dataset as shown in Table 2) in 10 randomly selected firmware images in which no misuse was reported, and we found no false negative.

4.5 Performance

Running CRYPTOREX on 1327 firmware images consumes 7 days and 3 hours (around 171 hours) in total. On average, each

Table 5: Performance analysis

Firmware Model	Firmware Size	# of Analyzed Files	Size of Analyzed Files	Time of Unpacking	Time of CFCG Construction	Time of IR Lifting	Time of ICFG Construction	Time of Taint Analysis	Total Time
E2500	7.0 MB	8	3.12 MB	<1s	3s	10m39s	16s	2m	13m10s
mipsbe-6.42.9	10.7 MB	9	1.11 MB	<1s	3s	1m52s	16s	<1s	2m19s
mipsbe-6.43.2	11 MB	10	1.2 MB	<1s	3s	2m12s	20s	<1s	2m25s
FW_EA6350	16.2 MB	53	3.99 MB	<1s	1m25s	5m18s	1m25s	<1s	9m8s
FW_WRT1900ACv2	32 MB	66	5.4 MB	34s	1m36s	6m41s	2m2s	<1s	11m15s
DSR-250_A2	25.8 MB	64	9.71 MB	3s	1m20s	11m39s	3m34s	4s	19m17s
DCS-960L_A1_FW	9.8 MB	144	6.3 MB	<1s	31s	14m20s	1m52s	<1s	16m19s
My_Cloud_KC2A	103.7 MB	120	26.73 MB	5s	1m57s	28m58s	9m42s	46m19s	98m53s
360P3	8.1 MB	61	4.6 MB	<1s	8s	13m44s	1m45s	<1s	15m43s
360POP-P1	6.7 MB	59	3.7 MB	<1s	8s	26m3s	1m35s	<1s	27m52s
B99_755025	5.9 MB	14	2.8 MB	<1s	4s	10m22s	57s	<1s	11m49s
IPC_V1.7	3.4 MB	1	0.03 MB	<1s	1s	8s	<1s	<1s	12s
NC250_1.0.10	7.6 MB	13	5 MB	2s	1s	16m27s	56s	<1s	17m45s
Archer_C9v1	15 MB	16	4.72 MB	1s	12s	4m58s	1m44s	<1s	7m41s

firmware analysis only costs 1120 seconds (for 521 firmware images that be successfully unpacked).

To understand which factors have significant contributions to the performance, we selected 14 representative firmware samples to investigate, which covers different situations. The firmware selection is based on three factors – firmware size, the number of analyzed files, and the size of analyzed files. In Table 5, we show the information about the analyzed firmware (i.e., the firmware size, the number of extracted files, the number and the size of executables that involve crypto) and the time consumption of each step. It turns out that, in general, IR lifting consumes most of the time. Additionally, we could observe that the larger the size of the analyzed files is, the more time it spends on ICFG construction. Moreover, the time cost of taint analysis is related to the complexity of ICFG directly, especially the number of path branches. For example, since *My_Cloud_KC2A* contains up to 26.73 MB files, it took 9m 42s to construct ICFG, and further spent 46m 19s on the taint analysis (due to its complex ICFG).

On the other hand, since CRYPTOREX is the first work focusing on the crypto misuse issue in IoT systems, it is inappropriate to make a crosswise performance comparison between our work and the previous ones on mobile platforms (see Section §6.2 for more discussions).

4.6 Case Studies

To further understand the effectiveness of CRYPTOREX and the risk of crypto misuse, we provide three case studies with in-depth analysis.

4.6.1 Tomato Shibby Router

In the firmware of the Tomato Shibby router, CRYPTOREX reported a crypto misuse that is also an authentication bypass vulnerability. We list the vulnerable function in Listing 3. At

Line 5, function `sub_2493C()` invokes `nvrkam_get()` to get the encryption key from the NVRAM parameter. However, if such the parameter is not filled, the function will use "admin"³ as the default encryption key (Line 11, 14, and 19). In the implementation of function `crypt`, if the second parameter starts with "\$1\$", it will combine the first parameter and the second parameter as the key to encrypt a constant string and output a token. This token will be further used in password checking. This vulnerability could allow attackers to bypass the authentication.

```

1 int sub_2493C()
2 {
3     ...
4     char *dest;
5     v3 = (const char *)nvrkam_get("
6         http_passwd");
7     v4 = v3;
8     strcpy(dest, "$1$");
9     f_read("/dev/urandom", dest + 3, 6);
10    if ( v3 ) {
11        if ( !*v3 )
12            v4 = "admin";
13    }
14    else {
15        v4 = "admin";
16    }
17    ...
18    v8 = fopen("/etc/shadow", "w");
19    if ( v8 ) {
20        v9 = crypt(v4, dest); //low-level
21        crypto function
22        fprintf(v8, "root:%s:0:0:99999:7:0:0:\
23            nnobody:*:0:0:99999:7:0:0:\n", v9)
24        ;
25        ...
26        fclose(v8);

```

³It is also the default user password.

```

23 }
24 ...
25 sprintf( (char *)&v11,
26 "root:x:0:0:root:/root:/bin/sh\n"
27 "%s:x:100:100:nas:/dev/null:/dev/null\n"
28 "nobody:x:65534:65534:nobody:/dev/null:/
  dev/null\n", v6);
29 f_write_string("/etc/passwd", &v11, 0,
  420);
30 }

```

Listing 3: Vulnerable code: Tomato Shibby router.

4.6.2 Open-source File Server Netatalk

CRYPTOREX reported crypto misuses on several NAS firmware images. Our further investigation shows that the misuses all occur in the same shared open-source file server called Netatalk. It supports five kinds of authentication ways, which involve random number exchange and Diffie-Hellman key exchange. As shown in Listing 4, for random number exchange, the server uses the user password as a key to encrypt a generated random number and send it to challenge the client. For this case, the ECB mode is used in the DES algorithms (Line 7), which allows attackers to guess random numbers and masquerade as legitimate users easily. In the other function `pwd_login()` (Line 16) that executes Diffie-Hellman key exchange to negotiate a shared key, we found that a constant IV (Line 20) is used for the CBC encryption (i.e., `CAST_cbc_encrypt()`). Without using random IV, attackers can easily brute-force the user password.

```

1  /***** uams_randnum.so *****/
2  static int randnum_logincont(void *obj,
3      struct passwd **uam_pwd,
4      char *ibuf, size_t ibuflen _U_,
5      char *rbuf _U_, size_t *rbuflen)
6  {
7      ...
8      DES_ecb_encrypt((DES_cblock *) randbuf,
9          (DES_cblock *) randbuf, &seskeysched,
10         DES_ENCRYPT);
11     ...
12     if (memcmp( randbuf, ibuf, sizeof(
13         randbuf) )) { /* != */
14         return AFPERR_NOTAUTH;
15     }
16     ...
17 }
18
19 /***** uams_dhx_passwd.so *****/
20 static int pwd_login(void *obj, char *
21     username, int ulen, struct passwd **
22     uam_pwd _U_,
23     char *ibuf, size_t ibuflen _U_,
24     char *rbuf, size_t *rbuflen)
25 {

```

```

20 unsigned char iv[] = "CJalbert";
21 ...
22 CAST_cbc_encrypt((unsigned char *)rbuf,
23     (unsigned char *)rbuf, CRYPTBUFLLEN,
24     &castkey, iv, CAST_ENCRYPT);
25 ...
26 }

```

Listing 4: Vulnerable code: Netatalk.

4.6.3 OpenSSL

To our surprise, CRYPTOREX also reported a crypto misuse bug in the standard OpenSSL library. As Listing 5 shows, the function `EVP_BytesToKey()` (Line 22) is used to generate keys and IVs. This function is then invoked with its sixth parameter being 1, which sets the round of iteration to 1. Such an operation violates Rule 5, allowing attackers to perform brute-force attacks to guess the keys and the IVs.

```

1  const OPTIONS enc_options[]=
2  {
3      ...
4      {"k", OPT_K, 's', "Passphrase"},
5      ...
6  };
7
8  int enc_main(int argc, char **argv)
9  {
10     while ((o = opt_next()) != OPT_EOF)
11     {
12         switch (o)
13         {
14             ...
15             case OPT_K:
16                 str = opt_arg();
17                 break;
18             ...
19         }
20     }
21     ...
22     if (!EVP_BytesToKey(cipher, dgst, sptr, (
23         unsigned char *)str, str_len, 1, key
24         , iv))
25     {
26         ...
27     }
28     if (!EVP_CipherInit_ex(ctx, NULL, NULL,
29         key, iv, enc))
30     {
31         ...
32     }
33 }

```

Listing 5: Vulnerable code: OpenSSL.

5 Discussions and Limitations

Though our framework achieves cross-architecture analysis and discovered several cryptographic misuse cases in the wild, there still exist some venues for improvements.

Firmware extraction. In our framework, the firmware will be unpacked to obtain the executables. In the implementation, this step is completed by Binwalk which is the de facto standard for firmware unpacking. However, Binwalk is not a silver bullet for every firmware format: (1) If the firmware images are packed with private compression algorithms which are not covered by Binwalk, the unpacking will fail; (2) Furthermore, if the firmware images are encrypted, Binwalk also cannot unpack them without the correct decryption keys. As a result, due to the non-standard firmware formats, only around 39.3% of firmware samples can be unpacked successfully. This situation limits the scope of our analysis.

Cryptographic function identification. CRYPTOREX uses the API data coming from seven popular cryptographic libraries, which have covered most cases. However, some developers may implement cryptographic algorithms by themselves, i.e., non-standard implementations. The current mainstream cryptographic function identification techniques in binary programs concentrate on dynamic analysis, especially comparing the I/O relationships [14,35]. However, since CRYPTOREX is static analysis solution, these techniques could not be applied directly.

Dynamically generated data. During taint analysis, CRYPTOREX only covers the data stored in binary files. However, if the vulnerable cryptographic parameters are generated dynamically (e.g., received from the Internet or got from NVRAM), CRYPTOREX cannot detect such misuse cases.

IoT apps. Except for firmware in IoT devices, some IoT vendors also provide a mobile app (Android or iOS) to assist the user in controlling the device. In our framework, we do not cover the cryptographic misuse issues in these IoT apps. Some previous work has proposed cryptographic misuse detection solutions for mobile platforms (see Section §6.2).

Automatic repair. As the first step, our technique can identify crypto misuse in IoT devices automatically. However, automated fixing the discovered crypto misuses remains a challenge. One possible solution is to fix the misused function arguments and rewrite the executables automatically. We leave this to future work.

6 Related Work

In this section, we review prior research about security analysis of firmware and misuse of crypto functions.

6.1 Security Analysis of Firmware

In recent years, researchers have a strong interest in the security analysis of firmware in IoT devices [15, 16, 18–21, 25, 27, 39, 45, 48, 49]. On the one hand, prior studies have emphasized addressing the challenges in dynamic analysis and static analysis of firmware. Regarding dynamic analysis, the main challenge lies in the emulation failures caused by diverse architectures and unavailable NVRAMs. To this end, researchers proposed different systems to support firmware emulation [15, 49]. For instance, Chen et al. [15] proposed a dynamic analysis tool which emulates the entire filesystem of Linux-based firmware. Avatar [49] is a framework that orchestrates the execution of an emulator with the real hardware, by forwarding I/O accesses from the emulator to the embedded devices. Apart from that, static firmware analysis also faces the challenge of different architecture [16]. As a result, the solutions to cross-architectural bug search have been widely investigated [25, 27, 45, 48]. For example, Xu et al. [48] proposed a neural network-based graph embedding system that supports multiple firmware architectures and can significantly speed up the vulnerability detection process. However, given that those techniques are designed for discovering general vulnerabilities and focus on scalability, they are less inaccurate for specific vulnerability types. On the other hand, another direction aims to detect specific types of vulnerabilities. For example, Costin et al. [19] performed a large-scale study and found that nearly 10% of the collected firmware images contain bugs in the web interfaces. Shoshitaishvili et al. [46] proposed a model to describe the authentication bypass vulnerability of firmware. David et al. [20] presented an IR-based static analysis solution for finding CVEs in stripped firmware images, which is the most relevant work to our CRYPTOREX. Nevertheless, none of the previous work has tackled the difficulty in identifying crypto misuse of firmware.

6.2 Misuse of Cryptographic Functions

Previous work has noticed the problem of crypto misuse on mobile platforms, say Android and iOS. However, how to detect crypto misuse in IoT systems is an open question.

Egele et al. [23] were the first to perform a large-scale experiment to measure cryptographic misuse in Android apps. Their result showed 88% of tested apps made at least one mistake. Wang et al. [47] analyzed crypto misuse issues in Android native libraries. Muslukhov et al. [40] studied how crypto APIs misuse in Android applications has changed between 2012 and 2016. It provides some updated findings, such as significantly fewer libraries and applications were using ECB mode in 2016. On the other hand, Ma et al. [37] proposed an approach, CDRep, to automatically repair vulnerable Android apps with cryptographic misuses.

Similarly, on the iOS platform, Li et al. [36] designed iCryptoTracer to check cryptographic usage, but the scale of their

dataset was quite small. Instead of inspecting a low-level representation of a binary, Feichtner et al. [26] proposed an LLVM-based approach to uncover cryptographic misuse in iOS apps. The result shows that 82% of apps are subject to at least one security misconception.

Also, more recently, Li et al. [34] proposed K-Hunt, a system for identifying insecure keys in x86/64 executables. Different from the above work, CRYPTOREX focuses on the crypto misuse problems in IoT devices. Also, it solves the challenge of cross-architecture crypto misuse checking, not just concentrating on a single platform.

7 Conclusion

In this paper, we introduced the automated cross-architecture analysis framework CRYPTOREX, for detecting cryptographic misuse bugs in IoT devices in a large-scale manner. It utilizes an intermediate representation to solve the issue of non-unified underlying architectures of IoT firmware. Following this trail, in the design of CRYPTOREX, we developed a series of practical techniques to achieve the purpose of reliable crypto misuse detection. Finally, we implemented CRYPTOREX and carried out experiments based on 1327 real-world firmware image samples (from 12 vendors, in 7 different architectures, 521 successfully unpacked ones). CRYPTOREX successfully identified 679 crypto misuse cases, which demonstrates the feasibility of our solution and sheds light on the worrisome situation in today's IoT development.

Acknowledgements

We are grateful to our shepherd Johannes Kinder and the anonymous reviewers for their insightful comments. This work was partially supported by Hong Kong S.A.R. Research Grants Council (RGC) Early Career Scheme / General Research Fund No. 24207815, No. 14217816, National Natural Science Foundation of China (NSFC) under Grant No. 61902148, No. 91546203, Key Science Technology Project of Shandong Province No. 2015GGX101046, Major Scientific and Technological Innovation Projects of Shandong Province, China No. 2018CXGC0708, No. 2017CXGC0704, and Qilu Young Scholar Program of Shandong University.

References

- [1] Angr. <https://github.com/angr/angr>.
- [2] Binwalk. <https://github.com/ReFirmLabs/binwalk>.
- [3] Buildroot. <https://buildroot.org/>.
- [4] Cryptlib. <https://www.cryptlib.com/>.
- [5] IDA F.L.I.R.T. Technology: In-Depth. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml.
- [6] Libgcrypt. <https://gnupg.org/software/libgcrypt/>.
- [7] LibTom. <https://www.libtom.net/LibTomCrypt/>.
- [8] Nettle - a low-level cryptographic library. <http://www.lysator.liu.se/~nisse/nettle/>.
- [9] wolfSSL. <https://www.wolfssl.com/>.
- [10] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium (USENIX-SEC), Vancouver, BC, Canada, August 16-18, 2017*, 2017.
- [11] Gogul Balakrishnan and Thomas W. Reps. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, 2004.
- [12] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011.
- [13] Matt Burgess. Google Home's data leak proves the IoT is still deeply flawed. <https://www.wired.co.uk/article/google-home-chromecast-location-security-data-privacy-leak>, 2018.
- [14] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS), Raleigh, NC, USA, October 16-18, 2012*, 2012.
- [15] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 21-24, 2016*, 2016.
- [16] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang.

- IoTfuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 18-21, 2018*, 2018.
- [17] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg City, Luxembourg, June 25-28, 2018*, 2018.
- [18] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Security Symposium (USENIX-SEC), San Diego, CA, USA, August 20-22, 2014.*, 2014.
- [19] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS), Xi'an, China, May 30 - June 3, 2016*, 2016.
- [20] Yaniv David, Nimrod Partush, and Eran Yahav. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Williamsburg, VA, USA, March 24-28, 2018*, 2018.
- [21] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium (USENIX-SEC), Washington, DC, USA, August 14-16, 2013*, 2013.
- [22] Thomas Dullien and Sebastian Porst. REIL: A Platform-independent Intermediate Representation of Disassembled Code for Static Code Analysis. In *The Annual CanSecWest Conference*, 2009.
- [23] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS), Berlin, Germany, November 4-8, 2013*, 2013.
- [24] Ericsson. Internet of Things forecast. <https://www.ericsson.com/en/mobility-report/internet-of-things-forecast>, 2018.
- [25] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discoverRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 21-24, 2016*, 2016.
- [26] Johannes Feichtner, David Missmann, and Raphael Spreitzer. Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec), Stockholm, Sweden, June 18-20, 2018*, 2018.
- [27] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, October 24-28, 2016*, 2016.
- [28] Josh Fruhlinger. How teen scammers and CCTV cameras almost brought down the internet. <https://www.csoonline.com/article/3258748/security/the-mirai-botnet-explained-how-teen-scammers-and-cctv-cameras-almost-brought-down-the-internet.html>, 2018.
- [29] Free Standards Group. Interfaces for libcrypt. http://refspecs.linuxbase.org/LSB_3.0.0/LSB-PDA/LSB-PDA/libcrypt.html.
- [30] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R. B. Butler. FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.
- [31] Selena Larson. FDA confirms that St. Jude's cardiac devices can be hacked. <https://money.cnn.com/2017/01/09/technology/fda-st-jude-cardiac-hack/>, 2017.
- [32] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO), San Jose, CA, USA, 20-24 March 2004*, 2004.
- [33] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *Proceedings of 5th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys), Beijing, China, June 25-26, 2014*, 2014.

- [34] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada, October 15-19, 2018*, 2018.
- [35] Xin Li, Xinyuan Wang, and Wentao Chang. CipherXRay: Exposing Cryptographic Operations and Transient Secrets from Monitored Binary Execution. *IEEE Transactions Dependable Secure Computing*, 11(2):101–114, 2014.
- [36] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. In *Network and System Security - 8th International Conference, NSS 2014, Xi'an, China, October 15-17, 2014, Proceedings*, 2014.
- [37] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS), Xi'an, China, May 30 - June 3, 2016*, 2016.
- [38] Roland Moore-Colyer. Samsung SmartThings Hub vulnerabilities leave smart home devices open to attack. <https://www.theinquirer.net/inquirer/news/3036719/samsung-smartthings-hub-riddled-with-20-security-bugs>, 2018.
- [39] 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 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 18-21, 2018*, 2018.
- [40] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source Attribution of Cryptographic API Misuse in Android Applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (AsiaCCS), Incheon, Republic of Korea, June 04-08, 2018*, 2018.
- [41] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI), San Diego, California, USA, June 10-13, 2007*, 2007.
- [42] OpenSSLWiki. Libcrypto API. https://wiki.openssl.org/index.php/Libcrypto_API.
- [43] Charlie Osborne. Meet Torii, a new IoT botnet far more sophisticated than Mirai variants. <https://www.zdnet.com/article/meet-torii-a-new-iot-botnet-far-more-sophisticated-than-mirai/>, 2018.
- [44] OWASP. Guideline: Testing for Weak Encryption. [https://www.owasp.org/index.php/Testing_for_Weak_Encryption_\(OTG-CRYPST-004\)](https://www.owasp.org/index.php/Testing_for_Weak_Encryption_(OTG-CRYPST-004)).
- [45] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, USA, May 17-21, 2015*, pages 709–724, 2015.
- [46] 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 Symposium (NDSS), San Diego, California, USA, February 8-11, 2015*, 2015.
- [47] Qing Wang, Juanru Li, Yuanyuan Zhang, Hui Wang, Yikun Hu, Bodong Li, and Dawu Gu. NativeSpeaker: Identifying Crypto Misuses in Android Native Code Libraries. In *Information Security and Cryptology - 13th International Conference, Inscrypt 2017, Xi'an, China, November 3-5, 2017, Revised Selected Papers*, 2017.
- [48] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.
- [49] 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 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 23-26, 2014*, 2014.
- [50] Nan Zhang, Soteris Demetriou, Xianghang Mi, Wenrui Diao, Kan Yuan, Peiyuan Zong, Feng Qian, XiaoFeng Wang, Kai Chen, Yuan Tian, Carl A. Gunter, Kehuan Zhang, Patrick Tague, and Yue-Hsun Lin. Understanding IoT Security Through the Data Crystal Ball: Where We Are Now and Where We Are Going to Be. *CoRR*, abs/1703.09809, 2017.

PAtt: Physics-based Attestation of Control Systems

Hamid Reza Ghaeini¹, Matthew Chan², Raad Bahmani³, Ferdinand Brasser³, Luis Garcia⁴, Jianying Zhou¹, Ahmad-Reza Sadeghi³, Nils Ole Tippenhauer⁵, and Saman Zonouz²

¹Singapore University of Technology and Design, ghaeini@acm.org, jianying_zhou@sutd.edu.sg

²Rutgers University, {matthew.chan, saman.zonouz}@rutgers.edu

³TU Darmstadt, {raad.bahmani, ferdinand.brasser, ahmad.sadeghi}@trust.tu-darmstadt.de

⁴University of California, Los Angeles, garcialuis@ucla.edu

⁵CISPA Helmholtz Center for Information Security, tippenhauer@cispa.saarland

Abstract

Ensuring the integrity of embedded programmable logic controllers (PLCs) is critical for the safe operation of industrial control systems. In particular, a cyber-attack could manipulate control logic running on the PLCs to bring the process of safety-critical application into unsafe states. Unfortunately, PLCs are typically not equipped with hardware support that allows the use of techniques such as remote attestation to verify the integrity of the logic code. In addition, so far remote attestation is not able to verify the integrity of the physical process controlled by the PLC.

In this work, we present PAtt, a system that combines remote software attestation with control process validation. PAtt leverages operation permutations—subtle changes in the operation sequences based on integrity measurements—which do not affect the physical process but yield unique traces of sensor readings during execution. By encoding integrity measurements of the PLC’s memory state (software and data) into its control operation, our system allows us to remotely verify the integrity of the control logic based on the resulting sensor traces. We implement the proposed system on a real PLC, controlling a robot arm, and demonstrate its feasibility. Our implementation enables the detection of attackers that manipulate the PLC logic to change process state and/or report spoofed sensor readings (with an accuracy of 97% against tested attacks).

1 Introduction

Industrial control systems (ICS) are a class of cyber-physical systems (CPS) that typically consist of industrial controllers sensing and actuating safety-critical applications, e.g., the power grid, water treatment plants, and factory automation [58]. In particular, ICS typically consist of programmable logic controllers (PLCs), which are embedded systems that act as a reliable and re-programmable cyber-physical interface between a monitoring entity, i.e., the Supervisory Control and Data Acquisition (SCADA) center, and field-level devices,

i.e., sensors and actuators that interface directly with the physical environment. As such, the security of these controllers is critical for ensuring the safe operation of the ICS [40].

Because of the safety-critical nature of PLCs, they have been typically targeted by nation-state malware, such as the infamous Stuxnet worm [18] against uranium enrichment facilities in Iran and the BlackEnergy crimeware [17] that targeted Ukrainian electric power and train railway systems [14, 47]. These attacks typically target the application-layer software, so-called *control logic*, due to the lack of security features in legacy industrial protocols. Although it has been shown that attackers can implement firmware-level attacks [6, 20], these attacks have been shown to be much more challenging to implement as they require a much more concerted effort for stealthiness.

Although these attacks are understood, they are challenging to defend against as security solutions need to be employed for legacy systems with fixed hardware. Currently, PLCs do not have hardware support to provide a hardware root-of-trust. Physical Unclonable Functions (PUFs) have been proposed in the past to enable software attestation for resource-constrained devices, but such modules are also not yet available for industrial devices. Existing integrity checks in industrial devices are limited to checksums that are preloaded onto the device when the program is initially loaded. In prior works, several solutions have been presented to either test the code that is being loaded onto the device [41] or verify the cyber and physical behavior of the overall CPS [2, 21, 22, 60]. However, these solutions treat the PLCs as black boxes and are not able to monitor the internal states at run-time. We conclude that a comprehensive solution to enable remote attestation of the logic running on a PLC is missing.

In this paper, we present PAtt, a remote attestation technique that combines software remote attestation with a physical PUF to attest the control-logic code is running on PLCs without trusted hardware. PAtt allows a verifier to challenge a PLC to generate an attestation report in the form of sensor values which are affected by a series of actuation commands (based on the challenge and checksums over the PLC logic).

The verifier can then attest the logic code integrity based on the measurement of the PLC memory, as well as the authenticity of the reply through the sensor values (similar to a PUF). In particular, we also show that PAtt can detect attempts of the attacker to replay the sensor readings with an accuracy of 97%. PAtt detects those manipulations based on an anomaly detector that is trained with data resulting from normal operation and does not need to be trained with prior attack examples.

Contributions. We summarize our contributions as follows.

- We present PAtt, a novel remote attestation technique for PLCs that combines software remote attestation with a PUF-like use of the physical process to attest the software and process state of the PLC.
- We theoretically investigate the performance of the proposed system and show that it is resilient against replay attacks which provide the sensor reading from a sensor record table. PAtt does not need to have prior knowledge of possible attacks, and it only requires the normal operational data during the training phase of the detector.
- We then implement and practically evaluate PAtt on a real ICS—a robotic arm in the context of a safety-critical process—and show that PAtt can detect the tested attacks with an accuracy of 97%.

The rest of the paper is structured as follows. First, we provide a background on previous techniques in remote attestation for CPS in Section 2. We describe the system model and design of PAtt in Section 3. Details on our implementation are provided in Section 4. We present our evaluation results in Section 5. We discuss the applicability of PAtt in Section 6 and we summarize related work in Section 7. Finally, we conclude in Section 8.

2 Background

In this section, we first provide an introduction to programmable logic controllers (PLCs) in the context of cyber-physical industrial control systems (ICS) as well as their security limitations. We then provide background on previous works in attestation of cyber-physical systems.

Industrial Control Systems. Modern industrial control systems consist of three major levels [29]:

- Supervisory Control And Data Acquisition (SCADA): this level of the ICS is mainly used for the control and monitoring of industrial process that may consist of large-scale geographical distributed computers. Five major components of the SCADA are the human-machine interface (HMI), data acquisition server, historian, engineer workstations, and remote workstation.
- Programmable Logic Controllers (PLC): The local control component that is mostly designed for managing a

single process in ICS. PLCs are industrial computers that are developed for handling the process level devices like sensors and actuators.

- Fieldbus: The physical elements like sensors and actuators are connected to the PLC at this level. Most of the recent Fieldbus implementations use the Device Level Ring (DLR) with two redundant PLCs and a ring topology between those PLCs and physical elements.

In cyber-physical systems, the term Programmable Logic Controller refers to computing devices, which control the industrial appliances. Each PLC consists of (1) computing modules, which are designed to perform industrial processes reliably, (2) input modules translating analog physical inputs to digital values, and (3) outputs modules, which map the PLC's digital outputs to analog physical outputs.

In a PLC, the next system state is computed based on the current state measured by the input and output modules. The main part of the logic executed on a PLC (*control logic*) is programmed in special-purpose industrial languages, e.g., ladder logic, designed to guarantee a reliable transfer between system states [9]. Control logic is compiled at a SCADA server and downloaded to the PLC. The PLC runs this program to perform a control task by processing a set of inputs, received from physical sensors, and generating outputs to be interpreted by actuators. Control logic runs on top of a privileged software layer, e.g., a real-time operating system (RTOS), which provides the required services. The control logic consists of *function blocks*, *data blocks*, and *organization blocks*. Function blocks contain reusable functions and data blocks include data structures holding *global* or *local* variables used in the control logic. Organization blocks serve as the entry point of a PLC program and execute in a fixed time interval, known as *scan cycles*.

Remote Attestation. Remote attestation is a technique, which provides an external verifier with proofs on the integrity of a system's software state. It is termed *software-based attestation* when the proof of integrity is generated with no hardware aid. This form of attestation is based on strong assumptions regarding the time and the authenticity of the communication channel between the system and the external verifier [3, 49]. The limitations of software-based attestation can be overcome by using trusted hardware. A software, protected by trusted hardware, could measure the systems software stack and authenticate the measurement using a secret key which is likewise protected by the trusted hardware. Attestation techniques can be used together with other security solutions like Transport Layer Security (TLS) to prevent eavesdropping and Man-in-the-Middle scenarios.

Physically Unclonable Functions. Physically Unclonable Functions (PUFs) are like a physical fingerprint used in semiconductors to generate key information with a level of randomness from a complex physical system [27]. PUFs leverage

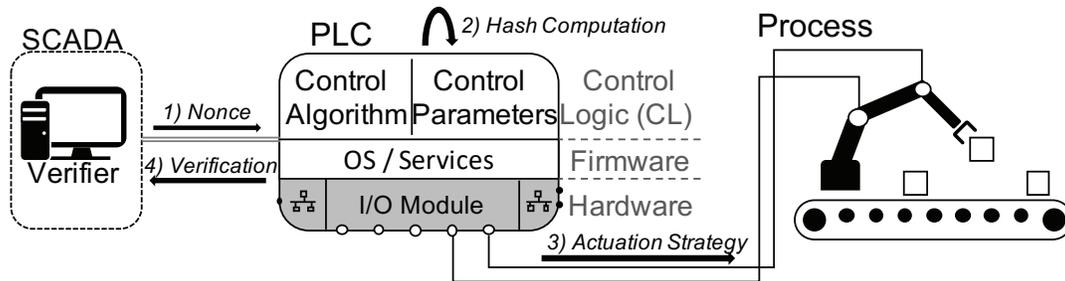


Figure 1: Overview of steps in PAtt framework.

unpredictable physical variations that occur naturally during semiconductor manufacturing. PUFs are used together with hash functions in many cryptographic applications [32, 50]. The latest versions of PUFs are equipped in integrated circuits and used in different security applications like software licensing. Recent industrial control systems do not include PUF-based integrated circuits inside the PLCs. However, there are some proposals to use the physical process as a PUF [62].

3 PAtt: Physics-based Attestation

PAtt is designed to allow remote attestation of logic code running on a PLC without a traditional trust anchor (such as a TPM or PUF). The devices that we target—PLCs in existing and legacy systems—are not usually equipped with trusted computing hardware to enable a hardware-based remote attestation process. While recent versions of PLC firmware (e.g., in the Siemens S7 series) include APIs that can be used for checksum generation over data blocks of control logic, the challenge is to authenticate such measurements.

This motivates the novel concept of PAtt: sensor readings from the physical process are used to authenticate the attestation response. The software attestation result is tied to the physical process readings through a derivation of an actuation path from the cryptographic hash of the control logic. We now introduce the system and attacker model and then provide an overview of the proposed system.

3.1 System and Attacker Model

The industrial control system considered in this work consists of a PLC that is controlling a dynamic local physical process such as a robotic arm, a laser, or extruder. The control logic of the PLC is responsible for real-time sensing and actuation of the dynamic process, e.g., periodic transport of a manufacturing component from one position to another position. The PLC does not provide onboard support for trusted execution or cryptographic signatures. Instead, the PLC does provide the capability to compute cryptographic hash functions over one or more data blocks used by the control logic (e.g., as possible on the Siemens S7 series PLCs). A remote attestation

server (the "verifier") is connected to the PLC over the local network and is attempting to verify the correct state of the system. Attacks that compromise the attestation server are out of the scope of this work. The verifier has a model of the physical processes that are trained during normal operation in the absence of the attacker.

We consider an adversary that has compromised the PLC. The attacker's goal is to change the way the physical process is actuated while hiding this compromise from the attestation server. The attacker is limited to executing code on the compromised PLC, which has limited computational power and memory. In particular, the attacker does not have additional computation devices inside the industrial network. As the attacker has compromised the PLC, attacks that would manipulate the PLC firmware is subsumed in our attacker model (as the data could also be manipulated by the PLC firmware when sent or received). We considered two types of attackers:

1. Hash approximation: This attack is designed for evaluating decoding precision. In this attack, a number hash bits will be flipped at a random offset of the hash.
2. Replay attack: In this attack, the attacker will replay a stored sensor reading inside the PLC that corresponds to a subset of the actual hash.

3.2 PAtt Framework

We now propose the Physics-based attestation (PAtt) framework, which allows the attestation server (AS) to perform remote attestation of the PLC's currently loaded control logic.

Overview. The main steps of PAtt are summarized in Figure 1: 1) the AS initiates the attestation process by sending a fresh challenge nonce over the network using industrial protocols; 2) the PLC stores this nonce as data block in the memory accessible to the logic code, and the PLC then computes a cryptographic hash over the PLC logic code blocks including the nonce; 3) the resulting hash is interpreted as an actuation strategy for the robotic arm (details on this are provided later) and the movement path is executed by the actuator; 4) the resulting sensor readings during this movement are collected (sent to the AS together with the hash), and the AS verifies

that the hash was derived from correct PLC logic and the nonce, and that the sensor measurements fit the specific physical process and hash. If the last step is successful, the AS has attested the integrity of the logic on the PLC. In practice, only a limited number of actuation can be executed within a scan cycle of a PLC (e.g., 10ms), which might require us to run multiple iterations (or rounds) of the protocol. We defer discussion of that implementation detail to Section 4. We now provide additional details on each step.

1) Attestation Request. The AS initiates the attestation request by sending the PLC a nonce (a randomly generated bit vector) using the standard industrial protocols. The PLC firmware receives the nonce and makes it available to the PLC control logic as normal data tag. Figure 2 shows a more detailed view of the interactions between the verifier, PLC, and the physical process during the attestation process.

2) Nonce Storage and Hash Computation. The PLC then computes a cryptographic hash function over a group of control logic objects (standard blocks, safety blocks, text lists, and the received nonce from the verifier). If the hash function is also able to cover the firmware memory space, that region should be included in the hash as well. We note that in our implementation, the specific PLC used is only able to cover logic accessible memory areas, and thus not the RTOS.

3) Derivation of Actuation Strategy, and Execution of Path. The cryptographic hash generated in the previous step is then encoded to an actuation strategy for the actuator controlled by the PLC. The intuition is that the execution of such strategies will (deterministically) create sensor readings unique for the physical process. To not impede on normal process operations, the actuation strategy should essentially represent an alternative way to reaching the normal operational goal of the process, without violating safety constraints. Additional details on the derivation of the actuation strategy are provided in the following subsections.

4) Verification of Hash and Measurements. Each physical process is unique, which will be reflected in a process-specific noise signal in all reported sensor values. In PAtt, the verifier knows the unique noise signature of all prover devices and can, therefore, identify all PLCs and the connected physical systems based on the sensor readings transmitted during attestation, i.e., the attested PLC is authenticated in the process through the reported sensor values. This noise includes any source of random noise e.g., the noise of the system, manufacturing imperfections, and differences. This ensures that the reported sensor values authenticate the attestation report. In Section 3.4, we propose to use machine learning techniques to compute the classifier prediction probability. Along with decoding the hash, we use the classifier prediction probability as a weight to compute the weighted distance between the reconstructed hash and the original hash. We describe the implementation details of the hash verification in Section 4.

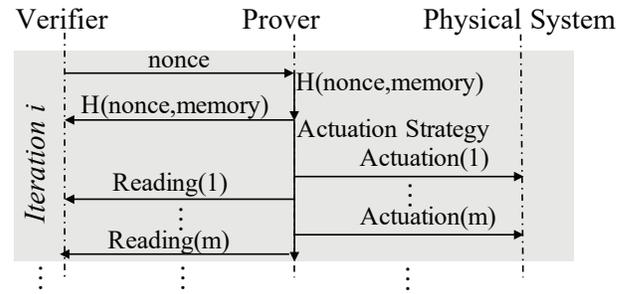


Figure 2: Interaction sequence of the verifier, the prover, and the physical system during attestation.

3.3 Generation of Actuation Strategy

We now discuss how the hash value of the logic-accessible memory areas and the nonce can be interpreted as an actuation strategy. In general, we assume that the cryptographic hash has length m bits. Assuming that there are two different potential actuation actions (e.g., a move horizontally or vertically), we interpret the hash as a sequence of those binary actuation commands, with m commands being executed per iteration. Figure 2 shows the interaction sequences of the verifier, PLC, and the physical process during the remote attestation process. We now provide additional details on individual commands in the strategy (which we call micro-commands).

Macro-commands and Micro-commands. Macro-commands are abstract movements to reach a goal from the start. The macro-command can be executed by different sequences of micro-commands (called a path strategy). Consider the following example: a robot arm with three stepper motors, which when actuated together will change the arm hand position in the x , or y , or z directions. The designed control logic of the robot arm will traverse the arm hand from the position (x_0, y_0, z_0) to the position (x_1, y_1, z_1) . We define the macro-command as move from (x_0, y_0, z_0) to the position (x_1, y_1, z_1) . The coordinate system is scaled such that each micro-command moves the arm by one unit (in particular, the arm does not move diagonally).

Path Strategy. Continuing our example, the path strategy now determines the sequence of micro-commands to execute the macro-command. To simplify things, we only consider micro-commands in direction x and y (and always use the same z direction commands). The arm hand will start from (x_0, y_0, z_0) and takes $x_1 - x_0$ steps towards the x direction, and $y_1 - y_0$ steps towards the y direction. Thus, the total number of steps (or micro-commands) is $(x_1 - x_0) + (y_1 - y_0)$. The order of micro-commands is defined by the path strategy.

We represent the path strategy as a binary vector (which we call a coding), with a micro-command in x -axis direction represented by a 0 bit, a micro-command in y -axis direction represented by a 1 bit. For example, our robot arm goal

might be to take five steps in the x direction and to take three steps in the y direction. Two possible path strategies could be 10100010, and 00000111. Figure 8 in Appendices 10.4 shows an example path strategy. We know that the number of unique paths u in a $x \times y$ grid can be computed as follows [57]:

$$u = \frac{(x+y)!}{(x!y!)} \quad (1)$$

Thus, we can enumerate all possible paths, and use an integer between 1 and u as an index to represent a specific path strategy in a $x \times y$ grid. In PAtt, this index is a random number resulting from the software attestation phase (i.e., the resulting cryptographic hash is interpreted as integer). The micro-commands of the chosen actuation strategy is then executed, and sensor readings are recorded to be sent as part of the attestation response.

3.4 Verification of the Measurement Traces

In the verification phase, the Verifier checks that the hash received from the Prover was derived from correct PLC logic and the nonce, and that the sensor measurements fit the specific physical process and hash. If the last step is successful, the Verifier has attested the integrity of the logic on the PLC.

Hash verification. As the Verifier has access to the logic that is supposed to run on the PLC (and the nonce), it is easy to check if the hash is correct. In particular, the Verifier computes the hash locally and compares with the received hash.

Replay Attack Detection. Next, the Verifier has to ensure that the received sensor reading sequence fits the received hash (which we call decoding) and that the sensor reading sequence was not spoofed (e.g., by simulation of the physical process or replay attack). The decoding process is designed to translate the sensor readings to the original hashes considering the physical behavior of the system and its non-deterministic noise. The features available are the actual sensor reading traces and information on the physical process (e.g., statistical properties of noise from the sensors) that were measured during the setup phase of the system. The decoding can be done by signal processing techniques (e.g., matched filters that detect movements and reconstruct the actuation strategy/hash), or machine learning approaches. The detection of replayed sensor traces can similarly be performed by statistical analysis, signal processing techniques, or machine learning.

3.5 Security Analysis

As the hash received is a result of weak software-based attestation, the verifier also needs to check if the hash was received within a specific time window. In particular, we need to prevent a compromised PLC from sharing the received challenge with a third party oracle that would provide the correct hash. As PLCs are only able to send and received messages synchronized with scan cycles, we use two scan cycles as a maximal

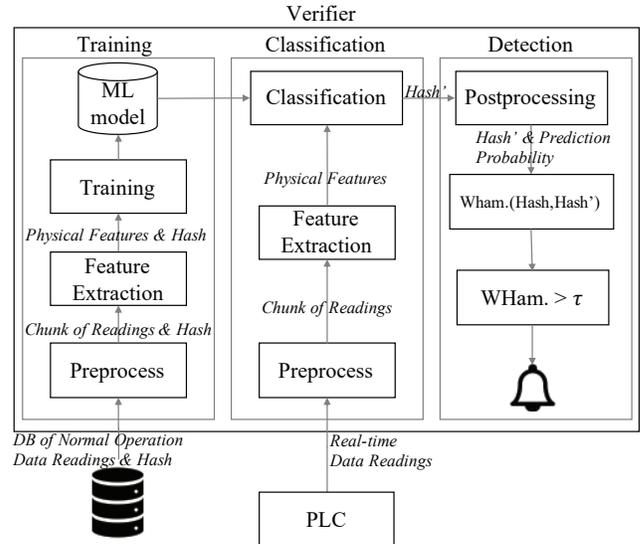


Figure 3: Overview of the validation process in the Verifier.

delay to provide the hash (in the first scan cycle, a challenge is received, and in the second the hash is computed and sent to the Verifier). In our implementation, scan cycles are 10ms long, so the maximal delay to provide the hash is 20ms.

Full replays of earlier actuation sequences are not feasible for the attacker, as she would need to eavesdrop and store a significant share of the hash space (i.e., actuation space and its corresponding sensor readings) in order to reliably be able to perform the replay attack. As the hash contains a fresh nonce, and the hash itself has 256-bit length, we consider this infeasible. As an example, a SIEMENS S7-1200 PLC might have work memory up to 125 Kilobytes, and load memory up to 4 Megabytes. A single trace of sensor readings of a single SHA-256 hash will require more than 300 Kilobytes, hence, storing the possible hashes inside the PLC is infeasible. As we will show in Section 5, in practice our decoding solution was able to detect attacks that could be produced on constrained devices such as PLCs, so there we are able to do both decoding and spoofing detection in one step.

4 Implementation

In this section, we provide details on our practical implementation of PAtt in an ICS use-case. We start by presenting our solution for hash verification and spoofing detection. Then, we present the ICS setup in which we evaluated PAtt.

4.1 Machine Learning Decoder

In our implementation, we chose to use (supervised) machine learning based approaches to both decode the sensor reading traces to the hash and detect spoofing of sensor reading traces. In Figure 3, we provide an overview

of the data processing at the verifier, including the offline training phase. The classifier decodes each individual micro-command from the sensor reading trace to its corresponding bit (or step) in the actuation strategy. To select an appropriate classifier, we implemented and evaluated a number of classifiers using WEKA [23]. In particular, we tested Random Forests [11], Multilayer Perceptrons [7], Decision Forests [25], FURIA [26], DTNB [24] NBTTree [31], LMT [34], J48 [48], PART [19], and REPTree [63]. To benchmark the machine learning model used in our verifier, we have evaluated a set of classifiers that are mostly used in related security research works e.g. [4, 5, 45, 51, 56] as shown in Appendix 5.2. Now we thoroughly discuss different parts of the PAtt as presented in Figure 3.

1) Training. The features that we used are trajectory position of the arm, average movement over a window of time, and statistical features that are mostly used in the signal processing to monitor the signal behavior. These statistical features are the mean of the signal over a window of time and the standard deviation of the signal. We applied the preprocessing of the signals that were reported by the sensor, and we use those features to recover the hash with a probability of prediction of the classifiers. We used the current position, the mean (AVG) and standard deviation (STD) of the Accelerometer and Gyroscope, and the change of mean of Accelerometer as features to create the profile of the physical system (see Table 1). Considering the three-dimensional trajectory of the robotic arm and two Accelerometers and two Gyroscopes, we used 33 features to train the machine learning model, and SciPy and NumPy libraries of Python [12] to automatically generate these features. In total, feature extraction was roughly 300 lines of code.

Table 1: The features classes used in PAtt.

Feature	Formula
Current Position	(x_A, y_A, z_A)
Mean of Accelerometer	$AVG_t(x_A, y_A, z_A)$
STD of Accelerometer	$STD_t(x_A, y_A, z_A)$
Accelerometer Difference	$AVG_t - AVG_{t-1}$
Mean of Gyroscope	$AVG_t(x_G, y_G, z_G)$
STD of Gyroscope	$STD_t(x_G, y_G, z_G)$

2) Classification. The classifiers assign a class to the test data set with a probabilistic model. Considering the sample set X , and class labels form a finite set Y , the classifier would assign a conditional distribution $\Pr(Y|X)$ which for a given sample $x \in X$, the classifier would assign a probability of being in $y \in Y$ class. Depending on the classification method, in hard classification, the sample $x \in X$ will be classified as $y \in Y$ class, where it holds:

$$\hat{y} = \arg \max_y \Pr(Y = y|X) \quad (2)$$

We use the highest classifier prediction probability of decoding a bit as a weight in the weighted distance computation.

3) Detection. In PAtt, the Prover generates a hash from the random nonce and memory block. Then it creates the actuation strategy based on the derived hash, and the physical system performs the actuation strategy and reports the sensor reading traces to the Prover. After decoding this hash (described above), the Verifier needs to decide whether it is authentic, for which we propose to use a weighted Hamming distance. To compute the weighted distance of the original hash and the recovered hash, we used the weighted Hamming distance with the classifier prediction probability as a weight of each bit of the hash. The Hamming distance is the number of non-matching positions between two equal-length string. Considering a noisy channel of transmitting bit arrays, the Hamming distance could be used to determine how many bits are different from the original bit arrays. In this paper, we used the Hamming distance to calculate the distance between the original hash and the decoded hash transmitted by actuation commands to the physical process and retrieved from the physical process by sensor readings. The Hamming distance between two-bit arrays of $a[1..k]$ and $b[1..k]$ is computed by:

$$\text{Ham}(a, b) = \sum_{i=1}^k a_i \oplus b_i \quad (3)$$

We use the weight of specific bits in the distance computation of the Hamming distance. The PAtt is using the classifier prediction probability as the weight for each bit of the hash. The weighted Hamming distance between two bit arrays of $a[1..k]$ and $b[1..k]$, and weight $w[1..k]$ is computed by:

$$\text{WHam}(a, b) = \sum_{i=1}^k w_i(a_i \oplus b_i) \quad (4)$$

The computed weighted Hamming distance over original hash and the recovered hash will be:

$$\text{WHam}(\text{Hash}, \text{Hash}') = \sum_{i=1}^k CPPr_i(\text{Hash}_i \oplus \text{Hash}'_i) \quad (5)$$

where Hash is the original hash, Hash' is the recovered hash, i is the index of the bit, and $CPPr_i$ is the classifier prediction probability of the i^{th} bit. PAtt will trigger an alarm when the computed weighted Hamming distance passes the detection threshold (τ).

4.2 Testbed Design and Setup

In this section, we describe our industrial robot arm testbed and our implementation of PAtt's Prover on the PLC controlling the robot arm.

The PLC and Process. The industrial control system used in this paper is a modern robotic arm controlled by a Siemens

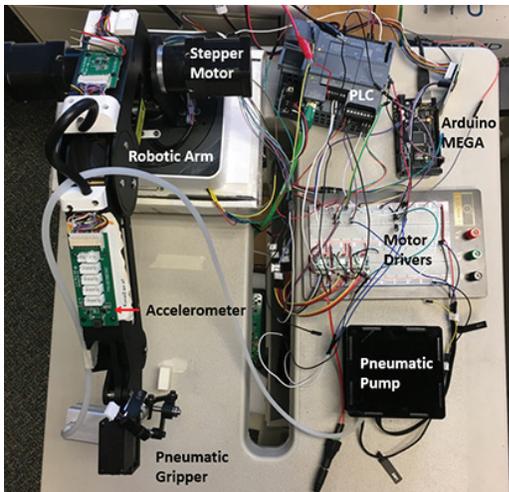


Figure 4: Implemented setup.

S7-1200 PLC with a remote attestation server with at least one GB storage, 16 GB memory, and an Intel Core i7 processor. The PLC is programmed with the controller code for the robotic arm. The verifier communicates with the PLC over an industrial network and uses the SNAP7 library. The verifier is written in Python, and has more than 3000 lines of code.

Our setup consists of a PLC-controlled Dobot robotic arm. The arm includes a rotating base with two arm segments, which we refer to as the rear and front arm segments respectively. At the end of the front arm, the segment is an end-effector, which can hold several different attachments, like a gripper or 3D-printing extruder head. The arm is actuated by three stepper motors, which we control via a Siemens S7-1200 PLC. Actuation is achieved through (i) a Pulse Width Modulation (PWM) Input/Output (IO) module for the PLC, and (ii) one A4988 stepper driver per stepper motor to correctly control the motor coils. Due to limitations in the amount of IO pins available in our setup, we utilize only two stepper motors in our implementation (see Figure 4). The PLC controls the stepper drivers with PWM signals generated by our designed function blocks of the PLC control logic that translate the macro-commands to micro-commands, including the analog conversion of the hash.

For sensing, the robotic arm uses an inertial measurement unit (IMU) on both of its segments, with accelerometer and gyroscope measurements collected during operation being routed to the PLC via a combination of an Arduino MEGA and MAX232 adapter, converting the accelerometers' I2C protocol to an RS232 module on the PLC.

On the PLC, we program the sensing-actuation control loop. We implement the low-level arm kinematics, and point-to-point movement functions adapted from the open-dobot project [1] translated to the Siemens SCL language. To simulate an application-focused environment, we use python

scripts to allow for automated control and link them to the PLC through the Snap7 communication library.

Function Blocks for Prover Functionality. To simplify the usage of the proposed method, in practical industrial control systems we programmed the whole path strategy and attestation functions inside the function blocks that could be easily used by non-experts. The core idea of the PAtt is to enable the remote attestation of a PLC's control logic (or the configuration data) without requiring changes to the PLC hardware or the manufacturer provided firmware. In other words, PAtt should be applicable to legacy systems by re-compiling/extending only the operator provided control logic programs of a PLC. We wrote 315 lines of code inside the designed function blocks to compute the hash and perform the attestation strategies. We used TIA 15 together with the Python-Snap7 libraries at the verifier side to communicate with the function blocks inside the PLC.

The attestation result, as soon as it is computed, will be encoded into the commands the PLC sends to the connected physical system as well as the verifier. The resulting sensor readings that depend on the attestation report are forwarded to the verifier. The verifier has a model of the physical system and can calculate which series of sensor readings to expect based on the actual functional commands, including the encoded attestation report.

Random Actuation Strategy. To implement the required random actuation strategy, we designed a set of function blocks to perform PAtt's micro-commands inside the function block within the bounded timing of the needed abstract macro-command by the process. As discussed before, the completion of the micro-commands will lead to a physical state as required by the macro-command. The random actuation strategy was written directly inside the function blocks, and it includes three function blocks and in total, 214 lines of codes inside the ladder logic. As only a limited number of micro-commands can be executed in each scan cycle, we are not able to run the complete actuation strategy based (with a number of steps determined by the length of our hash) within a single scan cycle. In particular, in our implementation, we generate eight micro-commands in one scan cycle. We now discuss how we addressed this implementation issue.

Splitting the Attestation over Multiple Protocol Rounds. Overall, we want to execute 256 micro-commands (to match the 256-bit output of our hash, SHA256). As we can only execute eight micro-commands in each scan cycle, we need to execute micro-commands over multiple scan cycles. Unfortunately, this increases the time required for the attestation, and potentially allows the attacker more time to compute precise simulations of executions (or communicate with a remote Oracle). To ensure that this is not possible for the attacker, we run the overall protocols in 32 rounds, with fresh nonces in each round. In each round, we execute eight micro-commands based on the most recent hash. Each round contributes 8 bits

of difficulty for the attacker to correctly guess the hash (or spoof sensor reading traces that are decoded to the expected hash). The Prover is attested by the Verifier only if all 32 rounds conclude successfully. We designed the encoding process to keep the PLC busy by the generation of hashes over the physical process and providing a new nonce in each scan cycle that convince the PLC to perform the random physical actuation over each scan cycle while the PLC computation power is devoted for the new hash generation.

5 Evaluation

In this section, we evaluate our proposal of the physics-based attestation of the control systems.

5.1 Dataset

Normal Operation. Our training dataset consists of 10 hashes and corresponding sensor reading traces, repeated six times for each hash (in total 60 runs). The traces are generated during normal operation (which captures the natural noise of the physical process) with a sampling rate of ten sensor readings for each micro-command. For our test evaluation dataset, we performed a number of attacks: a one-bit approximation attack, two-bit approximation attack, and simulation attacks (each attack was run 20 times). The overall size of sensor readings and the generated dataset was roughly 100 megabytes. We now describe the details of our attack implementation.

Hash Approximation Attacks. In this attack, an attacker modified the control logic of the PLC which resulted in a different hash (and thus a changed sequence of micro-commands). To show that PAtt can detect even minor changes in the hash/actuation sequence, we evaluated the performance of PAtt against an attacker that flips one or two random bits of the hash to modify the robot arm's trajectory. The attacker has full access to the stored hash and the actuation commands.

In particular, we performed ten one-bit approximation attacks and ten two-bit approximation attacks, and we evaluated the detection performance based on this attack. As the classifiers of our implementation PAtt are only trained on normal process behavior, we did not need to include the attack traces in the classifier training process. The implemented hash approximation attack was used as a test data-set, which consists of twenty hashes of sensor traces, and each hash repeated two times, and the normal operation was used as train data-set.

Replay Attacks. In addition to the effects of incorrect hashes or manipulated micro-command sequences, we also investigated whether the attacker would be able to produce sensor reading sequences by simulation (e.g., based on prior observations). For example, the attacker could try to record and replay earlier sensor reading traces, or try to re-assemble a new sensor reading trace from multiple earlier observations. We now argue why the former is infeasible and then describe

how we evaluated the latter. The attacker could generate a table of short sequences of micro-commands and corresponding sensor readings to simulate sensor reading traces of arbitrary hashes. Our intuition is that this is not feasible as the sensor readings are influenced by the trajectory position of the arm and the average move over a window of time. Considering the memory and computational resources available on the PLC, the attacker needs to regenerate the corresponding sensor reading that is influenced by the trajectory position of the arm and the average move over a window of time. Our results (presented next) confirm this. The implemented replay attack was used as a test data-set which consists of ten hashes with replayed sensor traces from the attack table, and each hash repeated two times, and the normal operation was used as train data-set.

5.2 Evaluation Results

To evaluate the performance of PAtt, we use the following metrics (see Section 10.2 in the Appendix): Sensitivity, Specificity, Precision, False Positive Rate (FPR), False Negative Rate (FNR), Accuracy, F1-score, and Matthews Correlation Coefficient (MCC).

Decoding. We now present our evaluation of different classifiers for decoding the sensor reading traces to the actuation strategy. The results in Table 2 summarizes the performance of our classifiers. The most promising classifiers for our data set were the Random Forest and Multi-Layer Perceptron (with accuracy 0.9923 and 0.9915, respectively). The tradeoff between false acceptance and false rejection can be seen in the ROC, provided in Appendix 10.3. Overall, it can be observed that our decoding is reliably able to translate the sensor reading traces back into the micro-commands.

Hash Authentication. For remaining analysis, we used the RF classifier for the decoding. The next step (the hash authentication) required an appropriate value for τ (the threshold for the weighted Hamming distance between the expected and decoded hash). We now show how we selected τ , based on analysis of the distributions of the hashes' weighted Hamming distances in normal and attack cases. We start with (more intuitively understandable) figures on the distributions of weighted Hamming distances in normal operations and during attacks. Figure 5 shows the weighted Hamming distance with the occurrence probability distribution of one-bit approximation attack, two-bit approximation attack, and replay attack, respectively. As shown in Figure 5a, the curve of the occurrence probability distribution of the normal operation overlap with the curve of the occurrence probability distribution of the one-bit approximation attack. That implies no value for τ will allow us to decide between the two cases without error correctly, and it means that there will be a non-negligible probability that the attacker can perform her attack while remaining undetected. In contrast, Figure 5b shows the curve of the occurrence probability distribution of the normal

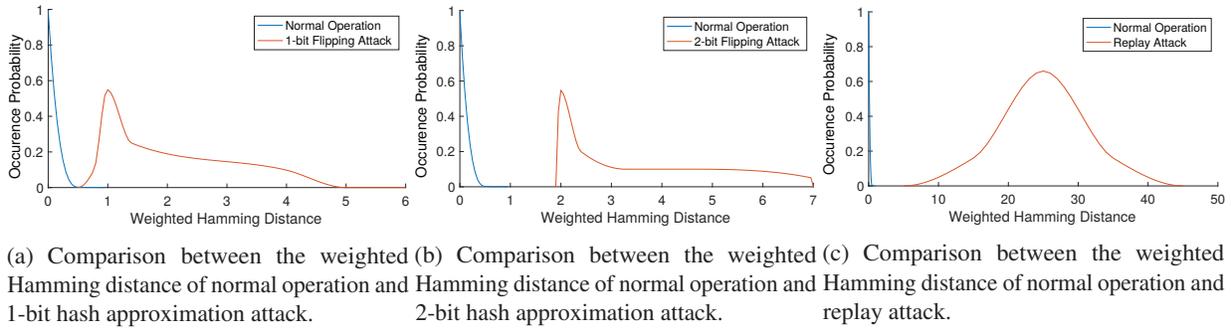


Figure 5: Weighted Hamming distance against the attack use-cases.

Table 2: Performance comparison of different decoding classifiers sorted by accuracy. FPR=False Positive Rate, FNR=False Negative Rate, MCC=Matthews Correlation Coefficient.

Algorithms	Sensitivity	Specificity	Precision	FPR	FNR	Accuracy	F1-score	MCC
Random Forest	0.9926	0.9921	0.9920	0.0079	0.0074	0.9923	0.9923	0.9847
Multilayer Perceptron	0.9915	0.9916	0.9915	0.0084	0.0085	0.9915	0.9915	0.9831
Decision Forest	0.9915	0.9895	0.9894	0.0105	0.0085	0.9905	0.9904	0.9810
FURIA	0.9878	0.9921	0.9920	0.0079	0.0122	0.9899	0.9899	0.9799
DTNB	0.9857	0.9910	0.9910	0.0090	0.0143	0.9884	0.9884	0.9767
NBTree	0.9873	0.9889	0.9889	0.0111	0.0127	0.9881	0.9881	0.9762
LMT	0.9873	0.9879	0.9878	0.0121	0.0127	0.9876	0.9875	0.9751
J48	0.9867	0.9868	0.9867	0.0132	0.0133	0.9868	0.9867	0.9735
PART	0.9893	0.9832	0.9830	0.0168	0.0107	0.9862	0.9862	0.9725
REPTree	0.9819	0.9810	0.9809	0.0190	0.0181	0.9815	0.9814	0.9630

operation does not overlap with the curve of the occurrence probability distribution of the two-bit approximation attack, which demonstrates that we can precisely detect the two-bit approximation attack (e.g., by choosing $\tau = 1$). Figure 5c shows the curve of the occurrence probability distribution of the normal operation did not overlap with the curve of the occurrence probability distribution of the replay attack, which means that we can also precisely detect replay attacks (e.g., by choosing $\tau = 1$).

In our experiments (see Table 3), first we studied the performance of PAtt with different values of the τ , from 0.8 to 1. We used the 60 traces of the normal operation and 60 traces of attacks in total. The true negative (TN) is the number of normal operation instances that are correctly classified as normal operation. The false positive (FP) is the number of normal operation instances that are wrongly classified as an attack. The true positive (TP) is the number of attack instances that are correctly classified as an attack. The false-negative (FN) is the number of attack instances that are wrongly classified as normal operation. We confirmed that $\tau = 1$ yields ideal performance in normal operations of the system. In the absence of attacks, our processing of the sensor reading traces always produces a hash that is classified as authentic. In addition,

we can detect the implemented attacks attack with Accuracy of 89%, sensitivity of 78%, and Matthews Correlation Coefficient (MCC) of 0.80. Both the two hash approximation attack and replay attack was detected without any false negative. Choosing the best value of the τ is dependent on the operational requirements of the control processes. However, if the operation of the system could tolerate the false positives (which would probably trigger the alarm even during the normal operation), we could choose the $\tau = 0.95$ which can detect the implemented attacks attack with Accuracy of 97%, the sensitivity of 96%, and MCC of 0.95.

6 Discussion

We now provide an additional discussion on the scalability of our approach, practical issues with critical zones, and the use-case scenarios.

6.1 Complexity/Scalability

The PAtt uses physical complexity (physical behavior) as a root-of-trust. By adding more sensors or actuators to the control processes, we could achieve a more robust model that

Table 3: Attack Detection performance comparison of implemented attacks (RF-based decoding). TN=True Negative, FP=False Positive, TP=True Positive, FN=False Negative, MCC=Matthews Correlation Coefficient.

Threshold	Normal		1-bit		2-bit		Replay		Sensitivity	Accuracy	F1-score	MCC
	TN	FP	TP	FN	TP	FN	TP	FN				
$\tau = 1$	60	0	7	13	20	0	20	0	0.7833	0.8917	0.8785	0.8024
$\tau = 0.95$	59	1	18	2	20	0	20	0	0.9667	0.9750	0.9748	0.9501
$\tau = 0.9$	58	2	18	2	20	0	20	0	0.9667	0.9667	0.9667	0.9333
$\tau = 0.8$	54	6	19	1	20	0	20	0	0.9833	0.9417	0.9440	0.8864

could verify the integrity of the control processes by wrapping this complexity over actuation strategy and the hash. In addition to the favor of complexity, the PAtt is designed to be scalable, and it could detect the change of physical complexity (physical behavior) as we have seen in the Section 5 by detecting the replay attacks which would report a table-based replay of physical behavior. These features of the PAtt would make it feasible to authenticate a physical process over an actuation strategy derived from a random hash.

6.2 Application to other PLCs

We used some APIs from the S7-1200 PLCs to perform the memory measurements and hash generation. The same functionality can be provided on other PLCs if they support the APIs. However, the memory measurement could be programmed directly in the control logic of the PLCs with some engineering effort.

6.3 Critical Zones

Given that this attestation routine is being integrated into safety-critical processes, there are restrictions on when the attestation process can be performed. We refer to these restrictions as critical zones. During a critical zone, the physical process is engaged in a fixed actuation and thus cannot be interrupted by an attestation. As an example, in 3D printing, a critical zone would be when the printer head is extruding filament. In addition to timing, the safety-critical zone must also include a spacial component, as actuation generated by the attestation process must not collide with anything and also stay within the range of motion of the system. A consequence of this is that processes that have no downtime or are always performing some critical action cannot utilize this augmented attestation method.

6.4 Example Applicable Use-Case Scenarios

In this section, we describe the application of our attestation scheme in several use-cases. We designed and implemented PAtt in a robotic arm controlled by a PLC, and we showed the applicability and security significance of the PAtt in a

real-world ICS. However, PAtt applies to other CPS as well, where the control process meets the following conditions:

- The control process has the ability to perform high-speed actuation (such as the 200 kHz PWM signal board that we used in the implementation of PAtt).
- The control process has powerful sensors that are able to report the current state of the physical process via a high-speed channel.

Automated Manufacturing. The first use case is automated manufacturing, which involves machinery similar to our implementation on a robotic arm. These types of setups are common in automotive assembly, where a robotic arm manipulates objects in 3D space. Actuation is carefully controlled and monitored by a PLC. In this situation, the critical zone occurs when the arm is manipulating an object. Conversely, when the arm is not gripping an object, the attestation process can be initiated. As described in the previous section, the conventional attestation report is encoded into micro-actuation of the robotic arm, resulting in sensor readings to corroborate the authenticity of the attestation report. In this case, the initiation of the attestation process must also take into account the spacial constraints given that the arm must not contact any objects during the attestation process.

Additive and Subtractive Manufacturing. Additive manufacturing processes, most commonly referring to 3D printing, consist of a printer head extruding heated filament in successive 2D layers to produce a product. The printing process is controlled by a micro-controller controlling several stepper motors. Designs are created in one of the various 3D printing programs and converted to a standard language of instructions known as G-code. In recent years, this technology has seen rapid growth leading not only to 3D printed parts used in a greater variety of applications, including safety-critical ones like medical prostheses. Consequently, it is essential to consider the security aspect of these applications, and much work has been done in this area already. The application of our attestation scheme to this process is complicated by the fact that conventional 3D printers lack the physical sensor channels leveraged in the robotic arm use-case, only being equipped with several limit switches for initial calibration,

then determining position relative to a "home point" through hard-coded characterization of the stepper motors controlling the motion of the printer head. However, this can easily be remedied through the placement of external sensors, like the accelerometers used in the robotic arm case, on to the printer head to provide a physical sensing channel to provide actuation feedback for the attestation procedure. In this use-case, the critical zone takes place when the printer head is extruding filament; its path is fixed. Thus the attestation procedure can take place between printing layers after the printer head is raised to satisfy the spacial aspect of the critical zone. A similar argument can be made for subtractive manufacturing applications, like laser engraving or CNC milling. Their setups are similar to 3D printing, with precise but relative motion control and an on/off-type process dictating their critical zones. Once again, the addition of cost-effective external sensors can allow the attestation of these processes.

7 Related Work

Remote Attestation. Remote attestation for embedded systems has been studied form many years, however, typically these approaches do not provide real-time execution guarantees for the system (at least while attestation is in progress) [10, 15, 30, 59]. The fundamental conflict between real-time execution and attestation has to be discussed by Carpent et al. [13]. Additionally, these approaches rely on a (hardware) root-of-trust, which is not available in today's CPS. Software-based attestation does not require a root-of-trust [28, 36, 37, 52–54], but can only provide uncertain security guarantees [55]. Valente et al. propose to leverage control-command variations to validate the correctness of a system's physical operations rather than the system software integrity, providing "a weak attestation" [61]. Our proposed scheme combines software-based attestation with control-command mutations to overcome the limitations of software-based attestation while providing guarantees regarding a device's software-integrity.

Machine Learning based Anomaly Detection. The authors of [42] discussed machine learning proposals for anomaly detection in the ICS. Also, the authors of [33] proposed to use convolutional neural networks for detecting cyber attacks on industrial control systems. Machine learning techniques for anomaly detection in industrial arm applications is discussed in [43]. Decision trees used in several security research areas like anomaly detection in network traffic data [8]. The authors of [44] used deep learning architecture in correlating Tor connections. The authors of [39] used the k-means clustering to detect traffic phase shifts inside the SCADA automatically. In [16], the authors discussed data mining and machine learning techniques for cybersecurity. There are many successful applications of machine learning in cyber-physical system security. PAtt uses the machine learning to learn the physi-

cal behavior alongside the other security measures like the integrity and authenticity checking of the control logic of the PLCs. Also, PAtt do not need to see the attack during the training phase. PAtt do not need to see the attack during the training phase, and it could detect the attack by a distance metric. Also, unlike most of machine learning anomaly detection techniques, PAtt is able to identify the replay attack.

Physical Unclonable Functions. The authors of [62] proposed to use the MEMS gyroscope as physical unclonable function, and they showed the feasibility of using the physical and electrical properties of the MEMS gyroscope for cryptographic key generation. The authors of [35] proposed an attested execution processor that do not need secure non-volatile memory, and it derives cryptographic identities from manufacturing variation measured by a PUF. Aging effects in PUFs have been discussed in the [38, 46], which were dominated by physical effects in the resistors. We evaluated the aging effect on industrial actuators and sensors used in our experimental evaluation in the Appendix 10.1, and show that on the scale of our sensors, no aging effects were observed.

8 Conclusion

In this paper, we presented PAtt, a novel remote attestation scheme for control processes of industrial control systems that integrates software-based attestation with physical behavior correlations, similar to a PUF. We used software-based attestation techniques to first generate a fresh hash over the loaded logic in the PLC, which is then translated into an actuation strategy for the physical process. The resulting sensor reading traces are then checked by the verifier to ensure that the correct logic is loaded and to detect spoofing of the process data. The proposed solution was able to detect the attack that is not seen before during the training phase of the verifier, and it could measure the anomalies by computing the distance over a cryptographic hash generated from the attestation of the control process. We implemented our solution (based on a robotic arm test-bed with Siemens PLC), and show that our proposed solution is accurate enough to detect the tested attacks with an accuracy of 97%, sensitivity of 96%, and Matthews Correlation Coefficient of 0.95 ($\tau = 0.95$).

9 Acknowledgment

The authors would like to thank the Singapore University of Technology and Design (SUTD), TU Darmstadt, DAAD, Rutgers University, CISPA- Helmholtz Center for Information Security, and UCLA for supporting this research by providing financial means and access to the laboratories. This work has been supported by the German Research Foundation (DFG) as part of projects HWSec, P3 and S2 within the CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Min-

istry for Higher Education, Research and the Arts (HMWK) within CRISP, by BMBF within the projects iBlockchain and CloudProtect, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS). Jianying Zhou's work is supported by the National Research Foundation (NRF), Prime Minister's Office, Singapore, under its National Cybersecurity R&D Programme (Award No. NRF2014NCR-NCR001-31) and administered by the National Cybersecurity R&D Directorate. This research is also funded in part by the National Science Foundation under awards CNS-1703782 and CNS-1705135. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, or the U.S. Government. We thank the National Science Foundation (NSF) - Cyber-Physical Systems (CPS) program - for their support of this project. Additionally, This material is partially based upon work supported by the Department of Energy's Office of Cybersecurity, Energy Security, and Emergency Response and the Department of Homeland Security's Security Science & Technology Directorate under Award Number DE-OE0000780.

References

- [1] open-dobot. <https://github.com/maxosprojects/open-dobot>. Accessed: 2018-12-06.
- [2] D. Antonioli, H. R. Ghaeini, S. Adepu, M. Ochoa, and N. O. Tippenhauer. Gamifying ics security training and research: Design, implementation, and results of s3. In *Proceedings of the 2017 Workshop on Cyber-Physical Systems Security and Privacy*, pages 93–102. ACM, 2017.
- [3] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13*, pages 1–12, New York, NY, USA, 2013. ACM.
- [4] S. Banescu, C. Collberg, and A. Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *Proceedings of the 26th USENIX Security Symposium*, 2017.
- [5] D. Barradas, N. Santos, and L. Rodrigues. Effective detection of multimedia protocol tunneling using machine learning. In *USENIX Security*, 2018.
- [6] Z. Basnight, J. Butts, J. Lopez, and T. Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76–84, 2013.
- [7] E. B. Baum. On the capabilities of multilayer perceptrons. *Journal of complexity*, 4(3):193–215, 1988.
- [8] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita. Network anomaly detection: methods, systems and tools. *IEEE communications surveys & tutorials*, 16(1):303–336, 2014.
- [9] W. Bolton. *Programmable logic controllers*. Newnes, 2015.
- [10] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *ACM DAC*, 2015.
- [11] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
- [12] E. Bressert. *SciPy and NumPy: an overview for developers*. " O'Reilly Media, Inc.", 2012.
- [13] X. Carpent, K. Eldefrawy, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. Reconciling remote attestation and safety-critical operation on simple iot devices. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.
- [14] E. Chien, L. OMurchu, and N. Falliere. W32.Duqu - The precursor to the next Stuxnet. Technical report, Symantic Security Response, nov 2011.
- [15] K. E. Defrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *NDSS*, 2012.
- [16] S. Dua and X. Du. *Data mining and machine learning in cybersecurity*. CRC press, 2016.
- [17] F-Secure Labs. BLACKENERGY and QUEDAGH: The convergence of crimeware and APT attacks, 2016.
- [18] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. Technical report, Symantic Security Response, Oct. 2010.
- [19] E. Frank and I. H. Witten. Generating accurate rule sets without global optimization. 1998.
- [20] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. Mohammed, and S. A. Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *Proceedings of the Network & Distributed System Security Symposium, San Diego, CA, USA*, pages 26–28, 2017.

- [21] H. R. Ghaeini, D. Antonioli, F. Brasser, A.-R. Sadeghi, and N. O. Tippenhauer. State-aware anomaly detection for industrial control systems. In *The 33rd ACM/SIGAPP Symposium On Applied Computing (SAC)*, Apr. 2018.
- [22] H. R. Ghaeini and N. O. Tippenhauer. Hamids: Hierarchical monitoring intrusion detection system for industrial control systems. In *Proceedings of the 2nd ACM Workshop on Cyber-Physical Systems Security and Privacy*, pages 103–111. ACM, 2016.
- [23] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.
- [24] M. A. Hall and E. Frank. Combining naive bayes and decision tables. In *FLAIRS conference*, volume 2118, pages 318–319, 2008.
- [25] T. K. Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.
- [26] J. Hühn and E. Hüllermeier. Furia: an algorithm for unordered fuzzy rule induction. *Data Mining and Knowledge Discovery*, 19(3):293–319, 2009.
- [27] S. Katzenbeisser, Ü. Kocabaş, V. Rožić, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann. PUFs: Myth, fact or busted? a security evaluation of physically unclonable functions (PUFs) cast in silicon. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 283–301, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [28] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *IEEE/IFIP DSN*, 2009.
- [29] E. D. Knapp and J. T. Langill. *Industrial Network Security: Securing critical infrastructure networks for smart grid, SCADA, and other Industrial Control Systems*. Synpress, 2014.
- [30] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *ACM EuroSys*, 2014.
- [31] R. Kohavi. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *Second International Conference on Knowledge Discovery and Data Mining*, pages 202–207, 1996.
- [32] J. Kong, F. Koushanfar, P. K. Pendyala, A.-R. Sadeghi, and C. Wachsmann. PUFatt: Embedded platform attestation based on novel processor-based PUFs. In *Proceedings of the 51st Annual Design Automation Conference, DAC '14*, pages 109:1–109:6, New York, NY, USA, 2014. ACM.
- [33] M. Kravchik and A. Shabtai. Detecting cyber attacks in industrial control systems using convolutional neural networks. In *Proceedings of the 2018 Workshop on Cyber-Physical Systems Security and Privacy*, pages 72–83. ACM, 2018.
- [34] N. Landwehr, M. Hall, and E. Frank. Logistic model trees. *Machine learning*, 59(1-2):161–205, 2005.
- [35] I. Lebedev, K. Hogan, and S. Devadas. Secure boot and remote attestation in the sanctum processor. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 46–60. IEEE, 2018.
- [36] Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-Based Attestation for Peripherals. In *TRUST*. Springer, 2010.
- [37] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the Integrity of PERipherals Firmware. In *ACM CCS*, 2011.
- [38] C. Q. Liu, Y. Cao, and C. H. Chang. Acro-puf: A low-power, reliable and aging-resilient current starved inverter-based ring oscillator physical unclonable function. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(12):3138–3149, 2017.
- [39] C. Markman, A. Wool, and A. A. Cardenas. Temporal phase shifts in scada networks. In *Proceedings of the 2018 Workshop on Cyber-Physical Systems Security and Privacy*, pages 84–89. ACM, 2018.
- [40] S. McLaughlin, C. Konstantinou, X. Wang, L. Davi, A.-R. Sadeghi, M. Maniatakos, and R. Karri. The cybersecurity landscape in industrial control systems. *Proceedings of the IEEE*, 104(5):1039–1057, 2016.
- [41] S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel. A trusted safety verifier for process controller code. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2014.
- [42] A. Meshram and C. Haas. Anomaly detection in industrial networks using machine learning: a roadmap. In *Machine Learning for Cyber Physical Systems*, pages 65–72. Springer, 2017.
- [43] V. Narayanan and R. B. Bobba. Learning based anomaly detection for industrial arm applications. In *Proceedings of the 2018 Workshop on Cyber-Physical Systems Security and Privacy*, pages 13–23. ACM, 2018.

- [44] M. Nasr, A. Bahramali, and A. Houmansadr. Deepcorr: Strong flow correlation attacks on tor using deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1962–1976. ACM, 2018.
- [45] X. Pan, Y. Cao, X. Du, G. Fang, R. Shao, and Y. Chen. Flowcog: Context-aware semantics extraction and analysis of information flow leaks in android apps. In *USENIX Security*, 2017.
- [46] M. T. Rahman, F. Rahman, D. Forte, and M. Tehranipoor. An aging-resistant ro-puf for reliable key generation. *IEEE Transactions on Emerging Topics in Computing*, 4(3):335–348, 2016.
- [47] J. Rrushi, H. Farhangi, C. Howey, K. Carmichael, and J. Dabell. A quantitative evaluation of the target selection of havex ics malware plugin. In *Industrial Control System Security (ICSS) Workshop*, 2015.
- [48] S. Ruggieri. Efficient c4. 5 [classification algorithm]. *IEEE transactions on knowledge and data engineering*, 14(2):438–444, 2002.
- [49] A.-R. Sadeghi and C. Stübke. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *Proceedings of the 2004 Workshop on New Security Paradigms, NSPW '04*, pages 67–77, New York, NY, USA, 2004. ACM.
- [50] A.-R. Sadeghi, I. Visconti, and C. Wachsmann. Enhancing rfid security and privacy by physically unclonable functions. In *Towards Hardware-Intrinsic Security*, pages 281–305. Springer, 2010.
- [51] S. Schüppen, D. Teubert, P. Herrmann, U. Meyer, and S. Sch. Fanci: feature-based automated nxdomain classification and intelligence. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 1165–1181. USENIX Association, 2018.
- [52] A. Seshadri, M. Luk, A. Perrig, L. V. Doorn, and P. Khosla. SCUBA: Secure Code Update By Attestation in Sensor Networks. In *ACM WiSec*, 2006.
- [53] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Using FIRE & ICE for Detecting and Recovering Compromised Nodes in Sensor Networks. Technical report, DTIC Document, 2004.
- [54] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *ACM SIGOPS OSR*, 2005.
- [55] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *USENIX Security*, May 2004.
- [56] A. K. Sikder, H. Aksu, and A. S. Uluagac. 6thsense: A context-aware sensor-based attack detector for smart devices. In *USENIX Security*, 2017.
- [57] R. P. Stanley. *Enumerative Combinatorics*, volume 1. Cambridge University Press, second edition, 2012. pp. 64-67.
- [58] K. Stouffer, J. Falco, and K. Scarfone. Guide to industrial control systems (ICS) security. *NIST special publication*, 800(82):16–16, 2011.
- [59] R. Strackx, F. Piessens, and B. Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *SecureComm*. Springer, 2010.
- [60] D. I. Urbina, J. A. Giraldo, A. A. Cardenas, N. O. Tippenhauer, J. Valente, M. Faisal, J. Ruths, R. Candell, and H. 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. ACM, 2016.
- [61] J. Valente, C. Barreto, and A. A. Cárdenas. Cyber-physical systems attestation. In *2014 IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 354–357. IEEE, 2014.
- [62] O. Willers, C. Huth, J. Guajardo, and H. Seidel. Mems gyroscopes as physical unclonable functions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 591–602. ACM, 2016.
- [63] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

10 Appendices

10.1 The Aging Effect on Classification Performance

The aging effect was reported in many PUF applications at the semiconductor level [38, 46]. However, such an effect is not common in mechanical actuators (the preciseness guarantee of the actuators could be found in the actuators data-sheet). We evaluated the aging effect by considering two data-sets of 6 months ago (old data-set) and a recent data-set (recent data-set). As we could see in Table 4 the tuned classifier would provide better classification results. We would recommend performing the tuning of the classifier by including the recent normal traces during the idle time of the CPS. However, the performance of the classifier that is built by training the old data-set is sufficient for the robot arm use-case that we evaluated in this paper. Figure 6 shows the true positive rate

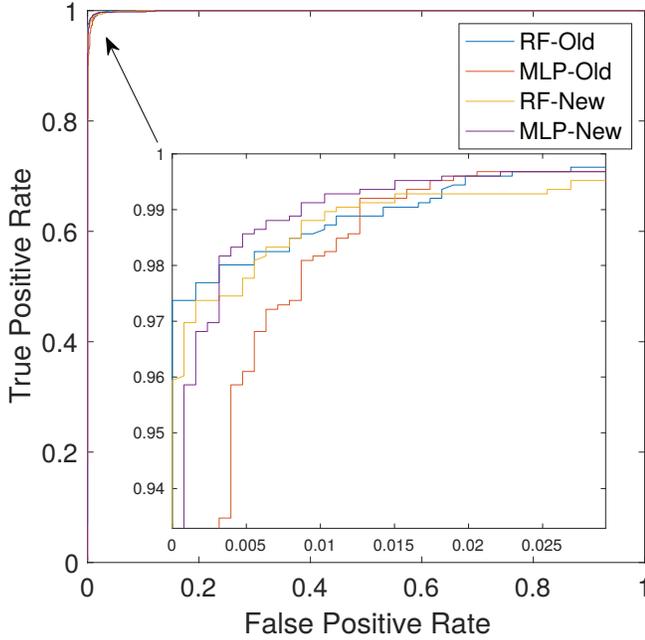


Figure 6: ROC curve of true positive rate (sensitivity) against false positive rate (1-precision), considering the aging effect.

(sensitivity) against the false positive rate (1-precision) in aging effect evaluation. As we could see in Figure 6, the classification performance of the classifiers with the two data-set of old and recent sensor traces are close to each other.

10.2 Performance Metrics

To evaluate the performance of the proposed method, we used eight performance metrics. The true positive (TP) is the number of retrieved relevant instances. The false positive (FP) is the number of retrieved nonrelevant instances. The true negative (TN) is the number of not retrieved nonrelevant instances. The false negative (FN) is the number of not retrieved relevant instances. The Sensitivity rate (Recall, eq. 6) presents the rate of retrieved relevant instances (TP) in overall relevant instances (TP + FN). The Precision rate (specificity, eq. 7) demonstrate the fraction of relevant instances (TP) in overall retrieved instances (TP + FP).

$$\text{Sensitivity rate} = \frac{TP}{TP + FN} \quad (6)$$

$$\text{Precision rate} = \frac{TP}{TP + FP} \quad (7)$$

The false positive rate (eq. 8) is the rate of retrieved non-relevant instances (FP) in overall nonrelevant instances (FP + TN). The false negative rate (eq. 9) is the rate of not retrieved

relevant instances (FN) in overall relevant instances (FN + TP). The false discovery rate (eq. 10) is the rate of retrieved nonrelevant instances (FP) in overall retrieved instances (FP + TP).

$$\text{False positive rate} = \frac{FP}{FP + TN} \quad (8)$$

$$\text{False negative rate} = \frac{FN}{FN + TP} \quad (9)$$

$$\text{False discovery rate} = \frac{FP}{FP + TP} \quad (10)$$

The accuracy (eq. 11) is the rate of retrieved relevant instances and not retrieved nonrelevant instances (TP + TN) in overall instances (TP + TN + FP + FN).

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (11)$$

The F1-score (eq. 12) is a metric for the test's accuracy. The F1-score (also F-score or F-measure) is defined as follows:

$$F1 - \text{score} = \frac{2 \times \text{Sensitivity} \times \text{Precision}}{\text{Sensitivity} + \text{Precision}} \quad (12)$$

The Matthews correlation coefficient (MCC) is a metric for the quality of two-class classification. The MCC metric is one of the most interesting metrics in anomaly detection where the physical feature will be classified to normal and abnormal classes. The MCC is defined as follows:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}}$$

10.3 Decoding ROC

Our ROC (true positive rate (sensitivity) against the false positive rate (1-precision)) for the decoding classifiers is presented in Figure 7.

10.4 An Example of Path Strategy

Figure 8 represents the path strategy of 10100010, and the path strategy of 00000111. We know that the number of unique paths u in a $x \times y$ grid can be computed as follows [57]:

$$u = \frac{(x+y)!}{(x!y!)} \quad (13)$$

Thus, we can enumerate all possible paths, and use an integer between 1 and u as an index to represent a specific path strategy in a $x \times y$ grid.

Table 4: Performance comparison of different classifiers with different metrics of Sensitivity & Specificity & Precision & FPR & FNR & Accuracy & F1-score & MCC (Matthews Correlation Coefficient), classifiers: Random Forest (RF) & Multilayer Perceptron (MLP).

Algorithms	Sensitivity	Specificity	Precision	FPR	FNR	Accuracy	F1-score	MCC
RF (old)	0.9857	0.9913	0.9912	0.0087	0.0143	0.9885	0.9884	0.9770
MLP (old)	0.9952	0.9826	0.9827	0.0174	0.0048	0.9889	0.9889	0.9779
RF (recent)	0.9912	0.9897	0.9897	0.0103	0.0088	0.9905	0.9905	0.9810
MLP (recent)	0.9881	0.9905	0.9904	0.0095	0.0119	0.9893	0.9892	0.9786

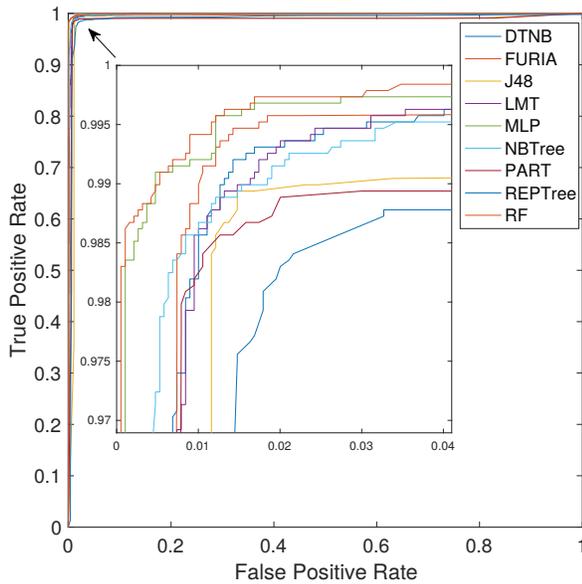


Figure 7: ROC curve of true positive rate (sensitivity) against false positive rate (1-precision).

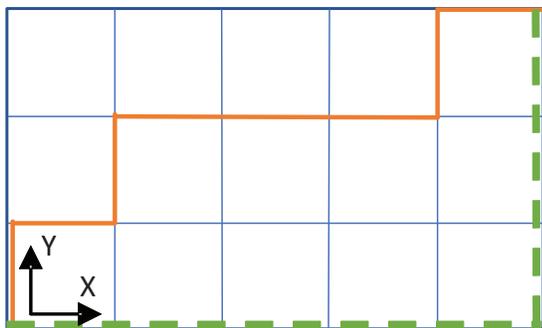


Figure 8: Two examples of path strategies: red=10100010, dashed green=00000111.

COMA: Communication and Obfuscation Management Architecture

Kimia Zamiri Azar*, Farnoud Farahmand*, Hadi Mardani Kamali*, Shervin Roshanisefat*,
Houman Homayoun[‡], William Diehl[†], Kris Gaj*, Avesta Sasan*

* *Department of ECE, George Mason University, VA, USA.*
{kzamiria, ffarahma, hmardani, sroshani, kgaj, asasan}@gmu.edu

[‡] *Department of ECE, University of California, Davis, CA, USA.* {hhomayoun}@ucdavis.edu

[†] *Department of ECE, Virginia Tech, VA, USA.* {wdiehl}@vt.edu

Abstract

In this paper, we introduce a novel *Communication and Obfuscation Management Architecture* (COMA) to handle the storage of the obfuscation key and to secure the communication to/from untrusted yet obfuscated circuits. COMA addresses three challenges related to the obfuscated circuits: First, it removes the need for the storage of the *obfuscation unlock key* at the untrusted chip. Second, it implements a mechanism by which the key sent for unlocking an obfuscated circuit changes after each activation (even for the same device), transforming the key into a dynamically changing license. Third, it protects the communication to/from the COMA protected device and additionally introduces two novel mechanisms for the exchange of data to/from COMA protected architectures: (1) a highly secure but slow double encryption, which is used for exchange of key and sensitive data (2) a high-performance and low-energy yet leaky encryption, secured by means of frequent key renewal. We demonstrate that compared to state-of-the-art key management architectures, COMA reduces the area overhead by 14%, while allowing additional features including unique chip authentication, enabling activation as a service (for IoT devices), reducing the side channel threats on key management architecture, and providing two new means of secure communication to/from an untrusted chip.

1 Introduction

The increasing cost of IC manufacturing has pushed several stages of the semiconductor device's manufacturing supply chain to the third-party facilities, which are identified as untrusted entities [4]. Fabrication of ICs in an untrusted supply chain has introduced multiple forms of security threats such as the possibility of overproduction, Trojan insertion, Reverse Engineering (RE), Intellectual Property (IP) theft, and counterfeiting [33, 34]. The stage that poses the utmost vulnerability is the fabrication stage, in which an untrusted foundry has the ultimate knowledge about a to-be-fabricated IC, and with minimal effort could reverse engineer the *GDSII* to its gate-level netlist, analyze, copy, and/or alter the design, creating

trust and security challenges for the original design house.

Considering that a foundry has the ultimate knowledge about the design, passive protection techniques such as watermarking, IC metering, or camouflaging [1, 28, 43, 52] are not well suited to protect against attacks initiated at this stage of supply chain, although they can be used to either identify counterfeits, or prevent reverse engineering of the manufactured ICs post fabrication. To protect the IP from being reverse engineered, overproduced, or stolen in the manufacturing supply chain, researchers have studied various means of hardware obfuscation [17, 18, 20, 21, 28, 35–37, 47, 53], which is the process of hiding the true functionality of an IC when no key, or an incorrect key, is present. Only once the correct key is provided, the IC behaves correctly. The requirement for obfuscated solutions is to resist various forms of attacks against such circuits including brute force, sensitization, Boolean satisfiability (SAT) or satisfiability modulo theories (SMT), removal, approximate-based, signal probability skew, functional analysis, etc. [11, 26, 29, 30, 38, 40, 41, 46].

To remain hidden, in addition to resisting the attacks against its obfuscated circuit(s), the IC should also resist passive, active, or destructive attacks that could be deployed to read the key values. Hence, neither the activation of such devices nor the storage of key values in them should expose or leak the key information. Activation of an obfuscated IC requires storing the activation key in a secure and tamper-proof memory. [25, 42]. However, there exist a group of applications that could use an alternative key storage. This alternative solution is to store the key outside the IC, where the IC is activated every time it is needed. This option requires constant connectivity to the key management source and a secure communication for key exchange to prevent any leakage of the key. This solution allows an IC designer to store the chip unlock key outside of an untrusted chip. So, no secure and tamper-proof memory is needed. Since the key is stored outside the untrusted chip, a constant connectivity to an obfuscation key-management solution is an indispensable requirement for this category of devices. This requirement could be easily met for two prevalent groups of architectures: (1) 2.5D package-stack

devices where a single trusted chip is used for key management and activation of multiple obfuscated ICs manufactured in untrusted foundries, and (2) IoT devices with constant connectivity to the cloud/internet.

In 2.5D package-integrated ICs, similar to DARPA SPADE architecture [10], a chip which is fabricated in a trusted foundry, but in a larger technology node, is packaged with an untrusted chip fabricated in an untrusted foundry in a smaller technology node. The resulting solution benefits from the best features of both technologies: The untrusted chip may be used as an accelerator, providing the resulting hybrid solution with the much-needed scalability (higher speed and lower power), while the trusted chip provides the means of trust and security. The untrusted chip is isolated from the outside world and any exchange of information to/from untrusted chip passes through the trusted chip.

The second group of devices in this category are IoT devices, where constant connectivity is their characterizing features. In these solutions the obfuscation key could be stored in the cloud, and activation of an IoT device could be done remotely. This model allows custom, monitored, and service oriented activation (Activation As A Service). An additional advantage is removing the possibility of extracting an unlock key from a non-volatile memory that otherwise would have to be used for storing the obfuscation unlock key. Examples of which are IoT devices used for providing various services, military drones activated for a specific mission, video decryption services for paid pay-per-view transactions, etc., where a device has to operate in an unsafe environment and is at risk of capture and reverse engineering. In these applications, the IC fabricated in an untrusted foundry is activated either every time it is powered up, or for certain time intervals. The key vanishes after the service is performed, or when the device is powered down. The activation of such devices is performed using a remote key management service (in the cloud or at a trusted base-station), and the key exchange to/from these devices should be secured.

In both 2.5D system solutions and IoT devices, the need for implementation of a tamper-proof memory, for storage of IC activation key, in an untrusted process is removed. Some reasons why implementing a secure memory in an untrusted foundry may be undesired, or practically unfeasible include:

Availability: The targeted foundry may not offer the required process for implementing a secure memory with the desired features. An example could be the requirement for storing sensitive information in magnetic tunnel junction (MTJ) memories to prevent probing and attacks that could be deployed against flash-based NVMs. Fabricating such ICs requires a hybrid process that supports both CMOS and MTJ devices, which may be unsupported by the targeted foundry.

Verified Security: The secure memory may be available in the targeted technology, however not be fully tested and verified in terms of its resistance against different attacks.

Cost: Implementing secure memory requires additional

fabrication layers and processing steps, increasing the cost of manufacturing. Increasing the silicon area is a far cheaper solution than increasing the number of fabrication layers.

Reusability: In 2.5D system solutions, a trusted chip could be shared by multiple untrusted chips, manufactured in different foundries. Moving the secure memory to the trusted chip removes the need for implementing the secure memory in all utilized processes. The trusted chip could be designed once with utmost security for protection and integrity of data. This also reduce the cost of manufacturing untrusted chips by removing the need for additional processing steps for implementing secure memory.

Ease of Design: Implementing secure memory requires pushing the design through non-standard physical design flow to implement the tamper-proof layers in silicon and package. In addition, the non-volatile nature of tamper-proof memory requires read and write at elevated voltages, increasing the burden on the power-delivery network design. Reuse of a trusted chip with a tamper proof memory that could manage activation of other obfuscated ICs, relaxes the design requirement of ICs to standard physical design and fabrication process.

In this paper, we propose the COMA key-management and communication architecture for secure activation of obfuscated circuits that are manufactured in untrusted foundries and meet the constant connectivity requirement, namely ICs that belong to a) 2.5 package-integrated and b) IoT solutions. We describe two variants of our proposed solutions: The first variant of COMA is used for secure activation of IPs within 2.5D package-integrated devices (similar to DARPA SPADE). The second variant of COMA is used for secure activation of connected IoT devices. The proposed COMA allows us to (1) push the obfuscation key and obfuscation unlock mechanism off of an untrusted chip, (2) make the key a moving target by changing it for each unlock attempt, (3) uniquely identify each IC, (4) remove the need to implementing a secure memory in an untrusted foundry, and (5) utilize two novel mechanisms for ultra-secure or ultra-fast encrypted communication.

The rest of this paper is organized as follows: Section 2 presents the background and related work to secure key-exchange and obfuscation schemes. Section 3 demonstrates how the proposed method has significant advantages in terms of security and performance. Both variants of the proposed architecture are evaluated in this Section. The security of the proposed architecture against various attacks is discussed in Section 4. The experimental results, as well as comparison with prior-art methods, is presented in Sections 5 and 6. Finally, Section 7 concludes the paper.

2 Background

Active metering, Secure Split-Test, logic obfuscation, and solutions such as Ending Piracy of Integrated Circuits (EPIC) have been proposed to protect ICs from supply chain-related security threats by initializing the HW control to a locked state

at power-up and hiding the design intent [2, 15, 16, 23, 27, 28, 44, 52]. Some of these techniques support single activation, while others support active metering mechanisms. Active metering techniques [15, 27, 44, 52] provide a mechanism for the IP owner to lock or unlock the IC remotely. In these solutions, the locking mechanism is a function of a unique ID generated for each IC, possibly and preferably by a *Physical Unclonable Function* (PUF) [42]. Only the IP owner knows the transition table and can unlock the IC. Active metering, combined with a PUF, makes the key a moving target from chip to chip. However, there exist a few issues with previous metering techniques: first, the key(s) to unlock each IC remains static. Second, these techniques unlock the chips before they are tested by the foundry. Hence, the IP owner can control how many ICs enter the supply chain, but not how many properly tested ICs exit the supply chain. Finally, these techniques do not respond well to the threat of the foundry requesting more IDs by falsifying the yield to be lower during the test process. Such shortcomings can potentially allow the foundry to ship more out-of-spec or defective ICs to the supply chain.

Many of these shortcoming were addressed in FORTIS [49] shown in Fig. 1. In FORTIS the registers that hold the obfuscation key are made a part of the scan chain, allowing the foundry to carry structural test by assigning test values to these registers prior to the activation of the IC. Authors of [49] argue that placing a DFT compression logic, not only reduces the test size, but also prevents the readout of the individual register values. After testing the IC, the obfuscation key is transferred and applied to unlock the circuit using two types of cryptographic modules: a public-key crypto engine, and a One Time Pad (OTP) crypto engine.

In FORTIS, the public and private keys are hardwired in the design. A TRNG is used to generate a random number (m) that is treated as a message. This message is encrypted using the private key of the chip to generate a signature $sig(m)$. The actual message and its signature are concatenated and later used as a mean for the authentication of the chip. At the same time, the TRNG generates another random number K_S . This random number is used as the key for OTP, and at the same time is encrypted using the public key of the designer to generate $KD_{pub}(K_S)$. OTP uses K_S for encrypting the $(m, sig(m))$, and the output of OTP is concatenated with the $KD_{pub}(K_S)$. The resulting string of bits is transmitted to the SoC designer. The SoC designer uses a OTP to obtain m and $sig(m)$ for the purpose of authentication. She then uses the private key of the designer to recover K_S . Finally, K_S is used by OTP to encrypt the chip unlock key (CUK). The encrypted CUK is transmitted to the chip, decrypted using OTP, and applied to the obfuscation unlock key registers to unlock the circuit.

FORTIS, however, suffers from several security issues including 1) using identical public and private keys in all manufactured chips, and thus its inability for unique device authentication, 2) being vulnerable to modeling attack in which the

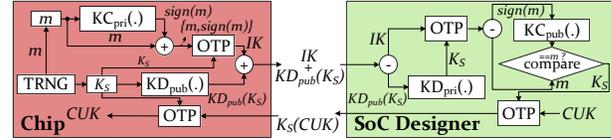


Figure 1: FORTIS: Overall Architecture.

FORTIS structure is modeled in software for requesting the CUK from SoC designer 3) being vulnerable to side channel attacks on public-key encryption engine aimed at recovering the private key of the chip, 4) being vulnerable to fault attacks in which the value of K_S is fixated, 5) requiring a secure memory for storage of the obfuscation unlock key, and 6) not addressing the mechanism for generating a unique and truly random seed to initialize PRNG. After describing our proposed solution, in section 6, we explain how these vulnerabilities are addressed in our proposed solution.

Our proposed solution fits the category of active metering techniques. The key is neither static nor stored in the untrusted chip. A key that is used to activate the IC at the test time cannot be reused to activate the same or a different IC in the future. Hence, the test facility is able to accomplish the test process using ATPG tools with a key which is valid for structural/functional test and it is not valid for any subsequent activation. Additionally, the communication to/from IC is secured using a side-channel protected cryptographic engine, combined with a dynamic switching and inversion structure that enhances the security of the chip against invasive and side-channel attacks. We demonstrate that COMA provides two useful means of secure communication to/from the untrusted chip, one for added security, and one for supporting a higher throughput. The proposed architecture is a comprehensive solution for the key management of the obfuscated IPs, where the challenges related to the activation of the IC and secure communication to/from the IC are addressed at the same time. However, as discussed earlier, it is not a universal solution and would fit within the context of IoT-based solutions or within 2.5D package-integrated solutions, as this solution requires constant connectivity.

3 Proposed COMA Architecture

The primary goal of the COMA is to remove the need for storing the *obfuscation key* (OK) on an untrusted chip while securing the communication flow used for activation of the obfuscated circuit in the untrusted chip. The additional benefits of the proposed architecture are the implementation of two new modes of 1) highly secure and 2) very high-speed encrypted communication. We propose two variants of the COMA architecture: The first variant is designed for securing the activation of the obfuscated IP and communication to/from an untrusted IC in 2.5D package-integrated architec-

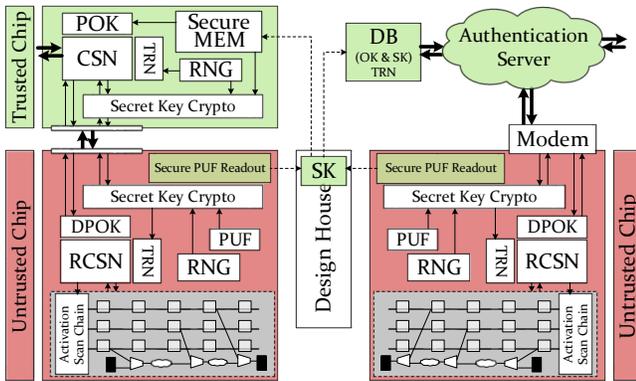


Figure 2: Proposed COMAs for (left) 2.5D and (right) IoT-based/remote devices.

tures similar to the DARPA SPADE architecture [10] (denoted by 2.5D-COMA). The second proposed architecture is designed for protecting IoT-based or remotely activated/metered devices (denoted by R-COMA). Fig. 2 captures the overall architecture of two variants of the proposed COMAs.

3.1 2.5D-COMA: Protecting 2.5D package integrated system solutions

The DARPA SPADE project [10] explores solutions in which an overall system is split-manufactured between two different technologies. In this solution, a trusted IC which is constructed in an older yet secure technology is packaged with an IC fabricated in an untrusted foundry in an advanced geometry. The purpose of this solution is to provide the best of two worlds: the security of older yet trusted technology and the scalability, power, and speed of the newer yet untrusted technology. The 2.5D-COMA is designed to work with an architecture similar to the DARPA SPADE architecture. The proposed solution allows an entire or partial IP in an untrusted chip to be obfuscated, while pushing the mechanism for unlocking and secure activation of the untrusted chip out to a trusted chip. In this solution, the trusted chip encapsulates the sensitive information, verifies the integrity of the untrusted chip, performs sensitive logic monitoring, and controls the activation of the untrusted chip. Also, the key to unlock the obfuscated circuit changes per activation, details of which will be explained shortly.

As shown in Fig. 2, the two variants of COMA contain two main parts, the trusted side (green) and the untrusted side (red). In both variants, the architectures of untrusted chips are identical, and only the architectures of trusted sides are different. In 2.5D-COMA, only the trusted chip is equipped with a secure memory. The secure memory stores the *Obfuscation Key* (OK) and the Secret Key (SK) used for encrypted communication between the trusted and untrusted chips. The SK is generated using a PUF in the untrusted chip, thus it is unique for each untrusted chip, and the untrusted chip does not need a secure memory to store the SK. The *Configurable*

Switching Network (CSN) and *Reverse CSN* (RCSN) are logarithmic routing and switching networks. They are capable of permuting the order and possibly inverting the logic levels of their primary inputs while these signals are being routed to different primary outputs. The RCSN is the exact inverse of the CSN. Hence, passing a signal through CSN-RCSN (or RCSN-CSN) will recover the original input. The switching and inversion behavior of CSN-RCSN is configured using a *True Random Number* (TRN). This TRN is generated in the trusted chip to avoid any potential weakening/manipulating of the TRNG. In addition, since the TRNG in COMA is equipped with standard-statistical-tests applied post-fabrication, such as Repetition-Count test and the Adaptive-Proportion test, as described in NIST SP 800-90B [12], any attempt at weakening the TRNG during regular operation (i.e. fault attack) can be detected by continuously checking the output of a source of entropy for any signs of a significant decrease in entropy, noise source failure, and hardware failure. By using TRN for the CSN-RCSN configuration, any signal passing through the CSN is randomized, and then by passing through the RCSN is recovered. Additional details are provided in section 3.3.1.

The untrusted chip unlock process in COMA is as follows: Prior to each activation, the CSN and RCSN are configured with the same TRN. Since the SK is a PUF-based key generated at the untrusted side, first the SK must be securely readout from untrusted chip. This is done by deploying public key cryptography, the details of which are described in section 3.3.4. Then, the trusted chip encrypts the TRN using the SK and sends it to the untrusted chip. To perform an activation, as shown in Fig. 2, the OK is read in segments, denoted as *Partial Obfuscation Key* (POK), and is passed through the CSN and encryption on the trusted side and the decryption and RCSN on the untrusted side. This process is repeated every time the obfuscated circuit in the untrusted chip is to be activated, each time using a different TRN for configuring the CSN-RCSN. Usage of a different TRN as the configuration input for the CSN-RCSN for each activation randomizes the input data to Secret key crypto engine. Hence, by using a different TRN for each activation, the obfuscation key (after passing through CSN) is transformed into a one-time license, denoted as *Dynamic Activation License* (DAL). Since the OK is read and sent in segments (from trusted chip), the DAL will be received (at untrusted chip) in segments, denoted as *Dynamic Partial Obfuscation Key* (DPOK), shown in Fig. 2, and is used as an input to RCSN. Passing DPOKs through RCSN recovers the POKs, and concatenating the POKs will generate the OK. Note that the DAL is only valid until the TRN is changed. So, the DAL cannot be used to activate other chips or the same chip at a later time.

In 2.5D-COMA, the untrusted chip(s) is used as an accelerator, and for safety reasons should not be able to directly communicate to the outside world. Hence, all communication to/from the untrusted chip must go through the trusted chip. In addition, it is possible that the computation, depend-

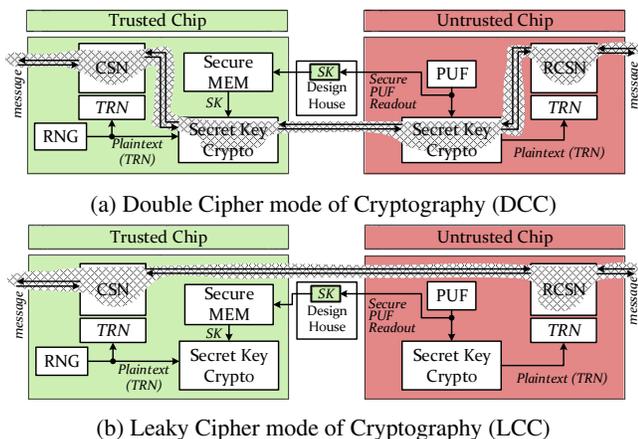


Figure 3: Modes of Encrypted Communication in COMA

ing on the sensitivity of processed data, is divided between the trusted and untrusted chips. Hence, there is a need for constant communication between the trusted and untrusted chips. The communication needed is sometimes for limited but highly sensitive data, and sometimes for vast amounts of less sensitive data. As illustrated in Fig. 3, the proposed architecture is designed to provide two hybrid means of encrypted communication : (1) Double-Cipher Communication (DCC) as ultra-secure communication, and (2) Leaky-Cipher Communication (LCC) as ultra-fast communication mechanism.

3.1.1 Double-Cipher Communication (DCC)

As shown in Fig. 3(a), DCC is implemented by passing each message through both CSN-RCSN and the secret key cryptography engine, where the TRN used in CSN-RCSN is renewed every U cycles. DCC provides the ultimate protection against side-channel attacks. In DCC mode, two necessary requirements for mounting a side channel attack are eliminated. The side channel attack aims to break the cryptography system by analyzing the leaked side channel information for different *input patterns*. Hence, (1) the degree of correlation between the input and the leaked side-channel information, and (2) the intensity of side-channel variation, are important. In COMA, the attacker cannot control the input to the secret-key cryptography. In addition, the input to the CSN is randomized using a TRN and then passed to the secret-key cryptography, removing the correlation between leaked side channel info (from secret-key cryptography) and the original input to the CSN. Additionally, the secret-key cryptography engine is side-channel protected to pass a t-test [6]. So, the intensity and variation in side-channel information is significantly reduced, making the DCC an extremely difficult attack target.

3.1.2 Leaky-Cipher Communication (LCC)

LCC is a fast and energy efficient mode of communication between the trusted (or remote device) and the untrusted chip. As illustrated in Fig. 3(b), in this protocol, the CSN-RCSN

pair is used for exchanging data. The secret key cryptography engine is used to transmit a TRN from one chip to the other. Since the throughput of TRNG is the bottleneck point compared to the performance of CSN-RCSN, the TRNG is used as a seed generator to the PRNG (which offers higher performance) on both sides. Hence, in LCC mode, PRNG is used to configure the CSN-RCSN to avoid any performance degradation on transmitting data. For U consecutive cycles, the PRNG is kept idle allowing the CSN to use the same PRNG output for U cycles. It not only reduces the power consumption of PRNG and TRNG, it also provides faster communication in LCC mode. However, using this model of communication is prone to algebraic and SAT attacks as each communicated message leaks some information about the TRN used to configure the CSN-RCSN pair. If an attacker can control the message and observe the output of the CSN, each communicated message leaks some information about the key, reducing its security. Extracting the key from such observations is possible by various attack models, including Satisfiability attacks. Hence, an attacker with enough time and enough traces could extract the TRN and retrieve the communicated messages. Preventing such attacks poses a minimum limit to U (the update frequency of the PRNG). U should be small to prevent SAT and other trace-based learning or analysis attacks, but large enough to be energy efficient. In Section 5, we deploy a SAT attack against LCC and will further elaborate on the required TRN update frequency.

3.2 R-COMA: for Protecting IoT devices

The R-COMA architecture in the untrusted chip is identical to that of 2.5D-COMA. However, the trusted chip is replaced with a remote key management service. The R-COMA provides a mechanism for an IP owner to remotely activate parts or entire functionality of the hardware. Similar to 2.5D-COMA, the DAL is different from chip to chip and from activation to activation. In R-COMA, the obfuscation unlock key is stored in a central database, while the CSN, the TRNG for configuring CSN-RCSN, and the secret key cryptography engine are implemented in software.

In R-COMA, an authentication server (AS) first securely receives the PUF-based SK from the untrusted chip. Then, it generates a TRN and sends it to the untrusted chip for RCSN configuration. Then, the AS starts sending the obfuscation key (OK). For the activation phase, the communication is double encrypted and authenticated using the CSN-RCSN and side-channel protected cryptography engine. Each COMA-protected device needs to be registered with the AS to receive the obfuscation key. The registration is done using the PUF-ID of the untrusted chip. Hence, the PUF is used for both authentication and generation of the secret key for communication. In R-COMA, the generation of DAL is granted after PUF authentication, and is based on the generated TRN, and the stored OK, which is generated at design time. The generation of DAL is algorithmic and takes linear time.

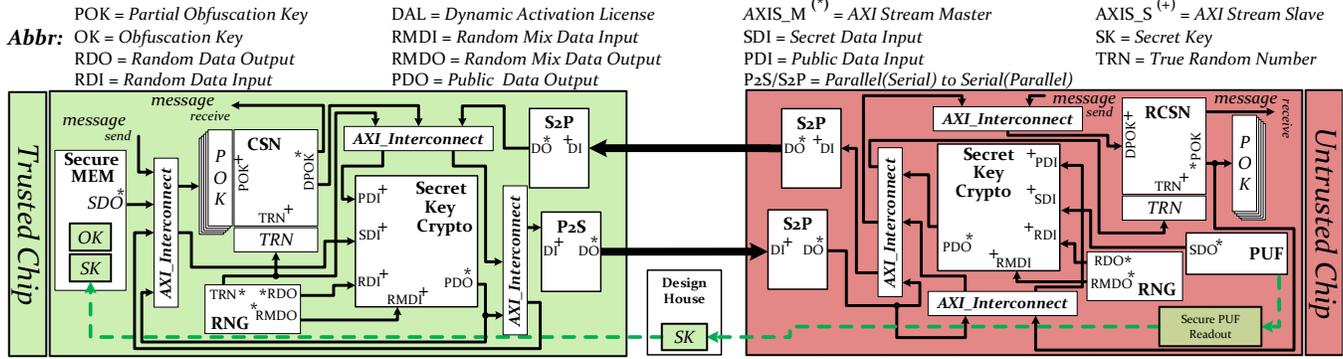


Figure 4: 2.5D-COMA Architecture.

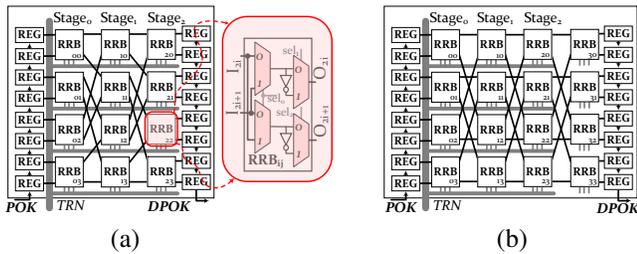


Figure 5: Logarithmic Network (a) Omega-based Blocking, (b) $LOG_{8,1,1}$ near Non-blocking.

3.3 Implementation Detail of COMA

Fig. 4 captures the overall architecture of COMA and relation and connectivity of its macros. As discussed, COMA supports both key-management and secure data communication. Based on the selected mode of communication (LCC/DCC), the message passes through {CSN \rightarrow RCSN} or {CSN \rightarrow encryption \rightarrow decryption \rightarrow RCSN}. RNG, which contains both TRNG and PRNG, is used in both sides. In the trusted chip, RNG is used for implementing side-channel protected cryptography engine, as well as generating the configuration of the CSN-RCSN (TRN). In the untrusted side, it is used only for implementing the side-channel protected cryptography engine. Finally, PUF is engaged in the untrusted chip for both unique IC authentication and for generation of the secret key for encryption. As shown in Fig. 4, all modules employ an AXI-stream interface to maximize the simplicity of the overall design, and minimize the overhead incurred by the controller of the top module in each side. The description of the behavior of each macro in COMA is provided next:

3.3.1 Configurable Switching Network (CSN)

The CSN is a logarithmic routing network that could route the signals at its input pins to its output pins while permuting their order and possibly inverting their logic levels based on its configuration. Fig. 5(a) captures a simple implementation of an 8-by-8 CSN using *OMEGA* [19] network. The network is constructed using routing elements, denoted as Re-Routing

Blocks (RRB). Each RRB is able to possibly invert and route each of the input signals to each of its outputs. The number of RRBs needed to implement this simple CSN for N inputs (N is a power of 2) is simply $N/2 * \log N$. Each CSN should be paired with an RCSN. The RCSN, is simply constructed by flipping the input/output pins of RRB, and treating the CSN input pins as its output pins and vice versa.

The *OMEGA* network along with many other networks of such nature (Butterfly, etc.) are blocking networks [19], in which we cannot produce all permutations of input at the network's output pins. This limitation significantly reduces the ability of a CSN to randomize its input. Also, we will show that a blocking CSN can be easily broken by a SAT attack within few iterations.

Being a blocking or a non-blocking CSN depends on the number of stages in CSN. Since no two paths in an RRB are allowed to use the same link to form a connection, for a specific number of RRB columns, only a limited number of permutations is feasible. However, adding extra stages could transform a blocking CSN into a strictly non-blocking CSN. Using a strictly non-blocking CSN not only improves the randomization of propagated messages through the CSN, but also improves the resiliency of these networks against possible SAT attacks for extraction of a TRN used as the key for a CSN-RCSN cipher. A non-blocking logarithmic network could be represented using $LOG_{n,m,p}$, where n is the number of inlets/outlets, m is the number of extra stages, and p indicates the number of copies vertically cascaded [9].

According to [9], to have a strictly non-blocking CSN for an arbitrary n , the smallest feasible values of p and m impose very large area/power overhead. For instance, for $n = 64$, the smallest feasible values, which make it strictly non-blocking, are $m = 3$ and $p = 6$, which means there exists more than $5 \times$ as much overhead compared to a blocking CSN with the same n , resulting in a significant increase in the area and delay overhead. To avoid such large overhead, we employ a *close to non-blocking CSN* described in [9] to implement the CSN-RCSN pair. This network is able to generate not all, but *almost all* permutations, while it could be implemented using a $LOG_{n, \log_2(n)-2, 1}$ configuration, meaning it needs $\log_2(n) - 2$

extra stages and no additional copy. Fig. 5(b), demonstrates an example of such a close-to-non-blocking CSN with $n = 8$. In the results section, we demonstrate that using these close-to-non-blocking CSNs enhances the resiliency of a CSN against SAT attack, even in small sizes of CSNs with significantly lower power, performance and area (PPA) overhead.

3.3.2 Authenticated Encryption with Associated Data

The Authenticated Encryption with Associated Data (AEAD) is used in the DCC mode for communicating messages, and in the LCC mode for the initial transmission of the CSN-RCSN key (TRN). Authenticated ciphers incorporate the functionality of confidentiality, integrity, and authentication. The input of an authenticated cipher includes Message, Associated Data (AD), Public Message Number (NPUB), and a secret key. The ciphertext is generated as a function of these inputs. A Tag, which depends on all inputs, is generated after message encryption to assure the integrity and authenticity of the transaction. This tag is then verified after the decryption process. The choice of AEAD could significantly affect the area overhead of the solution, the speed of encrypted communication, and the extra power consumption. To show the performance, power, and area trade-offs, we employ two AEAD solutions: a NIST compliant solution (AES-GCM), and a promising lightweight solution (ACORN).

AES-GCM is the current National Institute of Standards and Technology (NIST) standard for authenticated encryption and decryption as defined in [32]. ACORN is one of two finalists of the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR), in the category of lightweight authenticated ciphers, as defined in [22]. An 8-bit side-channel protected version of AES-GCM and a 1-bit side-channel protected version of ACORN are implemented as described in [51]. Both implementations comply with lightweight version of the CAESAR HW API [13].

Our methodology for side channel resistant is threshold implementation (TI), which has wide acceptance as a provably secure Differential Power Analysis (DPA) countermeasure [45]. In TI, sensitive data is separated into shares and the computations are performed on these shares independently. TI must satisfy three properties: 1) Non-completeness: Each share must lack at least one piece of sensitive data, 2) Correctness: The final recombination of the result must be correct, and 3) Uniformity: An output distribution should match the input distribution. To ensure uniformity, we refresh TI shares after non-linear transformations using randomness. We use a hybrid 2-share/3-share approach, where all linear transformations in each cipher are protected using two shares, which are expanded to three shares only for non-linear transformations.

To verify the resistance against DPA, we employ the Test Vector Leakage Assessment methodology in [6]. We leverage a "fixed versus random" non-specific t-test, in which we randomly interleave first fixed test vectors and then randomly-

generated test vectors, leading to two sequences with the same length but different values. Using means and variances of power consumption for our fixed and random sequences, we compute a figure of merit t . If $|t| > 4.5$, we reason that we can distinguish between the two populations and that our design is leaking information. The protected AES-GCM design has a 5-stage pipeline and encrypts one 128-bit input block in 205 cycles. This requires 40 bits of randomness per cycle. In ACORN-1, there are ten 1-bit TI-protected AND-gate modules, which consume a total of 20 random refresh, and 10 random refresh bits per state update. In a two-cycle architecture, 15 random bits are required per clock cycle.

3.3.3 Random Number Generator (RNG)

An RNG unit is required on both sides to generate random bits for side channel protection of AEAD units, a random public message number (NPUB) for AEAD, and TRNs for CSN-RCSN. We adopted the ERO TRNG core described in [39], which is capable of generating only 1-bit of random data per over 20,000 clock cycles. In our TI implementations, AES-GCM needs 40 and ACORN 15 bits of random data per cycle. So, we employed a hybrid RNG unit combining the ERO TRNG with a Pseudo Random Number Generator (PRNG). TRNG output is used as a 128-bit seed to PRNG. The PRNG generates random numbers needed by other components. The reseeding is performed only once per activation.

The choice of PRNG depends on the expected performance and overhead. To support COMA, we adopted two different implementations of PRNG: (1) AES-CTR PRNG, which is based on AES, is compliant with the NIST standard SP 800-90A, and generates 12.8 bits per cycle. (2) Trivium based PRNG, which is based on the Trivium stream cipher described in [7]. The Trivium-based PRNG is significantly smaller in terms of area and much faster than AES-CTR PRNG. It can generate 64 bits of random data per cycle, however, it is not compliant with the NIST standard.

3.3.4 PUF and Secure PUF Readout

The response of the PUF to a challenge selected randomly by Enrollment Authority (SoC designer) is used as the secret key in AEAD. Hence, the readout of the PUF-response should be protected. The simplest solution for the safe readout of a PUF-generated key is to enable the readout by burning one set of fuses, and disabling it by burning a second set of fuses. However, this solution, especially when combined with a weak PUF, is not likely to be resistant against the untrusted foundry, which may possibly burn the first set of fuses, read out PUF key, and then repair fuses before releasing the chip. To avoid this problem, we implement a lightweight one-sided public key cryptography (encryption only) based on Elliptic-Curve Cryptography (ECC). Considering the PUF readout is a one-time event, the performance of the public-key cryptography engine is not critical.

In order to prevent any attempts at fully characterizing a PUF in the untrusted foundry, only strong PUFs, e.g. an arbiter PUF, are considered. The secure readout of the PUF key is allowed only at the device enrollment time, in the secure facility. During the secure readout, the strong PUF is fed with multiple challenges selected by the Enrollment Authority. The corresponding PUF responses are encrypted by the untrusted chip using the public key of the Enrollment Authority, that is embedded in the chip layout or stored in the one-time programmable memory. Only the Enrollment Authority has access to the decrypted responses. Afterwards, one of the previously applied challenges is randomly selected and used for the generation of the secret key. This challenge is then hardwired on the untrusted chip, and the PUF response to that challenge is recorded by the Enrollment Authority. This PUF response is then stored in the secure memory of the trusted chip in 2.5D-COMA, or in the secure cloud directory in R-COMA. This process makes each PUF key unique to a given device, and resistant against any unauthorized readout by the untrusted foundry.

Still, additional precautions must be taken to protect this scheme against an attack aimed at replacing a real PUF by a pseudo-PUF, generating randomly looking responses that can be easily calculated by an attacker. An example of such a pseudo-PUF may be a lightweight symmetric-key cipher, with a fixed key known to the untrusted party, encrypting each challenge and outputting a ciphertext as the PUF response.

Such pseudo-PUF should be treated as a Trojan and detected by Enrollment Authority using the best known anti-Trojan techniques, e.g., those based on the measurement and analysis of the power consumption during the operation of the device [8]. Additional methods may be used to differentiate the outputs of a strong PUF from encrypted data, e.g., using known correlations between the PUF responses corresponding to closely-related challenges, such as challenges differing on only one bit position, or being mutual complements of each other [24]. These kinds of PUF-health tests may be specific to a particular strong PUF type, e.g., to an arbiter PUF, and will be the subject of our future work.

4 COMA Resistance against various Attacks

4.1 Assumed Attacker Capabilities

Different sources of vulnerability are considered in this section to demonstrate the COMA security. The attacker can be an adversary in the manufacturing supply chain, and has access to either the reverse engineered or design house-generated netlist of the COMA-protected untrusted chip. The attacker can purchase an activated COMA-protected IC from the market. The attacker can monitor the side channel information of chips at or post activation. The attacker can observe the communication between untrusted and trusted (or remote manager) chips and could also alter the communicated data. An Attack objective may be (1) extracting the obfuscation

key (OK), (2) illegal activation of the obfuscated circuit without extracting the key, (3) extracting the long-term secret key (SK), (4) extracting short-term CSN keys (TRNs), (5) eavesdropping on messages exchanged between the untrusted chip and the external sources, (6) removing the COMA protection, or (7) COMA-protected IC overproduction.

4.2 Side Channel Attack (SCA)

The objective of SCA on COMA is to extract either the secret key (SK) used by AEAD or the TRN used by CSN. Extracting a SK is sufficient to break the obfuscation; extracting a TRN reveals only messages sent in the LCC mode.

DCC significantly increases the SCA difficulty, since (1) the AEAD is side-channel protected, and (2) the attacker loses access to the input of AEAD. Fig. 6 captures our assessment of side channel resistance of AEAD using a t-test for unprotected and protected implementations of AES-GCM and ACORN [50]. As illustrated, both implementations pass the t-test, indicating increased resistance against SCA. On the other hand, the inability to control the input to AEAD comes from the COMA requirement of encryption in the DCC mode where a message first passes through the CSN. Hence, there exists no relation between the power consumption of the AEAD and the original input due to CSN randomization. CSN power consumption is also randomized as it is a function of n inputs (possibly known to the attacker) and $3n \times (\log_2 n - 1)$ TRN inputs unknown to the attacker, while the TRN is repeatedly updated based on the value of U . Note that during the physical design of COMA, the side channel information on power and voltage noise (IR drop) could be further mitigated using timing aware IR analysis [3], and voltage noise aware clock distribution techniques [5, 31].

The LCC mode is prone to side-channel, algebraic, and SAT attacks aimed at extracting the TRN. However, the attack must be carried out in a limited time while the TRN of the CSN/RCSN is unchanged. As soon as the TRN is renewed, the previous side-channel traces or SAT iterations are useless. The period of TRN updates (U) introduces a trade-off between energy and security and can be pushed to maximum security by changing the TRN for every new input. In section 5.2.2 we investigate the time required to break the LCC using side-channel or SAT attack and accordingly define a safe range for U to prevent such attacks.

4.3 Reverse Engineering

In COMA, reverse engineering (RE) to extract the secret key from layout is useless as the secret key is not hardwired in the design and is generated based on PUF. RE to extract the key from memory in an untrusted chip is no longer an option as the key is not stored in the untrusted chip. RE to extract the key from the trusted chip's memory is limited by the difficulty of tampering with secure memory in the trusted technology.

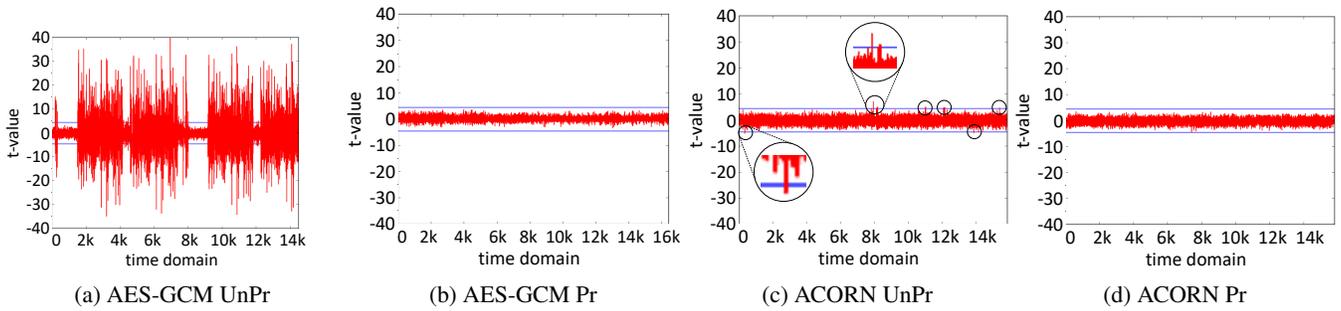


Figure 6: The t-test results for unprotected (UnPr) and protected (Pr) implementations of AES-GCM and ACORN.

4.4 Algebraic Attacks

Algebraic attacks involve (a) expressing the cipher operations as a system of equations, (b) substituting in known data for some variables, and (c) solving for the key. AES-GCM and ACORN have been demonstrated to be resistant against all known types of algebraic attacks, including linear cryptanalysis. Therefore, in the absence of any new attacks, the DCC mode is resistant against algebraic attacks. Using CSN and RCSN for fast encryption is new and requires more analysis. CSN can be expressed as an affine function of the data input x , of the form $y = A \cdot x + b$, where A is an $n \times n$ matrix and b is an $n \times 1$ vector, with all elements dependent on the input TRN. Although recovering A and b is not equivalent to finding the TRN, it may enable the successful decryption of all blocks encrypted using a given TRN. We protect against this threat in two ways: (1) The number of blocks encrypted using a given TRN is set to the value smaller than n , which prevents generating and solving a system of linear equations with A and b treated as unknowns, (2) We partially modify the TRN input of CSN with each block encryption (by a simply shifting the input TRN bits), so the values of A and b are not the same in any two encryptions, without the need of feeding CSN with two completely different TRN values.

4.5 Counterfeiting and Overproduction

COMA can be used to prevent the resale of used ICs, usage of illegal copies, and reproduction of a design. During packaging and testing, each COMA protected IC is first tested and then is matched with a trusted chip. So, the untrusted chip can only be activated by the matched trusted chip or the registered remote manager. Building illegal copies that work without the secure chip (or remote activation) and reproduction of the design requires successful RE. Blind reproduction is useless as its activation requires a matching trusted chip or passing PUF authentication of a remote manager. By receiving one or more DALs for testing, the manufacturer cannot activate additional IPs as the DAL changes from activation to activation.

4.6 Removal attacks

Removal of the TRNG fixates the DAL and breaks the LCC mode. In DCC mode, it gives an attacker control over the

input to the AEAD, increasing the chances of SCA on the cryptography engine. NIST standard SP 800-90B [12] dictates that continuous health testing must be performed on the TRNG. These tests include repetition counting to detect catastrophic failure and adaptive proportion testing to detect loss of entropy. Removal of the TRNG would be detected as this would result in insufficient entropy to satisfy the health test, assuming the test is implemented on the trusted chip. Removal of COMA architectural modules makes the chip non-functional as COMA is not a wrapper architecture, but a fused one. Complete removal of COMA requires successful RE. Removing the PUF can be made challenging by using a strong PUF, with a large number of challenge-response pairs. Replacing such a PUF with a deterministic function is challenging as such functions are likely to have a substantially different area and power, making them detectable.

Table 1: Main features of the two proposed COMA variants.

Feature	COMA1	COMA2
AEAD	AES-GCM	ACORN
PRNG	AES-CTR	Trivium
BUS Width	8	8
Pins used for Communication	8	8
CSN-RCSN Size	64	64
Trusted Memory	4 Kbits	4 Kbits
C_{init} : initialization overhead (cycles)	10,492	20,452
C_{byte} : cycles needed for encrypting each byte	72	17
PRNG _{perf} : Throughput of generating PRN	128bit/10cycles	64bit/cycle

5 COMA Implementation Results

For evaluation, all designs have been implemented in VHDL and synthesized for both FPGA and ASIC. For ASIC implementation we used Synopsys generic 32nm educational libraries. For FPGA verification, we targeted a small FPGA board, Digilent Nexys-4 DDR with Xilinx Artix-7 (XC7A100T-1CSG324).

5.1 COMA Area Overhead

We implemented two variants of COMA architecture: a NIST compliant solution (denoted by COMA1) and a lightweight solution (denoted by COMA2). The AEAD and PRNG in COMA1 is based on AES-GCM and AES-CTR respectively. The COMA2 is implemented by using ACORN for AEAD

Table 2: Resource Utilization of the COMA Architecture for NIST-compliant and lightweight solution.

Name	AES-GCM+AES-CTR			ACORN+Trivium		
	Slice	LUT	FF	Slice	LUT	FF
TRUSTED						
AEAD_EXT	1,336	3,804	4,432	333	1,067	591
RNG	712	2,226	618	215	601	450
CSN	257	540	739	257	540	739
Others	149	345	144	149	345	144
UNTRUSTED						
AEAD_EXT	1,336	3,804	4,432	333	1,067	591
RNG	738	2,352	628	241	683	460
RCSN	252	607	737	252	607	737
ECC	563	1569	1161	563	1569	1161
PUF [48]	177	—	—	177	—	—
Others	209	359	257	209	359	257

On Xilinx Artix-7 (XC7A100T-1CSG324) FPGA.

Table 3: Resource Utilization for ASIC implementation of NIST-compliant and lightweight COMA.

Name	AES-GCM+AES-CTR				ACORN+Trivium			
	Cells	Area _{um²}	Tclk _{ns}	Power _{mW}	Cells	Area _{um²}	Tclk _{ns}	Power _{mW}
COMA	25338	0.11	1.97	1.62	8681	0.046	1.18	0.84
▷ RNG	5684	0.025	1.43	0.431	1267	0.007	0.27	0.144
▷ CSN/RCSN	1749	0.008	0.08	0.11	1749	0.008	0.08	0.11
▷ AEAD	13675	0.061	1.67	0.704	2257	0.013	0.97	0.251
▷ ECC	3278	0.016	1.34	0.321	3278	0.016	1.34	0.321

Using Synopsys generic 32nm libraries.

and Trivium for PRNG, The details of these two variants are summarized in Table 1. The breakdown of area (in terms of Slices, LUTs, and FFs) for these solutions for an FPGA implementation in Xilinx Artix-7 (using Minerva [14]) is reported in Table 2. The breakdown of area (in terms of Cells and um^2), critical path, and power consumption for an ASIC implementation is reported in Table 3. Note that the 2.5D-COMA needs both the trusted and untrusted parts of the architecture, while the R-COMA only requires the untrusted part. Table 4 reports optimized area and frequency results on FPGA for top-level of trusted and untrusted sides. As illustrated, the total area of lightweight solution is around 1/3 of the NIST-compliant solution. The reported numbers in Table 2 include the overhead of all sub-modules including AEAD, CSN-RCSN, RNG, ECC, etc. Due to the optimization on the boundaries among the units, resource utilization in Tables 4 is less than the sum of row values in Table 2.

5.2 COMA Performance

Fig. 7 compares the performance of two solutions in DCC and LCC mode. As illustrated, for small data sizes, the COMA1 outperforms the COMA2 solution. However, as the size of data increases, the COMA2 outperforms the COMA1 solution. It is due to the fact that stream ciphers such as ACORN have a long initialization phase, making them inefficient for small data size. In addition, our AES-GCM implementation

Table 4: Optimized results of COMA Architecture for NIST-compliant and lightweight solution.

Name	AES-GCM+AES-CTR				ACORN+Trivium			
	Slice	LUT	FF	Freq[MHz]	Slice	LUT	FF	Freq[MHz]
Trusted	2,297	7,094	5,892	103	1,030	2,901	1,924	121
Untrusted	2,818	8,781	7,169	109	1451	4,182	3,156	120

On Xilinx Artix-7 (XC7A100T-1CSG324) FPGA.

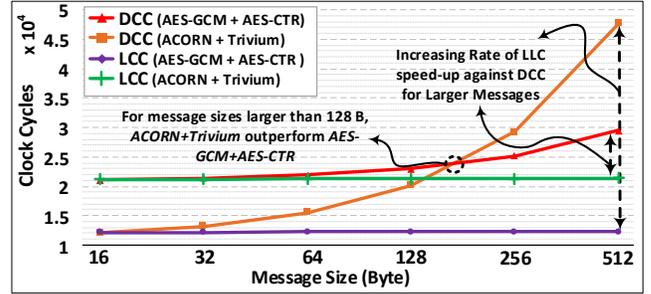


Figure 7: Total execution time in number of clock cycles for (AES-GCM + AES-CTR) and (ACORN + Trivium).

benefits from an 8-bit datapath, but the ACORN is realized by a 1-bit serial implementation. The total latency in terms of the number of clock cycles for COMA1 and COMA2 implementations can be calculated using equation (1), in which the number of cycles for the initialization and finalization is fixed and is given in Table 1. The C_{byte} is the number of cycles needed for encrypting each input message byte, which is 17 and 72 for COMA2 and COMA1, respectively. Hence, in spite of longer initialization, the COMA2 outperforms the COMA1 for message sizes larger than 128 Bytes.

$$T_{comm} = C_{fix} + Message_{size} \times C_{byte} \quad (1)$$

5.2.1 COMA performance in LCC mode

In the LCC mode, the AEAD is used to synchronize the initial seed of the PRNG, while the CSN is used for encrypting data. The random (TRN) configuration key for the CSN-RCSN is generated by PRNG, which is updated after transferring every U messages. In COMA, the PRNG has a limited buffer size, and as soon as the buffer is filled with random data, the PRNG stops producing additional bits. The consumption of TRNG output is synchronized (every U messages) and the generation of random inputs is limited to the size of buffer. Hence, the PRNGs in the trusted and untrusted sides are always in sync. The number of cycles it takes to initialize the LCC mode includes the time to initialize the secret key engine (C_{fix}), the encryption and transfer and decryption of PRNG seed (C_{ENC}), and the time for the PRNG to generate enough output from a newly received TRN (C_{PRNG}):

$$C_{LCC-init} = C_{fix} + C_{ENC} + C_{PRNG} \quad (2)$$

Depending on the AEAD used for transferring the original seed, the C_{fix} is obtained from Table 1. The seed size in our implementation is 16 Bytes, hence the C_{ENC} is simply $C_{bytes} \times 16$, and the C_{PRNG} is:

$$C_{PRNG} = \frac{Bit_{needed}}{PRNG_{perf}} = \frac{3n \times (\log_2 n - 1)}{PRNG_{perf}} \quad (3)$$

Finally, after initialization, and by using a CSN of size n when the bus width of COMA is BW , the number of cycles to encrypt and transfer one byte of information is:

$$C_{byte}^{LLC} = \frac{8}{n} \times \left(\frac{n}{BW} + 1 \right) \quad (4)$$

Using a 64-bit CSN and BW of 8 bits, the $C_{byte}^{LLC} = 9/8$. Compared to C_{byte}^{DCC} for the COMA1 ($C_{byte}^{DCC} = 72$), and for the COMA2 ($C_{byte}^{DCC} = 17$), the LCC mode is at least an order of magnitude faster. Fig. 7 compares the superior performance of LCC mode compare with DCC mode in both COMA variants.

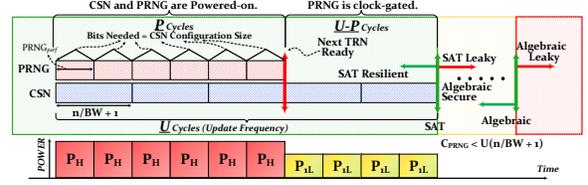
5.2.2 Frequency of TRN updates in LCC mode

The frequency of TRN update (U) for LCC is an important design feature. A large U reduces energy as PRNG/TRNG is kept idle for $U - P$ cycles. P is the number of required cycles to refill the PRNG buffer after a TRN read. However, when the TRN is fixated for a long duration of time, the possibility of a successful side-channel, algebraic, or SAT attack on the CSN increases. The minimum number of messages required for an algebraic attack (even if such attack is possible) is n , which is the CSN input size. Our experiments show that a SAT attack could recover the key with an even smaller number of inputs. Knowing the number of encryptions/decryptions needed by such attacks, we can set the U to a safe value smaller than the number of required messages to make it resistant against these attacks. So, the value of U should be between $P \leq U \leq n$.

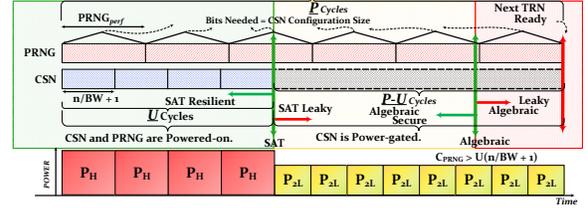
The SAT attack against CSN is implemented similar to [41]. In this attack the CSN gate-level netlist and an activated chip is available to the attacker, while the attacker aims to extract the CSN-RCSN configuration signals. Table 5 captures the results of the SAT attack against blocking and near non-blocking CSNs. As illustrated, the time to break a near non-blocking CSN is significantly larger. In each iteration SAT test one carefully selected input message. Hence, if the U is kept smaller than the number of required SAT iterations, the SAT attack could not be completed.

5.3 Energy saving in LCC mode

As illustrated in Fig. 8(a), in the LCC mode, the TRN is updated every U cycles. U is determined based on the fastest attack on CSN-RCSN pair, which is the SAT attack. After each TRN update, the PRNG takes P cycles to refill its buffer. Note that P cycles required for PRNG could be stacked at the beginning of U cycles, or distributed over U cycles depending



(a) While $P < U$. PRNG is kept idle (power-gated) after P cycles.



(b) While $U > P$. CSN/RCSN is kept idle after U cycles.

$P_H = \text{Power}_{CSN/RCSN} + \text{Power}_{PRNG}$; $P_{1L} = \text{Power}_{CSN}$; $P_{2L} = \text{power}_{PRNG}$;

Figure 8: The Power Consumption at LCC mode of operation.

on the size of PRNG buffer. As long as the TRN completely changes every U cycle, the possibility of attack is eliminated. Hence in each U cycles, for P cycles the PRNG/TRNG and CSN are active, and for $U - P$ cycles, the PRNG is clock gated, and only CSN is active. In both cases, the AEAD is active only for the initial exchange of PRNG seed, allowing us to express the power consumption of the LCC mode as:

$$E_{LLC} = C_{PRNG} \times P_H + \left(U \left(\frac{n}{BW} + 1 \right) - C_{PRNG} \right) \times P_L \quad (5)$$

Obviously, the number of required cycles to refill the PRNG buffer after TRN read (P) affects energy consumption and communication throughput. If $P < U$, as illustrated in Fig. 8(a), for $U - P$ cycles the PRNG is kept idle (power-gated). However, if $P > U$, as shown in Fig. 8(b), the communication should be stopped for $P - U$ cycles till the next TRN is ready and to resist SAT or algebraic attacks.

The energy consumption of LCC mode for COMA architectures constructed using NIST-compliant and lightweight solution when transmitting different size of messages is captured in Fig. 9. As illustrated, the LCC mode, for having to synchronize the two sides using a TRNG seed, is burdened with the initialization cost of AEAD. However, when the CSN-RCSN and PRNG are setup, the energy consumed for exchanging additional messages grow at a much lower rate compare to DCC mode (which is dominated by AEAD and PRNG power consumption as reported in table 3).

6 Comparing COMA with Prior Work

To the best of our knowledge, FORTIS [49] is the only comprehensive key-management scheme that was previously proposed. Table 6 compares our proposed solution against FORTIS. COMA addresses several shortcomings of the FORTIS:

- 1) In FORTIS, all chips use identical keys, hence there is no mean of differentiating between chips. In COMA each chip has a unique key generated by PUF.
- 2) In COMA, secret

Table 5: SAT Execution Time on *OMEGA*-based Blocking CSN and $LOG_{n,log_2(n)-2,1}$ as a Close to Non-blocking CSN .

CSN Size	4		8		16		32		64		128		256		512	
	blk	non-blk	blk	non-blk	blk	non-blk	blk	non-blk	blk	non-blk	blk	non-blk	blk	non-blk	blk	non-blk
SAT Iterations	6	14	7	18	8	25	12	31	14	TO	24	TO	25	TO	TO	TO
SAT Exe. Time (s)	0.01	0.01	0.03	0.15	0.2	2.35	0.8	79.18	5.9	TO	130.5	TO	1136.2	TO	TO	TO

TO: Timeout = 2×10^6 seconds; The SAT attack is carried on a Dell PowerEdge R620 equipped with Intel Xeon E5-2670 2.6 GHz and 64GB of RAM.

Table 6: COMA vs. FORTIS.

Scheme	Key Management	Data Comm	Private Key	SC Protected	Session Key	Activation	Need to TPM	RNG
FORTIS	Constant	\times^*	Embedded (known to the fab)	\times	Vulnerable to Fault Attack	Once	at Untrusted	PRNG
COMA	PUF-based Unique	\checkmark^+	No private key at untrusted	\checkmark	Secure	per Demand	at Trusted	TRNG

*: Not Implemented, but Naturally available using OTP. Limited Performance Due to Lightweight RSA

+ : Available in Two Variant: DCC (Fully Secure and Limited Performance) and LCC (Leaky yet Secure and High Performance).

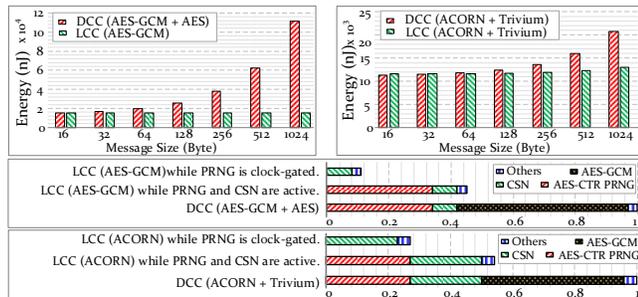


Figure 9: Energy Breakdown in COMA.

key for communication and authentication is generated by PUF, when FORTIS relies on embedding the private key and public key in GDSII. So, the private key in FORTIS will be known to the fabrication posing the risk that the entire process of activation could be faked in software. In COMA, such attack is prevented as secret key is generated by PUF and is securely read out using public key cryptography. 3) In FORTIS, the usage of the private key for chip authentication is vulnerable to SCA. In COMA, the secret-key cryptography is side channel protected, and the public-key encryption is only used once, making COMA secure against SCA. 4) In FORTIS, there is also the possibility of deploying a fault attack by fixing the value of session key K_s . In COMA, the same attack would require fixing the PUF output or replacing the PUF with a known function. This however could be tested by reading out the output of the PUF using multiple challenges and performing statistical test on the PUF response (PUF health check). 5) In FORTIS, the activation is done once, hence there is a need to store the obfuscation key in the untrusted chip. In COMA, the need to store the obfuscation key in untrusted chip is removed. In R-COMA, the activation takes place on demand, and the key is removed after power down or reset. In 2.5D-COMA, the activation key is stored in a trusted chip. 6) COMA provides two new mechanisms for communication: a) the DCC mode for added security, and b) the LCC mode for high-speed communication. 7) COMA uses a TRNG to produce the seed for PRNG, while FORTIS uses a PRNG without addressing a random source for its seed, increasing its vulnerability.

Table 7: Area Overhead of COMA vs. FORTIS.

Design	Gate Count	FORTIS/Design	COMA1/Design	COMA2/Design
b19	40,789	24.52%	62.1%	21.28%
VGA_LCD	43,346	23.07%	58.45%	20.02%
Leon3MP	253,050	3.95%	10.01%	3.43%
SPARC	836,865	1.19%	3.02%	1.03%
Virtex-7	2M	0.5%	1.26%	0.43%

In terms of area overhead, FORTIS [49] provides an estimate for the incurred overhead of their solution, which is around 10K gates. As shown in Table 3, the numbers of cells for implementing the NIST-compliant (COMA1) implementation is 25.4K gates, while the lightweight solution (COMA2) is implemented using 8.7K gates. Table 7 compares the area overhead of FORTIS against COMA1 and COMA2, when these architectures are deployed to protect a few mid- and large-size benchmarks. Using COMA2, which improves the overhead by 14% compared to FORTIS, requires between 0.43% and 21.3% of circuit area in selected benchmarks.

Acknowledgement

This research is funded by the Defense Advanced Research Projects Agency (DARPA #FA8650-18-1-7819) of the USA, and partly by Silicon Research Co. (SRC TaskID 2772.001) and National Science Foundation (NSF Award# 1718434).

7 Conclusion

In this paper we presented COMA, an architecture for obfuscation-key management and metered activation of an obfuscated IC that is manufactured in an untrusted foundry, while securing its communication. The proposed solution removes the need to store the key in the untrusted chip, makes the obfuscation unlock-key a moving target, allows unique identification of the protected IC, and secures the communication to/from the protected chip using two hybrid cryptographic schemes for ultra-high-speed and ultra-security. Our experimental results show that compared to the state-of-the-art key management architecture, FORTIS, COMA is able to reduce the area overhead by 14%, while addressing many of the shortcomings of the previous work.

References

- [1] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe. Watermarking Techniques for Intellectual Property Protection. In *Proceedings of the Annual Design Automation Conference (DAC)*, pages 776–781, 1998.
- [2] A. Baumgarten, A. Tyagi, and J. Zambreno. Preventing IC Piracy using Reconfigurable Logic Barriers. *IEEE Design & Test of Computers*, 27(1):66–75, 2010.
- [3] A. Vakil, H. Homayoun, and A. Sasan. IR-ATA: IR annotated timing analysis, a flow for closing the loop between PDN design, IR analysis & timing closure. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 152–159, 2019.
- [4] A. Yeh. Trends in the Global IC Design Service Market. *DIGITIMES research*, 2012.
- [5] B. Gunna, L. Bhamidipati, H. Homayoun, and A. Sasan. Spatial and Temporal Scheduling of Clock Arrival Times for IR Hot-Spot Mitigation, Eeformulation of Peak Current Reduction. In *IEEE/ACM Int’l Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2017.
- [6] B. J. Gilbert Goodwill, J. Jaffe, and P. Rohatgi. A Testing Methodology for Side-Channel Resistance Validation. In *NIST Non-Invasive Attack Testing Workshop*, volume 7, pages 115–136, 2011.
- [7] C. De Canniere and P. Bart. Trivium Specifications. In *eSTREAM, ECRYPT Stream Cipher Project*, 2005.
- [8] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan Detection using IC Fingerprinting. In *IEEE Symposium on Security and Privacy (SP)*, pages 296–310, 2007.
- [9] D.-J. Shyy and C.-T. Lea. Log/sub 2/(N, m, p) Strictly Nonblocking Networks. *IEEE Transactions on Communications*, 39(10):1502–1510, 1991.
- [10] D. S. Green. Leveraging the Commercial Sector and Providing Differentiation through Functional Disaggregation, 2013. https://www.darpa.mil/attachments/DisaggregatetheCircuit_Slides.pdf.
- [11] D. Sirone and P. Subramanyan. Functional Analysis Attacks on Logic Locking. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 936–939, 2019.
- [12] E. Barker and J. Kelsey. Recommendation for the Entropy Sources used for Random Bit Generation. *Draft NIST Special Publication*, pages 800–900, 2012.
- [13] E. Homsirikamol, W. Diehl, A. Ferozपुरi, F. Farahmand, P. Yalla, J.-P. Kaps, and K. Gaj. CAESAR Hardware API. *Cryptology ePrint Archive, Report 2016/626*, page 669, 2016.
- [14] F. Farahmand, A. Ferozपुरi, W. Diehl, and K. Gaj. Minerva: Automated Hardware Optimization Tool. In *Int’l Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2017.
- [15] F. Koushanfar. Provably Secure Active IC Metering Techniques for Piracy Avoidance and Digital Rights Management. *IEEE Transactions on Information Forensics and Security*, 7(1):51–63, 2012.
- [16] G. Contreras, Md. T. Rahman, and M. Tehranipoor. Secure Split-Test for Preventing IC Piracy by Untrusted Foundry and Assembly. In *IEEE Int’l Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 196–203, 2013.
- [17] G. Kolhe, H. M. Kamali, M. Naicker, T. D. Sheaves, H. Mahmoodi, S. M. PD, H. Homayoun, S. Rafatirad, and A. Sasan. Security and Complexity Analysis of LUT-based Obfuscation: From Blueprint to Reality. In *IEEE/ACM Int’l Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.
- [18] G. Kolhe, S. M. PD, S. Rafatirad, H. Mahmoodi, A. Sasan, and H. Homayoun. On custom lut-based obfuscation. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*, pages 477–482, 2019.
- [19] H. Ahmadi and W. E. Denzel. A Survey of Modern High-Performance Switching Techniques. *IEEE Journal on Selected Areas in Communications*, 7(7):1091–1103, 1989.
- [20] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan. Full-Lock: Hard Distributions of SAT Instances for Obfuscating Circuits Using Fully Configurable Logic and Routing Blocks. In *Proceedings of the Annual Design Automation Conference (DAC)*, pages 89:1–89:6, 2019.
- [21] H. M. Kamali, K. Z. Azar, K. Gaj, H. Homayoun, and A. Sasan. LUT-Lock: A Novel LUT-Based Logic Obfuscation for FPGA-Bitstream and ASIC-Hardware Protection. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 405–410, 2018.
- [22] H. Wu. ACORN: A Lightweight Authenticated Cipher (v3). *Candidate for the CAESAR Competition*, 2016. <https://competitions.cr.yt.to/round3/acornv3.pdf>.
- [23] J. B. Wendt and M. Potkonjak. Hardware Obfuscation using PUF-based Logic. In *IEEE/ACM Int’l Conference on Computer-Aided Design (ICCAD)*, pages 270–271, 2014.

- [24] J. Delvaux and I. Verbauwhede. Attacking PUF-based Pattern Matching Key Generators via Helper Data Manipulation. In *Cryptographers' Track at the RSA Conference*, pages 106–131, 2014.
- [25] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls. Physical Unclonable Functions and Public-Key Crypto for FPGA IP Protection. In *Int'l Conference on Field Programmable Logic and Applications (FPL)*, pages 189–195, 2007.
- [26] J. Rajendran, M. Sam, O. Sinanoglu, and R. Karri. Security Analysis of Integrated Circuit Camouflaging. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 709–720, 2013.
- [27] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri. Security Analysis of Logic Obfuscation. In *Proceedings of the Annual Design Automation Conference (DAC)*, pages 83–89, 2012.
- [28] J. Roy, F. Koushanfar, and I. L. Markov. Ending Piracy of Integrated Circuits. *Computer*, 43(10):30–38, 2010.
- [29] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan. SMT Attack: Next Generation Attack on Obfuscated Circuits with Capabilities and Performance Beyond the SAT Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pages 97–122, 2019.
- [30] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan. Threats on Logic Locking: A Decade Later. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*, pages 471–476, 2019.
- [31] L. Bhamidipati, B. Gunna, H. Homayoun, and A. Sasan. A Power Delivery Network and Cell Placement Aware IR-Drop Mitigation Technique: Harvesting Unused Timing Slacks to Schedule Useful Skews. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 272–277, 2017.
- [32] M. J. Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical report, 2007. NIST Technical Report.
- [33] M. M. Tehranipoor, U. Guin, and S. Bhunia. Invasion of the Hardware Snatchers. *IEEE Spectrum*, 54(5):36–41, 2017.
- [34] M. Rostami, F. Koushanfar, and R. Karri. A Primer on Hardware Security: Models, Methods, and Metrics. *Proceedings of the IEEE*, 102(8):1283–1295, 2014.
- [35] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu. Provably-Secure Logic Locking: From Theory to Practice. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1601–1618, 2017.
- [36] M. Yasin, B. Mazumdar, J. Rajendran, and O. Sinanoglu. SARLock: SAT Attack Resistant Logic Locking. In *Int'l Symposium on Hardware Oriented Security and Trust (HOST)*, pages 236–241, 2016.
- [37] M. Yasin, B. Mazumdar, J. Rajendran, and O. Sinanoglu. TTLock: Tenacious and traceless Logic Locking. In *IEEE Int'l Symposium on Hardware Oriented Security and Trust (HOST)*, pages 166–166, 2017.
- [38] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran. Security Analysis of Anti-SAT. In *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 342–347, 2017.
- [39] O. Petura, U. Mureddu, N. Bochard, V. Fischer, and L. Bossuet. A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices. In *Int'l Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, 2016.
- [40] P. Chakraborty, J. Cruz, and S. Bhunia. SAIL: Machine learning guided structural analysis attack on hardware obfuscation. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 56–61, 2018.
- [41] P. Subramanian, S. Ray, and S. Malik. Evaluating the Security of Logic Encryption Algorithms. In *IEEE Int'l Symposium on Hardware Oriented Security and Trust (HOST)*, pages 137–143, 2015.
- [42] P. Tuyls, G.-J. Schrijen, B. Škorić, J. Van Geloven, N. Verhaegh, and R. Wolters. Read-Proof Hardware from Protective Coatings. In *Int'l Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 369–383, 2006.
- [43] R. P. Cocchi, L. W. Chow, J. P. Baukus, and B. J. Wang. Method and Apparatus for Camouflaging a Standard Cell based Integrated Circuit with Micro Circuits and Post Processing, 2013. US Patent.
- [44] R. S. Chakraborty and S. Bhunia. HARPOON: An Obfuscation-based SoC Design Methodology for Hardware Protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1493–1502, 2009.
- [45] S. Nikova, C. Rechberger, and V. Rijmen. Threshold Implementations against Side-Channel Attacks and Glitches. In *Int'l Conference on Information and Communications Security*, pages 529–545, 2006.

- [46] S. Roshanisefat, H. K. Thirumala, K. Gaj, H. Homayoun, and A. Sasan. Benchmarking the Capabilities and Limitations of SAT Solvers in Defeating Obfuscation Schemes. In *IEEE Int'l Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 275–280, 2018.
- [47] S. Roshanisefat, H. M. Kamali and A. Sasan. SRClock: SAT-Resistant Cyclic Logic Locking for Protecting the Hardware. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*, pages 153–158, 2018.
- [48] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama. Implementation of Double Arbiter PUF and its Performance Evaluation on FPGA. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 6–7, 2015.
- [49] U. Guin, Q. Shi, D. Forte, and M. M. Tehranipoor. FORTIS: A Comprehensive Solution for Establishing Forward Trust for Protecting IPs and ICs. *ACM Transactions on Design Automation of Electronic Systems*, 21(4):63, 2016.
- [50] W. Diehl, A. Abdulgadir, F. Farahmand, J.-P. Kaps, and K. Gaj. Comparison of Cost of Protection against Differential Power Analysis of Selected Authenticated Ciphers. *Cryptography*, 2(3):26, 2018.
- [51] W. Diehl, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. Gaj. Face-Off between the CAESAR Lightweight Finalists: ACORN vs. Ascon. In *Int'l Conference on Field Programmable Technology (ICFPT)*, 2018.
- [52] Y. Alkabani and F. Koushanfar. Active Hardware Metering for Intellectual Property Protection and Security. In *USENIX Security Symposium*, pages 291–306, 2007.
- [53] Y. Xie and A. Srivastava. Mitigating Sat Attack on Logic Locking. In *Int'l Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 127–146, 2016.

PRO-ORAM: Practical Read-Only Oblivious RAM

Shruti Tople*
Microsoft Research

Yaoqi Jia
Zilliqa Research

Prateek Saxena
NUS

Abstract

Oblivious RAM is a well-known cryptographic primitive to hide data access patterns. However, the best known ORAM schemes require a logarithmic computation time in the general case which makes it infeasible for use in real-world applications. In practice, hiding data access patterns should incur a constant latency per access.

In this work, we present PRO-ORAM— an ORAM construction that achieves *constant latencies* per access in a large class of applications. PRO-ORAM theoretically and empirically guarantees this for *read-only* data access patterns, wherein data is written once followed by read requests. It makes hiding data access pattern practical for read-only workloads, incurring sub-second computational latencies per access for data blocks of 256 KB, over large (gigabyte-sized) datasets. PRO-ORAM supports throughputs of tens to hundreds of MBps for fetching blocks, which exceeds network bandwidth available to average users today. Our experiments suggest that dominant factor in latency offered by PRO-ORAM is the inherent network throughput of transferring final blocks, rather than the computational latencies of the protocol. At its heart, PRO-ORAM utilizes key observations enabling an aggressively parallelized algorithm of an ORAM construction and a permutation operation, as well as the use of trusted computing technique (SGX) that not only provides safety but also offers the advantage of lowering communication costs.

1 Introduction

Cloud storage services such as Dropbox [4], Google Drive [8], Box [2] are becoming popular with millions of users uploading Gigabytes of data everyday [6]. However, outsourcing data to untrusted cloud storage poses several privacy and security issues [5]. Although encryption of data on the cloud guarantees data confidentiality, it is not sufficient to protect user privacy. Research has shown that access patterns on encrypted data leak substantial private information such as secret keys

and user queries [25, 27]. One line of research to stop such inference is the use of Oblivious RAM (ORAM) [22]. ORAM protocols continuously shuffle the encrypted data blocks to avoid information leakage via the data access patterns.

Although a long line of research has improved the performance overhead of ORAM solutions [20, 32, 37, 40, 42, 43], it is still considerably high for use in practice. Even the most efficient ORAM solutions incur at least logarithmic latency to hide read / write access patterns [20, 34, 43], which is the established lower bound for the general case. Ideally, hiding access patterns should incur a *constant* access (communication) latency for the client, independent of the size of data stored on the cloud server, and constant computation time per access for the cloud server. To reduce the logarithmic access time to a constant, we investigate the problem of designing solutions to hide specific patterns instead of the general case.

We observe that a large number of cloud-based storage services have a read-only model of data consumption. An application can be categorized in this model when it offers only read operations after the initial upload (write) of the content to the cloud. For example, services hosting photos (e.g., Flickr, Google Photos, Moments), music (e.g., iTunes, Spotify), videos (e.g., NetFlix, Youtube) and PDF documents (e.g., Dropbox, Google Drive) often exhibit such patterns. Recently, Blass et al. have shown that designing an efficient construction is possible for “write-only” patterns wherein the read accesses are not observable to the adversary (e.g. in logging or snapshot / sync cloud services) [18]. Inspired by such specialized solutions, we ask *whether it is possible to achieve constant latency to hide read-only access patterns?* As our main contribution, we answer the above question affirmatively for all cloud-based data hosting applications.

1.1 Approach

We propose PRO-ORAM— a practical ORAM construction for cloud-based data hosting services offering constant latency for read-only accesses. PRO-ORAM incurs a constant computation and communication latency per access making it a promising

*Work done as a Ph.D student at National University of Singapore (NUS)

solution to use in a large class of real-world applications. The key idea to achieve constant latencies is to decompose every request to read a data block into two separate sub-tasks of “access” and “shuffle” which can execute in parallel. However, simply parallelizing the access and shuffle operations is not enough to achieve constant latencies. Previous work that employs such parallelization for the general case would incur a logarithmic slowdown even for read-only accesses due to the inherent design of the underlying ORAM protocols [41].

In designing PRO-ORAM, we make two important observations that allow us to achieve constant latency. First, we observe that there exists a simple ORAM construction — the square-root ORAM [22] — which can be coupled with a secure permutation (or shuffle) [33] to achieve idealized efficiency in the read-only model. A naïve use of this ORAM construction incurs a worst-case overhead of $O(N \log^2 N)$ to shuffle the entire memory with N data blocks. The non-updatable nature of read-only data allows us to parallelize the access and shuffle operations on two separate copies of the data. This results in a de-amortized $O(\sqrt{N})$ latency per access.

Second, we design a secure method to distribute the work done in each shuffle step among multiple computational units without compromising the original security guarantees. Our construction still performs $O(\sqrt{N})$ work per access but it is parallelized aggressively to execute in a constant time. Assuming a sufficient number of cores, PRO-ORAM distributes the total shuffling work among $O(\sqrt{N})$ threads without leaking any information. Although the total computation work is the same as in the original shuffle algorithm, the latency reduces to a constant for read streaks¹. With these two observations, we eliminate the expensive $O(N \log^2 N)$ operation from stalling subsequent read access requests in PRO-ORAM. Thus, we show that a basic ORAM construction is better for hiding read data access patterns than a complex algorithm that is optimized to handle the general case. Further, we present a proof for the correctness and security of PRO-ORAM. Our improved construction of the shuffle algorithm maybe of independent interest, as it is widely applicable beyond ORAM.

PRO-ORAM can be applied opportunistically for applications that expect to perform long streaks of read accesses intermixed with infrequent writes, incurring a non-constant cost only on write requests. Therefore, PRO-ORAM extends obliviousness to the case of arbitrary access patterns, providing idealized efficiency for “read-heavy” access patterns (where long streaks of reads dominate). To reduce trust on software, PRO-ORAM assumes the presence of a trusted hardware (such as Intel SGX [1], Sanctum [19]) or a trusted proxy as assumed in previous work on ORAMs [16, 28, 41].

1.2 Results

We implement PRO-ORAM prototype in C/C++ using Intel SGX Linux SDK v1.8 containing 4184 lines of code [9]. We eval-

¹A read streak is a sequence of consecutive read operations.

uate PRO-ORAM using Intel SGX simulator for varying file / block sizes and total data sizes. Our experimental results demonstrate that the latency per access observed by the user is a *constant* of about 0.3 seconds to fetch a file (or block) of size 256 KB. Our empirical results show that PRO-ORAM is practical to use with a throughput ranging from 83 Mbps for block size of 100 KB to 235 Mbps for block size of 10 MB. These results are achieved on a server with 40 cores. In real cloud deployments, the cost of a deca-core server is about a thousand dollars [10]; so, the one-time setup cost of buying 40 cores worth of computation seems reasonable. Thus, PRO-ORAM is ideal for sharing and accessing media files (e.g., photos, videos, music) having sizes of few hundred KB on today’s cloud platforms. PRO-ORAM’s throughput exceeds the global average network bandwidth of 7 Mbps asserting that the inherent network latency dominates the overall access time rather than computation latencies in PRO-ORAM [7].

Contributions. We summarize our contributions below:

- *Read-only ORAM.* We present PRO-ORAM— a practical and secure read-only ORAM design for cloud-based data hosting services. PRO-ORAM’s design utilizes sufficient computing units equipped with a trusted hardware primitive.
- *Security Proof.* We provide a security proof to guarantee that our PRO-ORAM construction provides obliviousness in the read-only data model.
- *Efficiency Evaluation.* PRO-ORAM is highly practical with constant latency per access for fixed block sizes and provides throughput ranging from 83 Mbps for a block size of 100 KB to 235 Mbps for a block size of 10 MB.

2 Overview

Our main goal is to ensure two important characteristics: a) hide read data access patterns on the cloud server; and b) achieve constant time to access each block from the cloud.

2.1 Setting: Read-Only Cloud Services

Many applications offer data hosting services for images (e.g., Flickr, Google Photos, Moments), music (e.g., iTunes, Spotify), videos (e.g., NetFlix, Youtube), and PDF documents (e.g., Dropbox, Google Drive). In these applications, either the users (in the case of Dropbox) or the service providers (such as NetFlix, Spotify) upload their data to the cloud server. Note that the cloud provider can be different from the service provider, for example, Netflix uses Amazon servers to host their data. After the initial data is uploaded, users mainly perform read requests to access the data from the cloud.

Let a data owner upload N files each having a *file identifier* to the cloud. A file is divided into *data blocks* of size B

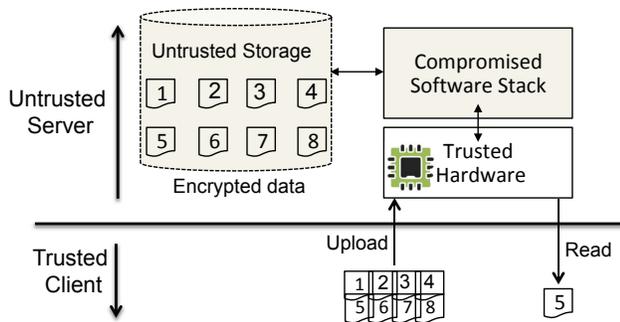


Figure 1: Baseline setting: Cloud-provider with trusted hardware and a compromised software stack. User uploads data and makes read requests

and stored in an *array* on the untrusted storage at the server. Each block is accessed using its corresponding *address* in the storage array. To handle variable length files, one can split large files into several data blocks and maintain a file to blocks mapping table. However, for simplicity, we assume each file maps to a single block and hence use the terms file and block interchangeably in this paper. When a user requests to fetch a file, the corresponding data block is read from the storage array and is sent to the user. To ensure confidentiality of the data, all the files are encrypted using a cryptographic key. The data is decrypted only on the user machine using the corresponding key.

2.2 Threat Model

Leakage of access patterns is a serious issue and has been shown to leak critical private information in several settings such as encrypted emails, databases and others [25, 27]. In our threat model, we consider that the adversary has complete access to the encrypted storage on the cloud. An attacker can exploit the vulnerabilities in the cloud software to gain access to the cloud infrastructure including the storage system which hosts encrypted content [5, 12]. Hence, we consider the cloud provider to be untrusted with a compromised software stack. The cloud provider can trace the requests or file access patterns of all the users accessing the encrypted data. We restrict each request to only read the data from the server. Essentially, the adversary can observe the exact address accessed in the storage array to serve each requested file. Along with access to the storage system, the adversary can observe the network traffic consisting of requested data blocks sent to each user.

Scope. Our main security goal is to guarantee obliviousness i.e., hide read access patterns of users from the cloud provider. Although we consider a compromised server, we do not defend against a cloud provider refusing to relay the requests to the user. Such denial of service attacks are not within the scope of this work. We only focus on leakage through *address* access patterns and do not block other channels of leakage

such as timing or file length [44]. For example, an adversary can observe the number of blocks fetched per request or the frequency of requesting files to glean private information about the user. However, our system can benefit from existing solutions that thwart these channels using techniques such as padding files with dummy blocks and allowing file requests at fixed interval respectively [15].

2.3 Baseline: Trusted H/W in the Cloud

A well-known technique to hide data access patterns is using Oblivious RAM (ORAM) [22]. In ORAM protocols, the encrypted data blocks are obliviously shuffled at random to unlink subsequent accesses to the same data blocks. Standard ORAM solutions guarantee obliviousness in a trusted client and an untrusted server setting. It generally uses a private memory called stash at the client-side to perform oblivious shuffling and re-encryption of the encrypted data. In the best case, this results in a logarithmic communication overhead between the client and the server [43]. To reduce this overhead, previous work has proposed the use of a trusted hardware / secure processor [16, 28] in the cloud or a trusted proxy [41]. This allows us to establish the private stash and a small trusted code base (TCB) to execute the ORAM protocol in the cloud. That is, instead of the client, the trusted component on the cloud shuffles the encrypted data, thereby reducing the communication overhead to a constant. Further, the trusted component can verify the integrity of the accessed data and protect against a malicious cloud provider [41]. Figure 1 shows the architecture for our baseline setting with a trusted hardware and a compromised software stack on the cloud.

In this work, we consider the above cloud setup with a trusted hardware as our baseline. Specifically, we assume the cloud servers are equipped with Intel SGX-enabled CPUs. SGX allows creating hardware-isolated memory region called *enclaves* in presence of a compromised operating system. With enclaves, we have a moderate size of private storage inaccessible to the untrusted software on the cloud. Further, we assume that the trusted hardware at the cloud provider is untampered and all the guarantees of SGX are preserved. We do not consider physical or side-channel attacks on the trusted hardware [26, 29, 31, 38, 45]. Defending against these attacks is out of scope but our system can leverage any security enhancements available in the future implementation of SGX CPUs [30]. In practice, SGX can be replaced with any other trusted hardware primitive available in the next-generation cloud servers.

2.4 Solution Overview

We present a construction called *PRO-ORAM*— a Practical Read-Only ORAM scheme that achieves *constant computation* latencies for read streaks. *PRO-ORAM* is based on square-root ORAM but can be extended by future work to other

ORAM approaches. It incurs default latency of the square-root ORAM approach in case of write operations. Thus, one can think of `PRO-ORAM` as a specialization for read streaks, promising most efficiency in applications that are read-heavy, but without losing compatibility in the general case.

Key Insight 1. The dominant cost in any ORAM scheme comes from the shuffling step. In square-root ORAM, the shuffling step is strictly performed after the access step [22]. This allows the shuffle step to consider any updates to the blocks from write operations. Our *main* observation is that for read-only applications, the algorithm need not wait for all the accesses to finish before shuffling the entire dataset. The key advantage in the read-only model is that the data is never modified. Thus, we can *decouple* the shuffling step from the logic to dispatch an access. This means the shuffle step can execute *in parallel* without stalling the read accesses. We give a proof for the correctness and security of `PRO-ORAM` in Section 5. Although prior work has considered parallelizing the access and shuffle step [41], our observations only apply to the read-only setting, and our specific way achieves constant latency which was not possible before.

Key Insight 2. Our second important observation allows us to reach our goal of constant latency. We observe that the Melbourne Shuffle algorithm performs $O(\sqrt{N})$ computation operations for each access where each operation can be executed independently [33]. Hence, the $O(\sqrt{N})$ computations can be performed in parallel (multi-threaded) without breaking any security or functionality of the original shuffle algorithm. This final step provides us with a highly optimized Melbourne Shuffle scheme which when coupled with square-root ORAM incurs constant computation latency per access. We further exploit the structure of the algorithm and propose pipelining based optimizations to improve performance by a constant factor (Section 4.4). We remark that our efficient version of the shuffle algorithm maybe of independent interest and useful in other applications [21, 33].

Note that `PRO-ORAM` is *compatible* with data access patterns that have writes after read streaks, since it can default to running a synchronous (non-parallel) shuffle when a write is encountered — just as in the original square-root ORAM. Of course, the constant latency holds for read streaks and read-heavy applications benefit from this specialized construction.

Comparison to Previous Work. The most closely related work with respect to our trust assumptions and cloud infrastructure is ObliviStore [41]. This protocol has the fastest performance among all other ORAM protocols when used in the cloud setting [17]. Similar to `PRO-ORAM`, ObliviStore parallelizes the access and shuffle operations using a trusted proxy for cloud-based data storage services.

We investigate whether ObliviStore’s construction can attain constant latency when adapted to the read-only model. We highlight that although the high-level idea of parallelizing ORAM protocol is similar to ours, ObliviStore differs

from `PRO-ORAM` in various aspects. ObliviStore is designed to hide arbitrary patterns in the general case and hence uses a complex ORAM protocol that is optimized for bandwidth. It uses a partition-based ORAM [42] where each partition is itself a hierarchical ORAM [22]. This design takes $O(\log N)$ time to access each block even if the protocol was restricted to serve read-only requests. Hence, our observations in the read-only model do not directly provide performance benefits to ObliviStore’s construction. The key factor in `PRO-ORAM` is that — “simple and specialized is better” — a simple ORAM construction which is non-optimized for the general case, is better suited for hiding read-only data access patterns.

3 Background

In designing an efficient `PRO-ORAM` scheme, we select square-root ORAM as our underlying ORAM scheme as it allows \sqrt{N} accesses before the shuffling step. To obviously shuffle the data in parallel with the accesses, we select the Melbourne shuffle scheme, that allows shuffling of data of $O(N)$ in $O(\sqrt{N})$ steps. Further, we use Intel SGX-enabled CPU present to create enclaves with $O(\sqrt{N})$ private storage. We provide a brief background on each of these building blocks.

3.1 Square-Root ORAM

We select the square-root ORAM scheme as the underlying building block in `PRO-ORAM`. The square-root ORAM scheme, as proposed by Goldreich and Ostrovsky [22], uses $N + \sqrt{N}$ permuted memory and a \sqrt{N} *stash* memory, both of them are stored encrypted on the untrusted cloud storage. The permuted memory contains N real blocks and \sqrt{N} dummy blocks arranged according to a pseudo-random permutation π .

To access a block, the protocol first scans the entire stash deterministically for the block. If the requested block is found in the stash then the protocol makes a fake access to a dummy block in the permuted memory. Otherwise, it accesses the real block from the permuted memory. The accessed block is then written to the stash by re-encrypting the entire \sqrt{N} stash memory. The key trick here is that all accesses exhibit a deterministic access order to the adversarial server, namely: a deterministic scan of the stash elements, followed by an access to a real or dummy block in permuted memory, followed by a final re-encrypted write and update to the stash. After every \sqrt{N} requests, the protocol updates the permuted memory with the stash values and obliviously permutes (shuffles) it randomly. This shuffling step incurs $O(N \log^2 N)$ overhead, resulting in an amortized latency of $O(\sqrt{N} \log^2 N)$ per request.

3.2 Intel SGX

Recently, Intel proposed support for a trusted hardware primitive called Software Guard Extensions (SGX). With SGX, we can create isolated memory regions called *enclaves* which are

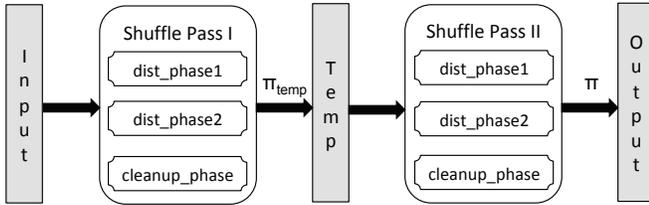


Figure 2: Overview of the Melbourne shuffle algorithm

inaccessible to the underlying operating system or any other application. In `PRO-ORAM`, we use the following two important features of Intel SGX. `PRO-ORAM` can be build using any other trusted hardware that provides these specific features.

Enclaved Memory. SGX allows the creation of hardware-isolated private memory region or enclaved memory. For SGX CPUs, BIOS allocates a certain region for processor reserved memory (PRM) at the time of boot up. The underlying CPU reserves a part of this PRM to create enclaves. All the code and data in the enclaved memory is inaccessible even to the privileged software such as the OS. Thus, an adversary in our threat model cannot access this protected memory. It guarantees confidentiality of the private data within enclaves from the adversary. At present, SGX supports 90 MB of enclaved memory. This allows us to use a moderate amount of private storage at the cloud provider. Further, we can create multiple threads within an enclave [39].

Attestation. Along with enclaved execution, SGX-enabled CPUs support remote attestation of the software executing within an enclave. This security features enables a remote party to verify the integrity of the software executing on an untrusted platform such as the cloud. Further, it supports local attestation between two enclaves executing on the same machine. These enclaves can then establish a secure channel and communicate with each other. One can perform such attestation of an enclave program as described in the SGX manual [1]. Thus, SGX-enabled CPUs at the cloud provider allows executing trusted code base (TCB) with a small amount of private storage at the cloud provider.

3.3 Melbourne Shuffle

Melbourne shuffle is a simple and efficient randomized oblivious shuffle algorithm [33]. Using this algorithm, we can obviously shuffle N data blocks with $O(N)$ external memory. The data is stored at the server according to a pseudo-random permutation. The encryption key and the permutation key π require constant storage and are stored in the private memory. This algorithm uses private storage of the size $O(\sqrt{N})$ and incurs a communication and message complexity of $O(\sqrt{N})$. We use this algorithm in `PRO-ORAM` to shuffle the encrypted data in parallel to accessing data blocks using enclave memory as the private storage.

The algorithm works in two passes as shown in Figure 2. It first shuffles the given input according to a random permutation π_{temp} and then shuffles the intermediate permutation to the desired permutation of π . Each pass of the shuffle algorithm has three phases, two distribution and a cleanup phase. The algorithm divides each N size array into buckets of size \sqrt{N} . Further, every $\sqrt[4]{N}$ of these buckets are put together to form a chunk. Thus, the N array is divided into total $\sqrt[4]{N}$ chunks. The first distribution phase (`dist_phase1`) simply puts the data blocks into correct chunks based on the desired permutation π_{temp} in the first pass and π in the second pass. The second distribution phase (`dist_phase2`) is responsible for placing the data blocks into correct buckets within each chunk. Finally, the clean up phase (`cleanup_phase`) arranges the data blocks in each bucket and places them in their correct positions based on the permutation key.

Choosing appropriate constants in the algorithm guarantees oblivious shuffling of N data blocks for any chosen permutation value π with a very high probability. The important point is that each of these phases can be implemented to have a “constant” depth and operate “independently” based only on the pre-decided π_{temp} and π values. This allows us to distribute the overall computation among multiple threads and parallelize the algorithm. Although the total work done remains the same, our design effectively reduces the overall execution time to a constant. We refer readers to the original paper for the detailed algorithm of each of these phases [33].

3.4 Encryption Algorithms

We use standard symmetric key and public key cryptographic schemes as our building blocks in `PRO-ORAM`. We assume that both these schemes guarantee IND-CPA security. The security guarantees of `PRO-ORAM` depends on the assumption of using secure underlying cryptographic schemes. We denote by $SE = (Gen_{SE}, Enc_{SE}, Dec_{SE})$ a symmetric key encryption scheme where Gen_{SE} algorithm generates a key which is used by the Enc_{SE} and Dec_{SE} algorithms to perform encryption and decryption respectively. $PKE = (Gen_{PKE}, Enc_{PKE}, Dec_{PKE})$ denotes a public key encryption scheme where the Gen_{PKE} algorithm generates a public-private key pair (Pb, Pr) . The Enc_{PKE} algorithm takes the public key Pb as input and encrypts the data whereas the Dec_{PKE} takes the private key Pr as input and decrypts the ciphertext.

4 PRO-ORAM Details

Today’s cloud platforms are equipped with a large amount of storage and computing units. In `PRO-ORAM`, we leverage these resources to achieve practical performance guarantees for hiding access patterns to read-only data such as photos, music, videos and so on.

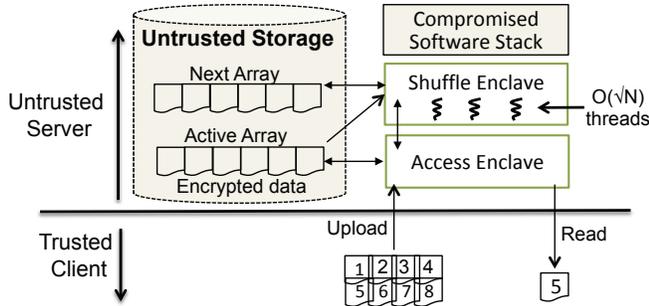


Figure 3: PRO-ORAM design overview with access and shuffle enclaves operating in parallel on active and next array.

4.1 Design Overview

Similar to any cloud storage service, we have a setup phase to establish user identities and upload initial data to the cloud. We outline the setup phase for users that directly upload their data to the cloud storage for e.g., Dropbox or Google Drive. However, it can be modified to accommodate applications such Netflix, Spotify where the initial data is uploaded by the service providers and not the users themselves.

Initialization. Each user registers with the cloud provider his identity uid and a public key Pb_{uid} mapped to their identity. Let the data structure Pub_map store this mapping on the server. The private key Pr_{uid} corresponding to the public key is retained by the user. Each of these registered users can upload their data to the server. To upload N data blocks to the untrusted server, a data owner first encrypts the data blocks with a symmetric key K and then sends them to the server. The order of these blocks during the initial upload does not affect the security guarantees of PRO-ORAM. On receiving the encrypted data, the server instantiates an “access” and a “shuffle” enclave. Next, the data owner attests the program running within these enclaves and secretly provisions the encryption key K to them on successful attestation.

System Overview. Figure 3 shows the overview of PRO-ORAM design for the read-only model. PRO-ORAM executes two enclaves called access and shuffle in parallel on the untrusted server. Each access and shuffle enclave has $O(\sqrt{N})$ private storage and corresponds to a set of N data blocks. These enclaves provide obliviousness guarantees to read from the N data blocks uploaded on the server. The enclaves locally attest each other and establish a secure channel between them [13]. They communicate over the secure channel to exchange secret information such as encryption and permutation keys (explained in detail in Section 4.2). The access enclave executes the square-root ORAM and the shuffle enclave performs the Melbourne shuffle algorithm. However, PRO-ORAM parallelizes the functioning of both these enclaves to achieve constant latency per read request.

Algorithm 1: Pseudocode for each round of shuffle enclave

```

Input: active_array: input data blocks ,
K_prev: previous key,
K_new: new key,
π: desired permutation,
r_num: current round number
Output: next_array: output permuted blocks
1 Let T1, T2, Otemp be temporary arrays;
2 Let πtemp be a random permutation;
3 Let Ktemp be an encryption key;
4 if r_num == 0 then
   | // Add dummy blocks
5   for j from N to N + √N do
6   |   d'_j ← EncSE(K_prev, dummy);
7   |   active_array = active_array ∪ d'_j;
8   end
9 end
   | // Two pass call to shuffle algorithm
10 mel_shuffle(active_array, T1, T2, πtemp, K_prev, Ktemp,
   |   Otemp);
11 mel_shuffle(Otemp, T1, T2, π, Ktemp, K_new, next_array);

```

PRO-ORAM algorithm consists of several rounds where each round is made of total \sqrt{N} requests from the users. In every round, the access enclave strictly operates on the permuted array of the uploaded data, which we refer as the active array. On every request, the access enclave fetches the requested data block either from the active array or the private stash (similar to the square-root ORAM), re-encrypts the block and sends it to the user. Simultaneously, the shuffle enclave reads data blocks in a deterministic pattern from the active array, performs the shuffle algorithm on them and outputs a new permuted array, which we refer as the next array. The shuffle enclave internally distributes the work using $O(\sqrt{N})$ separate threads. By the end of each round, i.e., after \sqrt{N} requests, the active array is replaced with the next array. Thus, for serving N data blocks, PRO-ORAM uses $O(N)$ space on the server to store the active and the next array.

Parallelizing the access and shuffle enclave enables PRO-ORAM to create a new permuted array while serving requests on the active array. This design is novel to PRO-ORAM and differs from previous ways of parallelizing access and shuffle operations [23, 41]. The algorithms for both the access and shuffle operations execute within SGX enclaves and are oblivious to the server. We give a detailed proof in Section 5.

4.2 Shuffle Enclave

The shuffle enclave starts its execution one round before the access enclave. We call this as the preparation round or round 0. The shuffle enclave uses this round to per-

Algorithm 2: Parallel pseudocode for `mel_shuffle` function

Input: I : input data blocks ,
 T_1, T_2 : Temporary arrays,
 K_{prev} : previous key,
 K_{new} : new key,
 π : desired permutation,
Output: O : output permuted blocks

- 1 Let K_1, K_2 be encryption keys;
// Place the blocks into correct chunks
- 2 `dist_phase1`($I, \pi, K_{prev}, K_1, T_1$): $O(\sqrt{N})$ threads;
// Place the blocks in correct buckets
- 3 `dist_phase2`(T_1, π, K_1, K_2, T_2): $O(\sqrt{N})$ threads;
// Arrange the blocks in each bucket
- 4 `cleanup_phase`(T_2, π, K_2, K_{new}): $O(\sqrt{N})$ threads;

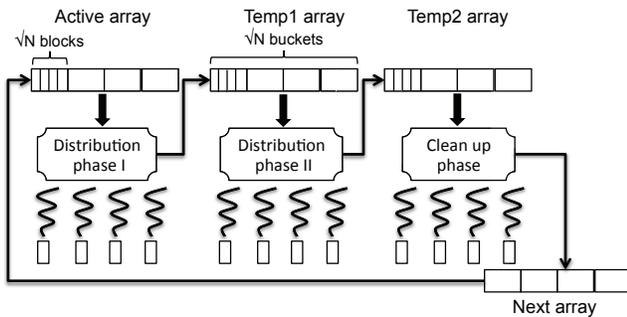


Figure 4: Multi-threaded Melbourne shuffle with constant latency per access

mute the data which is uploaded by the user during the initialization phase. The enclave permutes N encrypted real blocks (d'_1, \dots, d'_N) along with \sqrt{N} dummy blocks and adds them to the active array (as shown in lines 4-9 in Algorithm 1). In each round, the enclave executes the Melbourne shuffle algorithm with the active array as input and the next array as output. It makes a two pass call to the `mel_shuffle` function (lines 10 and 11). Internally, the function performs the three phases of `dist_phase1`, `dist_phase2` and `clean_up_phase` (lines 2, 3, 4 in Algorithm 2). Each phase performs \sqrt{N} steps, where each step fetches \sqrt{N} blocks of the input array, re-arranges and re-encrypts them and writes to the output array.

In PRO-ORAM, we distribute this computation over $O(\sqrt{N})$ threads and thus parallelize the execution of each phase (as shown in Figure 4). Carefully selecting the hidden constants in $O(\sqrt{N})$ allows us to securely distribute the work without compromising on the security of the original algorithm (see Lemma 5.1 in Section 5). Each thread re-encrypts and re-arranges only a single block in every step of the phase and writes them back in a deterministic manner. The operations on each block are independent of other blocks and

Algorithm 3: Pseudocode for Read Algorithm

Input: d_i : block identifier,
active_array: encrypted data blocks,
request: current request number
Output: d' : encrypted block

- 1 Lookup in the private stash;
- 2 **if** d_i **in** stash **then**
// access dummy value
3 | $addr \leftarrow \pi(N + request)$;
4 | $d' \leftarrow active_array(addr)$;
// select value from stash
5 | $d' \leftarrow stash(d_i)$;
- 6 **else**
7 | $addr \leftarrow \pi(d_i)$;
8 | $d' \leftarrow active_array(addr)$;
9 | Write d' to the stash;
- 10 **end**
- 11 **return** d' ;

have a constant depth. The threads use the private memory within the enclave as a stash to obviously shuffle the blocks. However, each thread reads and writes to its corresponding memory location during the shuffling step. We exploit this property and parallelize the computation on each of these blocks. In PRO-ORAM, we implement this approach using multi-threading with SGX enclaves. The `shuffle` enclave starts $O(\sqrt{N})$ threads in parallel to compute the re-encryption and rearrangement of data blocks. This results in a constant computation time per step. Thus, with parallelization imposed in each step, the total computation time for shuffling N data blocks is $O(\sqrt{N})$. Hence, the amortized computation latency per request over \sqrt{N} requests is reduced to $O(1)$. PRO-ORAM distributes the work in each shuffle step over $O(\sqrt{N})$ threads.

After the shuffle is completed, the `next_array` is copied to the active_array. The `shuffle` enclave sends the new keys (K_{new}) and permutation value (π) to the `access` enclave using a secure channel established initially. The latter enclave uses these keys to access the correct requested blocks from the active_array in the next round.

4.3 Access Enclave

Unlike the `shuffle` enclave, the `access` enclave begins execution from round 1. Each round accepts \sqrt{N} read requests from the users. Before the start of each round, the `access` enclave gets the permutation π and encryption key K_{new} from the `shuffle` enclave. The active array corresponds to data blocks shuffled and encrypted using the keys π and K_{new} . For each request, the `access` enclave takes as input the block identifier d_i and the requesting user id `uid`. The enclave first confirms that the requesting `uid` is a valid registered user and

Algorithm 4: Pseudocode for each round of `access` enclave

```

Input:  $d_i$ : request file identifier ,
Pub_map: User id and public key mapping table ,
uid: requesting user id,
 $K_{new}$ : encryption key
 $\pi$ : permutation key
active_array: permuted array
Output: response_msg
1 for request from 1 to  $\sqrt{N}$  do
2    $Pb_{uid} \leftarrow Pub\_map(uid)$ ;
3    $d' \leftarrow Read(d_i, active\_array, request)$ ;
4    $k' \leftarrow Gen_{SE}$ ;
5    $d'' \leftarrow Enc_{SE}(Dec_{SE}(d', K_{new}), k')$ ;
6    $key\_msg = Enc_{PKE}(Pb_{uid}, k')$ ;
7    $response\_msg = (d'', key\_msg)$ ;
8 end

```

has a public key corresponding to the user. On confirmation, the enclave invokes the read function in Algorithm 3.

The algorithm to read a block is same as the main logic of square-root ORAM. Algorithm 3 provides the pseudocode for reading a data block in PRO-ORAM. Note that, we do not store the private stash on the untrusted cloud storage as proposed in the original square-root ORAM approach. Instead, the stash is maintained in the private memory within the access enclave. The stash is indexed using a hash table and hence can be looked up in a constant time. The read algorithm checks if the requested data block is present in the private stash. If present, the enclave accesses a dummy block from the untrusted storage. Else, it gets the address for the requested block d_i using permutation π and fetches the real block from the untrusted storage. The output of the read algorithm is an encrypted block d' . The block is stored in the private stash.

After fetching the encrypted block d' , either from the private stash or the `active` array, the `access` enclave decrypts it using K_{new} . Algorithm 4 shows the pseudocode for this step. It then selects a new key k' and encrypts the block. The output message includes this re-encrypted block and the encryption of k' under public key of the requesting user Pb_{uid} . At the end of each round i.e., after serving \sqrt{N} request, the access enclave clears the private storage, permutation π and K_{new} . Note that unlike the original square-root ORAM, there is no shuffling after \sqrt{N} requests. The permuted `next` array from the `shuffle` enclave replaces the `active` array.

Performance Analysis. In PRO-ORAM, the `access` enclave sends only the requested block to the user. This results in a communication overhead of $O(1)$ with respect to the requested block size. Further, the `access` enclave computes i.e., re-encrypts only a single block for each request. Thus, the computation on the server for the `access` enclave is $O(1)$. The `shuffle` enclave computes a permuted array in $O(\sqrt{N})$

steps. It fetches $O(\sqrt{N})$ blocks for each request. Note that the total computation performed at the server is still $O(\sqrt{N})$ for each request. However, in PRO-ORAM, we parallelize the computation i.e., re-encryption on $O(\sqrt{N})$ blocks in $O(\sqrt{N})$ threads. This reduces the computation time required for each step to only a single block. Thus, the overall computation latency for the `shuffle` enclave is $O(1)$ per request.

4.4 Optimizations

We further propose optimizations to Melbourne shuffle algorithm such that the performance can be improved by a constant factor. Both these optimizations are possible by exploiting the design structure of the algorithm.

Pipelining. We observe that in the existing algorithm (shown in Algorithm 1) the three phases execute sequentially (see Figure 4). Once the `dist_phase1` function generates the `temp1` array, it waits until the remaining phases complete. On the other hand, the `dist_phase2` and the `cleanup_phase` functions have to wait for the previous phase to complete before starting their execution. To eliminate this waiting time, we separate the execution of these phases into different enclaves and execute them in a pipeline. Thus, instead of waiting for the entire shuffle algorithm to complete, `dist_phase1` enclave generates a new `temp` array in every round to be used as input by the `dist_phase2` enclave. Eventually, each phase enclave outputs an array in every round to be used by the next phase enclave, thereby pipelining the entire execution. Note that this optimization is possible because the input to the `dist_phase1` does not depend on any other phase. `dist_phase1` enclave can continue to use the initial uploaded data as input and generate different `temp` arrays based on a new permutation value selected randomly in each round. This allows us to continuously execute each of the phases in its own enclave without becoming a bottleneck on any other phase. Thus, the overall latency is reduced by a factor of 3. This optimization increases the external storage requirement by $2N$ to store the additional `temp` array.

Parallel Rounds using Multiple Enclaves. Another optimization is to instantiate multiple (possibly $O(\sqrt{N})$) enclaves and execute each of the $O(\sqrt{N})$ rounds in parallel in these enclaves. With this optimization, the latency for shuffling N data blocks reduces from $O(\sqrt{N})$ to $O(1)$. This observation is also discussed by Ohrimenko et al. [33]. However, the main drawback in implementing this optimization is the blow-up in the combined private storage. As each of $O(\sqrt{N})$ enclaves requires private memory of size $O(\sqrt{N})$, the combined private memory is linear in the total data size $O(N)$. Such a huge requirement of private storage may not be feasible even on very high-end servers. In our work, to use this optimization without requiring linear private storage, we propose using only a constant number of enclaves, thereby improving the performance by a constant factor.

5 Security Analysis

The observable access patterns in PRO-ORAM include accesses made both from `access` and `shuffle` enclave. We first show that the `shuffle` enclave executes an oblivious algorithm.

Lemma 5.1. *Given N data blocks, Melbourne Shuffle is an oblivious algorithm and generates a permuted array with very high probability $(1 - \text{negl}(N))$ in $O(\sqrt{N})$ steps, each exchanging a message size of $O(\sqrt{N})$ between a private memory of $O(\sqrt{N})$ and untrusted storage of size $O(N)$.*

This Lemma directly follows from Theorem 5.1 and 5.6 in [33]. In PRO-ORAM, the `shuffle` enclave executes the Melbourne Shuffle algorithm using $O(\sqrt{N})$ memory within an enclave. Thus, from Lemma 5.1, we get the corollary below,

Corollary 5.1. *The `shuffle` enclave generates a permuted array of $O(N)$ data blocks in $O(\sqrt{N})$ steps and the access patterns are oblivious to the server.*

From Corollary 5.1, the access patterns of the `shuffle` enclave are oblivious and the output is indistinguishable from a pseudo-random permutation (PRP) [33].

Further, the communication between `access` and `shuffle` enclave happens over a secure channel. This preserves the confidentiality of the permutation and encryption keys that `shuffle` enclave sends to the `access` enclave at the end of each round. Thus, no information is leaked due to the interaction between these enclaves in PRO-ORAM. Now, to prove that PRO-ORAM guarantees obliviousness for read access patterns, we first show that a request to the `access` enclave is indistinguishable from random for an adaptive adversary.

Let $\mathcal{E} = (\text{Gen}, \text{Enc}, \text{Dec})$ be a IND-CPA secure encryption scheme where Gen generates a key which is used by the Enc and Dec algorithms to perform encryption and decryption respectively. Let λ be the security parameter used in \mathcal{E} . $\text{Exp}_{\mathcal{A}_{\text{adt}}, \mathcal{E}}^{\text{PRO-ORAM}}$ refers to the instantiation of the experiment with PRO-ORAM, \mathcal{E} algorithms and adaptive adversary \mathcal{A}_{adt} . This experiment captures our security definition for read-only obliviousness. The experiment consists of:

- \mathcal{A}_{adt} creates request $r = (\text{read}, d_i)$ and sends it to a challenger \mathcal{C} .
- The challenger selects $b \xleftarrow{\$} \{1, 0\}$.
- If $b = 1$, then \mathcal{C} outputs the address access patterns to fetch d_i i.e., $A(d_1) \leftarrow \text{access}(d_i)$ and encrypted output $O_1 \leftarrow d_i'$
- If $b = 0$, then \mathcal{C} outputs a random address access pattern i.e., $A(d_0) \xleftarrow{\$} \{1, \dots, N + \sqrt{N}\}$ and $O_0 \xleftarrow{\$} \{0, 1\}^\lambda$
- Adversary \mathcal{A}_{adt} has access to an oracle $\mathcal{O}^{\text{PRO-ORAM}}$ that issues q queries of type (read, d) both before and after

executing the challenge query r . The oracle outputs address access patterns to fetch d i.e., $A(d) \leftarrow \text{access}(d)$

- \mathcal{A}_{adt} outputs $b' \in \{1, 0\}$.
- The output of the experiment is 1 if $b = b'$ otherwise 0. The adversary \mathcal{A}_{adt} wins if $\text{Exp}_{\mathcal{E}}^{\text{PO}}(\lambda, b') = 1$.

Based on the experiment and its output, we define read-only obliviousness as follows:

Definition 5.1. An algorithm satisfies read-only obliviousness iff for all PPT adversaries \mathcal{A} , there exists a negligible function negl such that:

$$\Pr[\text{Exp}_{\mathcal{A}_{\text{adt}}, \mathcal{E}}^{\text{PRO-ORAM}}(\lambda, 1) = 1] - \Pr[\text{Exp}_{\mathcal{A}_{\text{adt}}, \mathcal{E}}^{\text{PRO-ORAM}}(\lambda, 0) = 1] \leq \text{negl}(1) \quad (1)$$

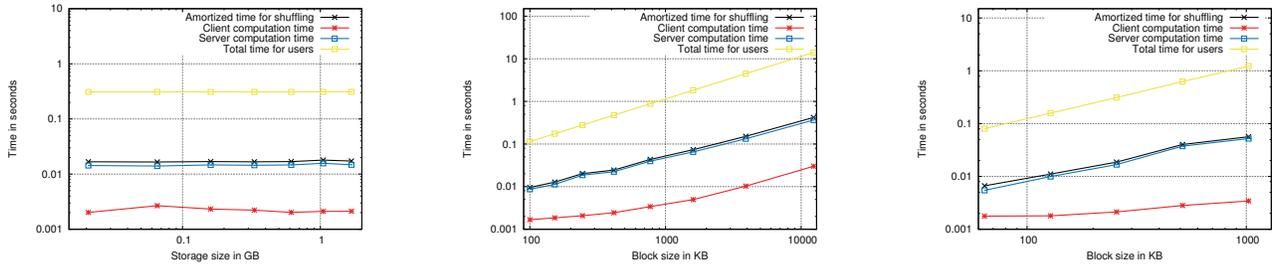
Theorem 5.1. *If `shuffle` enclave executes an oblivious algorithm and \mathcal{E} is a CPA-secure symmetric encryption then PRO-ORAM guarantees read-only obliviousness as in Def. 5.1.*

Proof. We present the proof in Appendix A

6 Implementation and Evaluation

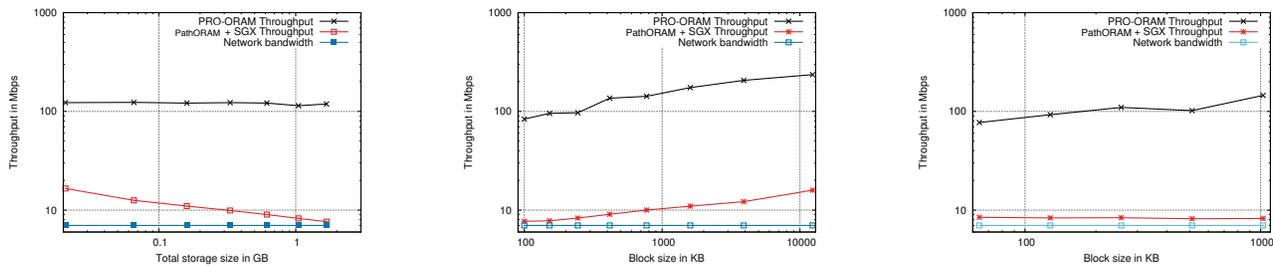
Implementation. We have implemented our proposed PRO-ORAM algorithm in C/C++ using Intel SGX Linux SDK v1.8 [9]. For implementing symmetric and public key encryption schemes, we use AES with 128-bit keys and Elgamal cryptosystem with 2048 bit key size respectively from the OpenSSL library [39]. We use SHA256 as our hash function. We implement the read logic of square-root ORAM and the parallelized shuffle algorithm as explained in Section 4.2. We use multi-threading with SGX enclaves to implement our parallel execution approach for each step. The prototype contains total 4184 lines of code measured using CLOC tool [3].

Experimental Setup & Methodology. To evaluate PRO-ORAM, we use SGX enclaves using the Intel SGX simulator and perform experiments on a server running Ubuntu 16.04 with Intel(R) Xeon(R) CPU E5-2640 v4 processors running at 2.4 GHz (40 cores) and 128 GB of RAM. As PRO-ORAM's design uses \sqrt{N} threads, our experimental setup of 40 cores can execute a total of 80 threads using Intel's support for Hyper-Threading, thereby handling requests with block-size of 256 KB for around 1 GB of data. Operating with data of this size is not possible with SGX in hardware mode available on laptops due to their limited processing capacity (8 cores). However, for real cloud deployments, the cost of a deca-core server is about a thousand dollars [10]; so, the one-time cost of buying 40 cores worth of computation per GB seems reasonable. To measure our results for gigabyte sized data, we chose to run 40 cores (80 threads) each with an SGX simulator.



(a) Execution time is constant for fixed $B = 256KB$. (b) Execution increases with B for $N.B = 1GB$. (c) Execution time increases with B where $N = 4096$.

Figure 5: Execution time for client, server, shuffle and total latency per access for a fixed block size (B), fixed total storage ($N.B$) and a fixed no. of blocks (N)



(a) Throughput for varying $N.B$ where $B = 256KB$. (b) Throughput for varying B where $N.B = 1GB$. (c) Throughput for varying B where $N = 4096$.

Figure 6: Throughput of PRO-ORAM in Mbps for fixed block size (B), fixed total storage ($N.B$) and fixed number of blocks (N)

As a baseline for comparisons of communication and network latencies, we take the bandwidth link of 7 Mbps as a representative, which is the global average based on a recent report from Akamai [7]. We perform our evaluation on varying data sizes such that the total data ranges from 20 MB to 2 GB with block sizes (B) varying from 4 KB to 10 MB. In our experiments for parallelized shuffle, as shown in Algorithm 1, we set temporary buffers as $2\sqrt{N}$ data blocks to ensure security guarantees. To make full use of computation power, we utilize all 40 cores for performing multi-threading for each distribution phase and cleanup phase. All results are averaged over 10 runs, reported on log-scale plots. We perform our evaluation with the following goals:

- To validate our theoretical claim of constant communication and computation latencies.
- To confirm that execution time per access in PRO-ORAM is dependent *only* on the block size.
- To show that the final bottleneck is the network latency, rather than computational latency.
- To show the effective throughput of PRO-ORAM for different blocks of practical data sizes.

6.1 Results: Latency

To measure the performance, we calculate the execution time (latency) at the user, server (i.e., access enclave) and the amortized shuffling time of the shuffle enclave for each request. We point out that the client computational latency, the amortized shuffle time, and the network latency are the three factors that add up to the overall latency.

Impact on Latency with Increasing Storage. We measure the execution time to access a block of fixed size $B = 256KB$, while increasing the total storage size from 20 MB to 2GB. The measurements are reported in Figure 5a. The dominant cost, as expected, is from the server computation. The access and shuffle enclave each incur a constant execution time of around 0.016 seconds per access, irrespective of the data sizes. The client computation time is constant at 0.002 seconds as the user only decrypts a constant-size encrypted block. Overall, these results confirm our theoretical claims of constant latency per request, and that the latency for large data size (in GBs) is practical (under 1 sec for 256KB blocks).

Computational vs. network bottleneck. An important finding from Figure 5a is that the latency per access observed by the user is a constant at 0.3 seconds, within experimental error, irrespective of the total data size. Even though the server computation cost is high, the final latency has primary

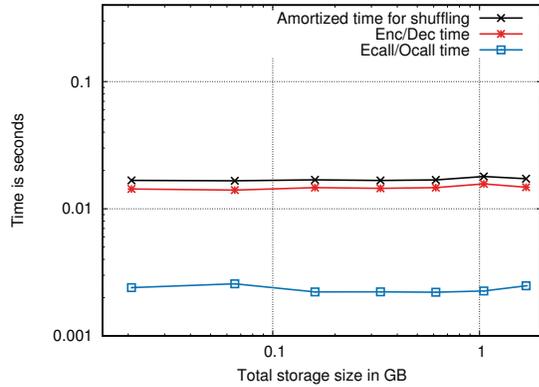


Figure 7: Overhead breakdown for shuffle step for fixed block-size $B = 256$

bottleneck as the network, not PRO-ORAM’s computation. In Figure 5a, the latency of shuffle per requested block is *lesser* than the network latency of sending a block from the server to the client on a 7Mbps link. This finding suggests that even for 256 KB block sizes, the network latency dominates the overall latency observed by the user, and is likely to be the bottleneck in an end application (e.g. streaming media) rather than the cost of all the operations in PRO-ORAM, including shuffling. This result suggests that PRO-ORAM is optimized enough to compete with network latency, making it practical to use in real applications.

Latency increase with block size. We perform three sets of experiments keeping (a) block size constant (B), (b) total storage size constant ($N \cdot B$), and (c) number of blocks constant (N), while varying the remaining two parameters respectively in each experiment. The results in Figure 5b and 5c show evidence that the computational latencies of server and client-side cost in PRO-ORAM depend primarily on the block size parameter, and is unaffected by the number of blocks or size of data. This is mainly because the cost of encryption and decryption per block increases these latencies.

6.2 Results: Throughput

We calculate throughput as the number of bits that PRO-ORAM can serve per second. PRO-ORAM can serve maximum \sqrt{N} blocks in the time the shuffle enclave completes permutation of N data blocks. Thus, to calculate throughput we use the following formula, $\text{Throughput} = \frac{\sqrt{N} \cdot B}{\text{total_shuffling_time}}$.

Throughput increase with block size. We find that throughput of PRO-ORAM increases with block size, ranging from 83 Mbps (for 100KB block size) to 235 Mbps (for 10MB block size), as shown in Figure 6b. Our experiments show that for data objects of the size larger than few hundred KB, the throughput is almost 10x larger than the global average network bandwidth (7Mbps). Such data object sizes are common for media content (e.g photos, videos, music) and cache web

page content [11]. Figure 6b and Figure 6c show the throughput measurements for increasing block sizes, keeping the total data size and the number of blocks fixed to 1 GB and 4096 respectively. We observe that the throughput increases with the blocksize. If we keep the block size fixed, the throughput is constant at almost 125 Mbps with the increase in the total data size, as seen in Figure 6a. Our evaluation shows that PRO-ORAM’s throughput exceeds reference throughput of 7 Mbps, re-confirming that network latency is likely to dominate latencies than computational overheads of PRO-ORAM.

Comparison to Tree-based ORAM. We compare the throughput of PRO-ORAM with the access overhead of using the simplest and efficient PathORAM scheme with SGX [43]. The client side operations in the original PathORAM scheme are executed within SGX. The throughput for PathORAM+SGX scheme decreases and almost reaches the network latency limit (7 Mbps) with increase in the number of blocks for fixed blocksize of 256 KB. Thus, the server computation overhead of $O(\log N)$ per access of PathORAM protocol becomes a bottleneck for reasonably large data sizes (e.g., 2 GB as shown in Figure 6a). Figure 6b shows that PathORAM’s throughput increases from 7 to 15 Mbps with a decrease in blocks.

6.3 Performance Breakdown

To understand the breakdown of the source of latency for the shuffle step, we calculate the time to perform the cryptographic operations and ECALLs/OCALLs to copy data in and out of memory. Such a breakdown allows us to better understand the time consuming operations in our system. We fix the block size to $B = 256$ KB and vary the total data size. Figure 7 shows the amortized shuffling time, time to perform encryption and decryption operations and the time to invoke ECALLs/OCALLs per access in PRO-ORAM. We observe that the dominant cost comes from the cryptographic operations 0.014 seconds out of the 0.016 seconds. Enclaves by design cannot directly invoke system calls to access untrusted memory. Each call to the outside enclave performed using OCALL. Similarly, a function within an enclave is invoked using an ECALL. Thus, invocation of ECALLs/OCALLs is necessary to perform multi-threading and for copying data in and out of memory. To fetch \sqrt{N} data blocks in parallel for each access, we use asynchronous ECALLs/OCALLs in PRO-ORAM similar to that proposed in a recent work [14]. These operations require 0.002 seconds (average) for a block of size 256 KB.

7 Related Work

First, we discuss ORAM constructions that guarantee constant latency per access for write-only patterns. Next, we summarize related work with similarities in our threat model.

Write-Only ORAMs. Recently, it is shown that constant computation and communication latency can be achieved

for applications with restricted patterns. Blass et. al show that some applications require hiding only write-patterns and hence proposed Write-Only ORAM in the context of hidden volumes [18]. Their work achieves constant latencies per write access to the data untrusted storage. Roche et al. propose a stash-free version of this Write-Only ORAM [35]. Further, Flat ORAM improves over this solution using secure processors to perform efficient memory management [24]. ObliviSync uses the write-only ORAM idea to support sharing of files on a Dropbox-like storage server that support auto-sync mechanisms [15]. These works that guarantee constant overhead for hiding write-only access patterns inspire our work. PRO-ORAM focuses on applications that exhibit read-only patterns and achieves constant latencies for them.

Improvements to square-root ORAM. Although square-root ORAM is known to have very high i.e., $O(N \log^2 N)$ worst-case overhead, Goodrich et. al provide a construction that reduces the worst-case overhead to $O(\sqrt{N} \log^2 N)$. Instead of shuffling the entire memory at once taking $O(N \log^2 N)$ computation time, their solution de-amortizes the computation over \sqrt{N} batches each taking $O(\sqrt{N} \log^2 N)$ time after every access step. This technique is similar to the distribution of shuffle steps in PRO-ORAM. However, our observations for the read-only setting allows us to execute the access and shuffle steps in parallel which is not possible in their solution. Ohrimenko et. al show that use of Melbourne Shuffle combined with square-root ORAM can reduce the worst-case computation time to $O(\sqrt{N})$ with the use of $O(\sqrt{N})$ private memory. In PRO-ORAM, we show that it is further possible to reduce the latency to a constant for applications with read-heavy access patterns. Further, Zahur et al. have shown that although square-root ORAM has asymptotically worse results than the best known schemes, it can be modified to achieve efficient performance in multi-party computation as compared to the general optimized algorithms [46]. In PRO-ORAM, we have a similar observation where square-root ORAM approach performs better in the read-only setting.

Solutions using Trusted Proxy. ObliviStore [41] is the first work that uses a trusted proxy to mediate asynchronous accesses to shared data blocks among multiple users, which was later improved by TaoStore [36]. A major differentiating point is that both ObliviStore [41] and TaoStore [36] assume mutually trusting users that do not collude with the server, thus operating in a weaker threat model than ours. The key contribution in these works is towards improving efficiency using a single ORAM over having separate ORAMs for each user. ObliviStore improves the efficiency of the SSS ORAM protocol [42] by making ORAM operations asynchronous and parallel. Similar to this work, their key idea is to avoid blocking access requests on shuffle operations, thereby matching the rate of access and shuffle operations using a trusted proxy. However, their underlying approach to achieve such parallelization largely differs from this work. Our observations

in designing PRO-ORAM are novel with respect to a read-only data setting that allows us to reduce the computation latency to a constant whereas ObliviStore has $O(\log N)$ computation latency per (read/write) access. TaoStore [36] is a more recent work that improves over ObliviStore using a trusted proxy and Path-ORAM [43] as its building block. Similar to [41], this approach has $O(\log N)$ computation latency per access.

Solutions using Trusted Hardware. An alternate line of work has shown the use of trusted hardware or secure processors with the goal to improve performance, as opposed to our use to strengthen existing ORAM protocols in a stronger threat model. Shroud uses trusted hardware to guarantee private access to large-scale data in data center [28]. ObliviAd is another system that makes use of trusted hardware to obliviously access advertisements from the server [16]. However, both these solutions do not optimize for read access patterns.

8 Conclusion

In this work, we provide a constant communication and computation latency solution to hide read data access patterns in a large class of cloud applications. PRO-ORAM guarantees a practical performance of 0.3 seconds to access a block of 256 KB leveraging sufficient storage and compute units with trusted hardware available on today's cloud platform. Our work demonstrates that simple ORAM solutions are better suited to hide read data access patterns than complex algorithms that are optimized for arbitrary read/write accesses.

Acknowledgments

We thank the anonymous reviewers for their feedback. This research is supported in part by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No. NRF2014NCR-NCR001-21).

References

- [1] Software Guard Extensions Programming Reference. software.intel.com/sites/default/files/329298-001.pdf, Sept 2013.
- [2] Box. <https://www.box.com/home>, Accessed: 2017.
- [3] Cloc. <http://cloc.sourceforge.net/>, Accessed: 2017.
- [4] Dropbox. <https://www.dropbox.com/>, Accessed: 2017.
- [5] Dropbox hacked. <https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach>, Accessed: 2017.

- [6] Dropbox usage statistics. <http://expandeddrablings.com/index.php/dropbox-statistics/>, Accessed: 2017.
- [7] Global average connection speed increases 26 percent year over year, according to akamai's 'fourth quarter, 2016 state of the internet report'. <https://www.akamai.com/us/en/about/news/press/2017-press/akamai-releases-fourth-quarter-2016-state-of-the-internet-connectivity-report.jsp>, Accessed: 2017.
- [8] Google drive. <https://drive.google.com/drive/>, Accessed: 2017.
- [9] Intel sgx linux sdk. <https://github.com/01org/linux-sgx>, Accessed: 2017.
- [10] Intel xeon processor pricing. https://ark.intel.com/products/92984/Intel-Xeon-Processor-E5-2640-v4-25M-Cache-2_40-GHz, Accessed: 2017.
- [11] Web content statistics. <http://httparchive.org/trends.php>, Accessed: 2017.
- [12] World's biggest data breaches. http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/Internet_and_cloud_services_-_statistics_on_the_use_by_individuals, Accessed: 2017.
- [13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.
- [14] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In *12th USENIX Symp. Operating Systems Design and Implementation*, 2016.
- [15] Adam J Aviv, Seung Geol Choi, Travis Mayberry, and Daniel S Roche. Oblivisync: Practical oblivious file backup and synchronization. In *NDSS*, 2017.
- [16] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 257–271. IEEE, 2012.
- [17] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 837–849. ACM, 2015.
- [18] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.
- [19] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.
- [20] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.
- [21] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *USENIX Security*, volume 15, pages 447–462, 2015.
- [22] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.
- [23] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100. ACM, 2011.
- [24] Syed Kamran Haider and Marten van Dijk. Flat oram: A simplified write-only oblivious ram construction for secure processor architectures. *arXiv preprint*, 2016.
- [25] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, volume 20, page 12, 2012.
- [26] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952*, 2016.
- [27] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.
- [28] Jacob R Lorch, Bryan Parno, James W Mckens, Mariana Raykova, and Joshua Schiffman. Shroud: ensuring private access to large-scale data in the data center. In *FAST*, volume 2013, pages 199–213, 2013.

- [29] Sinisa Matetic, Kari Kostianen, Aritra Dhar, David Sommer, Mansoor Ahmed, Arthur Gervais, Ari Juels, and Srdjan Capkun. Rote: Rollback protection for trusted execution. <https://eprint.iacr.org/2017/048.pdf>.
- [30] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, page 10. ACM, 2016.
- [31] Ming-Wei-Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS 2017*.
- [32] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication oram with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 862–873. ACM, 2015.
- [33] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming*, pages 556–567. Springer, 2014.
- [34] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious ram. In *USENIX Security Symposium*, pages 415–430, 2015.
- [35] Daniel S Roche, Adam Aviv, Seung Geol Choi, and Travis Mayberry. Deterministic, stash-free write-only oram. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 507–521. ACM, 2017.
- [36] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 198–217. IEEE, 2016.
- [37] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*. 2011.
- [38] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.
- [39] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26-March 1, 2017*, 2017.
- [40] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 247–258. ACM, 2013.
- [41] Emil Stefanov and Elaine Shi. Oblivstore: High performance oblivious cloud storage. In *IEEE S&P*, 2013.
- [42] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. In *NDSS’12*, 2011.
- [43] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.
- [44] Shruti Tople and Prateek Saxena. On the trade-offs in oblivious execution techniques. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.
- [45] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.
- [46] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root oram: efficient random access in multi-party computation. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 218–234. IEEE, 2016.

A Security Analysis

Theorem A.1. *If `shuffle` enclave executes an oblivious algorithm and \mathcal{E} is a CPA-secure symmetric encryption scheme then `PRO-ORAM` guarantees read-only obliviousness as in Definition 5.1.*

Proof. From Lemma 5.1, the access pattern of `shuffle` enclave are data-oblivious. To prove the theorem, we have to show that access pattern from `access` enclave are indistinguishable to the adversary. We proceed with a succession of games as follows:

- Game_0 is exactly the same as $\text{Exp}_{\mathcal{A}_{\text{adv}}, \mathcal{E}}^{\text{PRO-ORAM}}(\lambda, 1)$
- Game_1 replaces the O'_1 in Game_0 with a random string while other parameters are the same

- Game_2 is same as Game_1 except that $A(d_i)$ is selected using a pseudorandom permutation $\pi_s : \{0, 1\}^{(N+\sqrt{N})} \rightarrow \{0, 1\}^{(N+\sqrt{N})}$ where $s \leftarrow \{0, 1\}^\lambda$ and not from the access enclave.
- Game_3 is same as Game_2 except that $A(d_i)$ is selected at random from the entire data array.

From above description, we have

$$Pr[\text{Game}_0 = 1] = Pr[\text{Exp}_{\mathcal{A}_{\text{adv}}, \mathcal{E}}^{\text{PRO-ORAM}}(\lambda, 1) = 1], \quad (2)$$

For Game_1 , a distinguisher D_1 reduces the security of \mathcal{E} to IND-CPA security such that:

$$Pr[\text{Game}_0 = 1] - Pr[\text{Game}_1 = 1] \leq Adv_{D_1, \mathcal{E}}^{\text{IND-CPA}}(\lambda), \quad (3)$$

For Game_2 , according to Corollary 5.1, the advantage of a distinguisher D_2 is such that:

$$Pr[\text{Game}_1 = 1] - Pr[\text{Game}_2 = 1] \leq Adv_{D_2, \text{shuffle}}^{\text{Corollary 5.1}}, \quad (4)$$

This is because the access enclave uses the output of shuffle enclave to fetch the data for each request. The access enclave runs the square-root ORAM algorithm which selects a random address in each request. Hence, the advantage of the distinguisher D_2 depends on the correctness of the permuted output array from shuffle enclave.

For Game_3 , a distinguisher D_3 reduces the security of π to PRP security such that:

$$Pr[\text{Game}_2 = 1] - Pr[\text{Game}_3 = 1] \leq Adv_{D_3, \pi}^{\text{PRP}}(\lambda), \quad (5)$$

Also, we have,

$$Pr[\text{Game}_3 = 1] = Pr[\text{Exp}_{\mathcal{A}_{\text{adv}}, \mathcal{E}}^{\text{PRO-ORAM}}(\lambda, 0) = 1], \quad (6)$$

From 2, 3, 4, 5, 6 we get:

$$Pr[\text{Exp}_{\mathcal{A}_{\text{adv}}, \mathcal{E}}^{\text{PRO-ORAM}}(\lambda, 1) = 1] - Pr[\text{Exp}_{\mathcal{A}_{\text{adv}}, \mathcal{E}}^{\text{PRO-ORAM}}(\lambda, 0) = 1] \leq \quad (7)$$

$$Adv_{D_1, \mathcal{E}}^{\text{IND-CPA}}(\lambda) + Adv_{D_2, \text{shuffle}}^{\text{Corollary 5.1}} + Adv_{D_3, \pi}^{\text{PRP}}(\lambda)$$

The $Adv_{D_2, \text{shuffle}}^{\text{Corollary 5.1}}$ cannot be greater than negl as it would break the security of the underlying Melbourne Shuffle algorithm stated in Lemma 5.1. With this, we prove that the advantage of an adaptive adversary in distinguishing the access patterns induced by PRO-ORAM from random is negligible. Therefore, PRO-ORAM guarantees read-only obliviousness.

The DUSTER Attack: Tor Onion Service Attribution Based on Flow Watermarking with Track Hiding

Alfonso Iacovazzi¹, Daniel Frassinelli², and Yuval Elovici^{1,3}

¹ST Engineering-SUTD Cyber Security Laboratory – Singapore University of Technology and Design, Singapore

²CISPA Helmholtz Center for Information Security, Saarland University, Germany

³Department of Information Systems Engineering, Ben-Gurion University, Israel

Abstract

Tor is a distributed network composed of volunteer relays which is designed to preserve the sender-receiver anonymity of communications on the Internet. Despite the use of the onion routing paradigm, Tor is vulnerable to traffic analysis attacks. In this paper we present DUSTER, an active traffic analysis attack based on flow watermarking that exploits a vulnerability in Tor’s congestion control mechanism in order to link a Tor onion service with its real IP address. The proposed watermarking system embeds a watermark at the destination of a Tor circuit which is propagated throughout the Tor network and can be detected by our modified Tor relays in the proximity of the onion service. Furthermore, upon detection the watermark is cancelled so that the target onion service remains unaware of its presence. We performed a set of experiments over the real Tor network in order to evaluate the feasibility of this attack. Our results show that true positive rates above 94% and false positive rates below 0.05% can be easily obtained. Finally we discuss a solution to mitigate this and other traffic analysis attacks which exploit Tor’s congestion control.

1 Introduction

Anonymous networks are utilised by Internet users to privately surf the Web without being traced by a third party. Typically, when an Internet communication is intermediated by such anonymous network, any entity observing the traffic at a single point along the communication path cannot identify the communicating peers. Tor is the most popular anonymous network currently in use, and it is widely used to circumvent any kind of censorship or network activity monitoring.

Tor is based on the onion routing paradigm in which privacy is guaranteed by encapsulating messages into packets protected by several layers of encryption and transferring these packets through a chain of relays [1]. Tor is an open-source project in which the relays constituting the infrastructure of the network are voluntarily made available by Tor users, and are

distributed worldwide. In addition, Tor enables users to create “onion services” which allow them to offer content and/or services while preserving their anonymity. Onion services can only be reached by peers connected to the Tor network, and are identified only by their “onion address.” Onion services ensure sender-receiver anonymity as the server and client can only communicate through the Tor network.

Related works

Tor has been investigated by researchers and attackers over the past decades for the purpose of identifying vulnerabilities and/or improving its security and privacy aspects. The most relevant attacks against Tor fall into two categories: (i) denial of service (DoS) attacks, and (ii) de-anonymization attacks. In this paper we focus on the latter category.

DoS attacks are well known attacks usually investigated in traditional Internet. When targeting Tor, the main goal of these attacks is to exhaust resources (bandwidth, memory, etc.) of Tor relays or onion services in order to make the Tor network unavailable to its users [3, 6, 11, 15]. One example is the “Sniper” attack proposed by Jansen *et al.* which exploits a vulnerability in Tor’s congestion control system to anonymously and selectively disable arbitrary Tor relays. In the attack, a malicious client continuously sends SENDME cells to an exit relay without retrieving the returning data cells: this causes the relay to buffer those cells until exhausting its resources [11]. Another example is the “CellFlood” attack proposed by Barbera *et al.* in which the attacker overloads a target Tor relay by sending an excessive number of circuit CREATE requests, which leads the relay to start discarding the additional CREATE requests, even those originated from honest Tor clients [3].

In the second attack category we find de-anonymization attacks. In these attacks the aim of the attacker is to actively or passively analyse the network (traffic patterns, addresses, etc.) in order to breach the sender-receiver anonymity. Several attacks based on passive traffic analysis (TA) techniques have already been demonstrated to correlate sender and receiver

traffic and be able to uncover onion services [16, 18]. Fingerprinting techniques are a type of robust passive TA attack in which the adversary uses machine learning algorithms to fingerprint the traffic coming from a website [22] or an onion service [12, 17, 19]. As example, Kwon *et al.* demonstrate how a passive adversary can easily detect the presence of an onion service via “circuit fingerprinting.” Furthermore they apply website fingerprinting on the circuit traffic to infer which onion service is being visited.

In contrast to passive TA attacks, active TA attacks have been shown to be much more powerful in breaching the sender-receiver anonymity [4, 5, 14, 20, 21]. For example, in the attack proposed by Ling *et al.*, compromised exit relays pseudo randomly delay the cells of normal circuits; the added noise is detected by compromised guard relays which de-anonymize the clients [14]. Pappas *et al.* propose the “Packet spinning” attack which allows malicious Tor relays to overload legitimate relays: in this way the attacker increases the probability that its malicious relays are selected by the clients to build their circuits [20]. An approach similar to the Packet spinning is proposed by Borisov *et al.* in which malicious relays perform selective DoS attacks on Tor circuits; this attack generates traffic patterns which can be easily identified by other malicious relays [5]. The attack investigated by Biryukov *et al.* allows to de-anonymize the entry guard of an onion service by forcing it to connect to a rendezvous and middle relay controlled by the attacker [4]. Finally, Rochet and Pereira exploits Tor’s lax dropping policies to watermark a rendezvous circuit by sending numerous padding cells which are silently dropped by the targeted onion service [21].

The presented attacks, despite being successful, presents some limitations. First of all, although some DoS-based attacks can be very successful in selectively tracking/isolating a Tor peer, they are easy to detect by legitimate users and are therefore not suitable for stealthy tracking [5, 20]. Following, TA based attacks often require the attacker to control multiple relays of the same circuit (e.g. guard and exit), which is very unlikely in the current Tor implementation [4, 14]. Finally, most attacks were based on vulnerabilities which have been fixed by the Tor community over the years [5, 11].

Contribution

In this paper we show the feasibility of a new active TA attack against Tor which can be used to compromise onion services anonymity. The attack belongs to the family of network flow watermarking techniques in which the attacker modifies statistical features of the traffic to embed a watermark into the communication [8]. The proposed attack differs from previous works as it allows the attacker to cancel the tracks of the watermark once it has been detected, making it unnoticeable to the onion service – hence the name DUSTER. The DUSTER attack embeds a watermark into a Tor stream by exploiting Tor’s congestion control system. When detected by one of

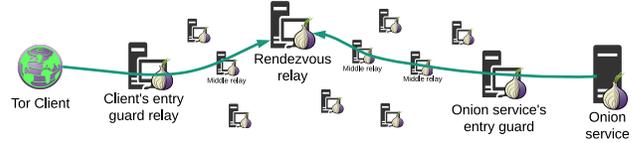


Figure 1: Tor rendezvous circuit.

the relays controlled by the attacker, the watermark is used to uncover the real IP address of the onion service.

The DUSTER attack has several advantages compared to traditional active traffic correlation attacks with the same privileged attacker position. Unlike the cell counter attack proposed by Ling *et al.* [14], which target only standard circuits, DUSTER works on both standard and rendezvous circuits, is hidden from the tracked endpoint, and does not affect the network performance. These last two features also make DUSTER stronger than the padding cell attack proposed by Rochet and Pereira [21]; in fact, their attack can be detected easily by the targeted onion service and requires additional traffic to be sent, which may affect the network performance. In addition, unlike the attack proposed by Biryukov *et al.* [4], the DUSTER attack does not require the attacker to control the rendezvous relay.

To summarise, the main novelties in DUSTER are:

- it exploits a vulnerability in Tor’s congestion control mechanism;
- it is hidden from the target endpoint;
- it applies to both standard and rendezvous circuits;
- it is lightweight and does not affect network performance;
- it works on up-to-date Tor implementations.

Paper organisation

The rest of the paper is organised as follows. An overview of the Tor network, its congestion control mechanism, and onion services is provided in Section 2. In Section 3 we describe the attack and the rules used to embed, detect, and cancel the watermark. The experimental evaluation is provided in Section 4. A solution to mitigate the attack is described in Section 5, which is followed by the conclusion in Section 6.

2 Tor, onion services, and congestion control

In this section we provide an overview of Tor’s key concepts useful for the understanding of the paper. Readers interested in more details may refer to the official project page for complete specifications [1].

In the Tor network, peers communicate via Tor *circuits*. A Tor circuit is a virtual path that traverses the Tor network and

is obtained by having the traffic routed via a series of Tor relays. Each Tor relay can be seen as a proxy which hides the IP addresses of the communication endpoints and applies encryption/decryption on the traversing data units according to the onion routing paradigm.

There are two types of circuits in Tor. First, standard circuits are used by Tor clients (e.g., Tor Browser) to communicate with Internet services outside of the Tor network (e.g., google.com), and they typically traverse 3 different relays. These relays are chosen by the client and are of three types: *guard*, *middle*, and *exit* relays. This categorisation is provided by the Tor consensus, a document containing all the information about the relays, compiled by the directory authorities and accessible to all clients. Each relay is given a flag on the basis of various metrics such as bandwidth, up-time, stability, configuration, etc. These flags are used by clients to select the relays for their circuits according to Tor’s circuit creation rules; For example, the first relay of a circuit must always have the guard flag.

The second type of Tor circuits are the “rendezvous” circuits. These circuits are used by Tor clients to communicate with Tor onion services. Apart for the circuit creation, which has no relevance for the attack discussed in this paper, the main difference between rendezvous and standard circuits is that the number of traversed relays is 6 instead of 3. In a rendezvous circuit, both the client and the onion service open a (respectively) two and three hop circuit towards a previously agreed relay called *rendezvous* relay. This relay acts as a bridge that interconnects the two independent circuits. This means that the client communicates to the rendezvous relay via a circuit composed of a guard and middle relay; The server (onion service) uses a three hop circuit consisting of a guard and two middle relays to communicate to the same rendezvous relay. Figure 1 presents an example of a rendezvous circuit.

Packets exchanged between peers in Tor are organised in fixed-size “cells” and grouped in logically separated “streams.” All streams sharing both sender and receiver are multiplexed into the same circuit and are processed according to the onion paradigm in which each traversed relay adds or removes an encryption layer. In a rendezvous circuit, the rendezvous relay has the role of routing the cells coming from one circuit to the other (with related encryption/decryption). In Tor jargon, cells travelling from one endpoint to the rendezvous relay are referred to as “OUT cells,” while cells moving in the opposite direction are referred to as “IN cells.”

Finally, to control the network congestion, Tor uses an end-to-end congestion avoidance mechanism based on dedicated SENDME cells. The congestion control rule is that an endpoint transmits one SENDME cell after receiving 50 data cells per stream and one SENDME cell after receiving 100 data cells per circuit. On the opposite, an endpoint can transmit a data cell if and only if there are less than 500 data cells per stream and 1000 data cells per circuit that have not yet been ac-

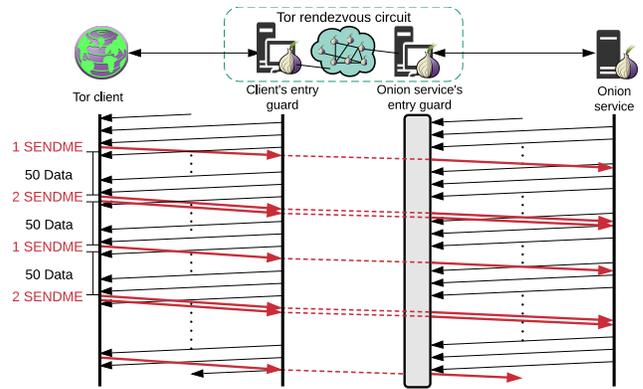


Figure 2: Traditional Tor cell exchange.

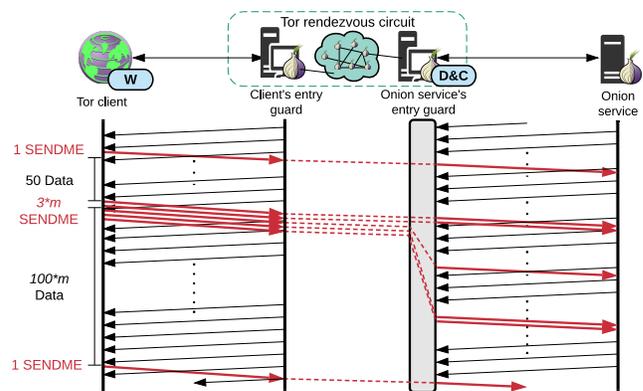


Figure 3: Watermarked Tor cell exchange.

knowledged. In a circuit with a single stream, assuming a continuous flow of data from the sender to the receiver, this results in a pattern of 1-2-1-2- SENDME cells from the receiver to the sender (Figure 2). In a rendezvous circuit, the SENDME and data cells are transferred between the client and the onion service, they are always encrypted along the path, and none of the traversed relays can distinguish between SENDME and data cells.

For simplicity, in the following we often refer to a SENDME triplet which consists of a set of stream-circuit-stream SENDME cells which acknowledge 100 data cells.

3 Watermarking system

We consider a scenario in which an adversary is interested in linking as many onion services as possible to their real IP addresses. For this purpose, the attacker creates a system to crawl the dark Web. The system is composed of (i) a watermark, which is a Tor client modified to inject a watermark into its incoming traffic, and (ii) multiple detectors, which are

Tor guard relays controlled by the adversary and modified to search for the watermark and, if detected, to cancel it.

The client builds a list of onion addresses and uses it to crawl all corresponding services. After establishing a rendezvous circuit with an onion service, the client downloads some content from it (e.g., HTML documents, files, multimedia) and in parallel injects a watermark into the data stream. At the same time, the detectors passively analyse all circuits passing through them. If one of the detectors detects the watermark, it provides the IP address of the circuit endpoint to the attacker, and cancel the watermark. The attacker can associate the onion address contacted by the client to its real IP address discovered by the detector.

After an instance of download is completed, the client moves to the next onion address on its list. The list of onion addresses can be built in various ways: most onion services' addresses are publicly listed by Tor's directory services whereas others require an invitation and/or some sort of manual set-up. How to gather the list of onion addresses is outside the scope of this paper. Ghosh *et al.* offer an example of how to automatically discover and categorise onion services [7].

The core watermarking/detection functions are described in the following subsections and in Figure 3. For sake of clarity, the corresponding pseudo-code is provided in Appendix A.

3.1 Embedding process

We detail the watermarking process by considering a single instance of DUSTER activity in which the adversary's Tor client knows an onion address and establishes a circuit with the corresponding onion service. Once the rendezvous circuit has been established, the client requests some content from the onion service. This generally happens via a client-server protocol (e.g., HTTPS) that encodes the content length together with the response. Knowing the content length is required to determine whether the content is large enough to embed a watermark into the data stream. Given a content provided by the onion service, if its length L is greater than or equal to a predefined threshold L_{min} , the watermark can be embedded. The selection of the minimum content length L_{min} is based on a couple of system parameters and is discussed in Subsection 3.4.

Assuming a resource that meets the mentioned criteria is found, the client starts the watermarking process. Figure 4 depicts the finite state machine of the watermarking process. The transitions between states are described by means of connecting arrows. Each arrow is labelled with the events causing the transition (above the horizontal line) and the actions executed because of the transition (below the horizontal line). The watermarking process consist of three states: INIT, SLEEP, and QUIT states.

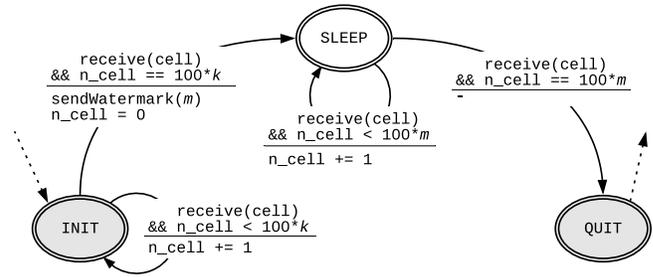


Figure 4: Finite-state machine of the watermarker.

INIT

The client starts the watermarking process at the INIT state. When in this state, the client strictly follows the Tor congestion protocol, counts the received data cells, and acts normally until it receives $100 \cdot k$ data cells. This initial waiting interval is needed to generate enough traffic that can be used by the detectors to validate the circuit and compute a set of bandwidth-dependent detection parameters. Upon receiving $100 \cdot k$ data cells the client executes the core of the attack consisting of the anticipation of m triplets of SENDME cells, which are sent in a single batch. Afterwards, the client transits to the SLEEP state.

SLEEP

The client stops sending SENDME cells until it receives $100 \cdot m$ data cells; any other cell is processed normally. Upon receiving $100 \cdot m$ data cells the client moves to the QUIT state.

QUIT

After receiving the $100 \cdot m$ data cells in the SLEEP state, the client quits the watermarking process and resumes the normal Tor SENDME process.

The watermarking process depends on two main variables: the number k of regular SENDME triplets that have to be sent by the client before embedding the watermark, and the watermark batch size m composing the watermark. The values given to these variables play an important role to the trade-off between accuracy and usability. We describe how we select the values of these variables in Subsection 3.4.

3.2 Detection and cancelling

A single detector consists of a modified Tor relay with guard capability that analyses every circuit passing through it. As any other relay in the circuit, the detector can only distinguish between IN and OUT cells (where IN cells are those going

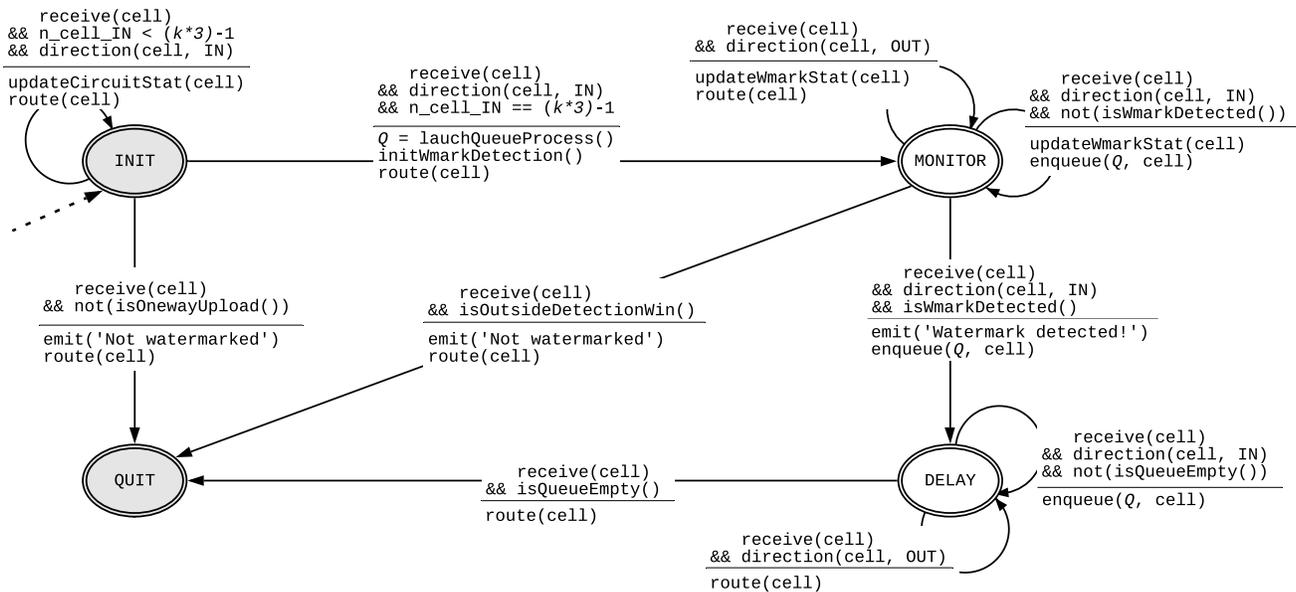


Figure 5: Finite-state machine of the detection and cancelling process.

towards the endpoint, and OUT cells those leaving the endpoint). When a new circuit is created, the detector checks that the IP address of the circuit’s origin does not belong to the known list of Tor relays retrieved from the consensus document; if it does not, the detector is certain of being the guard relay for that circuit.¹ Assuming a circuit meets the initial filtering criteria, the detector starts the detection and cancelling process. Figure 5 depicts a simplified version of the finite state machine of the process, which consists of four states: INIT, MONITOR, DELAY, and QUIT states.

INIT

When the detector is in this initial state, it performs two tasks. The first consists of verifying whether the traffic in the circuit is a one-way upload data transfer.² This is enforced by the *isOnewayUpload* function that checks if the circuit statistics respect a 3-IN:100-OUT cell ratio. This ratio is a distinctive feature of a one-way upload data transfer and permits to filter out all of the interactive or one-way download circuits which are the vast majority of the circuits in Tor and are not relevant for our detection. The one-way upload circuits are the only ones useful to analyse since the adversary’s Tor client is supposed to watermark only those circuits encapsulating onion

¹The detection process works even if the detector is a middle relay. However, the resulting IP address would point to the next relay in the circuit. This information can be used to pinpoint specific guard relays. If the onion service uses a bridge to connect to Tor, DUSTER will obtain the bridge’s IP address, and this occurs in 2-3% of all circuits.

²We use the term “one-way upload/download” to indicate those circuits in which the traffic is highly unbalanced in favour of one of the two directions (upload or download, respectively).

content transfer (i.e., content flowing from the onion server to the client).

The second task is the computation of few bandwidth-dependant statistics required for the watermark detection. These statistics are updated by the *updateStatistics* function and consist of (i) the average, minimum, and maximum number of IN and OUT cells per second, and (ii) the average, minimum, and maximum number of IN and OUT cell batch sizes.

The detector remains in this state until $3 \cdot k$ IN cells are detected. Upon receiving the $(3 \cdot k)$ -th IN cell, the detector starts the “core” detection process and calls the *initWmarkDetection* function which uses the statistics collected during the INIT state to compute the following detection parameters: (i) the observation time window d_{win} corresponding to the duration of the interval within which to look for the watermark, (ii) the minimum number $w_{th} \leq (3 \cdot m)$ of IN cells that are expected to be received for a successful detection, and (iii) the descriptors of the Markov model \mathcal{M} characterising the IN/OUT cell alternating process.

Following, the detector invokes the *launchQueueProcess* function. The function launches a process which (i) creates a queue where the IN cells are temporarily buffered and (ii) emits the IN cells according to the model \mathcal{M} in order to emulate the correct Tor IN/OUT cell timing process. This ensures that the SENDME cells belonging to the watermark batch are properly relayed (refer to Figure 3 which depicts the watermarking and cancelling process). In this way the watermark is cancelled, leaving the onion service unaware of the tracking process. Afterwards the detector moves to the MONITOR state.

MONITOR

In this phase the detector actively searches the watermark. It does this by analysing the circuit for an observation time window of duration d_{win} : if at least $w_{th} \leq 3 \cdot m$ IN cells are received within d_{win} , the circuit is considered watermarked. During this analysis, any received IN cell is not directly relayed but is instead forwarded to the queue process; the process takes care of the IN cells and emits them according to the model \mathcal{M} . To do that, the queue process needs to also receive information about the OUT cells routed by the relay. Upon receiving $3 \cdot m$ IN cells or at the end of the observation time window, if the watermark has been detected the detector moves to the DELAY state, otherwise it moves to the QUIT state.

DELAY

This state is coupled to the SLEEP state of the watermarker, and it can be reached only if the circuit has been labelled as watermarked. During this phase the onion service keeps sending OUT cells regularly. The OUT cells are observed by the detector and are used by the queue process to establish the timing of relaying the queued IN cells. Any additional IN cell that may arrive is still appended to the queue, since the IN cells must be forwarded in the order they come. All of the OUT cells are directly forwarded to the client. When all of the queued IN cells have been transmitted, the detector moves to the QUIT state.

QUIT

This is the final state and can be reached from both the MONITOR and DELAY states. If coming from the MONITOR state, the detector flushes eventually queued cells and destroys the queue process. If coming from the DELAY state, the detector destroys the already empty queue process. Afterwards it resumes routing cells normally.

As for the watermarking process, the detection depends on three variables: the duration of the observation time window d_{win} , the minimum number of IN cells required to detect the watermark w_{th} , and the set of descriptors of the Markov model \mathcal{M} characterising the IN/OUT cell process. We describe how we select the values of these variables in Subsection 3.4.

3.3 Attack reasoning

There are some vulnerabilities in the Tor protocol which make this attack possible. On the client side, the core of the watermarking algorithm is the anticipated transmission of a batch of SENDME cells. This is possible as Tor does not perform any

consistency or ordering control on the SENDME cells. In addition, the payload of a SENDME cell is composed by all zeros. This allows a malicious Tor client downloading a resource to craft and send all of the required SENDME cells in advance. The DUSTER attack exploits this vulnerability to anticipate a small batch of SENDME cells in a predefined manner: this generates a pattern which is propagated throughout the network with little variance, due to Tor's relay mechanism. For this reason the watermark can be identified by the detectors.

On the server side the SENDME cells are only used to acknowledge the correct reception of the data cells, and they are not authenticated.

Due to past vulnerabilities, the Tor community limited the maximum number of non-acknowledged data cells to 500 per stream and 1000 per circuit. If an onion service receives a SENDME cell acknowledging more than 500 data cells per stream or 1000 per circuit, it notices a protocol violation and drops the circuit. This means that if our *queueProcess* makes sure to never exceed this limit, the onion service won't drop the circuit or notice the attack. At the same time, if none of the detectors is the onion service's entry guard, the watermark won't be detected and cancelled. This implies the onion service will detect the SENDME anomaly and drop the circuit. The early interruption can be detected by the watermarker which can then move to the next onion address in the list.

3.4 System parameters

In this subsection we describe how the watermarking and detection parameters have been selected.

- Watermark batch size m . Number of SENDME triplets composing the watermark. A triplet is made of 3 SENDME cells (stream, circuit, and stream cells) which acknowledge 100 data cells received in the stream.³ The value of m can be tuned in order to achieve the desired trade-off between accuracy and detection time. In addition, the lower the value of m , the lower is the detection rate but the higher is the probability of finding a resource exceeding the minimum size required to embed the watermark. At the same time, a low value of m corresponds to a shorter interval during which the watermarker interferes with the circuit. We use m as a free parameter and we examine it in Section 4.
- Number k of monitored triplets. k represents the number of SENDME triplets that must be sent by the client and seen by the detector before running the "core" watermark embedding and detection, respectively. This waiting interval is necessary as it gives time to the detector to (i) recognise and filter out all of the circuits that do not convey a one-way upload data transfer, and (ii) collect the

³For simplicity we opted to always work in triplets of SENDME cells, which can be easily mapped to 100 data cells in the opposite direction. However one can choose any number of SENDME cells for both k and m .

circuit statistics needed by the detection and cancelling phase. The value of k can be tuned to achieve the desired trade-off between the accuracy of the collected statistics and the probability of finding a resource exceeding the minimum size required to embed the watermark. The greater the value of k the more accurate are the collected statistics, but the lower is the probability of finding a resource exceeding the minimum size to embed the watermark.

- Minimum content length L_{min} . Minimum content length required to embed the watermark onto the corresponding circuit. The value is computed as $L_{min} = 100 \cdot (m + k) \cdot L_c$ where L_c is the cell's payload length, that is $L_c = 498$ bytes.
- Detection size w_{th} . Minimum number of IN cells that must be received within the interval d_{win} to label the circuit as watermarked. w_{th} must be lower than or equal to $3 \cdot m$, and its value can be tuned in order to achieve the desired trade-off between true positive and false positive rates. The lower w_{th} , the higher the true positive and false positive rates.
- Duration of the observation time window d_{win} . Duration of the time window to search for the watermark. As for w_{th} , d_{win} can be tuned to achieve the desired trade-off between true positive and false positive rates. The higher d_{win} , the higher the true positive and false positive rates.
- Markov model \mathcal{M} . The model consists of an empirically computed Markov process that describes the probability of emitting the first IN cell from the queue, given that (i) the queue is not empty, (ii) b OUT cells have been routed (with $0 \leq b \leq 500$), and (iii) a time interval δ has elapsed since the last OUT cell. In our tests, a base model was first pre-computed from the IN/OUT cell statistics collected by a Tor guard relay over two days of logging at 1+Mbps. Then, during the INIT state of the detection process, each detector updates its model by tweaking the probabilities based on the updated traffic statistics of the analysed circuit.

4 System evaluation

We tested the attack on the real Tor network by setting up a guard relay on Amazon Web Services (AWS), an HTTPS onion service on a dedicated server, and a Tor client on a personal computer. We used an up-to-date Tor implementation (Tor version 0.3.2.10). The implementation was modified to perform the watermark detection on the entry guard and the watermark embedding on the client. For testing purposes, we configured our onion service to always select our testing relay as its entry guard.

A single test consisted of the client performing an HTTPS file download from the onion service with the detector trying to infer the presence of the watermark. Depending on the type of test, the client did or did not embed the watermark during the transfer. To ensure that different relays for circuit building were always chosen, we rebooted both the onion service and client after each test. Thanks to the worldwide distribution of AWS, we also moved our relay over three different locations during our experiments.

4.1 Numerical results

The plots in Figures 6–11 present the true positive (TPR – watermark present and correctly detected) and false positive (FPR – watermark detected although not present) rates obtained from our experiments. The results are shown by varying the parameters m , d_{win} , and w_{th} . Each value in the TPR plots is computed over 200 instances of experiments with the watermarker activated, while each value in the FPR plots is computed over 200 instances with the watermarker deactivated.

Experiments were carried out by setting $k = 15$, which entails that at least 1500 data cells (about 750 KB) are to be transferred from the onion service to the client before starting the watermarking process. Any variation of this parameter does not have an impact on the accuracy of the detection process.

In greater detail, the results in Figures 6 and 7 show the trend of TPR and FPR as a function of the ratio $u = \frac{d_{win}}{\phi(m)}$ between the observation window and the estimated time $\phi(m)$ to receive m IN cell triplets, providing different curves for different values of m . The estimation of $\phi(m)$ is based on the statistics collected from the detector during the INIT phase. As expected, both TPR and FPR increase when increasing the observation window; results show that for values of u between 0.125 and 0.5 we can get a good trade-off between TPR and FPR.

Similarly, the results in Figures 8–11 show the trend of TPR and FPR as a function of the ratio $t = \frac{w_{th}}{3 \cdot m}$ between detection size and watermark size, for two values of the observation window ($d_{win} = 0.25 \cdot \phi(m)$ and $0.5 \cdot \phi(m)$, respectively) and providing different curves for different values of m .

We observe that the TPR decreases when increasing the watermark detection size; it falls below 0.75 for t greater than 0.4, with $d_{win} = 0.25 \cdot \phi(m)$ (Figure 8). At the same time, we obtain FPR values close to zero when t is equal to or greater than 0.4. For a wider observation window ($d_{win} = 0.5 \cdot \phi(m)$) the TPR decreasing trends are slightly slower (Figure 10).

Observing the results as a whole, they demonstrate the strength of the DUSTER attack. This is also confirmed by the ROC function in Figure 12 in which the plotted ROC curve is close to the upper left corner of the graph. The ROC curves are obtained by varying u in the range $[0.125; 2]$ and w_{th} in $[0.2; 1]$, for different values of m . Considering the best

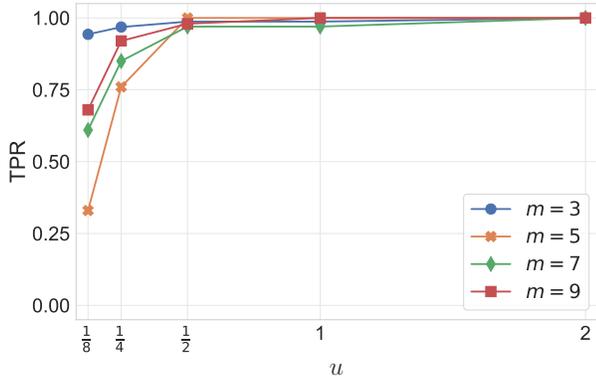


Figure 6: TPR as a function of d_{win} for different values of m and with $t = 0.4$.

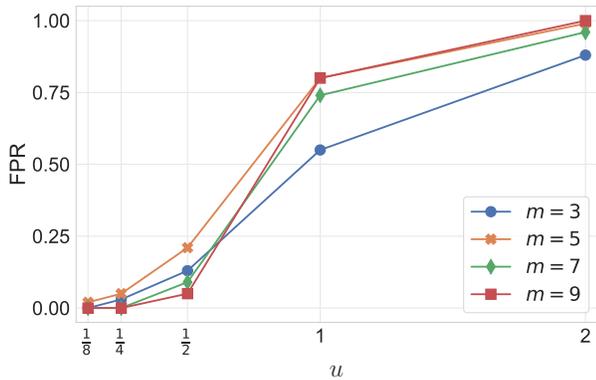


Figure 7: FPR as a function of d_{win} for different values of m and with $t = 0.4$.

combination of variables, that is $m = 5$, $w_{th} = 9$, and $u = 0.5$, we are able to obtain TPR equal to 0.98 with FPR equal to 0.03. We can reduce the FPR to 0.01 by selecting $m = 7$, $w_{th} = 13$, and $u = 0.5$ while maintaining the TPR above 0.94. We do not report results for $m < 3$ as a minimum of $m = 3$ is required to distinguish the watermark from natural cell batches.

For completeness, we have also verified the FPR assuming a real deployment scenario in which the detector analyses each relayed circuit. For the purpose we allowed the detector to analyse each circuit passing through it for few hours; of the approximately $2.5 \cdot 10^5$ relayed circuits, none were incorrectly labelled as watermarked. In the experiment, the detector was instantiated with $m = 5$, $w_{th} = 9$, and $u = 0.5$.

Finally, to analyse the effect of the watermark cancelling process, Figure 13 presents a statistical comparison between watermarked and un-watermarked circuits as observed by the onion service based on $2 \cdot 10^5$ statistical samples. The plots show the joint histogram of (i) the SENDME batch size along

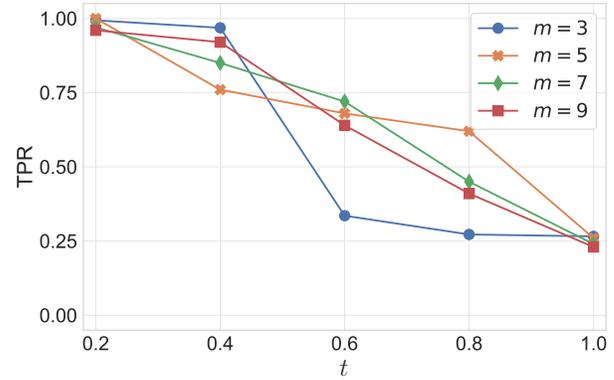


Figure 8: TPR as a function of w_{th} for different values of m and with $u = 0.25$.

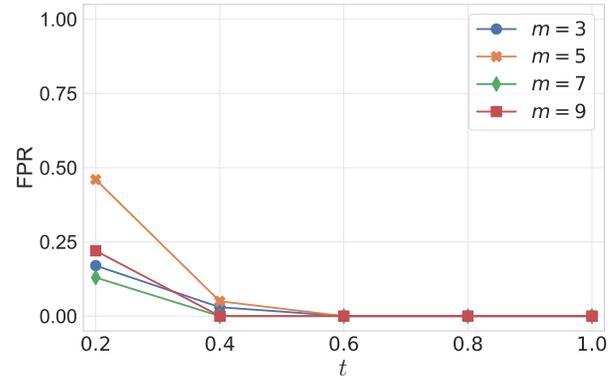


Figure 9: FPR as a function of w_{th} , for different values of m and with $u = 0.25$.

the ordinate and (ii) the distance between consecutive SENDME batches along the abscissa. The figure shows that the histograms of the watermarked circuits nearly match those of the un-watermarked circuits. This is confirmed by the Wasserstein distance⁴ computed between the two joint histograms, which is equal to $0.98 \cdot 10^{-5}$.

4.2 Attack applicability

The DUSTER attack, like the majority of known TA attacks against Tor, requires the attacker to control the guard relay of a circuit. The Tor community has put significant effort into mitigating these types of targeted attacks, mostly by applying strategies in which only few random entry guards are selected by each Tor endpoint, and they are kept for long periods of time. Nonetheless, the probability of being selected as a

⁴The Wasserstein distance returns a lower bound of zero in the case of two identical distributions and an upper bound of one in the case of disjoint distributions.

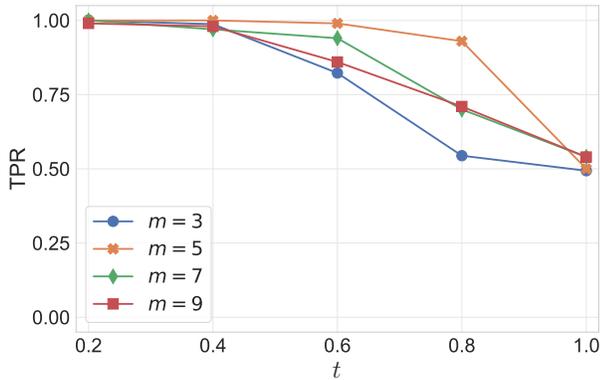


Figure 10: TPR as a function of w_{th} for different values of m and with $u = 0.5$.

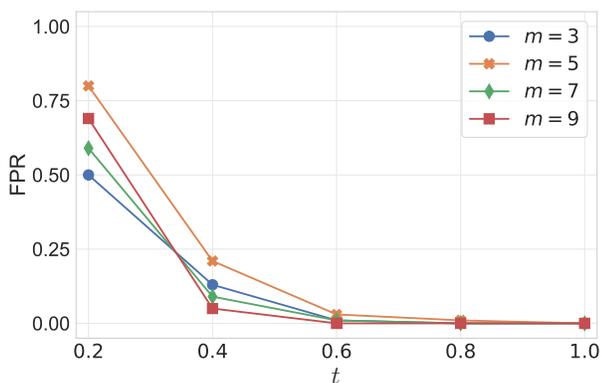


Figure 11: FPR as a function of w_{th} for different values of m and with $u = 0.5$.

guard by an endpoint is not negligible and this entails that the attack is successfully able to de-anonymize a number of onion services [13]. This number can be roughly estimated as the ratio $(\alpha \cdot D \cdot S)/G$ in which, given a generic instance of crawling, S is the number of onion services, D is the total guard bandwidth offered by the detectors controlled by the attacker, G is the total guard bandwidth offered by all of the entry guards, and α is the probability of the onion service being reached by the attacker’s crawler and providing at least a file to download of size greater than L_{min} . The values of S and G can be estimated from the metrics provided by Tor. For example, for values of $S = 10^5$, $G = 100 Gbps$, $D = 1 Gbps$, and $\alpha = 0.01$, the attacker would be able to de-anonymize about 10 onion services during an instance of crawling.⁵

We presented and analysed the DUSTER attack against Tor’s rendezvous circuits because it is, in our opinion, the

⁵The approximation assumes that the average bandwidth offered by a guard relay controlled by the attacker is the same as the bandwidth averagely offered by a regular guard relay.

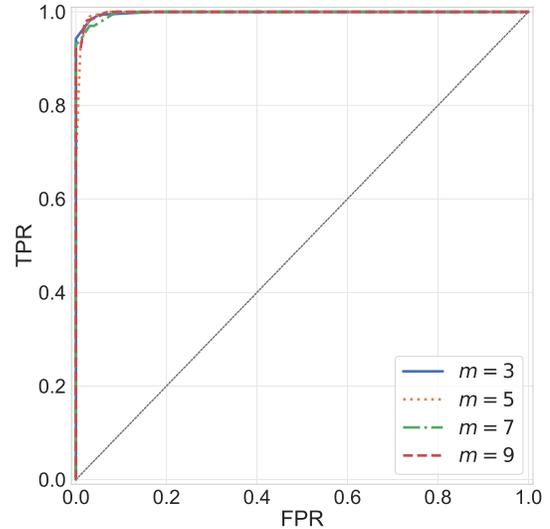


Figure 12: ROC curves for different values of m .

most compelling case. Nevertheless, the attack exploits a vulnerability of the congestion control mechanism; this means that it can also target standard circuits. In such cases, the purpose of the attack would be to correlate the flows of a Tor client with its visited websites. This scenario requires the watermarker implemented on the exit relay and the detector placed onto the client’s entry guard.

4.3 User safety and ethics

The Tor research board published a set of safety guidelines that researchers should respect to preserve legitimate users’ anonymity [2]. We relied on the guidelines to mount our experiments in order to (i) always work with only our modified onion service, client, and entry guard, (ii) test and prototype the implementation on the shadow simulator [10], (iii) actively modify only the traffic going to/from our onion service during active experiments, (iv) log and store only the minimal amount of data required for our analysis (in our case cells’ timing and direction). Finally, we disclosed the attack to Tor community before the publishing of this paper.

5 Attack mitigation

In the current Tor implementation, a stream is end-to-end encrypted using AES in counter block mode, which mitigates forgery and/or reordering attacks. However the congestion control system is predictable and can be exploited by a malicious client to anticipate `SENDME` cells and use them, as in DUSTER, to embed a detectable pattern in the traffic flow. This pattern can be analysed by detectors strategically placed in the network to de-anonymize the communicating parties.

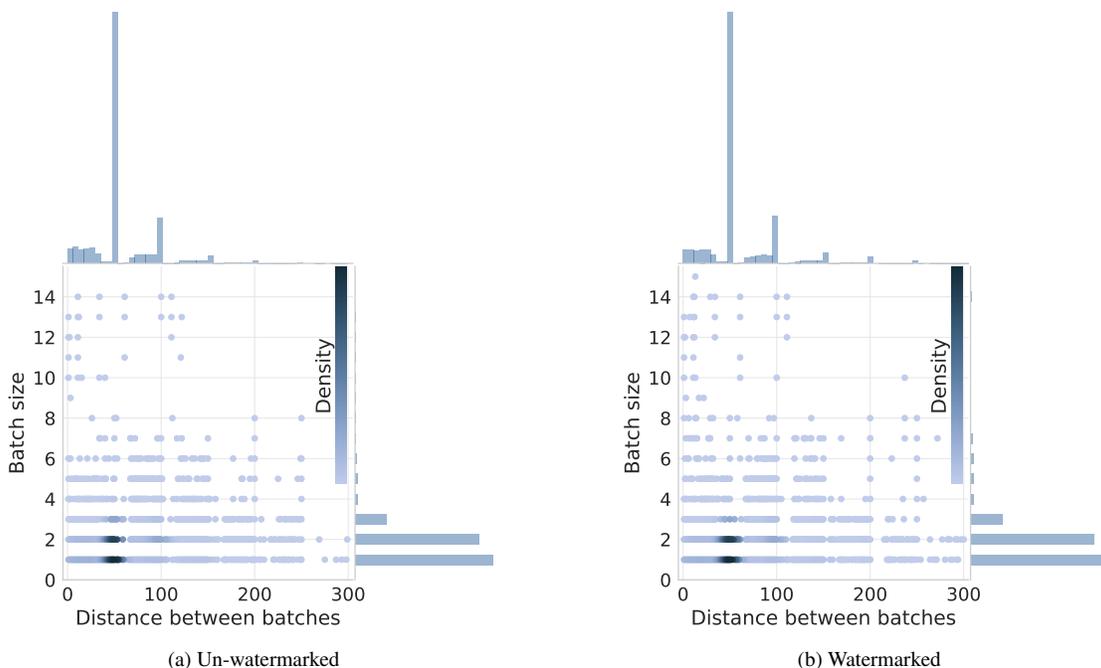


Figure 13: Tor traffic statistics.

Tor has had a history of attacks against its congestion control system. After each vulnerability disclosure the community implemented specific countermeasures. However, these countermeasures have never really solved the underlying problems and the congestion control system remains vulnerable. In fact, although we have implemented DUSTER proof of concept in Tor version 0.3.2.10 (March 2018), there was no countermeasure implemented until May 2019.

A possible solution to mitigate DUSTER and other attacks targeting Tor’s congestion control consists of a protocol ensuring that (i) the receiver computes a checksum of the acknowledged data cells and appends it to the payload of each SENDME cell, (ii) the sender verifies the checksum and drops the circuit if the checksum is incorrect, and (iii) both parties make use of some randomness to avoid reply attacks.

There has been a related proposal in late 2016 in which the authors suggested to integrate a SENDME authentication scheme in Tor [9]. As the referred document describes the protocol, requirement and guarantees, we won’t discuss it here. The Tor community released Tor version 0.4.1.1-alpha on May 22, 2019; this version incorporates the SENDME authentication scheme proposed in [9]. We inspected the implementation and verified that it is able to mitigate the DUSTER attack. However, the unauthenticated SENDME cells are expected yet to be supported until 2022 for backward compatibility, according to the documentation of the release. This entails that the vulnerability will be fully exploitable as long as unauthenticated SENDME cells will be accepted by Tor nodes. In fact, a

malicious client will always be able to force an onion service to downgrade to the unauthenticated congestion control mechanism.

6 Conclusions

In this paper we present DUSTER, an attack against Tor’s server-receiver anonymity. The attack is based on network flow watermarking and exploits a vulnerability in Tor’s congestion control mechanism. We implemented and evaluated the DUSTER attack on the real Tor network, and demonstrated that whenever a detector intercepts a watermarked circuit it can detect the watermark with a true positive rate up to 98%, while the false positive is close to zero; This enables the attacker to link the downloaded content with the IP address of the onion service. Further, as the watermark is cancelled by the detector, the onion service remains unaware of the tracking.

Finally we mention that, despite the partial countermeasures implemented over time, the underlying congestion control system remains vulnerable to the DUSTER attack and some possible other attacks. We stress that the community should deploy the solution proposed in [9] to once and for all mitigate attacks targeting Tor’s congestion control system.

References

- [1] Tor protocol specs. <https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt>.
- [2] Tor research safety board. <https://research.torproject.org/safetyboard/>.
- [3] Marco Valerio Barbera, Vasileios P Kemerlis, Vasilis Pappas, and Angelos D Keromytis. Cellflood: Attacking tor onion routers on the cheap. In *European Symposium on Research in Computer Security*, pages 664–681. Springer, 2013.
- [4] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. Trawling for tor hidden services: Detection, measurement, deanonymization. In *IEEE Symp. on Security and Privacy*, pages 80–94, 2013.
- [5] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 92–102. ACM, 2007.
- [6] Norman Danner, Sam Defabbia-Kane, Danny Krizanc, and Marc Liberatore. Effectiveness and detection of denial-of-service attacks in tor. *ACM Transactions on Information and System Security (TISSEC)*, 15(3):11, 2012.
- [7] Shalini Ghosh, Ariyam Das, Phil Porras, Vinod Yegneswaran, and Ashish Gehani. Automated categorization of onion sites for analyzing the darkweb ecosystem. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1793–1802. ACM, 2017.
- [8] Alfonso Iacovazzi and Yuval Elovici. Network flow watermarking: A survey. *IEEE Commun. Surveys & Tutorials*, 19(1):512–530, 2017.
- [9] Rob Jansen and Roger Dingledine. Authenticating sendme cells to mitigate bandwidth attack. <https://github.com/torproject/torspec/blob/master/proposals/289-authenticated-sendmes.txt>.
- [10] Rob Jansen and Nicholas Hooper. Shadow: Running tor in a box for accurate and efficient experimentation. Technical report, 2011.
- [11] Rob Jansen, Florian Tschorsch, Aaron Johnson, and Björn Scheuermann. The sniper attack: Anonymously deanonymizing and disabling the tor network. In *NDSS*, 2014.
- [12] Albert Kwon, Mashaal AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of tor hidden services. In *USENIX Security*, volume 20, 2015.
- [13] Z. Ling, J. Luo, K. Wu, and X. Fu. Protocol-level hidden server discovery. In *Proc. IEEE INFOCOM*, pages 1043–1051, 2013.
- [14] Zhen Ling, Junzhou Luo, Wei Yu, Xinwen Fu, Dong Xuan, and Weijia Jia. A new cell counter based attack against tor. In *Proc. ACM Conf. on Comput. and Commun. Security*, pages 578–589, 2009.
- [15] Nick Mathewson. Denial-of-service attacks in Tor: Taxonomy and defenses. Technical report, The Tor Project, 2015.
- [16] Milad Nasr, Amir Houmansadr, and Arya Mazumdar. Compressive traffic analysis: A new paradigm for scalable traffic analysis. In *Proc. ACM SIGSAC Conf. on Comput. and Commun. Security*, pages 2053–2069, 2017.
- [17] Rebekah Overdorf, Mark Juarez, Gunes Acar, Rachel Greenstadt, and Claudia Diaz. How unique is your .onion?: An analysis of the fingerprintability of tor onion services. In *Proc. ACM SIGSAC Conf. on Comput. and Commun. Security*, pages 2021–2036, 2017.
- [18] Lasse Overlier and Paul Syverson. Locating hidden servers. In *IEEE Symp. on Security and Privacy*, pages 15–pp, 2006.
- [19] Andriy Panchenko, Asya Mitseva, Martin Henze, Fabian Lanze, Klaus Wehrle, and Thomas Engel. Analysis of fingerprinting techniques for tor hidden services. In *Proc. on Workshop on Privacy in the Electron. Soc.*, pages 165–175, 2017.
- [20] Vasilis Pappas, Elias Athanasopoulos, Sotiris Ioannidis, and Evangelos P Markatos. Compromising anonymity using packet spinning. In *International Conference on Information Security*, pages 161–174. Springer, 2008.
- [21] Florentin Rochet and Olivier Pereira. Dropping on the edge: Flexibility and traffic confirmation in onion routing protocols. In *Proc. on Privacy Enhancing Technologies*, pages 27–46, 2018.
- [22] Tao Wang and Ian Goldberg. Improved website fingerprinting on tor. In *Proc. on Workshop on Privacy in the Electron. Soc.*, pages 201–212, 2013.

A Appendix

We present some simplified pseudo-code useful for the understanding of the paper. The code assumes each circuit has only one stream: in the real Tor network a circuit might have one or more data streams. Listing 1 and 2 present the use of Tor's SENDME cells and the client download process, respectively. Listing 3 describes how a Tor onion service sends data and validates SENDME cells. Listing 4 describes the DUSTER watermark which substitutes the default `tor_considerSendingSendme` showed in Listing 1.

Following, Listing 5 shows how a Tor guard relay routes cells. Listing 6 shows the DUSTER watermark detector which replaces the original `tor_routeIncomingCell` in Listing 5.

```
1 """
2 - circ: Tor circuit
3 - cell: Tor cell
4 """
5 def tor_considerSendingSendme(circ, cell):
6     if (circ.n_cell % 100) == 0:
7         tor_sendCircuitSendme(circ)
8     if (circ.stream.n_cell % 50) == 0:
9         tor_sendStreamSendme(circ.stream)
```

Listing 1: Tor SENDME handing function.

```
1 """
2 - addr: the onion service address.
3 - res: the resource to be fetched.
4 """
5 circ = tor_openRendezvousCircuit(addr)
6 circ.stream = tor_getResource(circ, res)
7 circ.n_cell = 0
8 circ.stream.n_cell = 0
9 while True:
10     cell = tor_getNextCell(circ.stream)
11     if tor_isEndCell(cell):
12         break
13     if tor_isDataCell(cell):
14         tor_commitPayload(cell)
15         circ.n_cell += 1
16         circ.stream.n_cell += 1
17         tor_considerSendingSendme(circ, cell)
18 tor_closeCircuit(circ)
```

Listing 2: Tor client resource download process.

```
1 circ = tor_acceptConnection()
2 circ.stream, req_res = tor_handleRequest()
3 bin_data = tor_fetchResource(req_res)
4 circ.cwin = 1000
5 circ.stream.cwin = 500
6 while True:
7     # Send data if congestion windows are not empty.
8     if circ.stream.cwin > 0 and circ.cwin > 0:
9         wr = tor_sendDataCell(circ.stream, bin_data)
10        bin_data = bin_data[wr:]
11        circ.cwin -= 1
12        circ.stream.cwin -= 1
13    # Send end-cell if transmission is completed.
14    if len(bin_data) == 0:
15        tor_sendEndCell(circ.stream)
16        break
17    # If a cell from the client has been received
18    if tor_isCellQueued():
19        cell = tor_getNextCell(circ.stream)
20        if tor_isCircuitSendmeCell(cell):
21            circ.cwin += 100
22        if tor_isStreamSendmeCell(cell):
23            circ.stream.cwin += 50
24    # Sanity checks to mitigate previous attacks.
25    if circ.cwin > 1000 or circ.stream.cwin > 500:
26        tor_print("Unexpected SENDME. Dropping circ.")
27        break
28 tor_closeCircuit(circ)
```

Listing 3: Onion service client handling.

```
1 """
2 - circ: Tor circuit
3 - cell: Tor cell
4 Duster parameters:
5 - {k, m}: attack variables.
6 - {INIT, SLEEP, QUIT}: Duster states.
7 """
8 def duster_considerSendingSendme(circ, cell):
9     duster = duster_getOrCreateNew(circ)
10    duster.n_cell += 1
11    # Default behaviour until K*100 received.
12    # Then send m*3 SENDME cells.
13    if duster.state == INIT:
14        tor_considerSendingSendme(circ, cell)
15        if duster.n_cell == k*100:
16            for i in range(M):
17                tor_sendStreamSendme(circ.stream)
18                tor_sendCircuitSendme(circ)
19                tor_sendStreamSendme(circ.stream)
20            duster.n_cell = 0
21            duster.state = SLEEP
22    # Wait m*100 data cells before moving to QUIT.
23    elif duster.state == SLEEP:
24        if duster.n_cell == m*100:
25            duster.state = QUIT
26    # Attack finished. Default Tor.
27    else:
28        assert(duster.state == QUIT)
29        tor_considerSendingSendme(circ, cell)
```

Listing 4: DUSTER watermark.

```

1 """
2 - circ: Source circuit.
3 - cell: Tor cell.
4 """
5 def tor_routeIncomingCell(circ, cell):
6     dest_circ = tor_getDestinationCircuit(circ)
7     tor_route(dest_circ, cell)

```

Listing 5: Onion service’s entry guard routing.

```

1 """
2 - circ: Source circuit.
3 - cell: Tor cell.
4 Duster parameters:
5 - {k, m}: attack variables.
6 - {INIT, MONITOR, DELAY, QUIT}: Duster states.
7 """
8 def duster_routeIncomingCell(circ, cell):
9     dest_circ = tor_getDestinationCircuit(circ)
10    duster = duster_getOrCreateNew(circ)
11    duster.updateCircuitStat(cell)
12    # Wait until 3*k IN-cells are seen.
13    # Meanwhile validate the circuit.
14    if duster.state == INIT:
15        if not duster.isOnewayUpload():
16            duster.state = QUIT
17            break
18        if duster.n_cell_IN == (3*k):
19            duster.state = MONITOR
20            duster.launchQueueProcess()
21            duster.initWmarkDetection()
22            tor_route(dest_circ, cell)
23    # Search for the watermark. Buffer IN cells
24    # and route OUT cells normally.
25    elif duster.state == MONITOR:
26        duster.updateWmarkStat(cell)
27        if tor_direction(cell, IN):
28            duster.enqueue(cell)
29        else:
30            tor_route(dest_circ, cell)
31        if duster.isWmarkDetected():
32            duster.print("WATERMARK DETECTED!!!")
33            duster.state = DELAY
34        elif duster.isOutsideDetectionWin(cell):
35            duster.state = QUIT
36            break
37    # Buffer IN cells until queue empties.
38    elif duster.state == DELAY:
39        if duster.isQueueEmpty():
40            duster.state = QUIT
41            break
42        elif tor_direction(cell, IN):
43            duster.enqueue(cell)
44        else:
45            tor_route(dest_circ, cell)
46    # Default Tor routing.
47    else:
48        assert(duster.state == QUIT)
49        tor_route(dest_circ, cell)

```

Listing 6: DUSTER watermark detector.

Talon: An Automated Framework for Cross-Device Tracking Detection

Konstantinos Solomos
FORTH, Greece
solomos@ics.forth.gr

Sotiris Ioannidis
FORTH, Greece
sotiris@ics.forth.gr

Panagiotis Ilija
Univ. of Illinois at Chicago, USA
pilia@uic.edu

Nicolas Kourtellis
Telefonica Reasearch, Spain
nicolas.kourtellis@telefonica.com

Abstract

Although digital advertising fuels much of today's free Web, it typically does so at the cost of online users' privacy, due to the continuous tracking and leakage of users' personal data. In search for new ways to optimize the effectiveness of ads, advertisers have introduced new advanced paradigms such as cross-device tracking (CDT), to monitor users' browsing on multiple devices and screens, and deliver (re)targeted ads in the most appropriate screen. Unfortunately, this practice leads to greater privacy concerns for the end-user.

Going beyond the state-of-the-art, we propose a novel methodology for detecting CDT and measuring the factors affecting its performance, in a repeatable and systematic way. This new methodology is based on emulating realistic browsing activity of end-users, from different devices, and thus triggering and detecting cross-device targeted ads. We design and build Talon¹, a CDT measurement framework that implements our methodology and allows experimentation with multiple parallel devices, experimental setups and settings. By employing Talon, we perform several critical experiments, and we are able to not only detect and measure CDT with average AUC score of 0.78-0.96, but also to provide significant insights about the behavior of CDT entities and the impact on users' privacy. In the hands of privacy researchers, policy makers and end-users, Talon can be an invaluable tool for raising awareness and increasing transparency on tracking practices used by the ad-ecosystem.

1 Introduction

Online advertising has become a driving force of the economy, with digital ad spending already surpassing the spending for TV-based advertising in 2017 [32], and expected to reach \$327 billion in 2019 [42]. This is because online advertising can be easily tailored to, and target specific audiences. In order to personalize ads, advertisers employ various tracking practices to collect user behavioral and browsing data.

¹<https://en.wikipedia.org/wiki/Talos>

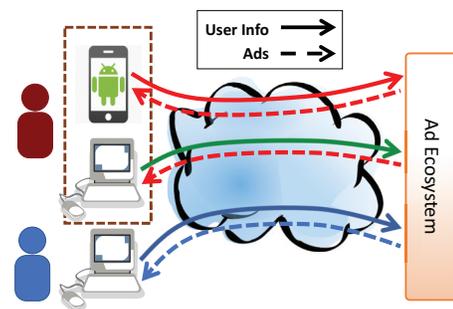


Figure 1: High level representation of cross-device tracking.

Until recently, the tracking of a user was confined to the physical boundary of each one of her devices. However, as users typically own multiple devices [2, 54], advertisers have started employing advanced targeting practices specifically designed to track and target users across **all** their devices. These efforts indicate a radical shift of the ad-targeting paradigm, from *device-centric* to *user-centric*. In this new paradigm, an advertiser tries to identify which devices (e.g., smartphone, tablet, laptop) belong to the same user, and then target her across all devices with ads related to her overall online behavior. Figure 1 illustrates a typical cross-device tracking (CDT) scenario, where a user is targeted with relevant ads in her second device (desktop), due to the behavior exhibited to the ad-ecosystem from her first device (mobile).

A recent FTC Staff Report [51] states that CDT can be deterministic or probabilistic, and companies engaging in such practices typically use a mixture of both techniques. Deterministic tracking utilizes 1st-party login services that require user authentication (e.g., Facebook, Twitter, Gmail). These 1st-party services often share information (e.g., a unique identifier) with 3rd-parties, enabling them to perform a more effective CDT. In the case of probabilistic CDT, there are no shared identifiers between the users' devices, and 3rd-parties attempt to identify which devices belong to the same user by considering network access data, common behavioral patterns in browsing history, etc. In fact, to understand the degree to which CDT trackers appear on the Web, we

measured their frequency of appearance on Alexa Top-10k websites: companies performing probabilistic CDT can be found in $\sim 27\%$ of the websites, and when also considering deterministic CDT, this coverage reaches $\sim 80\%$. Also, several advertising companies such as Criteo [16], Tapad [53], Drawbridge [20] etc., claim that they can track users across devices with very high accuracy (e.g., Drawbridge’s Cross-Device connected consumer graph is 97.3% accurate [19]).

In spite of its big impact on user privacy, apart from some empirical evidence about CDT, there is only a limited work investigating it. In the most close work to ours, Zimmeck et al. [56], designed an algorithm that correlates mobile and desktop devices into pairs by considering devices’ browsing history and IP addresses. While this approach shows that correlation of devices is possible when such data are available, it does not provide an approach for detecting and measuring CDT. In fact, to the best of our knowledge, there is no existing approach to audit the probabilistic CDT ecosystem and the factors that impact its performance on the Web. *Our work is the first to propose a novel methodology that enables auditing the CDT ecosystem in an automated and systematic way. In effect, our work takes the first and crucial step in understanding the inner workings of the CDT mechanics and measure different parameters that affect how it performs.*

The methodology proposed in this work is based on the following idea: we want to detect when CDT trackers successfully correlate a user’s devices, by identifying cross-device targeted behavioral ads they send, i.e., ads that are delivered on one device, but have been triggered because of the user’s browsing on a different device. In order to design this methodology, we first study browsing data of real users with multiple devices from [56] and extract topics of interest and other user behavioral patterns. Then, to make trackers correlate the different devices of the end-user and serve cross-device targeted ads, we employ artificially created personas with specific interests, to emulate realistic browsing activity across the user devices as extracted from the real data.

We build Talon, a novel framework that materializes our methodology in order to collect, categorize and analyze all the ads delivered to the different user devices, and evaluate with simple and advanced statistical methods the potential existence of CDT. Through a variety of experiments we are able to measure CDT with an average AUC of 0.78-0.96. Specifically, in the simplest experiment, where the user exhibits significant browsing activity mainly from the mobile device, the average value of AUC is 0.78 for the 10 different behavioral profiles used. When the user exhibits significant browsing activity from both devices (mobile and desktop), with a matching behavioral profile, we observe CDT with an average AUC of 0.83. In the case of visiting specifically chosen websites that employ multiple known CDT trackers, we achieve AUC score of 0.96. We also find that browsing in incognito can reduce the effect of CDT, but does not eliminate it, as trackers can perform device matching based

only on the current browsing session of the user, and not all her browsing history. Finally, we compare the data collected with our real user-driven artificial personas (such as CDT trackers found, types of ads detected, etc.) with corresponding distributions observed in the real user data from [56], offering a strong validation to the realistic design of Talon.

Overall, our main contributions in this work are:

- Design a novel, real data-driven methodology for detecting CDT by triggering behavioral cross-device targeted ads on one user device, according to specifically-crafted emulated personas, and then detecting those ads when delivered on a different device of the same user.
- Implement Talon, a practical framework for CDT measurements. Talon has been designed to provide scalability for fast deployment of multiple parallel device instances, to support various experimental setups, and to be easily extensible.
- Conduct a set of experiments for measuring the potential existence of CDT in different types of emulated users, with an average AUC score of 0.78-0.96, and investigate the various factors that affect its performance under different classes of experimental setups and configurations.

2 Related Work

The ad-industry continuously develops new mechanisms for making ads more relevant and effective. Such mechanisms include the delivery of contextual, targeted-behavioral, and retargeted ads. However, in order to serve such highly related ads, advertisers often employ questionable and privacy intrusive techniques for collecting user information. They typically apply techniques for tracking user visits across different websites, which allow them to reconstruct parts of the users’ browsing history. To that end, numerous works [48, 34, 39, 43, 17, 46, 45, 47, 44] investigate the various approaches employed by trackers, and propose protection mechanisms. Also, a large body of work investigates targeted behavioral advertising with regards to different levels of personalization, based on the type of information used to target the user [9, 5, 55], and its effectiveness [25, 35, 26, 12, 7, 29].

Some studies investigate CDT that utilizes technologies such as ultrasound and Bluetooth, and measure the prevalence of such approaches [40, 6, 33]. A study by Brookman et al. [10] provided initial insights about the prevalence of CDT on the web, identified 3rd-party CDT trackers and examined the transparency of the employed techniques.

Zimmeck et al. [56] conducted a small-scale exploratory study on CDT based on the observation of cross-device targeted ads in two paired devices using the same IP address (mobile and desktop) over the course of two months. Following this exploration, they collected real users’ browsing histories and device information and designed an algorithm

that correlates the devices into pairs. This approach shows that network information and browsing history can be used for correlating user devices, and thus potentially for CDT.

In general, research around CDT is still very limited; in fact, only [56, 10] initially studied some of its aspects, but without proving its actual existence or providing a methodology for detecting and measuring it. Our work builds on these early studies on CDT, as well as past studies on web tracking, and proposes a methodology that enables systematic investigation and measurements for detecting probabilistic CDT.

3 A methodology to measure CDT

The proposed methodology emulates realistic browsing activity of end-users across different devices, and collects and categorizes all ads delivered to these devices based on the intensity of the targeting. Finally, it compares these ads with baseline browsing activity to establish if CDT is present or not, at what level, and for which types of user interests.

3.1 Design Principle

In general, the CDT performed by the ad-ecosystem is a very complex process, with multiple parties involved, and a non-trivial task to dissect and understand. To infer its internal mechanics, we rely on probing the ecosystem with consistent and repeatable inputs (I), under specific experimental settings (V), allowing the ecosystem to process and use this input via transformations and modeling (F), and produce outputs we can measure on the receiving end (Y):

$$(I, V) \xrightarrow{F} Y$$

In this expression, the unknown F is the probabilistic modeling performed by CDT entities, allowing them to track users across their devices. Following this design principle, our methodology allows to push realistic input signals to the ad-ecosystem via website visits, and measure the ecosystem's output through the delivered ads, to demonstrate if F enabled the ecosystem to perform probabilistic CDT. An overview of our methodology is illustrated in Figure 2.

3.2 Design Overview

3.2.1 Input Signal (I)

To trigger CDT, we first need to inject to the ad-ecosystem some activity from a user's browsing behavior (I). This *input* can be visits (i) to pages of interest (e.g., travel, shopping), or (ii) to control pages of null interest (e.g., weather pages). Intuitively, the former can be used first to demonstrate particular behavior of a user from a given device (mobile), and the latter afterwards for collecting ads delivered as the *output* of the ecosystem (Y) due to I, to that device, or other device of the same user (desktop).

Persona Pages. We extract real users' interests from the dataset provided by Zimmeck et al. [56] and leverage an approach similar to Carrascosa et al. [12] to emulate browsing behavior according to specific web categories, and create multiple, carefully-crafted *personas* of different granularities. This design makes the methodology systematic and repeatable and produces realistic browsing traffic from scripted browsers. For each persona, our approach identifies a set of websites (dubbed as *persona pages*) that have, at the given time, active ad-campaigns. This "*training activity*" aims to drive CDT trackers into possible *device-pairing* between the user's two devices with high degree of confidence.

Control Pages. Following past works [12, 7], all devices in the system collect ads by visiting neutral websites that typically serve ads not related to their content, thus, reducing bias from possible behavioral ads delivered to specific type of websites. We refer to these websites as *control pages*. We detail the design of personas and control pages in § 4.1.

3.2.2 Experimental Setup (V)

No 1st-party logins. Since we focus on probabilistic CDT, we assume that the emulated user does not visit or log into any 1st-party service that employs deterministic CDT and thus, there is no common identifier (e.g., email address, social network ID) shared between the user's devices.

Devices, IP addresses & Activity. The approach we follow is based on triggering and identifying behavioral cross-device targeted ads, and specifically ads that appear on one of the user's devices, but have been triggered by the user's activity on a different device. For this trigger to be facilitated, the ad-ecosystem must be provided with hints that these two devices belong to the same user. Zimmeck et al. [56] suggest that in many cases, the devices' IP address is adequate for matching devices that belong to the same user. Also, according to relevant industrial teams [38, 4] more signals can be used, such as location, browsing, etc., for device matching.

Following these observations, our methodology requires a minimum of three different devices: one mobile device and two desktop computers, with two different public IP addresses. We assume that two devices (i.e., the mobile and one desktop) belong to the same user, and are connected to the same network. That is, these devices have the same public IP address, are active in the same geolocation as in a typical home network, and will be considered by the ad-ecosystem as producing traffic from the same user. The second desktop (i.e., *baseline PC*), which has a different IP address, is used for receiving a different flow of ads while replicating the browsing of the user's desktop (i.e., *paired PC*). This control instance is used for establishing a baseline set of ads to compare with the ads received by the user's paired PC.

CDT Direction. In principle, the design allows the investigation of both directions of CDT. That is, users may first browse on the mobile device, and then move to their desk-

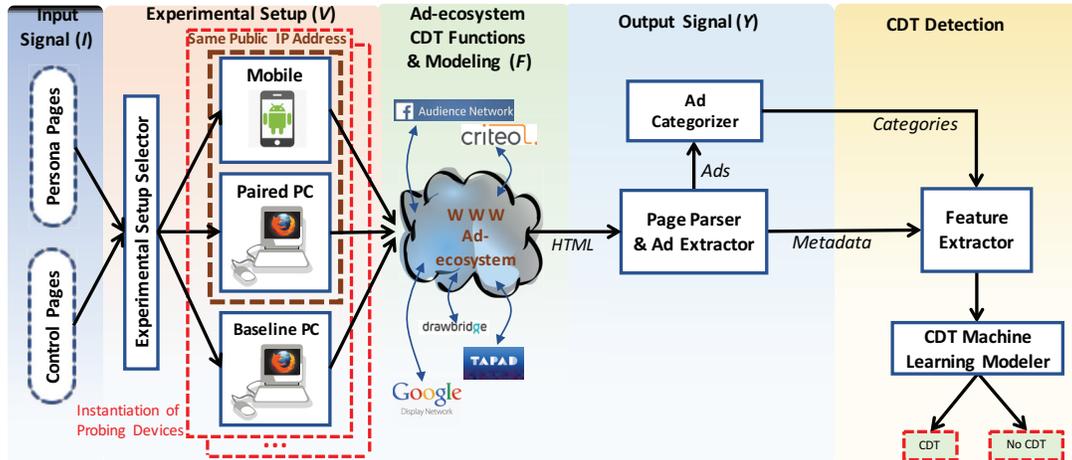


Figure 2: High level representation of methodology design principles and units for CDT measurements.

top, and vice versa. However, since ad-targeting companies such as AdBrain and Criteo support that the direction from mobile to desktop is more suitable for cross-device retargeting [49, 3, 15], in this work we focus on the mobile to desktop direction (*Mob* → *PC*). In essence, the mobile device performs a specifically instructed web browsing session to establish the persona, by visiting the set of *persona pages*, i.e., *training* phase; then, the two desktop computers perform web browsing, i.e., *testing* phase, where they visit the set of *control pages* and collect the delivered ads. The browsing performed by the desktops is synchronized by means of visiting the same pages and performing the exact same clicks.

3.2.3 Output Signal (*Y*)

In order to handle the Output Signal and transform it appropriately, we design and implement two different components: (i) Page Parser & Ad Extractor and (ii) Ad Categorizer. The first is responsible for the identification and extraction of ad elements inside the webpages. The module uses string matching techniques and a public list of common ad-domains (*Easylist* [21]) to identify the delivered ads. The second module assigns a keyword on each ad identified on the previous step, based on its type and content (e.g., “Online Shopping”, “Fashion”, “Recreation”, etc.). Using both modules, we store the ads delivered in all devices of our experimental setup along with their categories, as well as data related to the activity of the devices that attracted these ads.

3.2.4 CDT Detection

Comparing Signals. Various statistical methods can be used to associate the input signal *I* of persona browsing in the mobile device, with the output signal *Y* of ads delivered to the potentially paired-PC. For example, simple methods that perform similarity computation between the two signals in a

given dimensionality (e.g., Jaccard, Cosine) can be applied. These methods, as well as typical statistical techniques (e.g., permutation tests) capture only one dimension of each input/output signal and thus, might not be suitable for measuring with confidence the high complexity of the CDT signal. In this case, more advanced methods can be employed, such as Machine Learning techniques (ML) for classification of the signals as similar enough to match, or not. In our analysis, we mainly focus on ML to compute the likelihood of the two signals being the product of CDT, as it takes into consideration this multidimensionality in the feature space. We describe the modeling and methods used for ML in § 4.4.

4 Framework Implementation

A high level overview of our methodology, and its materialization by our framework Talon, is presented in Figure 2 and described in § 3. In the following, we provide more details about its building blocks, and argue for various design decisions taken while implementing this methodology into the fully-fledged automated system.

4.1 Input Signal: Control Pages & Personas

Persona Pages. A critical part of our methodology is the design and automatic building of realistic user personas. Each persona has a unique collection of visiting links, that form the set of *persona pages*. Since we do not know in advance which e-commerce sites are conducting cross-device ad-campaigns, we design a process to dynamically detect active persona pages of given interest categories. Our approach for persona generation is shown in Figure 3.

We first use the list of topics of Zimmeck et al. [56], that describe real user’s online interests. We perform a clustering based on the content of each interest and label the clusters appropriately (e.g., we group together: “Shopping” and

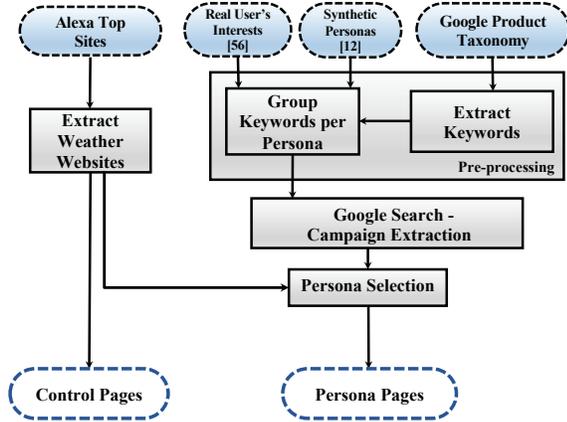


Figure 3: Persona design and automatic generation.

“Beauty and Fashion” under the label: “Shopping and Fashion”). Then, we use the persona categorization of Carrascosa et al. [12] for their top 50 personas, and select only those personas that describe similar interests with the previously formed list. For the resulting intersection of personas from the two lists, we iterate through the Google Product Taxonomy list [27] to obtain the related keywords for each one.

For increasing the probability to capture active ad-campaigns that can potentially deliver ads to the devices, we use Google Search as it reveals campaigns associated with products currently being advertised. That is, if a user searches for specific keywords (e.g., “men watches”), Google will display a set of results, including sponsored links for sites conducting campaigns for the terms searched. In this way, we use the keywords set for each persona, as extracted above, and transform them into search queries by appending common string patterns such as “buy”, “sell”, and “offers”. This process is repeated until between five and ten unique domains per persona are collected. If the procedure fails, no persona is formed.

As the effectiveness of a persona depends on the active ad-campaigns at the given time, in our experiments, we deploy personas in 10 categories related to shopping, traveling, etc. (full list shown in Table 3 in Appendix). With this procedure, we manage to design personas similar enough with real users, as well as with emulated users designed in previous works [12, 7, 8, 56].

Control Pages. For retrieving the delivered ads (after any type of browsing), we employ a set of webpages that contain: (i) easily identifiable ad-elements and (ii) a sufficient number of ads that remains consistent through time. These pages have neutral context and do not affect the behavioral profile of the device visiting them. For most of the experiments in § 5, we use a set of five popular weather websites² as control pages, similarly to [12]. We manually confirmed the neutral-

²accuweather.com, wunderground.com, weather.com, weather-forecast.com, metcheck.com

ity of these pages, by observing no contextual ads delivered to them. When visiting the set of control pages, our methods extract and categorize all the ads received, in order to identify those that have been potentially resulted from CDT.

4.2 Experimental System Setup

The experimental setup contains different types of units, connected together for replicating browsing activity on multiple devices. Typically, CDT is applied on two or more devices that belong to the same user, such as a desktop and a mobile device. Thus, the system contains emulated instances of both types, controlled by a number of experimental parameters.

Devices & Automation. The desktop devices are built on top of the web measurement framework OpenWPM [22]. This platform enables launching instances of the Firefox browser, performs realistic browsing with scrolling, sleeps and clicks, and collects a wide range of measurements in every browsing session. It is also capable of storing the browser’s data (cookies, local cache, temporary files) and exports a browser profile after the end of a browsing session, which can be loaded in a future session. With these options, we can perform *stateful* experiments, as a typical user’s web browser that stores all the data through time, or *stateless* experiments to emulate browsing in incognito mode.

For the mobile device, we use the official Android Emulator [28], as well as the Appium UI Automator [50] for the automation of browsing. We build the mobile browsing module on top of these components to automate visits to pages via the Browser Application. This browsing module provides functionalities for realistic interaction with a website, e.g., scrolling, click and sleep rate. Similarly to the desktop, it can run either in a *stateful* or *stateless* mode.

Experimental Setup Selector. As shortly described in § 3, we need two phases of browsing to different types of webpages (*training* and *testing*), in order to successfully measure CDT. For that reason, we set the two browsing phases in the following way: During the training phase, the selected device visits the set of *Persona Pages* for a specific duration, referred to as training time (t_{train}). The test phase is the set of visits to *control pages* for the purpose of collecting ads. During this phase, we control the duration of browsing (i.e., t_{test}). The experimental setup selector controls various parameters such as: which type of device will be trained and tested, the times t_{train} and t_{test} , the sequence of time slots for training and testing from the selected device, number of repetitions of this procedure, etc.

Timeline of phases. Each class of experiments is executed multiple times (or runs), through parallel instantiations of the user devices within the framework (as shown in Figure 2). Each experimental run is executed following a timeline of phases as illustrated in Figure 4. This timeline contains N sessions with three primary stages in each: Before, Mobile, and After. The *Before* (B_i) stage is when the two desktop de-

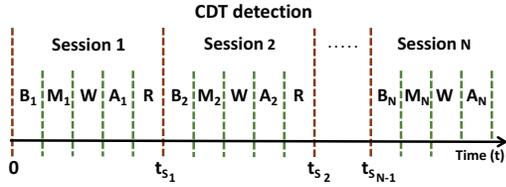


Figure 4: Timeline of phases for CDT measurement.
 M_i : mobile training time t_{train} + testing time t_{test} ;
 $B_i(A_i)$: desktop testing time t_{test} before (after) mobile phase;
 W : wait time (t_{wait}); R : rest time (t_{rest}); t_{s_i} : time of session i .

vices perform a parallel test browsing, with a duration of t_{test} time, to establish the state of ads before the mobile device injects signal into the ad-ecosystem. The *Mobile* (M_i) stage is when the mobile device performs a training browsing for t_{train} time, and a test browsing for t_{test} time. This phase injects the signal from the mobile during training with a persona, but also performs a subsequent test with control pages to establish the state of ads after the training. Finally, the *After* (A_i) stage is when the two desktops perform the final test browsing, with the same duration t_{test} as in *Before* (B_i) stage, to establish the state of ads after the mobile training.

After extensive experimentation, we found that a minimum training time $t_{train}=15$ minutes and testing time $t_{test}=20$ minutes are sufficient for injecting a clear signal over noise, from the trained device to the ad-ecosystem. There is also a waiting time ($t_{wait}=10$ minutes) and resting time ($t_{rest}=5$ minutes) between the stages of each session, to allow alignment of instantiations of devices running in parallel during each session. In total, each session lasts 1.5 hours and is repeated $N=15$ times during a run. Through the experimental setup selector, we define the values of such variables (t_{train} , t_{test} , t_{wait} , t_{rest} , N , type of device), offering the researcher the flexibility to experiment in different cases of CDT.

4.3 Output Signal

Page Parser. This component is activated when the visited page is fully loaded and no further changes occur on the content. To collect the display ads, we first need to identify specific DOM elements inside the visited webpages. This task is challenging due to the dynamic Javascript execution and the complex DOM structures generated in most webpages. For the reliable extraction of ad-elements and identification of the landing pages,³ we follow a methodology similar to the one proposed in [36]. The functionality of this component is to parse the rendered webpage and extract the attributes of display ads, which also contain the landing pages.

Ad Extractor. In most modern websites, the displayed ads are embedded in *iFrame* tags that create deep nesting layers, containing numerous and different types of elements. How-

ever, the ads served by the control pages are found directly inside the *iFrames* so the module does not have to handle such complex behavior. Therefore, the module firstly identifies all the active *iFrame* elements and filters out the invalid ones that have either empty content or zero dimensions. Then, it retrieves the *href* attributes of image and flash ads and parses the URLs, while searching for specific string patterns such as *adurl=*, *redirect=*, etc. These patterns are typically used by the ad-networks for encoding URLs in webpages. Next, the module forms the list of candidate landing pages, which are then processed and analyzed to create the set of true landing pages. The Ad Extractor is fully compatible with the crawlers, and does not need to perform any clicks on the ad-elements, since it extracts only the landing pages' URLs directly from the rendered webpage. After collecting the candidate landing pages, the module filters them with the *EasyList* [21], similarly to previous works [7, 22], and stores only the true active ad-domains. Finally, the Page Parser & Ad Extractor module also stores *metadata* from the crawls such as: time and date of execution, number of identified ads, number of categories, type and phase of crawl, etc.

Ad Categorizer. To associate landing pages or browsing URLs with web categories, we employ the McAfee Trusted-Sources database [41], which provides URLs organized into categories. This system was able to categorize 96% of the landing pages of our collection into a total of 76 unique categories, by providing up to four semantic categories for each page, while the remaining 4% domains were manually classified to the categories above. The final output contains the landing pages of collected ads, along with their categories.

4.4 CDT Detection

Probabilistic CDT is a kind of task generally suitable for investigation through ML. Previous work [56] and industry directions [38, 4] claim that probabilistic device-pairing is based on specific, well-defined signals such as: IP address, geolocation, type and frequency of browsing activity. Since we control these parameters in our methodology, by definition we construct the ground truth with our experimental setups. That is, we control (i) the devices used, which are potentially paired under a given IP address, geolocation and browsing patterns, (ii) the control instance of baseline desktop device, and (iii) the browsing with the personas.

Before applying any statistical method, every instance of the input data has to be transformed into a vector of values; each position in the vector corresponds to a feature. Features are different properties of the collected data: browsing activity of a user during training time, experimental setup used (persona, etc.), time-related details of the experiment, as well as information about the collected ads, which is the output signal received from the given browsing activity. These features can be studied systematically to identify statistical association between the input and output signals, given an ex-

³Destination websites the user is redirected to when clicking on the ads.

perimental setup. In effect, our feature space is comprised of a union of these vectors, since all features are either controlled, or measurable by us (detailed description of the features is given in Appendix, Table 5). The only unknown is whether the ad-ecosystem has successfully associated the devices, and if it has exhibited this in the output signal via ads.

One Dimension Statistical Analysis. At the first level of analysis, to measure the similarity of distribution of ads delivered in the different devices, we compare the signals using a two-tailed permutation test and reject the null hypothesis that the frequency of ads delivered (for a given category) comes from the same distribution, if the t-test statistic leads to a p-value smaller than a significance level $\alpha < 0.05$.

Multidimensional Statistical Analysis. Given that a uni-dimensional test such as the previous one does not take into account the various other features available in each experiment, we further consider ML, which take into account multidimensional data, to decide if the ads delivered in each device are from the same distribution or not. We transform the problem of identifying if the previously exported vectors are similar enough, into a typical binary classification problem, where the predicted class describes the existence of pairing or not, that may have occurred between the mobile device and one of the two desktop devices. As a paired combination we consider the desktop device that exists under the same IP address with the mobile device. The “not paired” combination is the mobile device and the baseline desktop. The analysis is based on three classification algorithms with different dependences on the data distributions. An easily applied classifier that is typically used for performance comparison with other models, is the Gaussian Naive Bayes classifier. Logistic Regression is a well-behaved classification algorithm that can be trained, as long as the classes are linearly separable. It is also robust to noise and can avoid overfitting by tuning its regularization/penalty parameters. Random Forest and Extra-Trees classifiers, construct a multitude of decision trees and output the class that is the mode of the classes of the individual trees. Also they use the Gini index metric to compute the importance of features.

A fundamental point when considering the performance evaluation of ML algorithms is the selection of the appropriate metrics. Pure Accuracy can be used, but it’s not representative for our analysis, since we want to report the most accurate estimation for the number of predicted paired devices, while at the same time measure the absolute number of miss-classified samples overall. For this reason, metrics like Precision, Recall and F_1 -score, and the Area Under Curve of the Receiver Operating Curve (*AUC*) are typically used, since they can quantify this type of information.

5 Experimental Evaluation

We use the Talon framework to perform various experiments and construct different datasets for each. Since every ex-

Table 1: Characteristics of the datasets used in each setup (S) of experiments. $S=\{1,2,3\}$ are the setups of experiments in § 5.2, § 5.3 and § 5.4, respectively; t_{total} : the total duration of experiment; t_{train} : the training duration; t_{test} : the testing duration; I: independent personas; C: data combined from personas; SF: stateful browser; SL: stateless browser; B: boosted CDT browsing.

S	Personas	Runs	t_{train}	t_{test}	t_{total}	Samples	Features
1a	10 (I, SF)	4	15min	20min	37 days	240	1100
1b	10 (C, SF)	-	-	-	-	2400	2201
2a	2 (I, SF)	4	480min	30min	6 days	192	600
2b	2 (C, SF)	-	-	-	-	384	750
2c	2 (I, SF, B)	4	480min	30min	6 days	192	500
2d	2 (C, SF, B)	-	-	-	-	384	576
3a	5 (I, SL)	2	15min	20min	9 days	120	450
3b	5 (C, SL)	-	-	-	-	600	880

perimental setup has different experimental parameters (i.e., training and testing time, number of personas, browsing functionalities), the datasets vary in terms of samples size and feature space. The datasets collected during our experiments and used in our analysis are presented in the Table 1.

5.1 Does IP-sharing allow CDT?

A first set of preliminary experiments were performed to demonstrate that our platform can (i) successfully identify and collect the ads delivered to our multiple devices (mobile and desktops), (ii) inject browsing signal from a device, thus biasing it to have a realistic persona and (iii) lead to matching/pairing of devices, which could be due to same behavioral ads, retargeting ads or CDT.

First, we use a simple experimental setup: we connect three instances of desktop devices and one mobile device under the same IP address. We create one persona (as in § 4.1), with an interest in “Online Shopping-Fashion, Beauty”, and following the described timeline of phases, we run this experiment for two days. Then, we perform one-dimensional statistical analysis, as introduced in § 4.4, and find that there is no similarity between the mobile with any of desktop devices (null hypothesis rejected with highest p-value=0.030), while all desktop distributions are similar to each other (null hypothesis accepted with lowest p-value=0.33). These statistical results indicate that there is no clear device-pairing (at the level of ad distribution for the given persona), and that we should consider controlling more factors to instigate it.

Consequently, we expand this experiment by also training one of the desktop devices using the same persona as with mobile. By repeating the same statistical tests, we find that the mobile and desktop with the same browsing behavior receive ads coming from the same distribution (null hypothesis accepted with lowest p-value=0.84), while the other desktop devices show no similarity with each other or the mobile (null hypothesis rejected with highest p-value=0.008). This

result indicates that browsing behavior under a shared IP address can boost the signal towards advertisers, which they can use to apply advanced targeting, either as CDT, or retargeting on each device or a mixture of both techniques.

Finally, these preliminary experiments and statistical tests provide us with evidence regarding the effectiveness of our framework to inject enough browsing signal from different devices under selected personas. Our framework is also able to collect ads delivered between devices, that can be later analyzed and linked back to the personas. Those are fundamental components for our system and importantly they are potentially causing CDT between the devices involved. Next, we present more elaborate experimentations with our framework, in order to study CDT in action.

5.2 Does short-time browsing allow CDT?

Independent Personas: Setup 1a. This experimental setup emulates the behavior of a user that browses frequently about some topics, but in short-lived sessions in her devices. Given that most users do not frequently delete their local browsing state, this setup assumes that the user’s browser stores all state, i.e., cookies, cache, browsing history. This enables trackers to identify users more easily across their devices, as they have historical information about them. In this setup, every experimental run starts with a clean browser profile; cookies and temporary browser files are stored for the whole duration of the experimental run (stateful). We use all personas of Table 3, and the data collection for each lasts 4 days.

We perform the same statistical analysis as in § 5.1, and find that in 4/10 personas, the mobile and paired desktop ads are similar (null hypothesis accepted with lowest p-value=0.13), while the mobile and baseline desktop ad distributions are different (null hypothesis is rejected with highest p-value=0.009). This inconsistency is reasonable since the statistical analysis is based only on one dimension (the frequency count of types of ads appearing in the devices), which may not be enough for fully capturing the existence of device-pairing. For this reason, we choose to use more advanced, multidimensional ML methods which take into account the various variables available, to effectively compare the potential CDT signals received by the two devices.

The classification results of the Random Forest (best performing) algorithm are reported in Table 2. We use AUC score as the main metric in our analysis, since the ad-industry seems to prefer higher Precision scores over Recall, as the False Positives have greater impact on the effectiveness of ad-campaigns.⁴ As shown in Table 2, the model achieves high AUC scores for most of the personas, with a maximum value of 0.84. Specifically, the personas 2, 4 and 8 scored

⁴Tapad [1] mentions: “Maintaining a low false positive rate while also having a low false negative rate and scale is optimal. This combination is a strong indicator that the Device Graph in question was neither artificially augmented nor scrubbed.”

Table 2: Performance evaluation for Random Forest in Setups 1a and 1b. Left value in each column is the score for Class 0 (C0=*not paired desktop*); right value for Class 1 (C1=*paired desktop*).

Persona (Setup)	Precision		Recall		F ₁ -Score		AUC
	C0	C1	C0	C1	C0	C1	
1 (1a)	0.89	0.60	0.57	0.90	0.70	0.72	0.73
2 (1a)	0.84	0.78	0.81	0.82	0.82	0.80	0.82
3 (1a)	0.81	0.73	0.78	0.76	0.79	0.74	0.76
4 (1a)	0.87	0.78	0.87	0.78	0.87	0.78	0.82
5 (1a)	0.94	0.65	0.68	0.93	0.79	0.76	0.80
6 (1a)	0.57	0.67	0.81	0.38	0.67	0.48	0.59
7 (1a)	0.81	0.87	0.89	0.76	0.85	0.81	0.81
8 (1a)	0.86	0.85	0.89	0.81	0.87	0.83	0.84
9 (1a)	0.74	0.90	0.91	0.73	0.82	0.81	0.81
10 (1a)	0.77	0.85	0.81	0.81	0.79	0.83	0.81
combined (1b)	0.77	0.84	0.81	0.84	0.82	0.84	0.89

highest in AUC, and also in Precision and Recall, whereas persona 6 has poor performance compared to the rest. These results indicate that for high scoring personas, we successfully captured the active CDT campaigns, but for the personas with lower scores, there may not be active campaigns for the period of the experiments.

In order to retrieve the variables that affect the discovery and measurement of CDT, we applied the feature importance method on the dataset of each persona, and selected the top-10 highest scoring features. For the majority of the personas (7 out of 10) the most important features were the number of ads (distinct or not) and the number of keywords in desktop. In some cases, there were also landing pages that had high scoring (i.e., specific ad-campaigns), but this was not consistent across all personas.

Combined Personas: Setup 1b. Here, we use all the datasets collected individually, for each persona in the previous experiment (Setup 1a), and combine them into one unified dataset. This setup emulates the real scenario of a user exhibiting multiple and diverse web interests, that give extra information to the ad-ecosystem about their browsing behavior. Of course, there is an increase in the possible feature space to accommodate all the domains and keywords from all personas. In fact, the dataset contains 2021 features as it stores the vectors of landing pages and keywords, for all the different types of personas. In total, there were 890 distinct ad-domains described by keywords in 76 distinct categories.

In this dataset, we apply feature selection with the Extra-Trees classifier to select the most relevant features and create a more accurate predictive model. This method reduced the feature space to 984 useful features out of 2201. Next, we use the three classification algorithms and a range of hyper-parameters for each one. Also, we apply a 10-fold nested cross-validation method for selecting the best model (in terms of scoring performance) that can give us an accurate, non overly-optimistic estimation [13]. Again, the

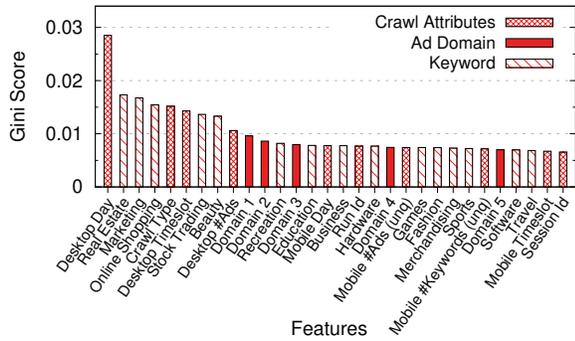


Figure 5: Top-30 features ranked by importance using Gini index, in the machine learning model.

best selected model was Random Forest, with 200 estimators (trees) and 200 depth of each tree, with AUC=0.89 (bottom row in Table 2). The model’s performance is high in all the mentioned scores, which indicates that the more diverse data the advertisers collect, the easier it is to identify the different user’s devices. This result is in line with Zimmeck et al. [56], who attempted a threshold-based approach for probabilistic CDT detection on real users’ data, lending credence to our proposed platform’s performance.

We also measure the feature importance for the top-30 features (shown in Figure 5). One third of the top features are related to crawl specific metadata, whereas about half of the top features are keyword-related. Interestingly, features such as the day and time of the experiment, as well as the number of received ads, are important for the algorithm to make the classification of the devices. Indeed, time-related features provide hints on when the ad-ecosystem receives the browsing signal and attempts the CDT, and thus, which days and hours of day the CDT is stronger. These results give support to our initial decision to experiment in a continuous fashion with regular sessions injecting browsing signal, while at the same time measuring the output signal via delivered ads.

5.3 Does long-time browsing improve CDT?

Independent Personas: Setup 2a. In this set of experiments, we allow the devices to train for a longer period of time, to emulate the scenario where a user is focused on a particular interest, and produces heavy browsing behavior around a specific category. This long-lived browsing injects a significantly higher input signal to the ad-ecosystem than the previous setup, which should make it easier to perform CDT. In order to increase the setup’s complexity, and make it more difficult to track the user, we allow all devices (i.e., 1 mobile, 2 desktops) to train in the same way under the same persona. In effect, this setup also tests a basic countermeasure from the user’s point of view, who tries to blur her browsing by injecting traffic of the same persona from all devices to the ad-ecosystem.

In this setup, while all devices are trained with the same behavioral profile, we examine if the statistical tests and ML modeler can still detect and distinguish the CDT. This experiment contains three different phases during each run. The mobile phase, where the mobile performs training crawls for $t_{train}=480$ mins, and a testing crawl for $t_{test}=30$ mins. In parallel with the mobile training, the two desktops perform test crawls for $t_{test}=30$ mins. After mobile training and testing, both desktops start continuous training and testing crawls alternately for 8 hours ($t_{train}=t_{test}=30$ min).

Due to the long time needed for executing this experiment, we focus on two personas constructed in the following way. We use the methodology for persona creation as described in § 4.1, and focus on active ad-campaigns, resulting to two personas in the interest of “Online Shopping-Accessories”, and “Online Shopping-Health and Fitness” (loosely matching the personas 1 and 4 from Table 3). Then, we performed 4 runs of 16 hours duration each, for each persona. In this setup, since all devices are uniformly trained, we do not include the keyword vector of the persona pages into the datasets, to not introduce any bias from repetitive features.

The statistical analysis for this experiment reveals potential CDT, since we accept the null hypothesis for the distribution of ads delivered in the paired desktop and mobile (lowest p-value=0.052), and reject it in the baseline desktop and mobile (highest p-value=0.006). This consistency is interesting, since for this setup all three devices are uniformly trained with the same persona, and thus all of them collect similar ads due to retargeting. However, there is no similarity between the distributions of ads in the devices that do not share the same IP address.

To clarify this finding, we applied the ML algorithms as in the previous experiment. The algorithms again detect CDT between the mobile and the paired desktop, even though all devices were exposed to similar training with the same persona. In fact, Logistic Regression performed the best across both personas, with $AUC \geq 0.81$, and F_1 -score ≥ 0.80 for both classes.⁵ When computing the importance of features, the desktop number of ads and keywords and the desktop time slot are in the top-10 features. Based on these observations, we believe that the longer training time allowed the ad-ecosystem to establish an accurate user profile, and retarget ads on the paired desktop, based on the mobile’s activity.

Combined Personas: Setup 2b. Similarly to § 5.2 we combine all data collected from the Setup 2a into a unified dataset. Under this scenario, in which we mix data from both personas, the classifier again performs well, with AUC=0.89. Important features in this case are the number of ads and keywords delivered to the desktops, the time of the experiment, and number of keywords for the desktop.

Boosted Browsing with CDT trackers and Independent Personas: Setup 2c. In the next set of experiments, we

⁵Detailed evaluation results of § 5.3 presented in Appendix, Table 4.

investigate the role of CDT trackers in the discovery and measurement of CDT. In particular, we attempt to boost the CDT signal, by visiting webpages with higher portion of CDT trackers. Therefore, the experimental setup and the preprocessing method remain the same as in the previous Setup 2a, but we select webpages to be visited that have active ad-campaigns and their landing pages embed the most-known CDT trackers (as we also show in the next section): Criteo, Tapad, Demdex, Drawbridge. We also change the set of our control pages, so that each one contains at least a CDT tracker. News sites have many 3rd-parties compared to other types of sites [22]. Thus, for this boosted browsing experiment, we choose the set of control pages to contain 3 weather pages and 2 news websites,⁶ while verifying they do not serve contextual ads.

Performing the same analysis as earlier, we find that mobile and paired desktop have ads coming from the same distribution (lowest p-value=0.10), and that there is no similarity between the ads delivered in the mobile and baseline desktop (highest p-value=0.007). For a clearer investigation of the importance of the CDT trackers, we also evaluate the findings with the ML models. For persona 1, Logistic Regression and Random Forest models perform near optimally, with high precision of Class 1, high recall for class 0, average F_1 -Score=0.93 for both classes, and AUC=0.93. For persona 4, the scores are even higher, outperforming the other setups, as all metrics for Logistic Regression scored higher than 0.98. Overall, these results indicate that we successfully biased the trackers to identify the emulated user in both devices, and to provide enough output signal (ads delivered) for the statistical algorithms to detect the CDT performed.

Boosted Browsing with CDT trackers and Combined Personas: Setup 2d. We follow a similar approach with before, and combine all data collected from the Setup 2c, into a unified dataset for Setup 2d. Under this scenario, the classifier (Logistic Regression) again performs very well, with AUC=0.93. Important features in this case are the number of ads delivered to the desktops, the time of the experiment in each desktop and the number of keywords. Interestingly, and perhaps unexpectedly, the existence of Criteo tracker in a landing page, is a feature appearing in the top-10 features.

5.4 Does incognito browsing help evade CDT?

Independent Personas: Setup 3a. In this final experimental setup, we investigate if it is possible for the user to apply some basic countermeasures to avoid, or at least reduce the possibility of CDT, by removing her browsing state in every new session. For this, we perform experiments where the traditional tracking mechanisms (e.g., cookies, cache, browsing history, etc.) are disabled or removed, emulating incognito browsing. We select the first five personas from Table 3,

⁶accuweather.com, wunderground.com, weather.com, usatoday.com, huffingtonpost.com

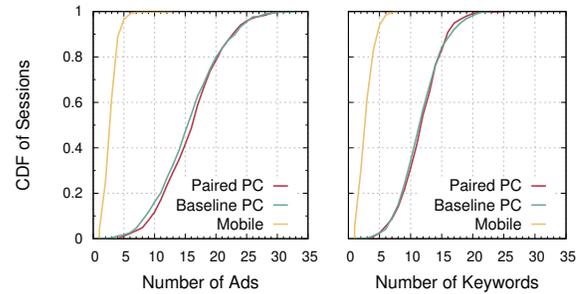


Figure 6: CDF of collected ads (left) and corresponding keywords of the ads (right) per crawling session for all devices.

which had the most active ad-campaigns and appeared to be promising due to the “online shopping” interest. Every desktop executed browsing in a stateless mode, while the mobile in a stateful mode. For each persona, we collected data for two runs, following the timeline of phases as in Setup 1a.

The distributions between mobile vs. paired desktop, as well as mobile vs. baseline desktop, were found to be different (highest p-value=0.034). Also, none of the ML classifiers performed higher than 0.7 (in all metrics), and thus we could not clearly extract any significant result. Specifically, the highest AUC score for personas 1 and 2 was 0.70 with the use of the Random Forest classifier, and for personas 3 and 4 was 0.73 using the Logistic Regression classifier. The worst scoring, independent of algorithm, was recorded for persona 5, with AUC=0.57, and Precision/Recall scores under 0.50.

Combined Personas: Setup 3b. When the data from all five personas are combined, the classifier performing best was Logistic Regression, with AUC=0.79. Overall, these results point to the semi-effectiveness of the incognito browsing to limit CDT. That is, by removing the browsing state of a user on a given device, the signal provided to the CDT entities is reduced, but not fully removed. In fact, when the data from various personas are combined, the CDT is still somewhat effective, since the paired devices have the same IP address.

6 Platform Validation

In this section we validate the representativeness of the data collected from the previous experiments, by examining: (i) the type and frequency of ads delivered in each device, and (ii) the type and number of trackers that our personas were exposed to. We compare the distributions of these quantities with past works and data on real users, to quantify if our synthetic personas successfully emulate real users’ traffic, and if our measurements of the CDT ad-ecosystem are realistic.

We first measure the frequency of ads delivered to our devices in the experiment § 5.2, since it follows a well-crafted timeline that is suitable for this kind of measurement. The ads delivered in the three devices during these sessions are shown in Figure 6 (left). For most sessions (~90%), the mobile device was exposed to fewer than five ads, since the

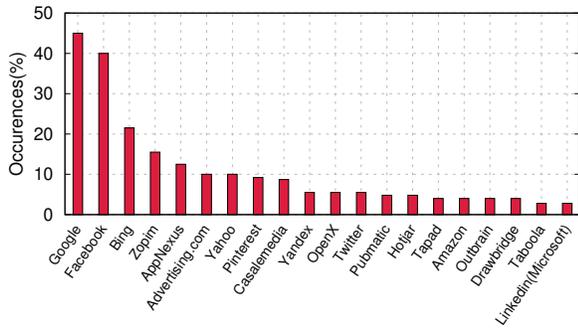


Figure 7: Top-20 trackers (grouped based on organization) and their coverage in persona pages. For example, all the Google-owned domains, such as Doubleclick, Googleapis, Google-Analytics, are grouped under the “Google” label.

mobile version of websites typically delivers a smaller number of ads, designed for smaller screens and devices. On the contrary, the desktop devices had a higher exposure to ads compared to the mobile device. Also, the two desktops receive a similar number of ads (on average 2 to 4 ads on every visit to the control pages). Similar observations can be made for the keywords categories of ads (Figure 6 (right)). The ad-industry has reported that ~ 300 ads everyday, on average, are being displayed to desktop users [30, 11, 31, 14], while they also recommend the delivery of 5 ads per mobile domain [52], which proportionally match the number of ads we have collected in our mobile and desktop sessions.

We also validate the representativeness of the data collected from the experiments § 5.2 and § 5.3, by examining the trackers appearing in the webpages visited by the personas. We use Disconnect List [18] to detect them and measure their frequency of appearance (i.e., Figure 7). From the trackers detected in the set of persona pages, and using the list provided by [56], 37% was found to be CDT related, including both deterministic and probabilistic. In fact, the top CDT trackers found in our data, which may perform both types of CDT, include Google-owned domains, Facebook, Criteo, Zopim, Bing, Advertising.com(AOL), and are in-line with the top CDT trackers found in [56, 10] (66% overlap of top-20 with [56] and 55% overlap with [10]). In addition, 17% of these trackers are mainly focused on probabilistic CDT, including Criteo, BlueKai, AdRoll, Cardlytics, Drawbridge, Tapad, and each individual tracker is found at least in 2% of the persona pages, again in-line with the results in [56].

7 Discussion & Conclusion

Through extensive experiments with the proposed framework Talon, we were able to trigger CDT trackers into pairing of the emulated users’ devices. This allowed us to statistically verify that CDT is indeed happening, and measure its effectiveness on different user interests and browsing behaviors, independently and in combination. In fact, CDT was

prominent when user devices were trained to browse pages of similar interests, reinforcing the behavioral signal sent to CDT entities, and specifically when browsing activity is related with online shopping, since those types of users seem to be more targeted by advertisers. The CDT effect was further amplified when the visited persona and control pages had embedded CDT trackers, pushing the accuracy of detection up to 99%. We also found that browsing in a stateless mode showed a reduced, but not completely removed CDT effect, as incognito browsing obfuscates somewhat the signal sent to the ad-ecosystem, but not the network access information. Indeed, our data collection was performed across relatively short time periods, in comparison to the wealth of browsing data that advertising networks have at their disposal. In fact, we anticipate that CDT companies collect data about users and devices for months or years, and even buy data from data brokers, to have the capacity of targeting users with even higher rates. To that end, we believe that high accuracies self-reported by CDT companies (e.g., Lotame: $>90\%$ [37], Drawbridge: 97.3% [19]), are possible.

Impact on user privacy: Undoubtedly, CDT infringes on users’ online privacy and minimizes their anonymity. But the actual extent of this tracking paradigm and its consequences to users, the community, and even to the ad-ecosystem itself, are still unknown. In fact, since CDT is heavily depended on user’s browsing activity, and the ad-ecosystem employs such collected data for targeting purposes, one major line of future work is the study of targeting sensitive user categories (e.g., gender, sexual orientation, race, etc.) via CDT. This is especially relevant nowadays with the enforcement of recent EU privacy regulations such as GDPR [24] and ePrivacy [23]. This is where Talon comes in play, as it provides a concrete, scalable and extensible methodology for experimenting with different CDT scenarios, auditing its mechanics and measuring its impact. In fact, the modular design of our methodology allows to study CDT in depth, and propose new extensions to study the CDT ecosystem: new plugins, personas and ML techniques. To that end, our design constitutes Talon into an enhanced transparency tool that reveals potentially illegal biases or discrimination from the ad-ecosystem.

Acknowledgments

The research leading to these results has received funding from the European Union’s Horizon 2020 Research and Innovation Programme under grand agreement No 786669 (project CONCORDIA), the Marie Skłodowska-Curie grant agreement No 690972 (project PROTASIS), and the Defense Advanced Research Projects Agency (DARPA) ASED Program and AFRL under contract FA8650-18-C-7880. The paper reflects only the authors’ views and the Agency and the Commission are not responsible for any use that may be made of the information it contains.

References

- [1] Measuring Cross-Device: The Methodology. <https://www.tapad.com/resources/cross-device/measuring-cross-device-the-methodology>, 2018.
- [2] Pew Research Center - Mobile Fact Sheet. <http://www.pewinternet.org/fact-sheet/mobile/>, 2018.
- [3] ADBRAIN. Demystifying cross-device. essential reading for product management, business development and business technology leaders. <https://www.iabuk.com/sites/default/files/white-paper-docs/Adbrain-Demystifying-Cross-Device.pdf>, 2016.
- [4] ADELPHIC. How cross-device identity matching works. <https://adelphic.com/how-cross-device-identity-matching-works-part-1/>, 2016.
- [5] AGUIRRE, E., MAHR, D., GREWAL, D., DE RUYTER, K., AND WETZELS, M. Unraveling the personalization paradox: The effect of information collection and trust-building strategies on online advertisement effectiveness. *Journal of Retailing* 91, 1 (2015), 34–49.
- [6] ARP, D., QUIRING, E., WRESSNEGGER, C., AND RIECK, K. Privacy threats through ultrasonic side channels on mobile devices. In *IEEE European Symposium on Security and Privacy (EuroS&P)* (2017), pp. 35–47.
- [7] BASHIR, M. A., ARSHAD, S., ROBERTSON, W., AND WILSON, C. Tracing information flows between ad exchanges using retargeted ads. In *25th USENIX Security Symposium* (2016), pp. 481–496.
- [8] BASHIR, M. A., FAROOQ, U., SHAHID, M., ZAFFAR, M. F., AND WILSON, C. Quantity vs. quality: Evaluating user interest profiles using ad preference managers. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2019).
- [9] BLEIER, A., AND EISENBEISS, M. Personalized online advertising effectiveness: The interplay of what, when, and where. *Marketing Science* 34, 5 (2015), 669–688.
- [10] BROOKMAN, J., ROUGE, P., ALVA, A., AND YEUNG, C. Cross-device tracking: Measurement and disclosures. *Proceedings on Privacy Enhancing Technologies*, 2 (2017), 133–148.
- [11] BRYCE SANDERS. Do we really see 4,000 ads a day? <https://www.bizjournals.com/bizjournals/how-to/marketing/2017/09/do-we-really-see-4-000-ads-a-day.html>, 2017.
- [12] CARRASCOSA, J. M., MIKIANS, J., CUEVAS, R., ERRAMILLI, V., AND LAOUTARIS, N. I always feel like somebody’s watching me: measuring online behavioural advertising. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CONEXT)* (2015).
- [13] CAWLEY, G. C., AND TALBOT, N. L. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research* 11 (2010).
- [14] CHRISTOPHER ELLIOTT. Yes, there are too many ads online. yes, you can stop them. heres how. https://www.huffingtonpost.com/entry/yes-there-are-too-many-ads-online-yes-you-can-stop_us_589b888de4b02bbb1816c297, 2017.
- [15] CRITEO. The State of Cross-Device Commerce. <https://www.criteo.com/wp-content/uploads/2017/07/Report-criteo-state-of-cross-device-commerce-2016-h2-SEA.pdf>, 2016.
- [16] CRITEO. The 5 top attribution methodologies for cross-channel roi. <https://www.criteo.com/insights/top-attribution-methodologies-for-cross-channel-roi/>, 2018.
- [17] DAS, A., BORISOV, N., AND CAESAR, M. Tracking mobile web users through motion sensors: Attacks and defenses. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2016).
- [18] DISCONNECT. Disconnect lets you visualize and block the invisible websites that track your browsing history. <https://disconnect.me/>, 2019.
- [19] DRAWBRIDGE. Cross-Device Consumer Graph. <https://go.drawbridge.com/rs/454-0RY-155/images/Drawbridge-Cross-Device-Consumer-Graph.pdf>, 2015.
- [20] DRAWBRIDGE. Drawbridge Cross-Device Connected Consumer Graph Is 97.3% Accurate. <https://go.drawbridge.com/rs/454-0RY-155/images/Drawbridge-Cross-Device-Consumer-Graph.pdf>, 2015.
- [21] EASYLIST. Easylist is the primary filter list that removes most adverts from international webpages, including unwanted frames, images and objects. <https://easylist.to/>, 2018.

- [22] ENGLEHARDT, S., AND NARAYANAN, A. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the ACM SIGSAC conference on computer and communications security (CCS)* (2016), pp. 1388–1401.
- [23] EUROPEAN PARLIAMENT, COUNCIL OF THE EUROPEAN UNION. Directive 2002/58/EC of the European Parliament and of the Council of 12 July 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector (Directive on privacy and electronic communications), 2002.
- [24] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union L119* (2016), 1–88.
- [25] FARAHAT, A., AND BAILEY, M. C. How effective is targeted advertising? In *Proceedings of the 21st ACM International Conference on World Wide Web* (2012), pp. 111–120.
- [26] GIRONDA, J. T., AND KORGAONKAR, P. K. ispy? tailored versus invasive ads and consumers perceptions of personalized advertising. *Electronic Commerce Research and Applications* 29 (2018), 64–77.
- [27] GOOGLE. Google Product Taxonomy. <https://www.google.com/basepages/producttype/taxonomy.en-US.txt>, 2015.
- [28] GOOGLE. Run apps on the Android Emulator. <https://developer.android.com/studio/run/emulator/>, 2018.
- [29] IORDANOU, C., KOURTELLIS, N., CARRASCOA, J. M., SORIENTE, C., CUEVAS, R., AND LAOUTARIS, N. Beyond content analysis: Detecting targeted ads via distributed counting. *arXiv preprint arXiv:1907.01862* (2019).
- [30] JON SIMPSON. Finding brand success in the digital world. <https://www.forbes.com/sites/forbesagencycouncil/2017/08/25/finding-brand-success-in-the-digital-world/#734eaba626e2>, 2018.
- [31] JUSTIN MALLINSON. How many ads do we really see each day? <http://www.tcsmedia.co.uk/many-ads-really-see-day/>, 2018.
- [32] KAFKA, P., AND MOLLA, R. Recode - 2017 was the year digital ad spending finally beat TV. <https://www.recode.net/2017/12/4/16733460/2017-digital-ad-spend-advertising-beat-tv>, 2017.
- [33] KOROLOVA, A., AND SHARMA, V. Cross-app tracking via nearby bluetooth low energy devices. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY)* (2018), pp. 43–52.
- [34] LÉCUYER, M., DUCOFFE, G., LAN, F., PAPANCEA, A., PETSIOS, T., SPAHN, R., CHAINTREAU, A., AND GEAMBASU, R. Xray: Enhancing the webs transparency with differential correlation. In *23rd USENIX Security Symposium* (2014), pp. 49–64.
- [35] LEWIS, R. A., RAO, J. M., AND REILEY, D. H. Here, there, and everywhere: correlated online behaviors can lead to overestimates of the effects of advertising. In *Proceedings of the 20th ACM International Conference on World Wide Web* (2011), pp. 157–166.
- [36] LIU, B., SHETH, A., WEINSBERG, U., CHANDRASHEKAR, J., AND GOVINDAN, R. Adreveal: improving transparency into online targeted advertising. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks* (2013), p. 12.
- [37] LOTAME. Cross-Device ID Graph Accuracy: Methodology. <https://www.lotame.com/cross-device-id-graph-accuracy-methodology/>, 2016.
- [38] LOTAME. Cross-device bridging the gap between screens. <https://www.lotame.com/products/cross-device/>, 2018.
- [39] MATHUR, A., VITAK, J., NARAYANAN, A., AND CHETTY, M. Characterizing the use of browser-based blocking extensions to prevent online tracking. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS)* (Baltimore, MD, 2018), USENIX Association, pp. 103–116.
- [40] MAVROUDIS, V., HAO, S., FRATANONIO, Y., MAGGI, F., KRUEGEL, C., AND VIGNA, G. On the privacy and security of the ultrasound ecosystem. *Proceedings on Privacy Enhancing Technologies* (2017).
- [41] MCAFEE. Customer URL Ticketing System. <https://www.trustedsource.org/>, 2018.
- [42] MCNAIR, C. Global Ad Spending Update. <https://www.emarketer.com/content/global-ad-spending-update>, 2018.
- [43] MELICHER, W., SHARIF, M., TAN, J., BAUER, L., CHRISTODORESCU, M., AND LEON, P. G. (do not) track me sometimes: Users contextual preferences for web tracking. *Proceedings on Privacy Enhancing Technologies*, 2 (2016).

- [44] PACHILAKIS, M., PAPADOPOULOS, P., MARKATOS, E. P., AND KOURTELLIS, N. No more chasing waterfalls: A measurement study of the header bidding ad-ecosystem. In *19th ACM Internet Measurement Conference* (2019).
- [45] PAN, X., CAO, Y., AND CHEN, Y. I do not know what you visited last summer - protecting users from third-party web tracking with trackingfree browser. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2015).
- [46] PAPADOPOULOS, E. P., DIAMANTARIS, M., PAPADOPOULOS, P., PETSAS, T., IOANNIDIS, S., AND MARKATOS, E. P. The long-standing privacy debate: Mobile websites vs mobile apps. In *Proceedings of the 26th ACM International Conference on World Wide Web* (2017), pp. 153–162.
- [47] PAPADOPOULOS, P., KOURTELLIS, N., AND MARKATOS, E. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *The World Wide Web Conference* (2019), ACM, pp. 1432–1442.
- [48] PAPADOPOULOS, P., KOURTELLIS, N., RODRIGUEZ, P. R., AND LAOUTARIS, N. If you are not paying for it, you are the product: How much do advertisers pay to reach you? In *Proceedings of the ACM Internet Measurement Conference* (2017), pp. 142–156.
- [49] PATRICK HOLMES. Mobile and Desktop Advertising Strategies Based on User Intent. <https://instapage.com/blog/adwords-search-device-user-intent>, 2018.
- [50] PROJECT, J. F. Automation for Apps. <http://appium.io/>, 2018.
- [51] RAMIREZ, E., OHLHAUSEN, M., AND MCSWEENEY, T. Cross-device tracking: An FTC staff report. Tech. rep., 2017.
- [52] RENE HERMENAU. Adsense max allowed number of ads - 2018 Rules. <https://wpquads.com/google-adsense-allowed-number-ads/>, 2018.
- [53] TAPAD. Tapad device graph - creating a unified view of the consumer. <https://www.tapad.com/device-graph/>, 2018.
- [54] TAPAD. The expert’s guide to cross-device conversion & attribution. <https://www.tapad.com/uses/the-experts-guide-to-cross-device-conversion-attribution>, 2018.
- [55] TUCKER, C. E. Social networks, personalized advertising, and privacy controls. *Journal of Marketing Research* 51, 5 (2014), 546–562.
- [56] ZIMMECK, S., LI, J. S., KIM, H., BELLOVIN, S. M., AND JEBARA, T. A privacy analysis of cross-device tracking. In *26th USENIX Security Symposium* (2017), pp. 1391–1408.

A Appendix

Table 3: Behavioral personas used in our experiments.

Persona	Category - Description
1	Online Shopping - Accessories, Jewelry.
2	Online Shopping - Fashion, Beauty.
3	Online Shopping - Sports and Accessories.
4	Online Shopping - Health and Fitness.
5	Online Shopping - Pet Supplies.
6	Air Travel.
7	Online Courses and Language Resources.
8	Online Business, Marketing , Merchandising.
9	Browser Games - Online Games.
10	Hotels and Vacations.

Table 4: Performance evaluation for Logistic Regression in total components of Setup 2. Left value in each column is the score for Class 0 (C0=not paired desktop); right value for Class 1 (C1=paired desktop).

Persona (setup)	Precision		Recall		F_1 -Score		AUC
	C0	C1	C0	C1	C0	C1	
1 (2a)	0.90	0.79	0.82	0.88	0.86	0.83	0.85
4 (2a)	0.83	0.79	0.81	0.81	0.82	0.80	0.81
combined(2b)	0.87	0.92	0.92	0.87	0.89	0.90	0.89
1 (2c)	0.87	1.0	1.0	0.88	0.93	0.93	0.93
4 (2c)	1.0	0.98	0.98	1.0	0.99	0.99	0.99
combined(2d)	1.0	0.86	0.88	1.0	0.93	0.93	0.93

Table 5: Description of features used by datasets. The type of desktop crawl values are in range $\{0,1\}$, where 0 represents the before/test sessions, while 1 the after/train sessions. The time of crawl is divided in 30 minutes timeslots and is encoded in range $\{0,48\}$. The day of crawl is encoded in range $\{1,7\}$. V represents the (enumerated) vectors of values in the sets of: landing pages, training pages, ads and ad categories.

Feature Label	Description
Crawl.Type	The type of desktop crawl.
Run_ID	The indexed number of run $\{1,4\}$.
Session_ID	The index of session $\{1,15\}$.
Persona.Keywords	V: keyword categories of training pages.
Mobile.Timeslot	Time of crawl (Mobile).
Desktop.Timeslot	Time of crawl (Desktop).
Desktop.Day	The day of crawl (Desktop).
Mobile.Number_of_Ads	# ad domains (Mobile).
Desktop.Number_of_Ads	# ad domains collected (Desktop).
Mobile.Unique_Number_of_Ads	# distinct ad domains (Mobile).
Desktop.Unique_Number_of_Ads	# ad domains (Desktop).
Mobile.Number_of_Keywords	# ad categories (Mobile).
Desktop.Number_of_Keywords	# ad categories (Desktop).
Mobile.Unique_Number_of_Keywords	# distinct ad categories (Mobile).
Desktop.Unique_Number_of_Keywords	# distinct ad categories (Desktop).
Mobile.Keywords	V: keyword categories of landing pages (Mobile).
Desktop.Keywords	V: keyword categories of landing pages (Desktop).
Mobile.Landing_Pages	V: landing pages of delivered ads (Mobile).
Desktop.Landing_Pages	V: landing pages of delivered ads (Desktop.)

Analysis of Location Data Leakage in the Internet Traffic of Android-based Mobile Devices

Nir Sivan, Ron Bitton, Asaf Shabtai
Department of Software and Information Systems Engineering
Ben Gurion University of the Negev

Abstract

In recent years we have witnessed a shift towards personalized, context-based services for mobile devices. A key component of many of these services is the ability to infer the current location and predict the future location of users based on location sensors embedded in the devices. Such knowledge enables service providers to present relevant and timely offers to their users and better manage traffic congestion control, thus increasing customer satisfaction and engagement. However, such services suffer from location data leakage which has become one of today's most concerning privacy issues for smartphone users. In this paper we focus specifically on location data that is exposed by Android applications via Internet network traffic in plaintext without the user's awareness. We present an empirical evaluation involving the network traffic of real mobile device users, aimed at: (1) measuring the extent of relevant location data leakage in the Internet traffic of Android-based smartphone devices; (2) understanding the value of this data and the ability to infer users' points of interests (POIs); and (3) deriving a step-by-step attack aimed at inferring the user's POIs under realistic, real-world assumptions. This was achieved by analyzing the Internet traffic recorded from the smartphones of a group of 71 participants for an average period of 37 days. We also propose a procedure for mining and filtering location data from raw network traffic and utilize geolocation clustering methods to infer users' POIs. The key findings of this research center on the extent of this phenomenon in terms of both ubiquity and severity; we found that over 85% of the users' devices leaked location data, and the exposure rate of users' POIs, derived from the relatively sparse leakage indicators, is around 61%.

1 Introduction

In recent years, there has been a trend towards the personalization of services in many areas. This is particularly true for services provided on mobile devices, where a plethora of context-based applications (e.g., Yelp, Uber, Google Maps)

are used daily by millions of people. These devices possess a tremendous amount of private information, ranging from users' personal and financial data to their location data, making such devices the target of personalized advertisements (by commercial entities) and intelligence gathering. A key property of many of these services is the ability to understand the users' current location, infer points of interest, and predict the future location of users based on location sensors embedded in the devices. Such knowledge enables service providers to present relevant and timely services (such as navigation recommendations, weather forecasts, advertisements, and social networks) to their users, thus increasing customer satisfaction and engagement. Methods for deriving the location of a mobile device can be categorized into the following two approaches.

In the **host-based** approach, an installed application can infer the location of the device by probing built-in sensors or evaluating data provided when a user checks in to a place on social media. Local sensors that can provide location data include hotspot (Wi-Fi) information such as the SSID and BSSID [1], connected cell tower, as well as GPS [2]. The location can also be inferred by using various side-channel attacks such as power supply variance analysis [3].

In the **network-based** approach, the location of the mobile device can be derived by using cell tower triangulation [4] (i.e., using radio location by analyzing signals received by the cell towers the device is connected to) or by analyzing the CDRs [5]. This requires high privileged access to the data which is usually available to service providers and law enforcement agencies.

In order to utilize location traces as a meaningful information source, it is imperative to analyze the data and aggregate it to location clusters that are important to the user, such as home, shopping, or work [2]. These locations are also known as the users' points of interest (POIs). The most common approach for inferring a user's POIs is by clustering the location traces by distance and time thresholds; eventually, a cluster will be produced if the user stays in the same place

for a sufficient amount of time. POIs are identified by understanding which clusters are important to the user and omitting less important data such as transit data [6, 7].

Location data collected on the mobile device may be provided to third party services by applications or leaked by a malicious application [8]. Recent research has reported a high rate of personal data leakage by popular applications over insecure communication channels without users' awareness. These studies also showed that location data is one of the most "popular" leaked personally identifiable information (PII), as 10% of the most popular applications leak location data in plain text [8]. In fact, according to Trend Micro, the location permission was identified as the most abused Android application permission.¹ This privacy breach was also acknowledged during the 2018 DEFCON workshops, when applications of both iOS and Android-based devices were detected sending accurate location data in unencrypted formats.²

Recent research investigated the privacy risks to mobile device users emanating from legitimate or malicious applications granted permission to access the user's location. Most of these works however, focused on analyzing and quantifying the exposure of private data to the *specific application* (or location-based service provider) granted access to the location data [9–12]. However, in practice, multiple applications with access to location data are installed on each mobile device. Hence, there is a need to explore the implications of location data leakage by multiple applications on the user's privacy.

One main challenge in conducting research that focuses on understanding the privacy risks to mobile device users is the collection of location data accessed and/or sent by applications, which might require root access. Recent researches collected data either by running applications in a controlled environment (e.g., sandboxes) [11, 12] or provided subjects with alternative (rooted) devices that were not necessarily used as their primary devices [9]. Previous studies also analyzed the privacy risk associated with an adversary that can eavesdrop on the communication of the mobile device. These works focused mainly on automatically identifying PII.

In this research, we investigate the phenomenon of location data leakage in the Internet traffic of Android-based smartphones, by multiple applications. The main goals of this research are as follows. First, to understand the *amount and quality (relevancy)* of location leakage detected in plain text in the device's network traffic. Second, to analyze the location leaks in order to infer the user's POIs. This task is not trivial due to the fact that the vast majority of existing POI detection methods assume a consistent and high rate of location sampling (e.g., via GPS); therefore, they cannot be directly applied on noisy and sparse location data, like the data we focus on in this study (i.e., location data leaked over mobile

device network traffic). Third, to understand the privacy exposure level of users in terms of the number of identified POIs, amount of data required for identifying the POIs, accuracy of the detected POIs, and time spent in the POIs.

In order to achieve these goals, we collected and analyzed the Internet traffic of 71 smartphone users for an average of 37 days, while the devices were being used routinely. In addition, we collected the location of the mobile devices by using a dedicated Android agent (application) that was installed on the devices and sampled the location sensor. The data collected by the agent was used as the ground truth for the actual location of the mobile device. The results of our experiment showed that over 85% of the users' devices leaked location data. Furthermore, the exposure rate of users' POIs, derived from the relatively sparse leakage indicators, is around 61%. Even cases of low location leakage rates (once in six hours) and coverage of only 20% (i.e., 20% of the time during which location data was accessed by the applications it was leaked) can expose approximately 70% of the weighted POIs.

In summary, the contributions of this paper are as follows. First, we explore and discuss the scope, volume and quality of location-based data leaked via insecure, unencrypted network traffic of smart mobile devices. We are specifically interested in exploring the implication of location data leakage by multiple applications on the user's privacy, a case which has not been investigated before. Second, we conduct an empirical evaluation based on real data from mobile devices. The evaluation involves a unique dataset that was collected simultaneously from the device itself and the network traffic sent from the device; such a dataset from real users' devices is very difficult to obtain. Third, we present a methodological process for collecting, processing, and filtering location-based data from the network traffic of mobile devices in order to infer the users' POIs. We use POI clustering on a sparse, inconsistent data stream by modifying available clustering algorithms, and discuss the experimental results and the effectiveness of the applied process. Finally, to the best of our knowledge, we are the first to present a step-by-step attack aimed at leaking location data and inferring users' POIs from a device's network traffic under realistic assumptions.

2 Related Work

2.1 Privacy in location based services

According to the GDPR definition [13], personal data or personally identifiable information (PII) is any data that can be used to identify a person, including name, ID, social media identity, and location. PII leaks are a major privacy concern for mobile device users. Along with device and user identifiers, location leakage is among the private data most commonly and extensively leaked from mobile devices [8]. As a case in point, recent studies showed that it is possible to de-anonymize users using location traces [14]. Furthermore,

¹<http://about-threats.trendmicro.com/us/library/image-gallery/12-most-abused-android-app-permissions>

²<https://arstechnica.com/information-technology/2018/09/dozens-of-ios-apps-surreptitiously-share-user-location-data-with-tracking-firms/>

location permission is commonly requested by most mobile apps (25% of apps use precise location, and an even greater number of apps use coarse location) [15]. Without accurate knowledge about how each application uses and handles the location data, access of applications to location API pose a real threat to users' privacy.

A summary of related works in this domain is provided in Table 8. The research can be broadly categorized by the threat actor that abuses network traffic access in order to disclose users' personal information.

The first type of threat actor considered in related studies includes applications that are installed by the user on the mobile device [1, 9–12, 16–20]. These studies focus on analyzing the private data that is available to the installed application or the location-based service provider with which the application is communicating, and consequently understanding the privacy risks to which the user is exposed. This can be attempting to de-anonymize the user by using location traces [9], inferring the user's POIs [9, 10, 19], or identifying other PII leaks [11, 12, 16–18, 20]. In most cases, the assumption is that the user has granted permission to access the private data [1, 9, 10, 16, 17, 20]. Razaghpanah *et al.* [18] and Song *et al.* [19] analyzed the potential exposure of private data to third party services, such as advertisement and tracking services, usually in the form of libraries that are added to the applications. In these cases, the user grants the permission to an installed application without being aware of the third party library that is included within the application and without knowing that the data is available and used by these third-party services.

In order to perform the analysis, the abovementioned studies collected data from mobile devices provided to users for the purpose of the experiment [9, 10] or executed the monitored applications on a dedicated mobile device or emulator [11, 12, 19, 20].

Another threat actor considered in previous research is an adversary that can analyze publicly available location data in online social networks. Such data includes tweets metadata in Twitter, check-in information in Facebook, and proximity services. The adversary in these cases analyzes the data in order to identify users' POIs [21–24], mobility patterns [25–27], and location [28]. Like in our case, the spatial-temporal location data analyzed in these studies is also sparse and inconsistent. Nevertheless, in our research we consider a different adversary model, namely an eavesdropper that can monitor and analyze the network traffic sent from the mobile device (to the LBSs).

In our research we focus on a different adversary (threat actor), such as an ISP, VPN service provider, or proxies (i.e., an eavesdropper on network traffic) that is able to intercept all of the mobile device's network communication. We assume that this threat actor is exposed to *all* data leaks by all of the applications and services installed on the mobile device (as opposed to leaks of data that is available to individual

applications or a location-based service provider). In addition, unlike the works mentioned above in which the user actively installed the applications and granted access to the private data, in our case, the user is unaware that an the adversary may be observing the network traffic.

Several previous studies also focused on a network eavesdropper adversary [8, 20, 29]. Taylor *et al.* [29], for example, showed that by sniffing the network traffic of well-known and heavily downloaded apps, an attacker can obtain a broad spectrum of personal and device identifying information without the user's awareness. They searched for predefined strings of PII (e.g., phone identifiers such as the IMEI and MAC address) and conducted a controlled experiment by running applications on a dedicated mobile phone and collecting the network traffic using a dedicated setup. They did not, however, analyze location data leakage. In contrast, in this research, our focus is on analyzing the extent of location data leakage from devices that are owned and used by real users and by *all* applications that are installed on the users' phones. Ren *et al.* [8] presented Recon, a system which is based on machine learning technique used to automatically identify PII sent in plain text network traffic; the authors tested the proposed approach in a controlled lab environment (i.e., running applications in a sandbox), using devices owned by real users who were required to manually tag potential PII leaks, including location data. The authors, however, didn't focus on estimating the extent and value of the location data leaked by all of the applications running on the device, which is a focus of our study. In addition, we present a step-by-step process that can be applied by the adversary in order to derive valuable information about the user from the raw network traffic.

The protocol used to transfer PII may be vulnerable to cyber attacks and thus creating an additional privacy breach regardless of user awareness. As a case in point, Ren *et al.* [20] presented a review on data leakage in mobile apps by an unsecured HTTP protocol which can be used to identify the user. As part of this process, the adversary can use the PII detector presented by the authors in order to improve the detection of location data within network traffic.

To summarize, in this study we focus specifically on an adversary that can monitor network traffic. While in previous studies the adversary did not consider any noise in the observed data, in this study we discuss a realistic case where the attacker collects and analyzes raw network traffic in a completely unsupervised approach.

2.2 POI Identification Methods

Inferring meaningful locations from aggregated location traces is a field of research that has rapidly evolved since the appearance of cheap mobile GPS devices for civilian use. These devices, as well as smartphones (in the proper navigation and sampling mode) which have become ubiquitous, have a high and constant sampling rate. The availability of a

constant data stream is a common assumption for most studies in location analytics.

Inferring meaningful locations relies upon several major algorithm families: Ester *et al.* introduced DBSCAN [30], a density-based algorithm for spatial data, which provides the ability to determine clusters with undefined shapes and is not bound to a specific number of clusters and does not use temporal data as a parameter. Birant *et al.* extended DBSCAN into ST-DBSCAN [31], which not only uses the spatial data of the database points but also uses its temporal data which is more suitable for spatio-temporal data sets. Another approach introduced by Kang *et al.* [6] clustered places based on time and distance thresholds to differentiate stay points from transit to improve analysis of trajectories. In the current research we refer to this method as the "incremental method." Kang *et al.* [6] used predefined constraints in order to prevent incorrect clustering due to missing information between traces. Montoliu *et al.* [32] also deal with inconsistency and missing data by adding maximum time constraints between traces' constraints. Alvares *et al.* [33] proposed the use of semantic data to better understand the meaning of the collected data, and their method can be used to determine whether a place could be important to the user.

3 Threat Model

The threat model serving as the focus of this research is an adversary that is able to eavesdrop on the mobile device's network traffic and is exposed to personal, sensitive information that is transmitted in plaintext.

Previous research has discussed personal information disclosure and inference from network traffic leakage. However, those studies mainly dealt with inferring static information, such as demographic attributes or other PII which can be observed when the user is connected to a single malicious hotspot. In our case, since we are analyzing location data over time in order to obtain contextual information, capturing network traffic from a single hotspot is insufficient. Thus, in this paper it is assumed that a threat actor can continuously capture the network traffic of the user's mobile device. This special capability is granted to the following threat actors:

Internet service providers and mobile network operators (MNOs) who are exposed to a vast amount of the users' network traffic. The ISP threat model is strong, however we believe that location data leakage is a significant problem in and of itself and therefore should be explored. In addition, although ISP's can derive location information from the cell ID, the derived location is very coarse. On the other hand, the location that can be derived from the leaked data is much more accurate.

VPN and proxy servers, which relay mobile device network traffic to a third party server, can also misuse the unencrypted data sent in the network traffic; these solutions are increasingly being used by mobile users to protect their privacy or

consume restricted entertainment content [34].^{3 4}

Tor-like solutions, are commonly used to protect privacy [35]; in this case, the location leakage data can be used by the exit node to expose the real user location (and possibly the user's identity as well).

The main goal of this study is to estimate the potential privacy exposure of a user by such threat actors if they choose to misuse this data, or by an attacker. Note that previous research (e.g., [14, 36, 37]) proved that it is possible to de-anonymize users based on location traces. Therefore, an adversary that is able to monitor a device's network traffic and infer the user's POIs will be able to apply similar methods (together with information publicly available in phone books and social networks) in order to breach the user's anonymity.

4 Data Collection

In the interest of exploring the extent of location data leakage in the Internet traffic of Android-based smartphones, we developed a dedicated data collection framework. Using this framework, we collected data from 71 participants for an average period of 37 days.

4.1 Data collection framework

The framework consists of three main components: a VPN client that collects all of the network traffic transmitted by the device to the Internet, a dedicated Android agent application that obtains location readings from the device's location API, and a light-weight server.

VPN client. The high volume of Internet traffic on smartphones makes it prohibitive to store this information locally on the device for a long period of time. Alternatively, caching the Internet traffic locally and transmitting it daily to a remote server adds overhead to the device's battery, CPU, and network. In addition, due to security concerns, such operations require super-user privileges, which necessitate rooting the user's device. For these reasons, we opted to use VPN tunneling to redirect the traffic through a dedicated VPN server, where we could record and store the traffic. It should be mentioned that a VPN is the only user space API provided by the Android operating system that can be used to intercept network traffic that does not require super-user privileges.

Android agent (client) application. To understand the quality of location leakage detected in plaintext within the device's network traffic and to estimate the privacy exposure level of users, we developed a dedicated Android application that obtains the actual location of users. The application uses the network location provider API which utilizing three sources of information: GPS, Wi-Fi, and Cell ID (obtained from the cellular network) to assesses the location. The data collected

³<https://www.forbes.com/sites/forbestechcouncil/2018/07/10/the-future-of-the-vpn-market/>

⁴<https://blog.globalwebindex.com/chart-of-the-day/vpn-usage/>

by the agent application was used as the ground truth for the actual location of the users. Applying clustering algorithms on data collected from the mobile device was shown to be accurate and effective for deriving users' POIs [33]. Therefore, we opt to use this approach as our baseline and not to rely on the (subjective) collaboration of the participants in providing the actual/labeled POIs.

Light weight server. This server has two primary objectives. First, it operates as an application server which communicates with the Android agent application and stores all location data in a database. Second, it acts as a VPN server which communicates with the VPN clients, records all of their network traffic, and redirects it to the Internet. To provide VPN connectivity and record the traffic, we created a dedicated LAN (local area network) on that server where every VPN client was assigned to a different IP address in the LAN. The Internet traffic was recorded using "tshark," - a network analysis tool that is capable of capturing packet data from real-time network traffic.

We opted to use this data collection approach for the following reasons. First, because we were performing exploratory research where we didn't know exactly what data we could expect and what data we would want to collect and analyze in advance, we wanted to be able to collect all of the network traffic transmitted by the device. Second, it was important for us to be able to collect data from devices owned and regularly used by the participants. Therefore, we could not use a data collection approaches that requires root access. This also introduced another challenge to our research which is the ability to link network traffic (specifically, location data) with the sending application. Finally, since we wanted to collect data for a long period of time, we followed prior research and opted to use the data collected by the agent as our baseline for the users' POIs and not to rely on the (subjective) collaboration of the participants in providing the actual/labeled POIs which would require significant overhead on the participants (and consequently cannot scale). As noted, this approach was shown to be accurate and effective in prior research.

4.2 Experiment

We conducted an experiment involving 71 participants. The participants were current and former students at two universities located in two different cities in Israel. Additional information about the participants is as follows: 60% are male, and 40% female; 44% are in the 18 to 24 age range, and 56% in the 25 to 30 age range; 51% are undergraduate students, and 49% are graduate student.

In Figure 2 we present the distribution of the number of distinct application installations, focusing only on applications that are granted with location permission. As can be seen, more than 40% of the applications are installed on one device. Furthermore, most of the applications are installed on less

Application name	Percentage of users that installed the application	Application Type	Installations (from Google play store)	Number of apps in this category (Wikipedia)
Whatsapp	100%	User	1-5B	31
Google Maps	100%	Pre-installed	5B+	6
Google Play Services	100%	Depends on Android version	5B+	6
YouTube	100%	Pre-installed	5B+	6
Chrome	90%	Pre-installed	5B+	6
Waze	85%	User	100-500M	334
Facebook	80%	Depends on Android version	1-5B	31
Shazam	70%	User	100-500M	334
Facebook Messenger	67%	Depends on Android version	1-5B	31
Skype	65%	User	1-5B	31
Ynet	52%	User	1-5M	25K
Duolingo	47%	User	100-500M	334
Moovit	47%	User	50-100M	484
IsraelTrain	43%	User	1-5M	25K
GetTaxi	42%	User	10-50M	3925
Instagram	38%	User	1-5B	31
Istudent	32%	User	100-500K	334
Viber	32%	User	500M-1B	43

Table 1: Exploring the popularity of applications installed on the mobile devices of the participants.

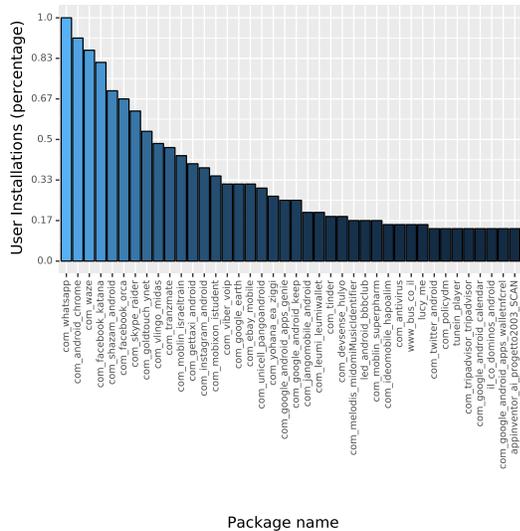
than 20% of the devices. Only a small number of applications are installed on more than 50% of the devices.

We also analyzed the popularity of applications installed on the participants' devices. As can be seen, most of the heavily adopted applications used by the participants in the experiment (shown in Figures 1a and 1b) are extremely popular, with more than one billion installations worldwide (see Table 1). These applications (e.g., WhatsApp, Google Maps, Chrome, YouTube, Facebook, Skype, and Instagram) are not typical to a specific user profile. In addition, because of the diversity of the applications installed on the participants' devices (as shown in Figures 1a, 1b, and 2), we believe that the insights derived from our study are valid and can be generalized to other user profiles.

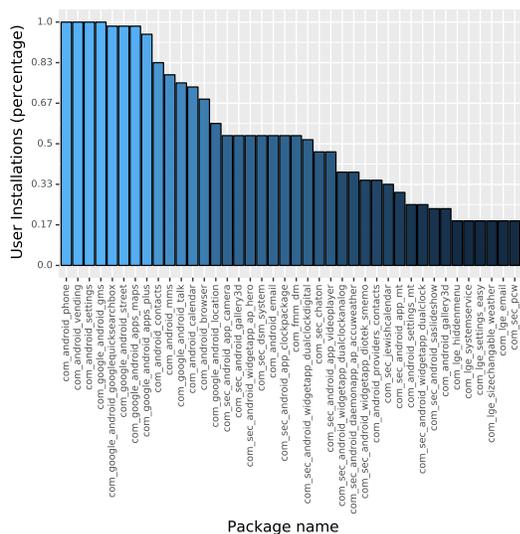
The participants were required to install the two client applications (VPN and Android monitoring agent) on their personal mobile device throughout the experiment, which lasted for an average period of 37 days, depending on the participants' actual engagement. To ensure that the location sampling had minimal impact on the battery, the Android client sampled the location provider for 60 seconds every 20 minutes for most of the experiment. In addition, the server was deployed on an AWS (Amazon Web Services) EC2 instance to ensure high availability.

In Figure 3 we present the amount of time the users participated in the experiment. The distribution of the agent application's actual location sample rate (as observed in the experiment) is presented in Figure 4.

Note that the framework we developed, and specifically the VPN and monitoring agent applications installed on the participants smartphones, were used for data collection, research, and validation only and are not assumed to be part of the threat model.



(a) user applications.



(b) system applications.

Figure 1: Distinct installations of applications among the participants that request location permissions; for each application group (user and system) we sorted the applications by their distinct installations and select the top 40.

4.3 Privacy and ethical considerations

The experiment involved the collection of sensitive information from real subjects for a long period of time. To preserve the subjects' privacy, we took the following steps.

The subjects participated in the experiments at their own will and provided their formal consent to participate in the research. In addition, they were fully aware of the type of data that would be collected and were allowed to withdraw

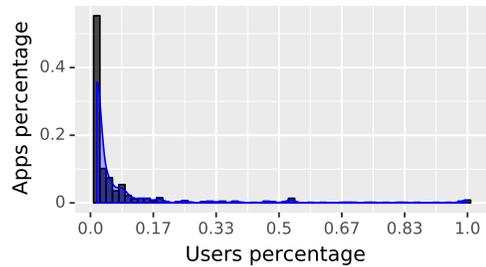


Figure 2: Distribution of distinct installations among applications that request location permission.

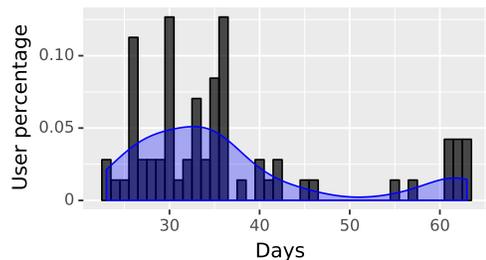


Figure 3: The length of time (in days) users participated in the experiment.

from the study at any time. The subjects received a one-time payment as compensation for their participation.

Anonymization was applied to the data. At the beginning of the experiment, a random user ID was assigned to each subject; this UID served as the identifier of the subject, rather than his or her actual identifying information. The mapping between the UID and the real identity of the subjects was stored in a hard copy document kept in a safe box; we destroyed this document at the end of the experiment.

During the experiment, **the communication between the agents and servers was fully encrypted.** In addition, the data collected was stored in an encrypted database. At the end of the experiment, the data was transferred to a local server (i.e., within the institutional network) that was not connected to the Internet. Only anonymized information of the subjects was kept for further analysis.

Based on these steps, the research was approved by the institutional review board (IRB).

5 Extracting Location Traces from Raw Network Traffic

Location data can be transmitted over network traffic in many formats including: explicit geolocation coordinates, names of cities or POIs, Wi-Fi networks (BSSID), and cellular network data (Cell ID). In this research we focus on explicit geolocation data that is transmitted in plaintext; such structured data

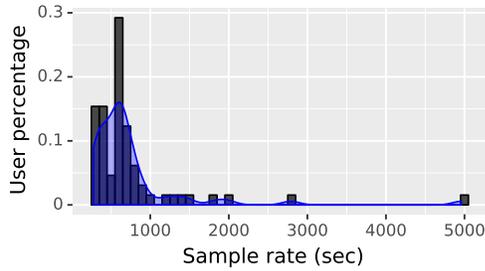


Figure 4: The actual (averaged) location sample rate of the agent installed on the users' devices.

can (potentially) provide more accurate location and is easier to extract and analyze; thus, it introduces a greater risk to the user's privacy if leaked.

5.1 Process description

In order to automatically detect location traces within the network traffic of a mobile device, we capture the data at the IP network layer. Geographic coordinates can be represented in different formats [38]. We perform regex search of the standard Android API [39] representation of the geographic coordinates which is decimal degrees in the following format: XX.YYYYYYYY. We specifically used this regular expression for two main reasons. First, this is the standard format of the Android location API, and thus such expressions are very likely to be a location data. Second, our manual exploration of other location data formats (e.g., names of cities, POIs, and Wi-Fi networks) indicated that they can dramatically increase the number of false positives. For example, city names sent by a weather forecast application do not indicate the true location of the user. Note that as proposed by Ren *et al.* [8], machine learning techniques can be used to automatically identifying PII (including location data) within the network traffic; this, however, still does not ensure that the identified data indicates the true location of the user.

Each result is assigned a timestamp based on the packet capture time. This regular expression may retrieve irrelevant results of simple float numbers which have no geographic meaning and can appear within the network traffic (e.g., the location of an object on the screen). Therefore, in the next step we apply the following heuristics in order to filter out irrelevant results.

Outgoing traffic filter. Extracting geographic coordinates from outgoing traffic (incoming traffic may contain geographic data that is not relevant to the real location of the user such as recommendations of POIs and weather forecasts).

Latitude/longitude pair filter. Extracting only pairs of valid geographic coordinates that were assigned with the same timestamp; this is because a coordinate is represented by two values indicating the latitude and longitude of the location.



Figure 5: The geo-fencing of identified coordinates; filtering coordinates that are outside of a predefined geographical area.

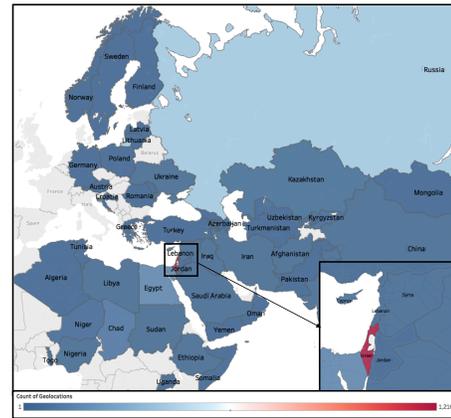


Figure 6: Worldwide distribution of randomly selected geo-locations found in the network traffic for all of the participants.

Geo-fencing filter. Filtering out geographic coordinate that are outside a predefined geo-fence (e.g., the geographical boundaries of a given country or city). In our case, all users were located within the geographical boundaries of Israel during the data collection period, and therefore we filtered out all geographic coordinates that are not within this area (an illustration of the geographical boundaries is presented in Figure 5). Note that to apply the geo-fencing filter in a general and practical case, the attacker can perform reverse geo-coding on the leaked location data and identify the geographical areas that are most likely to be relevant to a user or a group of users. We demonstrate this approach in Figure 6, in which we performed reverse geo-coding on randomly selected samples from the leaked location data (1%). As can be seen, most of the samples are located within the state of Israel.

5.2 Analysis and results

In order to evaluate the amount of location data leakage, we have to determine the accuracy and correctness of the geographic coordinates detected within the network traffic. In our experiment we could compare the geographical coordinates detected within the network traffic to the location data that was sampled by the agent application (installed on the participants' mobile phones). By analyzing the data collected by the agent application, we observed that the location was sampled only 70% of the time (see Figure 4 for the distribution of the location sampling rate). Possible reasons for this are that the device was off, the agent was shut down, or the location service was disabled. Following the above observation, we define the *active time* of a given user as the number of hours at which the agent observed at least one location sample.

Validating the correctness of leaked samples. Using the location samples observed by the agent, we validated the geographic coordinates that were detected within the network traffic. Specifically, a location that was observed in network traffic was classified as a 'true' location only if (1) the timestamp of the detected coordinate was within a predefined time threshold to a location sampled by the agent application, and (2) the measured distance between the two coordinates was below a predefined distance threshold. If just the timestamp of the detected coordinate was close enough (within the time threshold) to a location sampled by the agent application, we labeled the detected coordinate as 'false'; otherwise, it was labeled as 'unknown.'

We tested the labeling of the leaked location data when setting the distance threshold to 250, 500, and 1000 meters and the time threshold to 10 and 30 minutes (see Table 2). As can be expected, when increasing the distance and/or time threshold more leaked location coordinates are labeled as 'true.' On the other hand, doing so may result in incorrect labeling. Therefore, we opted to use the most strict labeling rules in which the distance threshold was set to 250 meters and the time threshold to 10 minutes.

Volume of leaked location data. A total of approximately 474K geolocations (complying with the standard API location regex) were identified within the network traffic of all of the monitored mobile devices. After applying the geo-fencing filter, approximately 347K geolocations remained.

		Distance threshold (meters)		
		250	500	1000
Time threshold (minutes)	10	89K/203K (0.44)	121K/203K (0.60)	157K/203K (0.77)
	30	129K/210K (0.61)	129K/210K (0.61)	165K/210K (0.79)

Table 2: The results of labeling leaked location data when setting the distance threshold to 250, 500, and 1000 meters and the time threshold to 10 and 30 minutes.

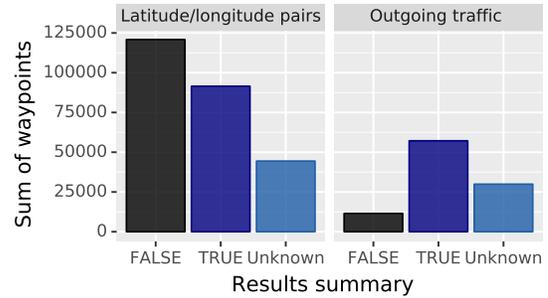


Figure 7: The classification of geolocations detected within the network traffic after applying the latitude/longitude pair filter and after applying both the latitude/longitude pair and outgoing traffic filters.

Figure 7 presents the classification of geo-locations detected within the network traffic after applying the latitude/longitude pair filter (left column) and after applying both the latitude/longitude pair filter and the outgoing traffic filter (right column). Each column presents the distribution of labels ('true,' 'false,' and 'unknown') of the remaining geolocations according to the labeling process described above. In total, after applying the latitude/longitude pair filter, 257K geolocations remained; 36% of them were labeled as 'true,' 47% as 'false,' and the rest could not be labeled. After also applying the outgoing traffic filter, 100K geo-locations remained; 58% of them were labeled as 'true,' 11% as 'false,' and the rest could not be labeled.

These results support our hypothesis that incoming traffic is unlikely to contain relevant geolocations of the mobile device. We can also see that after applying all three filters, 85% of the geolocations that could be labeled (either as 'true' or 'false') indicated the true location of the mobile device. We can assume that the same rate also exists for the geolocations that could not be labeled (i.e., 'unknown').

Rate of data leakage. By analyzing the validated geolocations (i.e., labeled as 'true') of the 71 users, we could see that the mobile devices of about 90% of them were leaking location traces. The *rate of data leakage* of a given user (device) is calculated by dividing the user's *active time* by the number of validated leaked locations. We partitioned the calculated leakage rate into the following groups: 'high,' 'medium,' 'low' and 'no leakage' as presented in Table 3. As can be seen in Table 3, 55% of the devices were leaking location data at a medium rate (once every one to six hours) or high (once every one hour) rate.

5.2.1 Leakage coverage

We define an exposed hour as an hour within the collected data (network or agent) in which at least two valid location leaks were detected.

In order to analyze the coverage over time of relevant (validated) leaked location data, we define the *coverage rate measure* as the total number of exposed hours of network traffic divided by the total number of hours of agent data (i.e., excluding hours in which the agent was not active):

$$CoverageRate = \frac{\#ofExposedTrafficHours}{\#ofAgentDataHours}$$

We assume that a high coverage rate will result in high exposure and discovery rates of users' important places. Figure 8 presents the distribution of the coverage rates of the mobile devices in the collected dataset. As can be seen, for almost 70% of the users the coverage rate is below 0.2.

5.2.2 Leakage inconsistency.

While Table 3 and Figure 8 present the overall average leakage rate of location data, our manual exploration of the data showed that the leaked data exhibits inconsistent, non-uniform, and bursty behavior. As an example, Figure 9 depicts the number of location samples per hour of a single user observed by the agent application (blue line) and within the network traffic (black line). It can be seen that while the agent's sample rate is relatively stable (around 12 samples per hour) excluding minor changes (such as phone shutdown or agent crash), the leaked location data within the network traffic is unstable, ranging from only a few or no leaks to a high rate of leakage.

Thus, in order to analyze and understand the inconsistency in the amount of leaked location data, we computed the relative standard deviation measure for each mobile device by dividing the standard deviation of 'leaks per hour' by the average number of 'leaks per hour.'

Figure 10 depicts the distribution of the values of the relative standard deviation measure of the mobile devices; a value of zero (0) indicates a constant leakage rate.

We were also interested in understanding the distribution of location leakage data during the day. As can be seen in Figure 11, the leakage rate during different hours of the day is correlated with the participants' normal activity during the day (e.g., sleeping at night, attending classes, and taking a lunch break).

Group	Leakage rate	Number of devices	Percentage
High	under 1hr	20	28%
Medium	1-6hrs	19	27%
Low	6+ hrs	23	32%
No leakage	∞	9	13%

Table 3: The leakage rate of different mobile devices. As can be seen, 55% of the devices were leaking location data at a medium rate (once every one to six hours) or high (once every one hour) rate.

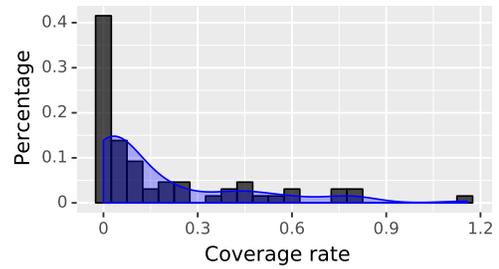


Figure 8: The distribution of the coverage rate measure. The coverage rate measure is defined as the total number of exposed hours of network traffic (i.e., hours in which location leakage was observed) divided by the total number of hours of agent application data.

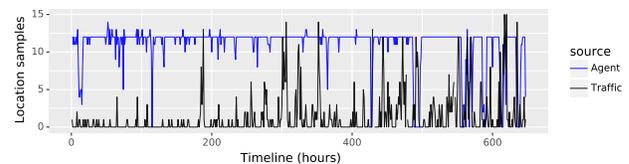


Figure 9: An example of a single user location leakage rate observed by the agent installed on the device (blue line) and within the network traffic (black line).

6 Inferring POIs from Leaked Location Traces

We are also interested in understanding how an attacker can infer meaningful insights from the geolocations (coordinates) that were detected as leaks within the mobile device's network traffic. Specifically, we are interested in identifying a user's POIs and differentiating them from transit or noise data [40].

The most common approach for identifying stay points (or POIs) is by applying clustering algorithms that are not usually bound to a predetermined number of clusters (e.g., k-means) and clustering stay points by spatial or spatio-temporal parameters. In this research we opted to use three different algorithms: incremental clustering [6], DBSCAN [7], and ST-DBSCAN [31]. These algorithms usually make some assumptions about the data. Specifically, it is assumed that the data arrives at a constant rate, which is not correct in our case. Therefore, we made several modifications to the algorithms.

First, for the incremental algorithm, we added the notion of time by calculating the time between samples and defined a bound on that time interval. The pseudo-code of the modified incremental algorithm is presented in Algorithm 1 (see Cluster procedure). As can be seen, the procedure receive three inputs: the distance threshold (denoted by D), the time threshold (denoted by T), and a list of location samples sorted by their timestamps (denoted by WP). The distance threshold specifies the maximal distance (in meters) between the center

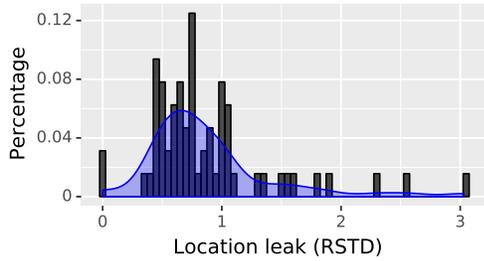


Figure 10: The leakage rate variability as indicated by the leak relative standard deviation measure.

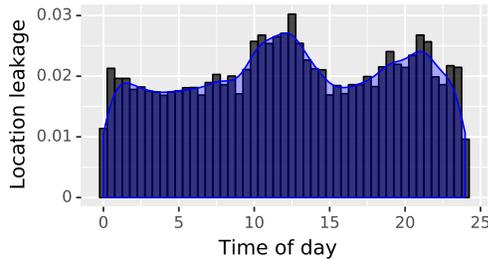


Figure 11: Location leakage rate (in percentage) within different time (hour) of day.

of a cluster to a given location sample. If this distance is less than the distance threshold, the location sample is added to the cluster (lines 13-17); otherwise it is considered an instance of a different cluster, or as a transition state. The time threshold specifies the minimal time for a list of way points to be considered as a cluster. If this time is greater than the time threshold, the list of way points is considered a cluster (lines 18-21); otherwise, it is not considered as cluster.

Second, for both the incremental and ST-DBSCAN algorithms we applied a backtracking procedure, which iterates over the created clusters and merges clusters that are within the distance threshold (lines 28-36). The main benefit of the backtracking procedure is to increase the confidence for repeated clusters over time.

Another approach uses semantic data to determine when a user is at an important place; for example, a location trace at a famous landmark will identify it as an important place for that user [33]. This approach is not relevant in our case, because the POIs are not known in advance; however, we use semantic data from reverse geocoding to eliminate transit geolocations (e.g., highways).

One of the main challenges when applying clustering algorithms in fully unsupervised data is selecting the parameters correctly (namely, the distance and time threshold), since their values directly affect the number and size of clusters detected by the algorithm. In order to address this challenge we tested several time threshold values (15, 30, and 60 minutes) and distance threshold values (100, 250, and 500 meters).

Algorithm 1 Incremental clustering with backtracking and time constraint

```

1: Inputs:
2:    $D \leftarrow$  Distance threshold
3:    $T \leftarrow$  Time threshold
4:    $WP \leftarrow$  List of location samples sorted by their timestamps
5: procedure CLUSTER( $WP, D, T$ )
6:    $cluster \leftarrow \emptyset$ 
7:    $clusterList \leftarrow \emptyset$ 
8:   for  $wp \in WP$  do
9:     if  $cluster = \emptyset$  then
10:       $cluster.add(wp)$ 
11:       $cluster.startTime \leftarrow wp.timestamp$ 
12:     else
13:       if  $distance(cluster.center, wp.location) < D$  then
14:          $cluster.add(wp)$ 
15:          $cluster.updateCenter()$ 
16:          $cluster.endTime \leftarrow wp.timestamp$ 
17:       else
18:         if  $(cluster.endTime - cluster.startTime) > T$  then
19:            $clusterList.add(cluster)$ 
20:            $cluster \leftarrow \emptyset$ 
21:            $cluster.add(wp)$ 
22:         end if
23:       end if
24:     end if
25:   end for
26:   return( $clusterList$ )
27: end procedure
28: procedure BACKTRACK( $ClusterList, D$ )
29:   for  $cluster1, cluster2 \in WP$  do
30:     if  $distance(cluster1.center, cluster2.center) < D$  then
31:        $cluster1.merge(cluster2)$ 
32:        $clusterList.remove(cluster2)$ 
33:     end if
34:   end for
35:   return( $clusterList$ )
36: end procedure

```

Note that we did not have any information about the users' real (confirmed) POIs in order to understand the nature and validity of the POIs identified in the network traffic's leaked locations. Therefore, as a benchmark (and ground truth) we used the POIs identified by applying the incremental clustering algorithm on the Android agent location data (denoted as Incremental-agent). Because the location traces collected by the mobile agent application indicate the true location of the user with a high degree of accuracy, and POIs clustering methods have been shown to be effective in previous work, we found this benchmark sufficient for our purposes.

In Table 4, we present the number of clusters (POIs) detected by the incremental algorithm for different distance and time threshold values. As can be seen, reducing the distance and time thresholds resulted in a large number of clusters, and increasing them resulted in fewer clusters. Nevertheless, it can be seen that the detection rates are not affected by the different parameters. For the rest of the evaluation we set the thresholds for the algorithms at 500 meters and 30 minutes.

The POIs identified by applying the different clustering algorithms on the network traffic's leaked locations (denoted as Incremental-traffic, DBSCAN-traffic, and STDBSCAN-traffic) were compared with the agent-based POIs, namely the Incremental-agent. We calculated the total amount of time spent at each user POI and assigned a weight representing the

		Distance threshold (meters)		
		100	250	500
Time threshold (minutes)	15	371/1402 (0.26)	238/996 (0.24)	184/724 (0.25)
	30	317/1162 (0.27)	218/847 (0.26)	171/627 (0.27)
	60	228/906 (0.25)	183/718 (0.25)	150/532 (0.28)

Table 4: The number of clusters (i.e., POIs) detected by the incremental algorithm when selecting different distance and time thresholds.

POI's significance based on its part of the user's total amount of time spent in all POIs.

POI detection rate. A total of 1,053 POIs (across all users) were identified using the Incremental-agent method. For each traffic-based method we calculated: (1) the total number of POIs identified; (2) the true positive measure (number of POIs that were also detected by the Incremental-agent method); (3) the precision (the true positive value divided by the total number of POIs identified); and (4) the recall (the true positive value divided by the number of POIs identified by the benchmark method, i.e., Incremental-agent).

In addition, previous work on location data analysis showed that previously obtained semantic information (e.g., landmarks, shopping centers, roads, etc.) can be used in order to determine if a location trace is a user's POI or a transit location [33]. Thus, in order to further improve the POI inference process, we used previously obtained semantic information in order to better determine the real POIs. Specifically, we used Google's reverse geo-coding API in order to remove geo-location clusters (i.e., POIs) that are located on highways.

The results are presented in Table 5. As can be seen, the recall, which represents the POI discovery rate, is approximately 20% for all methods; the Incremental-traffic method yields the best results, compared to the other methods, with slightly lower recall but much higher precision. The values within the parentheses represents the results of detected POIs (true positive, precision, and recall) when using this semantic information. As can be seen, using semantic information to eliminate irrelevant location clusters can improve the precision with no effect on the recall. This can be explained by the fact that due to the lower and inconsistent location leakage within the network traffic, the irrelevant location clusters (e.g., highways) are poorly reflected within the network traffic but better captured by the agent.

In a real-life scenario, the adversary will not be able to label the extracted geolocations as 'true,' 'false,' or 'unknown' (as described in Section 5). Therefore, in Table 5 we applied the clustering algorithm on *all* of the geolocations. Nevertheless, in order to evaluate the best results that an adversary could achieve, we applied the POI identification process only on the users' 'true' geolocations. In this case, the Incremental-traffic clustering method achieved a similar recall, however the precision improved dramatically to 95%.

The importance of the 25% identified POIs. The number of identified POIs alone does not necessarily provide a good estimation of the exposure rate of users' whereabouts. For example, let's assume a user with ten different significant locations (POIs). If a user spends 50% of his/her time at home and is at work 35% of the time, by determining the user's home and work locations, we are able to identify the locations at which the user spends 85% of his/her time (although we identified only 20% of the user's POIs).

Thus, in order to estimate the significance of the identified locations (POIs), we computed a weighted measure for the POI detection rate as follows. For each POI detected we computed the relative time spent by the user at that location (i.e., the total time that the user was at the POI divided by the total time the user spent at all POIs). The weighted measure of the POIs was computed from the baseline Incremental-agent method. Then, the POI discovery rate measure was computed by using the weights computed for each POI identified. The results presented in Table 6 show a high weighted POI discovery ratio for the medium and high leakage rates, and a total of 61% weighted POI's exposure rate.

The attack. To conclude, an adversary that can eavesdrop on the user's network traffic can apply the following step-by-step attack in order to infer the user's POIs (and consequently reveal his or her identity). First, the attacker identifies geographic coordinates within the outgoing network traffic (using regex or pre-trained machine learning models). Next, the attacker applies the latitude/longitude pair and geo-fencing filters on the identified coordinates. Finally, the attacker applies our proposed incremental-traffic clustering algorithm on the remaining geo-locations (after applying the filters) in order to identify the user's POIs.

In a real-life scenario an attacker would not have a benchmark to relate to, and inferring a user's POI exposure rate would be based on captured data alone. By deriving a regression model (Table 7), we can see that the user weighted POI exposure ratio parameter has a high correlation with the leakage and coverage rate measures and no significant correlation with the relative standard deviation. Therefore, the attacker can compute the leakage rate and coverage measures from the analyzed traffic and estimate the potential exposure rate of the user's POIs.

	Incremental	DBSCAN	STDBSCAN
Total	282 (263)	470 (374)	339 (264)
True positive	205 (193)	213 (201)	148 (141)
Precision	0.73 (0.73)	0.45 (0.54)	0.43 (0.53)
Recall	0.20 (0.2)	0.20 (0.2)	0.14 (0.14)

Table 5: Recall and precision measures of the three methods (Incremental-traffic, DBSCAN-traffic, STDBSCAN-traffic), when considering the Incremental-agent as the ground truth of the users' POIs. The values within the parentheses represents the results of detected POIs when using semantic information.

Leakage rate	POI discovery ratio	Weighted POI discovery ratio
High	48%	81%
Medium	26%	67%
Low	8%	37%

Table 6: The POI and weighted POI discovery ratios of the Incremental-traffic method. The POI discovery rate is the number of places found in the network data divided by the number of places found by the Incremental-agent method. The weighted POI discovery rate is the amount of time spent at the identified POIs out of the total user time.

	Dependent variable:	
	Weighted POI's exposure rate	
Coverage	0.559***	(0.172)
Leak rate	-0.0000013**	(0.00000)
Relative standard deviation	-0.048	(0.078)
Constant	0.597***	(0.106)
R ²	0.315	
Adjusted R ²	0.283	
Residual Std. Error	0.331 (df = 65)	
F Statistic	9.955*** (df = 3; 65)	

Note: *p<0.1; **p<0.05; ***p<0.01

Table 7: The linear regression model for estimating POIs' exposure rate from network traffic measures.

7 Identifying leaking applications

The goal of the analysis presented in this section is to determine which applications are responsible for leaking the location data and whether the leakage occurs as a result of intentional misuse or a benign application sending location data in plaintext.

One approach for obtaining such information is real-time *on device* monitoring of the installed applications' outgoing traffic [17, 41]. However, starting from Android version no' 8.0, such operations require super-user privileges (which necessitate rooting the user's device). In our experiment, we analyzed the network traffic of the user's personal device, therefore such an approach was not an option.

Deriving information about the leaking applications from the network traffic captured is also challenging for three main reasons: (1) the network traffic captured does not provide an explicit indication of the sending application/service, (2) because of the growing use of cloud services and content delivery networks, many destination IPs are hosted by services such as AWS, Akamai or Google, and (3) location data can be sent to advertisement and intelligence service domains by components embedded in many Android applications.

Given this, we opted to analyze the destination host names observed within the HTTP traffic. We focused specifically on outgoing traffic containing location leaks. By extracting the host names from the HTTP requests that contained the leaked

location data, we identified 112 different services. Then, we analyzed the host names using a public security service (such as VirusTotal), search engines (Google and Whois), and security reports. Based on the results of this analysis, we were able to classify each host name by its reported usage (e.g., weather forecast, navigation, location analytics) and whether it appears to be a legitimate or unwanted/suspicious service. Services with clear/reasonable location usage and no reported security issues were classified as 'benign'; the rest of the services were classified as 'suspicious.'

Figure 12 presents the top 12 host names classified by their category (color) and level of suspiciousness (size of circle). Each host name is placed on the graph according to the average number of detected leakage events (*x*-axis) and the number of participants sending location data to that host name (*y*-axis).

Some of the suspicious domain names include `samsung-buiars.vlingo.com` which was previously published as a Samsung pre-installed speech recognition application called Vlingo. This application was found to be leaking sensitive information. Other example includes the domains, `n129.epom.com` and `mediation.adnxs.com`, which are reported to provide personalized advertisements. An additional significant suspicious domain is `app.woorlids.com`, which was reported to be a location analytics service. Interestingly, our analysis concludes that the Google Maps JavaScript API (`maps.googleapis.com`), which lets Android application developers customize maps with user locations, is also responsible for sending location data in plaintext. This is particularly noteworthy since Google recommends that application developers use the secured Maps JavaScript API (which operates over HTTPS) whenever possible.⁵ Nonetheless, our analysis shows that in practice, developers also use the unsecured Map JavaScript API. Overall, based on the analyzed data, we found that the set of unwanted services is responsible for more than 60% of location leakage events.

Another observation is that although the number of location data leakage events (*x*-axis in Figure 12) for each individual host name is not high, based on the analysis presented in Section 6 we were still able identify the users' significant POIs from the data. We attribute that finding to the fact that there are multiple applications installed on each individual smartphone, which together leak a sufficient amount of information that can be analyzed in order to infer the POIs.

Although the identity of the leaking application is not explicitly indicated in the network traffic, we performed further analysis in an attempt to link the identified host names with the applications installed on the users' mobile devices. In order to do so, we first extracted (using our Android agent) the set of all applications installed on the mobile devices of the users which require both location and network permissions. Next, we computed a modified *tf-idf* measure for each pair

⁵<https://developers.google.com/maps/documentation/javascript/tutorial>

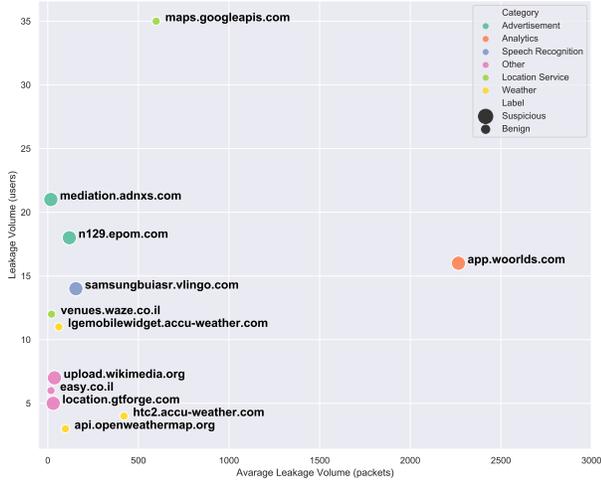


Figure 12: Presenting the top 12 host names classified by their category (color) and level of suspiciousness (size of circle). The x-axis represents the average number of detected leakage events, and the y-axis represents the number of participants sending location data to that host name.

consisting of an application and host name. A well-known measure in the field of text categorization, *tf-idf* is often used as a weighting factor in information retrieval and text mining [42]. The *tf-idf* is a numerical statistic intended to reflect how important a term (i.e., word) is to a document in a collection or corpus. The *tf-idf* value increases proportionally to the number of times a term appears in the document, but it is offset by the frequency of the term in the corpus, which helps to adjust for the fact that some terms appear more frequently in general. In our case, a document is a host name, and a term is an application. The term frequency (denoted by TF) of an application (denoted by a) with respect to a given host name (denoted by h) is calculated as follows:

$$TF_h(a) = \frac{|U_h^a|}{|U_h|}$$

where U_h represents the set of users for which we identify a location leakage (from their devices to h), and U_h^a represents the subset of users from U_h that have application a installed on their devices.

The inverse document frequency (denoted by IDF) of an application is calculated as follows:

$$IDF(a) = -\log_{10} \left(\frac{|U^a|}{|U|} \right)$$

where U^a represents the set of users for which application a was installed on the devices; and U represents the set of all users.

Given the above, the *tf-idf* of an application with respect to a given host name is calculated as follows:

$$TFIDF_h(a) = TF_h(a) * \max(1, IDF(a))$$

The reason for limiting the inverse document frequency (IDF) value to one is to prevent rare applications from achieving a very high *tf-idf* score and consequently be erroneously linked with the host name. In addition, we applied min-max normalization to the *tf-idf* scores of applications in order to keep them within the range of zero to one. A high *tf* value for an application a with respect to a host h indicates that a was frequently observed in devices that transmit location data in plaintext to h .

On the other hand, a high *idf* value for a indicates that a was not observed frequently on the devices in general. Thus, a high *tf-idf* score for application a with respect to host h may indicate that a is related to the location leakage to h .

The raw results of this analysis are presented in Figure 13, where the *tf-idf* values are shown for each host name (x-axis) and application (y-axis). Based on these results, we classified the applications into two categories. The first category includes applications that send the location data to their own hosting service. In this category we can find multiple HTC, LG, and Samsung pre-installed applications found to be related to their own hosting services (htc2.accu-weather.com, lgmobilewidget.accu-weather.com, and samsungbuiasr.vlingo.com), as well as the GetTaxi (com.gettaxi.android) and Easy (easy.co.il.easy3) applications which were found to be related to their hosting services (location.gtforge.com and easy.co.il respectively).

The second category includes applications that send (via integrated "software plug-ins") location data to third party services such as advertisement APIs (n129.epom.com) or analytical services (app.woorlids). In this category we identified a popular student application named com.mobixon.istudent. We also identified applications that send location data to Google Maps services, some of which potentially use the Google Maps JavaScript API in an unsecured manner (the HTTP protocol instead of the HTTPS protocol).

8 Mitigation strategies

In order to reduce the privacy risk associated with location leakage and POI inference presented in our paper, the following countermeasures are suggested and could be further investigated and developed in future work.

Awareness. The most basic approach is increasing the awareness of mobile device users to such risks and providing them with the tools and means to reduce the risk [43]. For example, reducing the risk can be achieved by installing only trusted applications (from trusted sources), monitoring and limiting sensitive permissions such as location and Internet access, and disabling location service on the device while not in use and turning it on only on demand. Tools such as Recon [8] and LP-Doctor [10] can also be used to increase

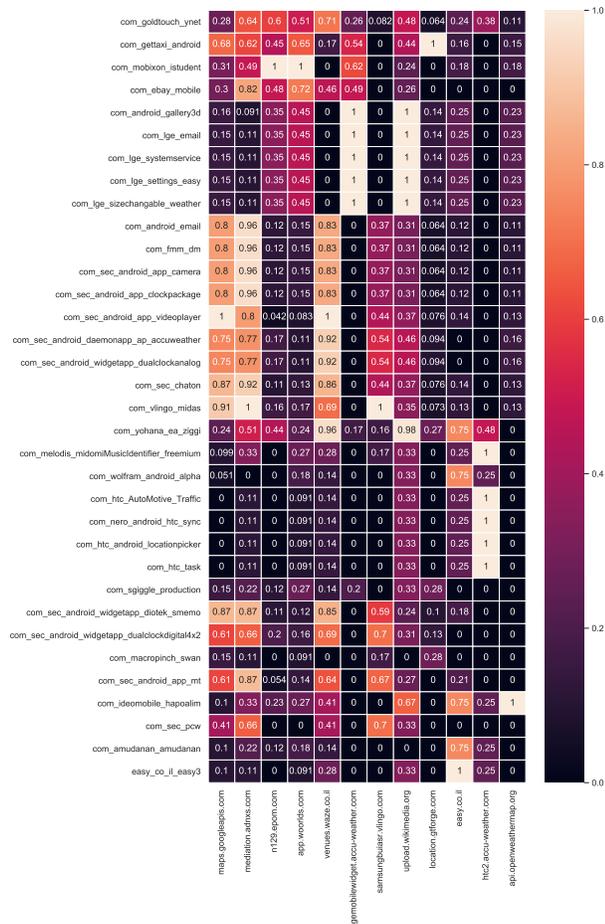


Figure 13: The tf-idf values for each host name and app.

the awareness of users to the privacy risks by presenting alerts and providing the ability to override sensitive transmitted information.

OS policy and tools. The Android OS provides built-in standard security solutions features such as isolation, encryption, memory management, and user-granted permissions for phone resources and sensors. Once permission has been granted by the user, PII handling is done solely by best practice recommendations [44]. This may include enforcing encryption on applications that send sensitive information such as location over the Internet or applying a tool such as PrivacyGuard [19] which is a VPN-based solution that can detect data leakage over the network traffic, modify the leaked information, and replace it with data crafted for privacy protection. Other tools such as MockDroid [45] and LP-Guardian [46] can be used to block an application from accessing the location sensor at runtime. In addition, future versions of the Android OS could consider improving privacy by enabling user decisions regarding background operation involving sensitive data.

Monitoring. Monitoring network traffic by third party security providers in order to detect PII leaks was proposed by [8] and is a useful approach for identifying applications that misuse users’ private information. The proposed Recon application [8] also allows the user to replace the transmitted data with another value selected by the user.

Anonymization and PII obfuscation. Various techniques and algorithms were presented to ensure k-anonymity in LBSs [47–53]. Most of the methods rely on a proxy that filters, manipulates, or generalizes the user’s location data before sending it to the LBS. Such an approach is difficult to apply when the LBS requires accurate or frequent location samples in order to provide the service. In addition, these techniques do not consider the unique threat model of an adversary that has access to the location data of multiple LBSs. Puttaswamy *et al.* [54] suggested that LBSs should move the application functionality to the client devices in order to preserve their privacy. This is however, impractical because in most cases (particularly for free applications or third-party SDK) collecting personal data (e.g., location) is the LBS’s main business model.

An alternative approach can be in the form of an application installed on the mobile device that monitors the location sampling or location leakage over the network traffic and intelligently inject spoofed locations that can make it difficult for a threat actor to infer the true POIs of the user.

Due to the inherent trade-off between providing the required location data to location-based applications (or services) on the one hand, and protecting user privacy on the other hand, we believe that a privacy preserving solution should be: (1) implemented at the OS level and thus have visibility to all running applications and can apply access control policies as well as obfuscation techniques; (2) automated with minimal user involvement and decision making; (3) as proposed in our study, it should analyze the privacy exposure level not only by a single application but also by looking at the network traffic as a whole in order to mitigate the risk of a network eavesdropper adversary; and (4) it should apply intelligent machine learning techniques to automatically identify location data (and other PII) leaks, thereby profiling applications in order to understand the level of granularity of location data required by each application, and to intelligently obfuscate the POIs that can be inferred from the transmitted data. Since the experimental setup of this research was complex and resource and time consuming, we opted to leave the design, development and evaluation of the defense approaches, especially applying PII obfuscation as an adversarial learning approach on the clustering algorithms [55], for future research.

9 Conclusions and Future Work

In this paper, we showed the extent of location leakage from mobile devices and presented a systematic process of extracting location traces from raw network traffic. We analyzed the results of three geo-clustering methods applied over inconsistent network traffic data and showed that existing algorithms yield good results, even with inconsistent location data; this makes any user location privacy vulnerable, even with a low rate of location leakage. Our work enables location exposure assessment by monitoring network traffic and reveals that even relatively low location leakage rates (once in six hours) and coverage of only 20% can expose approximately 70% of the weighted POIs. In future work we plan to further automate the POI identification process by automatically setting up the clustering algorithm parameters, evaluate the extent to which the leaked data can be used to predict the users future location, and develop a mitigation approach based on adversarial learning techniques, which will be based on injecting a minimal number of false locations in clear text to the network traffic, in order to deceive the clustering algorithms (i.e., PII obfuscation).

References

- [1] L. Nguyen, Y. Tian, S. Cho, W. Kwak, S. Parab, Y. Kim, P. Tague, and J. Zhang, “Unlocin: Unauthorized location inference on smartphones without being caught,” in *2013 International Conference on Privacy and Security in Mobile Systems (PRISMS)*. IEEE, 2013, pp. 1–8.
- [2] I. Hazan and A. Shabtai, “Dynamic radius and confidence prediction in grid-based location prediction algorithms,” *Pervasive and Mobile Computing*, vol. 42, pp. 265–284, 2017.
- [3] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly, “Powerspy: Location tracking using mobile device power analysis,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 785–800.
- [4] L. Arigela, P. Veerendra, S. Anvesh, and K. Satya, “Mobile phone tracking & positioning techniques,” *International Journal of Innovative Research in Science, Engineering and Technology*, vol. 2, no. 4, 2013.
- [5] S. Isaacman, R. Becker, R. Cáceres, S. Kobourov, M. Martonosi, J. Rowland, and A. Varshavsky, “Identifying important places in people’s lives from cellular network data,” in *International Conference on Pervasive Computing*. Springer, 2011, pp. 133–151.
- [6] J. H. Kang, W. Welbourne, B. Stewart, and G. Borriello, “Extracting places from traces of locations,” *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 3, pp. 58–68, 2005.
- [7] M. Umair, W. S. Kim, B. C. Choi, and S. Y. Jung, “Discovering personal places from location traces,” in *16th International Conference on Advanced Communication Technology*. IEEE, 2014, pp. 709–713.
- [8] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, “Recon: Revealing and controlling pii leaks in mobile network traffic,” in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2016, pp. 361–374.
- [9] J. Freudiger, R. Shokri, and J.-P. Hubaux, “Evaluating the privacy risk of location-based services,” in *International conference on financial cryptography and data security*. Springer, 2011, pp. 31–46.
- [10] K. Fawaz, H. Feng, and K. G. Shin, “Anatomization and protection of mobile apps’ location privacy threats,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 753–768.
- [11] C. Leung, J. Ren, D. Choffnes, and C. Wilson, “Should you use the app for that?: Comparing the privacy implications of app-and web-based online services,” in *Proceedings of the 2016 Internet Measurement Conference*. ACM, 2016, pp. 365–372.
- [12] E. P. Papadopoulos, M. Diamantaris, P. Papadopoulos, T. Petsas, S. Ioannidis, and E. P. Markatos, “The long-standing privacy debate: Mobile websites vs mobile apps,” in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 153–162.
- [13] P. Regulation, “General data protection regulation,” *Official Journal of the European Union*, vol. 59, pp. 1–88, 2016.
- [14] S. Gambis, M.-O. Killijian, and M. N. del Prado Cortez, “De-anonymization attack on geolocated data,” *Journal of Computer and System Sciences*, vol. 80, no. 8, pp. 1597–1614, 2014.
- [15] V. F. Taylor and I. Martinovic, “A longitudinal study of app permission usage across the google play store,” *CoRR*, abs/1606.01708, 2016.
- [16] A. Shuba, A. Le, M. Gjoka, J. Varmarken, S. Langhoff, and A. Markopoulou, “Antmonitor: Network traffic monitoring and real-time prevention of privacy leaks in mobile devices,” in *Proceedings of the 2015 Workshop on*

Wireless of the Students, by the Students, & for the Students. ACM, 2015, pp. 25–27.

- [17] A. Shuba, A. Le, E. Alimpertis, M. Gjoka, and A. Markopoulou, “Antmonitor: A system for on-device mobile network monitoring and its applications,” *arXiv preprint arXiv:1611.04268*, 2016.
- [18] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill, “Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem,” 2018.
- [19] Y. Song and U. Hengartner, “Privacyguard: A vpn-based platform to detect information leakage on android devices,” in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices.* ACM, 2015, pp. 15–26.
- [20] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez, “Bug fixes, improvements,... and privacy leaks,” 2018.
- [21] K. Drakonakis, P. Ilija, S. Ioannidis, and J. Polakis, “Please forget where i was last summer: The privacy risks of public location (meta) data,” *arXiv preprint arXiv:1901.00897*, 2019.
- [22] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang, “Towards social user profiling: unified and discriminative influence model for inferring home locations,” in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2012, pp. 1023–1031.
- [23] J. Lin and R. G. Cromley, “Inferring the home locations of twitter users based on the spatiotemporal clustering of twitter data,” *Transactions in GIS*, vol. 22, no. 1, pp. 82–97, 2018.
- [24] T.-r. Hu, J.-b. Luo, H. Kautz, and A. Sadilek, “Home location inference from sparse and noisy data: models and applications,” *Frontiers of Information Technology & Electronic Engineering*, vol. 17, no. 5, pp. 389–402, 2016.
- [25] Z. Cheng, J. Caverlee, K. Lee, and D. Z. Sui, “Exploring millions of footprints in location sharing services,” in *Fifth International AAAI Conference on Weblogs and Social Media*, 2011.
- [26] F. Luo, G. Cao, K. Mulligan, and X. Li, “Explore spatiotemporal and demographic characteristics of human mobility via twitter: A case study of chicago,” *Applied Geography*, vol. 70, pp. 11–25, 2016.
- [27] E. Cho, S. A. Myers, and J. Leskovec, “Friendship and mobility: user movement in location-based social networks,” in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2011, pp. 1082–1090.
- [28] I. Polakis, G. Argyros, T. Petsios, S. Sivakorn, and A. D. Keromytis, “Where’s wally?: Precise user discovery attacks in location proximity services,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2015, pp. 817–828.
- [29] V. Taylor, J. R. Nurse, and D. Hodges, “Android apps and privacy risks: what attackers can learn by sniffing mobile device traffic,” 2014.
- [30] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, “A density-based algorithm for discovering clusters in large spatial databases with noise.” in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.
- [31] D. Birant and A. Kut, “St-dbscan: An algorithm for clustering spatial-temporal data,” *Data & Knowledge Engineering*, vol. 60, no. 1, pp. 208–221, 2007.
- [32] R. Montoliu and D. Gatica-Perez, “Discovering human places of interest from multimodal mobile phone data,” in *Proceedings of the 9th International Conference on Mobile and Ubiquitous Multimedia.* ACM, 2010, p. 12.
- [33] L. O. Alvares, V. Bogorny, B. Kuijpers, J. A. F. de Macedo, B. Moelans, and A. Vaisman, “A model for enriching trajectories with semantic geographical information,” in *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems.* ACM, 2007, p. 22.
- [34] M. T. Khan, J. DeBlasio, G. M. Voelker, A. C. Snoeren, C. Kanich, and N. Vallina-Rodriguez, “An empirical analysis of the commercial vpn ecosystem,” in *Proceedings of the Internet Measurement Conference 2018.* ACM, 2018, pp. 443–456.
- [35] A. Mani, T. Wilson-Brown, R. Jansen, A. Johnson, and M. Sherr, “Understanding tor usage with privacy-preserving measurement,” in *Proceedings of the Internet Measurement Conference 2018.* ACM, 2018, pp. 175–187.
- [36] Y.-A. De Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel, “Unique in the crowd: The privacy bounds of human mobility,” *Scientific reports*, vol. 3, p. 1376, 2013.
- [37] P. Golle and K. Partridge, “On the anonymity of home/work location pairs,” in *International Conference on Pervasive Computing.* Springer, 2009, pp. 390–397.

- [38] “Standard representation of geographic point location by coordinates, volume = 2008, address = Geneva, CH, institution = International Organization for Standardization,” Standard, Jul. 2008.
- [39] “Location-Android developer documentation,” <https://developer.android.com/reference/android/location/Location.html/>, 2018, [Online; accessed 12-April-2018].
- [40] D. Ashbrook and T. Starner, “Using gps to learn significant locations and predict movement across multiple users,” *Personal and Ubiquitous computing*, vol. 7, no. 5, pp. 275–286, 2003.
- [41] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson, “Haystack: A multi-purpose mobile vantage point in user space,” *arXiv preprint arXiv:1510.01419*, 2015.
- [42] F. Sebastiani, “Machine learning in automated text categorization,” *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.
- [43] L. Kraus, T. Fiebig, V. Miruchna, S. Möller, and A. Shabtai, “Analyzing end-users’ knowledge and feelings surrounding smartphone security and privacy,” *S&P. IEEE*, 2015.
- [44] “Security Tips-Android developer documentation,” <https://developer.android.com/training/articles/security-tips.html/>, 2018, [Online; accessed 01-April-2018].
- [45] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, “Mockdroid: trading privacy for application functionality on smartphones,” in *Proceedings of the 12th workshop on mobile computing systems and applications*. ACM, 2011, pp. 49–54.
- [46] K. Fawaz and K. G. Shin, “Location privacy protection for smartphone users,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 239–250.
- [47] X. Zhao, L. Li, and G. Xue, “Checking in without worries: Location privacy in location based social networks,” in *2013 Proceedings IEEE INFOCOM*. IEEE, 2013, pp. 3003–3011.
- [48] S. Mascetti, C. Bettini, D. Freni, X. S. Wang, and S. Jajodia, “Privacy-aware proximity based services,” in *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*. IEEE, 2009, pp. 31–40.
- [49] M. Gruteser and D. Grunwald, “Anonymous usage of location-based services through spatial and temporal cloaking,” in *Proceedings of the 1st international conference on Mobile systems, applications and services*. ACM, 2003, pp. 31–42.
- [50] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, D. Boneh *et al.*, “Location privacy via private proximity testing,” in *NDSS*, vol. 11, 2011.
- [51] T. Xu and Y. Cai, “Feeling-based location privacy protection for location-based services,” in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 348–357.
- [52] R. Shokri, G. Theodorakopoulos, C. Troncoso, J.-P. Hubaux, and J.-Y. Le Boudec, “Protecting location privacy: optimal strategy against localization attacks,” in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 617–627.
- [53] M. E. Andrés, N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi, “Geo-indistinguishability: Differential privacy for location-based systems,” *arXiv preprint arXiv:1212.1984*, 2012.
- [54] K. P. Puttaswamy and B. Y. Zhao, “Preserving privacy in location-based mobile social applications,” in *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*. ACM, 2010, pp. 1–6.
- [55] B. Biggio, K. Rieck, D. Ariu, C. Wressnegger, I. Corona, G. Giacinto, and F. Roli, “Poisoning behavioral malware clustering,” in *Proceedings of the 2014 workshop on artificial intelligent and security workshop*. ACM, 2014, pp. 27–36.
- [56] J. Krumm, “Inference attacks on location tracks,” in *International Conference on Pervasive Computing*. Springer, 2007, pp. 127–143.

	Platform	Threat actor	Permissioned access?	Data	Goal	Evaluation environment	Evaluation method	Comments
[9]	cars (24), Nokia phones (40)	location-based service (LBS)	yes - data sent by the device to the service	continuous active sampling of GPS traces	user de-anonymization and POI inference	data collected from cars and phones provided to users for data collection	simulating location streams to the LBS by undersampling the location traces	simulating location exposure to LBSs
[10]	Android (119), iPhone (34) phones	applications	yes - data sampled by the application	continuous active sampling of GPS traces	user POI inference	data collected from devices provided/owned by users	analyzing location traces sampled by the applications	-
[8]	iPhone (63), Android (33), Windows phones	eavesdropper	no	network traffic	identify PII's leaked by specific applications	network traffic collected using a VPN connection while running applications in sandbox (no human activity); running Recon on rooted user devices	using machine learning for automatically identify PII's within unencrypted network flows	-
[11]	iPhone (1), Android (2)	mobile and Web applications	yes - data sampled by the application	network traffic	identify PII's leaked by specific applications	network traffic collected using a VPN/Proxy connection while running applications on the devices (including human activity)	using Recon tool [8] for automatically identify PII's within unencrypted network flows	controlled experiment; PII's are known
[12]	Android (1)	mobile and Web applications	yes - data sampled by the application	network traffic	identify PII's leaked by specific applications	network traffic collected using a man-in-the-middle device while running applications on the device (no human activity)	analyze network traffic and search for predefined strings of PII's	controlled experiment; PII's are known
[16]	Android phones (9)	applications	yes - data sampled by the application	network traffic of applications	block PII's sent by applications	network traffic captured and inspected (using virtual VPN) on the device	analyze network traffic and search for predefined strings of PII's	no evaluation or performance analysis
[17]	Android phones (12)	applications	yes - data sent by the application	network traffic	proposing an on-device framework for analyzing apps' network traffic for performance monitoring, data leakage detection, and user profiling	network traffic collected using a VPN connection	search for predefined strings representing PII's	focus on the efficient data analysis framework
[29]	Android phones	eavesdropper	no	network traffic	identify PII's sent by individual applications	controlled experiment - run applications on a dedicated mobile phone and collect the network traffic using a dedicated network setup	identify PII's within network traffic by searching for predefined strings of PII's	do not focus on location data
[18]	Android phones (11,384)	third-party advertisement and tracking services	yes - without user awareness	network traffic	analyze the extent of PII's leaks to third-parties advertisement and tracking services	network traffic collected using a virtual VPN connection on the device	analyze network traffic and search for predefined strings of PII's	do not analyze location leakage
[1]	Android phone (1)	applications	yes - app installed by the user	WiFi readings	transparently leak location information by using WiFi state permission	data collected from one device owned by a user	comparing the locations inferred from the WiFi state readings to GPS readings	propose a transparent approach for leaking location data
[19]	Android phone and emulator	applications	yes - without user awareness	network traffic	detect location leaks within network traffic and identify user's POIs	network traffic collected using a VPN connection	using predefined filters (rules) to identify PII's and location data	-
[20]	Android phones (5)	applications	yes - data sent by the application	network traffic	detect PII's and location leaks within network traffic sent by an application; analyze leakage trends with application versions	controlled experiment; run applications on dedicated mobile phones and collect the network traffic by forwarding the traffic to a proxy; mimic user interaction using software	using string matching and machine learning models to label traffic containing leaked PII's	-
[56]	cars (172)	location-based services	yes - data sampled by the service provider	network traffic	user POI inference	network traffic collected from the cars	apply simple heuristics for inferring home location	-
[40]	Garmin model 35-LVS wearable (7)	location-based services	yes - data sampled by the service provider	user POI inference and location modeling	data extracted from the devices	apply simple heuristics for inferring home location	cluster GPS data to extract locations that are then incorporated into a Markov model	the research is based on frequent and fresh GPS samples
Our paper	Android phones (71)	eavesdropper	no	network traffic	detect relevant location leaks within network traffic and identify user's POIs	network traffic collected using a VPN connection (to a remote server) as well as location data collected simultaneously from the device	measure the extent of location data leakage in the Internet traffic of Android devices and analyze the value of the leaked data (i.e., infer POIs)	-

Table 8: Summary of related works.

Kindness is a Risky Business: On the Usage of the Accessibility APIs in Android

Wenrui Diao^{*†}, Yue Zhang[‡], Li Zhang[‡], Zhou Li[§], Fenghao Xu[¶], Xiaorui Pan^{||}, Xiangyu Liu[‡],
Jian Weng[‡], Kehuan Zhang[¶], and XiaoFeng Wang^{||}

**Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education,
Shandong University, diaowenrui@sdu.edu.cn*

†School of Cyber Science and Technology, Shandong University

‡Jinan University, {zyueinfosec, zhanglikernel, cryptjweng}@gmail.com

§University of California, Irvine, zhou.li@uci.edu

¶The Chinese University of Hong Kong, {xf016, khzhang}@ie.cuhk.edu.hk

||Indiana University Bloomington, xiaopan@umail.iu.edu, xw7@indiana.edu

‡Alibaba Inc., eason.lxy@alibaba-inc.com

Abstract

The assistive technologies have been integrated into nearly all mainstream operating systems, which assist users with disabilities or difficulties in operating their devices. On Android, Google provides app developers with the accessibility APIs to make their apps accessible. Previous research has demonstrated a variety of stealthy attacks could be launched by exploiting accessibility capabilities (with `BIND_ACCESSIBILITY_SERVICE` permission granted). However, none of them systematically studied the underlying design of the Android accessibility framework, making the security implications of deploying accessibility features not fully understood.

In this paper, we make the first attempt to systemically evaluate the usage of the accessibility APIs and the design of their supporting architecture. Through code review and a large-scale app scanning study, we find the accessibility APIs have been misused widely. Further, we identify a series of fundamental design shortcomings of the Android accessibility framework: (1) no restriction on the purposes of using the accessibility APIs; (2) no strong guarantee to the integrity of accessibility event processing; (3) no restriction on the properties of custom accessibility events. Based on these observations, we demonstrate two practical attacks – installation hijacking and notification phishing – as showcases. As a result, tens of millions of users are under these threats. The flaws and attack cases described in this paper have been responsibly reported to the Android security team and the corresponding vendors. Besides, we propose some improvement recommendations to mitigate those security threats.

1 Introduction

The assistive technologies have been integrated into nearly all mainstream operating systems, which assist users with disabilities in operating their devices. It is not only the kindness of OS vendors but also the requirements of federal law [20]. On mobile platforms, tremendous efforts have been spared

into developing assistive technologies. For example, Android provides TalkBack [7] for users who are blind and visually impaired. They could perform input via gestures such as swiping or dragging on the screen and listen to the feedback in an artificial voice. Other supports include Switch Access (an alternative to using the touchscreen), Voice Access (control device with spoken commands), etc.

Besides the built-in accessibility features provided by Android OS, to support the development of accessible apps, Google also provides app developers with the *accessibility APIs* to develop custom accessibility services. The mission of these APIs is to provide user interface enhancements to assist users with disabilities, or who may temporarily be unable to fully interact with a device [11]. With the accessibility APIs, an app could observe user actions, read window content, and execute automatic GUI operations, which improves the interactions between users and apps. Since these APIs are quite powerful, as a restriction, the accessibility service must be protected by the `BIND_ACCESSIBILITY_SERVICE` permission to ensure that only the system can bind to it.

On the other hand, the powerful capabilities of the accessibility APIs can be exploited for malign purposes. Previous research demonstrated a variety of stealthy attacks could be launched with the accessibility capabilities [35, 37, 43] and investigated the inadequate checks on accessibility I/O paths [36]. However, previous works focused on exploring what kinds of attacks could be achieved through a malicious app with the `BIND_ACCESSIBILITY_SERVICE` permission, and none of them touched the design of the Android accessibility framework.

Our Work. Motivated by the significant security implications of the accessibility service, in this work we perform the first comprehensive study to evaluate the usage of the accessibility APIs and the design of their supporting architecture in Android. In particular, we first conducted a large-scale study on 91,605 Android apps crawled from Google Play to measure the accessibility APIs usage in the wild. The result shows the accessibility APIs have been misused widely. Most assis-

tive apps utilize them to bypass the permission restrictions of Android OS, which deviates from the original mission.

Then, we reviewed the Android accessibility framework to investigate the fundamental reasons for misuse and the potential security risks. Finally, we identify a series of fundamental design shortcomings that can lead to severe security threats. (1) Specifically, we find that there is no restriction on the purposes of using the accessibility APIs. Any app can invoke the accessibility APIs even when the purpose is not for helping the disabled users. (2) Also, we notice that the Android accessibility architecture is event-driven. Under such design, the event receivers do not communicate with the event senders directly. The execution logic of accessibility services only could rely on the received accessibility events. However, the information contained in the events does not provide a strong guarantee to the integrity of event processing. (3) Even worse, Android allows zero-permission apps to inject arbitrary custom accessibility events into the system, which brings the possibility of constructing fraudulent activities.

Exploiting these design shortcomings, we demonstrate two real-world attacks as showcases. That is, a malicious app without any sensitive permission can hijack the execution logic of assistive apps installed on the same phone to *install arbitrary apps* and *send phishing notifications*. Note that, different from the previous works [35, 37, 43], in our attacks, we consider a more general model and do not assume the malicious app has been granted with the `BIND_ACCESSIBILITY_SERVICE` permission.

Following the responsible disclosure policy, we reported our findings to the Android security team and the corresponding vendors. At present, Google has confirmed our discovery and rewarded us with \$200 under the Android Security Rewards Program. The latest update could be tracked through `AndroidID-79268769` and `CVE-2018-9376`.

As mitigations, we also propose some improvement recommendations for each shortcoming. However, to thoroughly address the current security threats, a new accessibility architecture may be needed. How to trade off the security and usability is still an open question.

Contributions. We summarize the contributions of this paper as follows:

- *Data-driven Analysis.* We perform the first large-scale study to measure the usages of the accessibility APIs in the wild (based on 91,605 app samples from Google Play). Our study shows the accessibility APIs have been misused widely.
- *Discovery of New Design Flaws.* After reviewing the design of the Android accessibility supporting architecture, we identify a series of fundamental design shortcomings which may bring serious security risks. We also propose improvement recommendations.

- *Demonstration of Proof-of-Concept Attacks.* We demonstrate two concrete attacks exploiting the design shortcomings of the accessibility framework as showcases: installation hijacking and notification phishing.

Roadmap. The rest of this paper is organized as follows. Section §2 provides the background of the accessibility service on Android. Section §3 introduces the threat model and methodology of this paper. In Section §4, we measure the usage status of the accessibility APIs on a large-scale app dataset. Section §5 summarizes the discovered design shortcomings. Section §6 demonstrates two practical attacking exploiting these shortcomings. Section §7 discusses some attack conditions and limitations. Section §8 proposes some improvement recommendations. Section §9 reviews related works, and Section §10 concludes this paper.

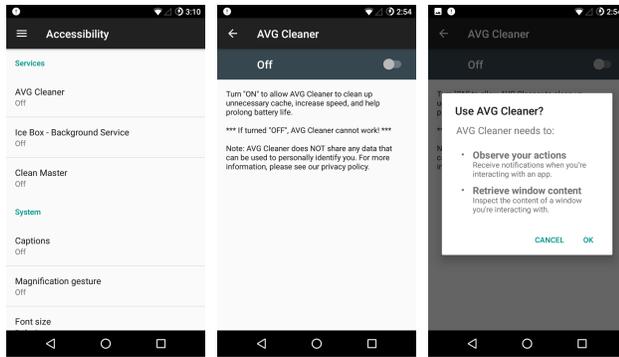
2 Accessibility Service on Android

The accessibility service was introduced by Google starting from Android 1.6 (API Level 4) and gained significant improvement since Android 4.0 (API Level 14). It is designed to implement *assistive technology* with two main functionalities: (1) receiving input from alternative input devices (e.g., voice into microphone) and transforming it to commands accepted by OS or apps; (2) converting system output (e.g., text displayed on screen) into other forms which can be delivered by alternative output devices (e.g., sound through a speaker).

Capabilities. To this end, Android provides a set of capabilities for the accessibility APIs, and they can be grouped into eight categories [5], as listed below:

- *C0: Receive AccessibilityEvents.* This is the default capability ensuring the accessibility service can receive notifications when the user is interacting with an app.
- *C1: Control display magnification.*
- *C2: Perform gestures,* including tap, swipe, pinch, etc.
- *C3: Request enhanced web accessibility enhancements¹.* Such extensions aim to provide improved accessibility support for the content presented in a `WebView`.
- *C4: Request to filter the key event stream,* including both hard and soft key presses.
- *C5: Capture gestures from the fingerprint sensor.*
- *C6: Request touch exploration mode.* In this mode, a single finger moving on the screen behaves like a mouse pointer hovering over the user interface.
- *C7: Retrieve interactive window content.* An interactive window is a window that has input focus.

¹This capability was deprecated in Android 8.0 (API level 26).



(a) Accessibility settings (b) AVG Cleaner (c) Security warning

Figure 1: Turn on accessibility service for AVG Cleaner.

Building An Assistive App. Android provides standard accessibility services (such as TalkBack), and developers can create and distribute their own custom services. An app providing the accessibility service is called *assistive app*. To build an assistive app, developers need to create a service class that extends `AccessibilityService`.

Permission. For security reasons, the accessibility service must be protected by the `BIND_ACCESSIBILITY_SERVICE` permission (protection level: signature) to ensure that only the system can bind to it. An extra requirement is that the user must manually turn on the accessibility switch and confirm the security implications for every accessibility service (assistive app), as demonstrated in Figure 1(c). The listed items in this picture are the capabilities declared by this service.

Service Interaction. The internal mechanism of the accessibility framework is quite complicated, and here we focus on how accessibility events are processed. As illustrated in Figure 2, three components are involved in the Android accessibility service framework: topmost app, system-level service `AccessibilityManagerService`, and multiple assistive apps with custom accessibility services. A typical and simplified invocation process of accessibility service is illustrated as below:

1. *Generate & Send Events:* An accessibility event [2] is fired by the topmost app which populates the event with data for its state (e.g., the changes in the UI) and requests from its parent to send the event to interested parties. The event delivery is based on the binder IPC mechanism via `IAccessibilityManager`.
2. *Dispatch Events:* All generated accessibility events will be sent to the centralized manager of the Android OS – `AccessibilityManagerService`. After some basic checkings (such as event types), it dispatches events to each bound accessibility services through binder `IAccessibilityServiceClient`.

3. *Receive & Handle Events:* Through the callback function `onAccessibilityEvent`, assistive apps obtain the dispatched events and further process them following their own programmed logics. If the assistive app needs to inject actions (e.g., clicking), it could reversely lookup the view hierarchy from the source contained in an event, locate a specific view node (e.g., a button), and then perform actions on the topmost app.

Accessibility Events². As described above, the accessibility architecture is event-driven. The `AccessibilityEvent` is generated by a view and describes the current state of the view. The main properties of an `AccessibilityEvent` include [2]: `EventType`, `SourceNode`, `ClassName`, and `PackageName`. Note that, each event type has an associated set of different or unique properties.

3 Threat Model and Methodology

According to whether the assistive apps are malicious, the security threats related to the accessibility APIs could be classified into two groups. In this paper, we focus on the normal use cases of the accessibility APIs and, therefore, consider the security threats assuming benign assistive apps.

It has been well-studied that malicious apps can exploit the powerful capabilities of the `BIND_ACCESSIBILITY_SERVICE` permission to launch attacks. Previous research [35,37,43] has demonstrated a variety of stealthy attacks could be launched, even complete control of the UI feedback loop (see Section §9 for more details). Such kind of attacks is built on the dominant feature through the accessibility APIs which could achieve the cross-app operations. As shown in Figure 1, the key point of a successful attack is how to induce victim users to turn on the accessibility service.

Threat Model. In our study, we consider a more general model: the attacker only could control a malicious app installed on the victim’s phone without any sensitive permissions. Also, there is a benign assistive app installed on the same phone. We assume the malicious app attempts to hijack the execution logic of the assistive app to perform malicious activities. In this process, the assistive app becomes the confused deputy, and its assistive capabilities are abused.

Methodology. In our study, we employed the following two-step methodology:

- *Measuring the usage of the accessibility APIs in the wild.* With the data collected by ourselves, we could answer whether the accessibility APIs are used correctly by developers as expected.
- *Reviewing the design of Android accessibility supporting architecture.* If the answer of the first step is “no”, we

²We use “`AccessibilityEvent`” and “accessibility event” interchangeably in this paper.

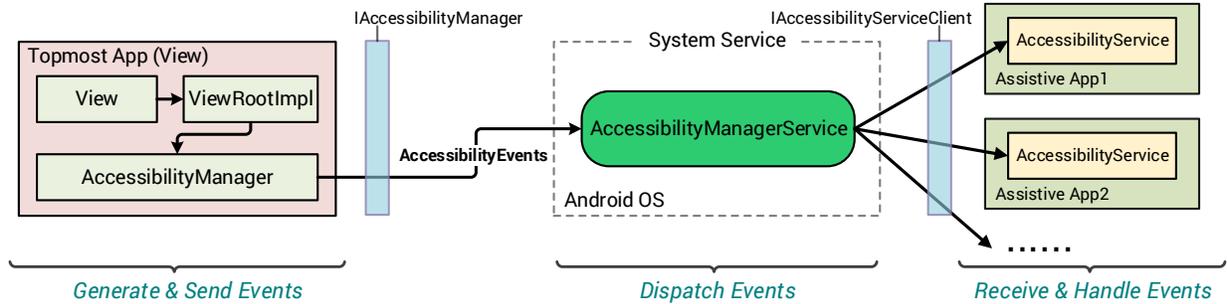


Figure 2: Android accessibility service framework.

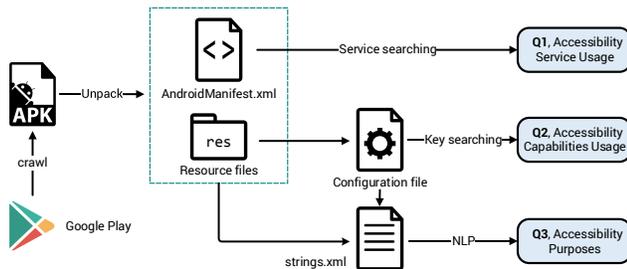


Figure 3: App analysis.

need to investigate the fundamental reasons of misuse and the potential security risks.

4 Accessibility APIs Usage

As the first step, to understanding the accessibility APIs usage status, we carried out a large-scale study on Android apps in the wild. In particular, we try to answer the following three questions.

- Q1:** How many apps use the accessibility APIs?
Q2: What kinds of accessibility capabilities are used?
Q3: What are the purposes of using accessibility APIs?

Since we are interested in the usage status of legitimate apps, an APK sample dataset (91,605 samples, around 1.12 TB) crawled from Google Play was used in our experiment. These samples were collected in 2018, covering most popular apps in each category except for games. As preparation, we used Apktool [9] to disassemble them and obtained the corresponding manifest and resource files (the res folder). To each question, we deployed different analysis approaches, and Figure 3 illustrates the overall analysis process.

4.1 Accessibility APIs Usage

To Q1, we wrote a shell script to search the services protected by the `BIND_ACCESSIBILITY_SERVICE` permission in manifest files.

Result. The result shows around 0.37% apps (337 / 91,605) from Google Play use the accessibility APIs. Also, these 337 assistive apps provide 342 accessibility services³. Though the percentage looks quite low, it does not mean these assistive apps receive little attention. On the contrary, more than half of them (56.7%) have over 1 million installations.

4.2 Accessibility Capabilities Usage

To Q2, accessibility services must declare the needed accessibility capabilities (listed in Section §2) in advance and ask users to confirm, like Figure 1(c).

In particular, every assistive app must prepare a configuration file for its accessibility service, which declares some meta information, such as needed capabilities, expected event types, and timeout. Here we take the configuration file of Network Master (`com.lionmobi.netmaster`) as an example, as shown in Listing 1. The key-value pair `[android:canRetrieveWindowContent="true"]` indicates it needs to invoke the capability of retrieving the active window content. Note that the default capability of receiving accessibility events will always be granted automatically.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <accessibility-service android:description
   = "@string/
   boost_tag_acc_kill_service_description
   " android:accessibilityEventTypes="
   typeWindowStateChanged"
   android:accessibilityFeedbackType="
   feedbackGeneric"
   android:notificationTimeout="100"
   android:canRetrieveWindowContent="true
   "
3 xmlns:android="http://schemas.android.com/
   apk/res/android" />
4 </service>

```

Listing 1: Accessibility configuration of Network Master.

Based on this observation, we obtain the capability usage data by analyzing the accessibility service configuration files.

³Note that, one app could provide multiple accessibility services.

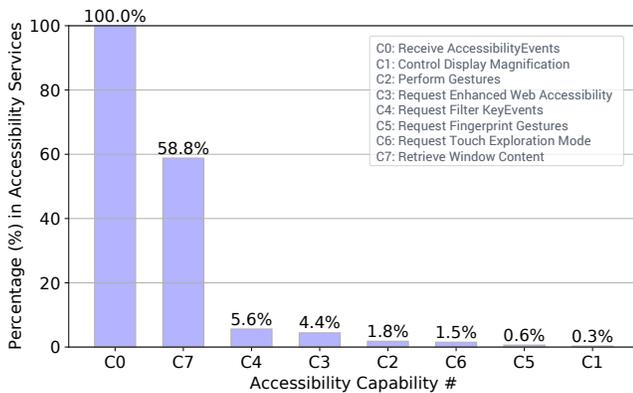


Figure 4: Statistics of capabilities usage.

Among the collected 342 accessibility services, we did not obtain the configuration files from 8 of them. The reasons for the failed retrieval are two-fold: (1) some apps declare the accessibility services in manifest files but do not implement them in code; (2) the other apps deploy anti-analysis protections (e.g., packer), and the resource files cannot be disassembled successfully.

Result. The statistical result is plotted in Figure 4. It shows, besides the default capability of receiving accessibility events (C0), the capability of retrieving the active window content (C7) is the most popular one, say 58.8%. The use cases of the other capabilities are not common. Also, 128 accessibility services (37.4%) only use the default capability.

Our assessment: Most assistive apps only use the accessibility APIs to receive accessibility events (C0) and execute automated clicking operations (C7). Also, the deployment scenarios of some accessibility APIs (C1 and C5) are extremely rare, of which design may be ill-considered.

4.3 Purposes of Using Accessibility APIs

To Q3, it is indeed a non-trivial task. An intuitive solution is to analyze the disassembled code of assistive apps. Through building a context-sensitive call graph, we could track the accessibility APIs invocation and related code executions. However, the challenge is how to identify the ultimate purpose of the accessibility code. For example, we could identify the assistive app injects some clicking actions to the foreground app, but the purpose of the injection operations is not easy to identify, especially for the various nonstandard implementations. To achieve it, we have to manually analyze several implementation samples and build a series of models. When facing obfuscated apps, it will become a tough process. Therefore, static code analysis is not practical for this task.

NLP-based Analysis. As an alternative, we designed a light-weight solution based on natural language processing (NLP)

techniques. We notice that every accessibility service must provide a description to explain why it needs the accessibility service, as shown in Figure 1(b). If the assistive app is legitimate (and appears on Google Play), it has no incentive to provide a fake description in most cases. Also, since this description should be understood by ordinary users, it is usually written in plain languages, like Listing 2 from Network Master. Also, we give more examples in the Appendix.

```

1 <string name="
    boost_tag_acc_kill_service_description
">"Turn it on will help Network Master
stop apps and extend your battery
life. Network Master uses
accessibility service to optimize your
device only. We will never use it to
collect your privacy information. If
you receive warnings about privacy,
please ignore."</string>

```

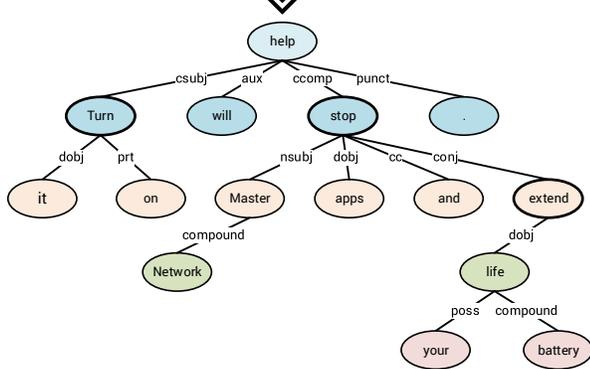
Listing 2: Service description of Network Master.

Motivated by this observation, the purposes of accessibility APIs invocations could be extracted through analyzing their usage descriptions. Here we describe our four-step approach:

1. *Service Description Crawling.* Through analyzing the service configuration file, we could locate and extract the service description from `res/values/strings.xml` (or similar paths). If the description is not in English, we translate it to English through the Google Translate API.
2. *Part-of-Speech Tagging.* Part-of-speech (PoS) tagging is the operation of tagging a word in a text as corresponding to a particular part of speech, based on both its definition and context [17]. Since we are concerned with the actions mentioned in the description, we need to extract the contained action phrases, like “stop apps”. With PoS tagging, we could obtain all verbs in a sentence as preparatory knowledge. In the implementation, we use spaCy [21] to complete this step, and the result is shown in Figure 5 (based on Listing 2).
3. *Semantic Relationship Extraction.* In this step, we extract the [action + object] relationship from the sentences. Our method is to build a semantic relationship tree for each sentence based on spaCy. Then, through breadth-first searching from each verb, we could get the [action + object] relationships. As illustrated in Figure 5, we obtain “help apps”, “stop apps”, and “extend battery life” from the first sentence. Note that, the negative statements have been excluded in this step because the contained actions will not happen. Therefore, “(never) collect private information” is not extracted from the third sentence.
4. *Matching and Classification.* The last step is to build a series of matching rules for classifying the [action +

(Turn VERB) (it PRON) (on PART) (will VERB) (help VERB) (Network PROPN) (Master PROPN) (stop VERB) (apps NOUN) (and CCONJ) (extend VERB) (your ADJ) (battery NOUN) (life NOUN) (. PUNCT) (Network PROPN) (Master PROPN) (uses VERB) (accessibility NOUN) (service NOUN) (to PART) (optimize VERB) (your ADJ) (device NOUN) (only ADV) (. PUNCT) (We PRON) (will VERB) (never ADV) (use VERB) (it PRON) (to PART) (collect VERB) (your ADJ) (privacy NOUN) (information NOUN) (. PUNCT) (If ADP) (you PRON) (receive VERB) (warnings NOUN) (about ADP) (privacy NOUN) (. PUNCT) (please INTJ) (ignore VERB) (. PUNCT)

Part-of-Speech Tagging



Semantic Relationship Extraction

help apps | stop apps | extend battery life | uses accessibility service | optimize your device | receive warnings | ignore warnings

Remarks: The other three relationship trees are omitted due to space limitations. The definitions of grammatical relations (like csbj, aux, ccomp, and prt) are based on the Stanford typed dependencies [31].

Figure 5: [action + object] relationship extraction.

object] relationship sets. We apply a heuristic method to build rules⁴. That is, when an app is not matched by any rule, we will check its accessibility service description and add new rules. If an app is classified incorrectly, we will adjust the existing rules. The formats of rules are [v] for matching a single verb, [n] for matching a single noun, and [v n] for matching action phrases. For example, the rules for the usage of killing processes contain:

[v kill n app; v stop n app; v block n app; v kill n applic; v stop n applic; v block n applic; n batteri; n cach; n acceler; n power]

Note that, we have applied the stemming in matching to avoid the interference of inflected words. Therefore, in this example, “applic” could match “application” and “applications”. Similarly, “stop” could match “stop”, “stops”, “stopping”, and so forth.

After the first step, we obtained 321 descriptions from 342 accessibility services. Among the failure samples, 8 of them lacked available configuration files (the reason has been given in Section §4.2), and 13 of them did not provide the service descriptions.

Result. Finally, we classified the descriptions into 10 categories, and the result is plotted in Figure 6. Among them,

⁴We did not use machine learning-based algorithms (like k-means) in this step because they do not work well on short text due to insufficient features.

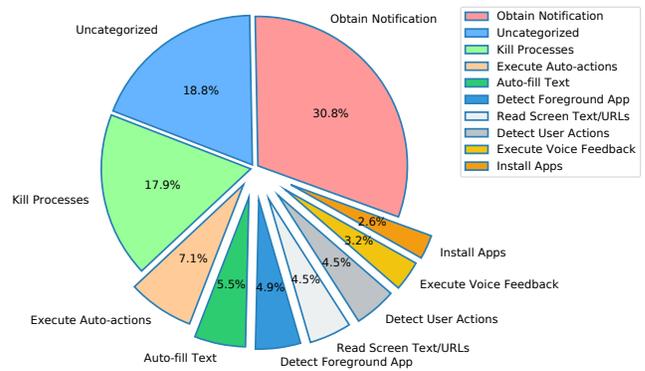


Figure 6: Statistics of purposes.

58 accessibility services (18.8%) are labeled “uncategorized” because their descriptions do not provide useful information⁵.

Among the collected descriptions, only 11 apps mention they are designed for users with disabilities, i.e., 3.2%. It means most accessibility service invocation behaviors are suspicious to some extent. Also, according to Android developers documents: “*accessibility services should only be used to assist users with disabilities in using Android devices and apps* [4].” Here we define that if an app uses the accessibility APIs not for helping the disabled people, it should be treated as a **misuse** behavior. Note that, even such usage is not for malicious purposes, it also could be classified into misuse behaviors.

Through the categorical data analysis and manual confirmation, we identify some typical misuse implementations.

(1) Around 30.8% of assistive apps use the accessibility APIs to *obtain system notifications*, which occupies the most significant share. Most of them belong to the launcher, lockscreen, or status bar apps. Though Android has provided the notification reading APIs and the `BIND_NOTIFICATION_LISTENER_SERVICE` permission in Android 4.3, several assistive apps still keep the accessibility-based approach to avoid compatibility issues.

(2) Another significant category is the purpose of *killing background processes*, say 17.9%. The apps falling in category use the accessibility service to click the “FORCE STOP” button on the app info menu in the system setting. This method could terminate a background process and prevent it from restarting again. Also, the regular `KILL_BACKGROUND_PROCESSES` permission cannot achieve preventing apps restart. The `FORCE_STOP_PACKAGES` permission could achieve it but not available for third-party apps. Therefore, such implementation is popular in battery saver or system booster apps.

⁵For example, “Tap on the top right hand toggle to enable CM Launcher. Attention: You may receive standard privacy warnings. There’s no need to worry, no personal data will be collected.” from CM Launcher (com.ksmobile.launcher).

(3) Executing auto-actions (7.1%) means accessibility services could *automatically complete a series of clicking actions* without user operations. A typical case is that input method apps use it to send GIFs. Users with motor impairments also could benefit such usage.

(4) Auto-filling text (5.5%) is mainly implemented to *automatically fill username and password*. It has become the standard feature in nearly all password manager apps.

(5) Around 4.9% of assistive apps use the accessibility APIs to *detect foreground apps*, such as measuring game playing time. The information about which app is running in the foreground is sensitive because it may be abused for phishing attacks. Therefore, Google has replaced the GET_TASKS permission by the system-level permission REAL_GET_TASKS in Android 5.0 to block accessing from third-party apps. However, with the accessibility service, assistive apps could still obtain the foreground app information.

Our assessment: The accessibility APIs have been misused widely. Most assistive apps utilize them to bypass the permission restrictions of Android OS, which deviates from the original mission.

5 Design Shortcomings

Motivated by the less optimistic results of app scanning, we further reviewed the design of Android accessibility supporting architecture. Finally, we identify a series of design shortcomings lying in the Android accessibility framework, which may bring serious security risks.

Design Shortcoming #1⁶. The accessibility service is designed for users with disabilities and, therefore, enhances the user interactions (i.e., input and output). However, *there is no restriction on the purposes of using the accessibility APIs*. Any app can invoke the accessibility APIs even it is not designed for disabled users. Naturally, in practice, how to use these APIs depends on the developers' understanding and users' demand.

Since the accessibility APIs are very powerful, the assistive apps can know the current foreground app, displayed texts, and user's actions, and even operate arbitrary other apps. On the other hand, due to the increasingly strict restriction of the Android permission system [51], some apps need to find a new code implementation approach to meet the requirements of their function designs. As a result, through combining the accessibility APIs and programming tricks, several dangerous permissions could be bypassed, as summarized in Table 1.

Design Shortcoming #2. The accessibility architecture of Android is event-driven. The execution logic of accessibility services only could rely on the received accessibility events. The assistive app extracts the event properties (like

⁶DS#1 for short. Similarly, we have DS#2 and DS#3.

EventType, ClassName, PackageName) and further judges what happens in the foreground. However, *the information contained in the events cannot provide a strong guarantee to the integrity of event processing*. The current design cannot guarantee that two events with the same properties are definitely generated by the same view, i.e., uniqueness guarantee.

In the accessibility framework, the event receivers (assistive apps) do not communicate with the event senders (topmost app) directly. Such a design ensures the centralized management and efficient event dispatching. On the other hand, the integrity of event processing flow only relies on the checkings implemented by the assistive apps themselves. The unreliable provenance information may confuse the checkings.

Design Shortcoming #3. Android allows zero-permission apps to inject custom AccessibilityEvents into the system. This function is provided to developers to make their custom view components accessible [12]. However, *there is no restriction on how to set the properties of a custom AccessibilityEvent*, which brings the possibility of constructing fraudulent events. Also, though Android OS requires the AccessibilityEvent only could be sent by the topmost view in the view tree [3], this restriction is not enforced.

Any app could implement the following code to construct and inject a custom AccessibilityEvent:

```
1 AccessibilityManager manager = (  
    AccessibilityManager) getSystemService  
    (ACCESSIBILITY_SERVICE);  
2 AccessibilityEvent event =  
    AccessibilityEvent.obtain();  
3  
4 event.setEventType(...);  
5 event.setClassName(...);  
6 event.setSource(...);  
7 event.setParcelableData(...);  
8 ... // Other properties are omitted  
9  
10 manager.sendAccessibilityEvent(event);
```

Listing 3: Inject custom AccessibilityEvent.

6 Attack Case Studies

In this section, we discuss how to exploit the discovered design shortcomings to launch real-world attacks. Specifically, we present installation hijacking and notification phishing as showcases. The attack demos are available at <https://sites.google.com/site/droidaccessibility/>.

6.1 Case Study: Installation Hijacking

In this case, a malicious app without sensitive permission could hijack the execution logic of assistive apps to *install arbitrary apps silently*.

Table 1: Bypassed permissions through the accessibility APIs.

Usage	Bypassed Permission	Protection Level
Obtain notification	BIND_NOTIFICATION_LISTENER_SERVICE [†]	Signature
Kill processes	FORCE_STOP_PACKAGES	Not for third-party apps
Execute auto-actions	INJECT_EVENTS	Not for third-party apps
Auto-fill text	BIND_AUTOFILL_SERVICE [†]	Signature
Detect the foreground app	REAL_GET_TASKS	Not for third-party apps
Install / Uninstall apps	INSTALL_PACKAGES	Not for third-party apps
	DELETE_PACKAGES	Not for third-party apps

[†]: The corresponding service must be protected by this permission to ensure that only the system can bind to it.

The popularity of Android is primarily due to a wide variety of apps provided by Google Play – the official Android app store. However, due to the policy restriction, the Google service framework (including Google Play) is not available in some countries. Also, some apps on Google Play are region-locked. Therefore, third-party app stores (*store app* for short) become an alternative choice, such as 1Mobile [1], Amazon Appstore [6], and APKPure [8].

In Android, the `INSTALL_PACKAGES` permission is designed to prevent the apps from unknown sources to be installed silently. Also, it is a system-level permission and not available for third-party apps. As a result, third-party store apps have to work as APK downloaders and ask the user to click the “INSTALL” button of the Installer by themselves, as shown in Figure 7(a). However, with the accessibility APIs, store apps can achieve the automatic installation by clicking the “INSTALL” button programmatically, which saves user clicks and bypasses the `INSTALL_PACKAGES` permissions. Such implementation improves the user experience but disobeys the mission of the accessibility APIs [DS#1].

Logic Analysis. Here we describe the logic implementation of the automated installation of store apps, as illustrated in Figure 7(c). After the target APK file has been downloaded to the device, the store app utilizes the Intent mechanism [13] to load this APK file. Then Android OS will invoke a proper program (i.e., `PackageInstaller` [16] in this case) to process it. After that, the Installer requests the user to confirm the installation and required permissions. Note that, during this process, the store app continuously monitors the change of foreground UI through filtering `AccessibilityEvents`. When it finds that the `PackageInstaller` is launched to process the APK file just downloaded, it will invoke the accessibility service to click the “INSTALL” button automatically.

However, we find that, before deciding whether to click the “INSTALL” button, the checking logic (Step ③ in Figure 7(c)) of the store app is vulnerable. This step checks four parameters of incoming `AccessibilityEvents`:

1. `SourceNode != null?` (If null, the store app cannot retrieve the window content, locate the “INSTALL” button, and execute the clicking action.)

2. `EventType == TYPE_WINDOW_CONTENT_CHANGED?` (This type of events is usually triggered by adding or removing views.)
3. `PackageName == com.android.packageinstaller?` (This ensures that the app running in the foreground is the `PackageInstaller`.)
4. `Text` is the name of the downloaded app (e.g., “WhatsApp” in Figure 7(a))? (This ensures that the app being installed is the one just downloaded.)

If all four conditions pass, the store app will believe the `PackageInstaller` is processing the downloaded APK file [DS#2]. Unfortunately, this `AccessibilityEvent`-based checking is not complete, and a malicious app installed on the same phone can construct a scenario passing the checking conditions to hijack the work-flow of store apps.

Attack. In this attack, the malicious app only declares the `READ_EXTERNAL_STORAGE` permission, a very common permission. Its payload contains a repackaged Trojan APK file disguised as a popular app, like WhatsApp. This Trojan app can execute various malicious operations with many dangerous permissions, like Figure 7(b).

First, the malicious app running in the background monitors the download folder of the victim store app. In general, since the downloaded APK files are not sensitive data, this folder is usually located in the public storage of the device [19]. Therefore, any app with the `READ_EXTERNAL_STORAGE` permission could access it. Note that, if the store app keeps the downloaded APK files in its private folder, the malicious app will not be able to monitor the file downloading status and further launch the hijacking attack. (Un)fortunately, using the public storage for saving temporary data is a widespread operation in Android apps [34, 40], including at least 11 popular store apps as listed in Table 2.

During the monitoring, if a new cache file appears in this folder, it means the store app starts to download a new APK file. The code implementation could be based on the `Runnable` interface [18] for periodic file existing checking. Through identifying the name of the cache file, the malicious app could know what app is being downloaded because the file

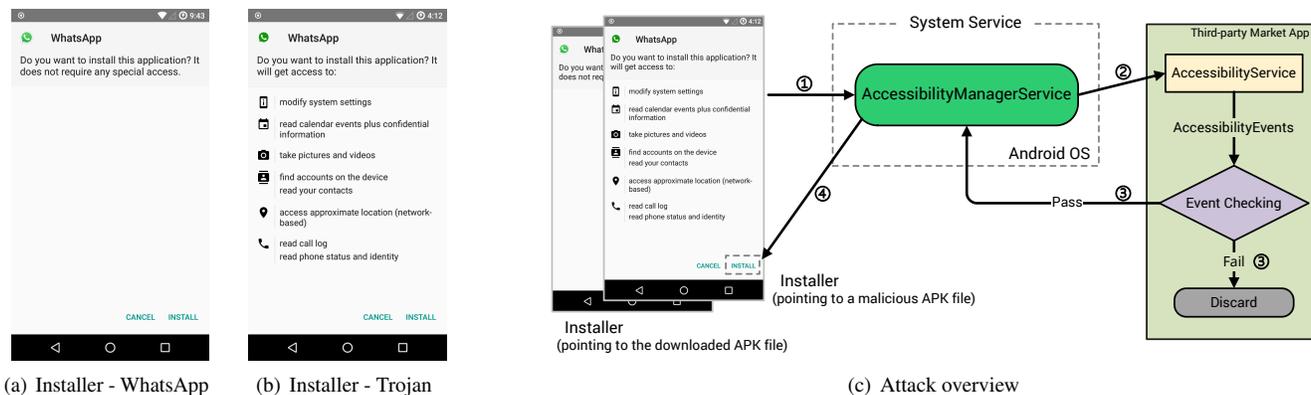


Figure 7: Installation hijacking attack.

name is usually the hash value (MD5 or SHA-1) of the being downloaded APK file or contains the package name. Taking APKPure as an example, the default path for saving APK files is /Download on the external storage, and the format of the cache file name is `WhatsApp_Messenger_[xxx].apk.tmp`, e.g., `WhatsApp_Messenger_2a1417b0.apk.tmp`.

Next, if the malicious app finds the store app is downloading the target app (i.e., WhatsApp in our case), the hijacking attack will be launched. When the downloading completes (`WhatsApp_Messenger_[xxx].apk.tmp` becomes `WhatsApp_Messenger_[xxx].apk`), the malicious app utilizes the same Intent mechanism as the store app to load its Trojan APK file immediately, as shown in Figure 7(b). The *PackageInstaller* pointing to the Trojan APK file (*Installer-A*) will happen to cover the *PackageInstaller* pointing to the downloaded APK file (*Installer-B*), as illustrated in Figure 7(c). Note that, this step creates a race condition (*Installer-A* vs. *Installer-B*), and the attack may fail if the Intent from the malicious app is not processed by the OS at the right time. In practice, the success rate of attacks could be significantly improved through adjusting the point in time of launching *Installer-A*. For example, on Motorola Moto G3 (the device used in our attack demo), when the downloading completes, the malicious app will wait 400ms before launching *Installer-A*. Following this trail, in experiments, we achieved nearly 100% success rate (*Installer-A* covering *Installer-B*). Also, it is an empirical time value and may be different on other devices with varying computing performance.

At this moment, the `AccessibilityEvent` from the *Installer-A* is almost the same as the one from *Installer-B*, which meets all four conditions listed previously. As a result, the store app will be deceived into thinking the *Installer-A* is processing the APK file it just downloaded, so it decides to click the "INSTALL" button. Finally, the repackaged Trojan app prepared by the attacker is installed on the phone.

Summary. The checking logic of store apps entirely depend on the information contained in `AccessibilityEvents`.

Table 2: Vulnerable third-party app stores.

Store	Package Name	Version
APKPure	com.apkpure.aegon	2.12.2
1Mobile Market	me.onemobile.android	6.8.0.1
360 Mobile Assistant	com.qihoo.appstore	7.1.90
Baidu Mobile Assistant	com.baidu.appsearch	8.5.1
Sogou Mobile Assistant	com.sogou.androidtool	6.7.2
MoboMarket	com.baidu.androidstore	4.1.9.6222
PP Assistant	com.pp.assistant	6.0.8
AppChina	com.yingyonghui.market	2.1.62716
Lenovo Le Store	com.lenovo.leos.appstore	9.8.0.88
2345 Mobile Assistant	com.market2345	5.6
Wandoujia	com.wandoujia.phoenix2	5.74.21

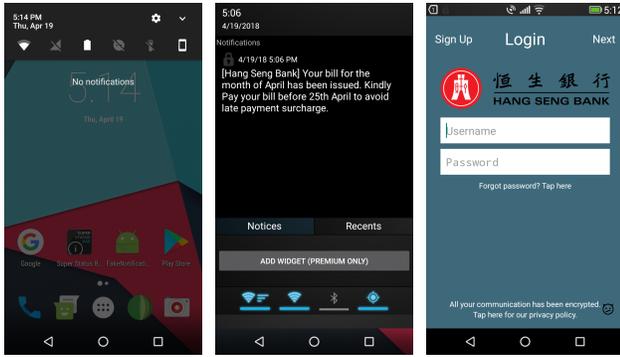
However, the verified factors cannot guarantee which APK file is being processed, which results in the possibility of creating a race condition.

Scope of Attacks. We checked popular third-party store apps and found at least 11 of them (with tens of millions of users [23, 24]) suffer from the security risk of installation hijacking, as listed in Table 2.

6.2 Case Study: Notification Phishing

In this case, a zero-permission malicious app could exploit the execution logic of assistive apps to *send phishing notifications to users*. Also, this attack is different from the direct notification abuse attack [47]. Even the attack app has been blocked for sending system notifications, this attack still works.

Here we consider the apps with the function of notification management, such as status bar app. On Android, the system default status bar could be replaced by third-party status bar apps for a better experience. They could provide several advanced features, like replaceable theme styles, customized fonts, spams filtering, and so forth. On Google Play, there are several popular status bar apps with over one million in-



(a) Original status bar (b) Phishing notification (c) Phishing Activity

Figure 8: Notification phishing attack.

stallations, such as Super Status Bar⁷ (`com.firezenk.ssb`), Status (`com.james.status`), and Material Notification Shade (`com.treydev.msb`).

As an essential function, status bar apps need to obtain system notifications and notify the user. After Android 4.3 (API level 18), third-party apps could obtain system notifications through the `NotificationListenerService` [15] with the `BIND_NOTIFICATION_LISTENER_SERVICE` permission. However, for the devices equipped with old Android versions, the only method of obtaining system notifications is to utilize the accessibility service. Due to the Android fragmentation problem and the consideration of backward compatibility, this accessibility-based notification obtaining method is still very popular. This observation also has been confirmed by our study in the accessibility purpose analysis, say 30.8% usage (see Section §4.3). Again, such implementation is not designed for disabled users and disobeys the mission of the accessibility APIs [DS#1].

Logic Analysis. Here we describe the execution logic of accessibility-based notification obtaining. First, the status bar app filters the received accessibility events for notifications. If the `EventType` is `TYPE_NOTIFICATION_STATE_CHANGED`, it will believe the system just dispatches a new notification [DS#2]. Then the status bar app further extracts the properties of this event, and parses the necessary information, like the notification title, content, parcelable data. However, we find this process is vulnerable. A zero-permission malicious app could construct a custom `AccessibilityEvent` with phishing information and cheat the event receiver.

Attack. As preparation, our zero-permission malicious app has been installed on the user’s phone and runs in the background. Taking Super Status Bar as an example (Figure 8), the attack app intends to send a phishing notification disguised as a message from a bank app. Therefore, it needs to construct

⁷At present, this app is unavailable on Google Play, see the “Impact” part of this subsection for more details.

and inject a custom `AccessibilityEvent` with the following properties [DS#3]:

1. `EventType = TYPE_NOTIFICATION_STATE_CHANGED`.
2. `ClassName = android.app.Notification`.
3. `PackageName = com.hangseng.rmobile`, notification sender, a bank app.
4. `SourceNode = null`, there is no source for the type of `TYPE_NOTIFICATION_STATE_CHANGED` [DS#2].
5. `ParcelableData` is set as a `Notification` [14] instance which contains the phishing message and an `Intent` pointing to a phishing `Activity` (prepared by the malicious app) disguised as the bank app.

When Super Status Bar receives the custom (phishing) `AccessibilityEvent`, it will think the bank app just posts a new notification to the system. Then it parses the properties of this event and displays the phishing notification in its status bar, like Figure 8(b). After the user notices this new notification and clicks it, the status bar app will load the contained `Intent`. Finally, the phishing `Activity` is launched and induces the user to fill her credentials, as shown in Figure 8(c).

Summary. As mentioned in Section §2, different types of accessibility events may have different properties. To some specific types, like `TYPE_NOTIFICATION_STATE_CHANGED`, there is no `SourceNode` property, which results in that assistive apps cannot identify the original senders.

Impact. All assistive apps implementing the accessibility-based notification obtaining suffer from the security risk of notification phishing. This attack method and the design shortcoming on custom `AccessibilityEvent` have been reported to the Android security team and assigned tracking id `AndroidID-79268769`. At present, they have acknowledged our report and rewarded us with \$200. They mentioned it was also “*reported by an internal Google engineer*”. In other words, they confirmed we discovered the problem independently. Also, a CVE-ID has been assigned – `CVE-2018-9376`. Besides, after our report, the vulnerable app Super Status Bar (`com.firezenk.ssb`) was removed from Google Play.

7 Discussions and Limitations

Here we discuss some attack conditions and the limitations existing in our experiment and analysis.

Attacks without accessibility services. To the installation hijacking attack, if the accessibility service is not presented, the user will be involved in the installation process. To an experienced user, she may notice the unusual permission requests (see Figure 7(b)) and rejects the installation. To the notification phishing attack, if the accessibility service is not presented, how to select the time of showing the phishing

Activity will become a problem. This is the primary technical challenge of Activity phishing attacks on Android.

APK dataset. The dataset for app scanning experiment contains 91,605 samples, and 337 assistive apps (containing 342 accessibility services) were identified. Also, due to anti-analysis protection and legacy code, only 334 service samples could be used for subsequent analysis. Our dataset could be extended to obtain more apps for analysis.

Dishonest descriptions. In Section §4.3, our analysis is based on the accessibility service descriptions provided by assistive apps. Though these apps are legitimate, it is still possible that their descriptions are not honest. They may conceal (parts of) their true intentions for some reasons. Such a situation may affect the accuracy of our purpose analysis.

Misuse Identification. In some cases, it is difficult to judge whether the usage behaviors are misuse, especially executing auto-actions. For example, Automate (com.llamalab.automate) could help users create their automations using flowcharts. The supported actions include automatically sending SMS or E-mail, copy files to FTP or Google Drive, play music or take photos, etc. According to the introduction on its website [10], we believe this app is not designed for disabled users, but it is difficult to judge based on its usage descriptions or behaviors.

8 Recommendations to More Secured Accessibility APIs and Framework

In this paper, we systematically analyze the usage and security risks of the accessibility APIs. Given the design shortcomings of Section §5, we propose some possible improvements to mitigate these security risks.

At the high level, the accessibility APIs are very special because they are designed for the users with disabilities. Therefore, the usability is essential in the framework design. It cannot be too complicated for disabled people. The trade-off between security and usability is still an open question. The shortcomings (**DS#1**, **DS#2**, and **DS#3**) discovered in this paper are the fundamental design issues of the event-driven accessibility framework. A new architecture may be needed to completely solve them. At this moment, it is out of the scope of this paper, and here we discuss some targeted improvements for each shortcoming.

To **DS#1**, ideally, if an app is not designed for disabled users, it should not invoke the accessibility APIs. The problem is that some assistive apps belong to the killer apps with millions of installations, and (parts of) their core functionalities rely on the accessibility service, such as LastPass (com.lastpass.lpandroid) and Universal Copy (com.came1.corp.universalcopv). On November 2017, Google required the assistive app developers must explain how their apps are using the accessibility APIs to help users with disabilities, or their apps will be removed from the Play

Store [32]. However, according to our observation, this plan was not executed smoothly, and Google gave up due to the public outcry about favorite apps will stop working [27]. The lesson to be learned here is that whether something is a “misuse” is mostly determined if the users are happy with how that something is used.

We recommend designing new APIs for the requirements of misuse cases. The existence of misuse cases reflects the current Android APIs cannot meet the requirements of developers. New APIs and permissions could be added to make developers give up using the accessibility APIs. Such an improvement will be once and for all. Google also has made such an attempt. On Android 8.0, a new permission `BIND_AUTOFILL_SERVICE` is added, and password manager apps could utilize this new permission to achieve the auto-fill feature [41]. Due to Android fragmentation, it may take a long time before all relevant issues are fixed. On the other hand, the introduction of new APIs will bring some compatibility problems inevitably. For example, the apps developed with the new APIs cannot run on old devices directly. As a result, the developers have to use the Android Support Library [22] to achieve backward compatibility. Even so, due to the limitations of the host device platform version, the full set of functionality may still be unavailable.

To **DS#2**, under the current architecture, it is nearly impossible to fix this design shortcoming. Since the accessibility event senders and receivers do not interact directly, it is difficult for an assistive app (receiver) to identify the event sender.

We recommend improving the execution logic of assistive apps as short-term mitigation. For example, in the case of installation hijacking, the store app should save the downloaded APK files to its private data folder (i.e., internal storage) [19], which would significantly reduce the chance of being identified what APK file is being downloaded. Also, in the case of notification hijacking, the status bar app should not launch the (unreliable) Intent contained in the received `TYPE_NOTIFICATION_STATE_CHANGED` events.

To **DS#3**, the basic information of custom accessibility events should not be filled by third-party apps, including `SourceNode`, `ClassName`, and `PackageName`. Only the OS could fill such information. This restriction ensures the sender information cannot be tampered with.

On the other hand, a new permission could be added for restricting sending custom accessibility events. Since this functionality is provided to developers to make their custom views accessible, it should not be used by any app without restrictions. At least, more restrictions should be applied to the allowed types and numbers of custom events.

9 Related Work

Assistive technologies do not come at no cost. In this section, we review the related works on the security issues of accessibility techniques.

Jang et al. [36] presented the first security evaluation of accessibility support for four mainstream platforms (Microsoft Windows, Ubuntu Linux, iOS, and Android). Their study demonstrated that inadequate security checks on I/O paths make it possible to launch attacks from accessibility interfaces. It is the closest work to us. The difference is that this study focused on the accessibility module I/O and did not touch the underlying design of Android accessibility framework.

On the Android platform, Kraunelis et al. [37] first noticed the possibility of attacks leveraging the Android accessibility framework. More recently, Fratantonio et al. [35] designed the “cloak and dagger” attack. Their attack combines the capabilities of the `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE` permissions, which achieves the complete control of the UI feedback loop. Aonzo et al. [28] uncovered the design issues of mobile password managers and mentioned the misuse of the accessibility service (though it is not the focus of this work). Naseri et al. [43] investigated the sensitive information leakage through the accessibility service. They found 72% of the top finance and 80% of the top social media apps are vulnerable. Different from our work, previous works focused on exploring what kinds of attacks could be achieved through a malicious app with the `BIND_ACCESSIBILITY_SERVICE` permission. Our study focused on evaluating the usage of the accessibility APIs and the design of their supporting architecture. Also, the demonstrated attacks do not need any sensitive permission.

To the security risks of voice control, Diao et al. [33] first discovered the Android built-in voice assistant module (Google Now) could be injected malicious voice commands by a zero-permission app. Some subsequent improved attacks are designed, like hidden voice commands [25, 29, 46, 48] and inaudible voice commands [45, 49]. The corresponding defense mechanisms also have been proposed, like articulatory gesture-based liveness detection [50], tracking the creation of audio communication channels [44], using the physical characteristics of loudspeaker for differentiation [30], utilizing the wireless signals to sense the human mouth motion [42].

In this paper, we present installation hijacking and notification phishing as showcases. Some other works also achieve similar attacks on Android with different approaches or adversary models, such as abusing the notification services [47], exploiting push-messaging services [39], ghost installer attack [38], and UI redressing attacks [26].

10 Conclusion

In this paper, we systematically studied the usage of the accessibility APIs and the design of their supporting architecture. Through code analysis and a large-scale apps scanning study, we identified a series of fundamental design shortcomings that may bring serious security risks. As showcases, we presented two concrete attacks exploiting these shortcomings: installation hijacking and notification phishing. As mitiga-

tions, we also propose improvement recommendations. We believe the security threats reported in this paper are not just isolated incidents. A new accessibility architecture may be needed to completely solve these flaws.

Acknowledgements

We are grateful to our shepherd Jason Polakis and the anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (NSFC) under Grant No. 61902148, No. 61572415, Hong Kong S.A.R. Research Grants Council (RGC) General Research Fund No. 14217816, and Qilu Young Scholar Program of Shandong University.

References

- [1] 1Mobile. <http://www.lmobile.com/>.
- [2] AccessibilityEvent. <https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent.html>.
- [3] AccessibilityEvent.java. https://android.googlesource.com/platform/frameworks/base/+android-8.1.0_r27/core/java/android/view/accessibility/AccessibilityEvent.java.
- [4] AccessibilityService. <https://developer.android.com/reference/android/accessibilityservice/AccessibilityService>.
- [5] AccessibilityServiceInfo. <https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo.html>.
- [6] Amazon Appstore. <https://www.amazon.com/androidapp>.
- [7] Android accessibility overview. <https://support.google.com/accessibility/android/answer/6006564>.
- [8] APKPure. <https://apkpure.com/>.
- [9] Apktool. <https://ibotpeaches.github.io/Apktool/>.
- [10] Automate. <https://llamalab.com/automate/>.
- [11] Building Accessibility Services. <https://developer.android.com/guide/topics/ui/accessibility/services.html>.
- [12] Building Accessible Custom Views. <https://developer.android.com/guide/topics/ui/accessibility/custom-views.html>.

- [13] Intents and Intent Filters. <https://developer.android.com/guide/components/intents-filters>.
- [14] Notification. <https://developer.android.com/reference/android/app/Notification.html>.
- [15] NotificationListenerService. <https://developer.android.com/reference/android/service/notification/NotificationListenerService>.
- [16] PackageInstaller. <https://developer.android.com/reference/android/content/pm/PackageInstaller>.
- [17] Part-of-speech tagging. https://en.wikipedia.org/wiki/Part-of-speech_tagging.
- [18] Runnable. <https://developer.android.com/reference/java/lang/Runnable>.
- [19] Save files on device storage. <https://developer.android.com/training/data-storage/files>.
- [20] Section 508 Law and Related Laws and Policies. <https://section508.gov/content/learn/laws-and-policies>.
- [21] spaCy. <https://spacy.io/>.
- [22] Support Library. <https://developer.android.com/topic/libraries/support-library/>.
- [23] 30 Best Google Play Store alternative as of 2018. <https://www.slant.co/topics/2175/~google-play-store-alternative>, 2018.
- [24] Top 20 Chinese Android App Stores. <https://www.appinchina.co/market/>, January 2019.
- [25] Hadi Abdullah, Washington Garcia, Christian Peeters, Patrick Traynor, Kevin R. B. Butler, and Joseph Wilson. Practical Hidden Voice Attacks against Speech and Speaker Recognition Systems. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 24-27, 2019*, 2018.
- [26] Efthimios Alepis and Constantinos Patsakis. Trapped by the UI: The Android Case. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, 2017.
- [27] Ron Amadeo. Public outcry causes Google to rethink banning powerful “accessibility” apps. <https://arstechnica.com/gadgets/2017/12/google-pauses-android-accessibility-app-crackdown-after-public-outcry/>, December 2017.
- [28] Simone Aonzo, Alessio Merlo, Giulio Tavella, and Yanick Fratantonio. Phishing Attacks on Modern Android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada, October 15-19, 2018*, 2018.
- [29] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David A. Wagner, and Wenchao Zhou. Hidden Voice Commands. In *Proceedings of the 25th USENIX Security Symposium (USENIX-SEC), Austin, TX, USA, August 10-12, 2016*, 2016.
- [30] Si Chen, Kui Ren, Sixu Piao, Cong Wang, Qian Wang, Jian Weng, Lu Su, and Aziz Mohaisen. You Can Hear But You Cannot Steal: Defending Against Voice Impersonation Attacks on Smartphones. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, June 5-8, 2017*, 2017.
- [31] Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. Technical report, 2008.
- [32] Steve Dent. Google cracks down on apps that misuse accessibility features. <https://www.engadget.com/2017/11/13/google-cracks-down-accessibility-features/>, 2017.
- [33] Wenrui Diao, Xiangyu Liu, Zhe Zhou, and Kehuan Zhang. Your Voice Assistant is Mine: How to Abuse Speakers to Steal Information and Control Your Phone. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM), Scottsdale, AZ, USA, November 03 - 07, 2014*, 2014.
- [34] Shaoyong Du, Pengxiong Zhu, Jingyu Hua, Zhiyun Qian, Zhao Zhang, Xiaoyu Chen, and Sheng Zhong. An Empirical Analysis of Hazardous Uses of Android Shared Storage. *IEEE Transactions on Dependable and Secure Computing*, 2018.
- [35] Yanick Fratantonio, Chenxiong Qian, Simon P. Chung, and Wenke Lee. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, USA, May 22-26, 2017*, 2017.
- [36] Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. A11y Attacks: Exploiting Accessibility in Operating Systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS), Scottsdale, AZ, USA, November 3-7, 2014*, 2014.

- [37] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. On Malware Leveraging the Android Accessibility Framework. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services - 10th International Conference, MobiQuitous 2013, Tokyo, Japan, December 2-4, 2013, Revised Selected Papers*, 2013.
- [38] Yeonjoon Lee, Tongxin Li, Nan Zhang, Soteris Demetriou, Mingming Zha, XiaoFeng Wang, Kai Chen, Xiao-yong Zhou, Xinhui Han, and Michael Grace. Ghost Installer in the Shadow: Security Analysis of App Installation on Android. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Denver, CO, USA, June 26-29, 2017*, 2017.
- [39] Tongxin Li, Xiao-yong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS), Scottsdale, AZ, USA, November 3-7, 2014*, 2014.
- [40] Xiangyu Liu, Zhe Zhou, Wenrui Diao, Zhou Li, and Kehuan Zhang. An Empirical Study on Android for Saving Non-shared Data on Public Storage. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, 2015.
- [41] Joe Maring. Accessibility Services: What they are and why Google is cracking down on their misuse. <https://www.androidcentral.com/android-accessibility-services>, 2017.
- [42] Yan Meng, Zichang Wang, Wei Zhang, Peilin Wu, Haojin Zhu, Xiaohui Liang, and Yao Liu. WiVo: Enhancing the Security of Voice Control System via Wireless Signal in IoT Environment. In *Proceedings of the Nineteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc), Los Angeles, CA, USA, June 26-29, 2018*, 2018.
- [43] Mohammad Naseri, Nataniel P. Borges Jr., Andreas Zeller, and Romain Rouvoy. AccessiLeaks: Investigating Privacy Leaks Exposed by the Android Accessibility Service. *Proceedings on Privacy Enhancing Technologies*, 2019(2):291–305, 2019.
- [44] Giuseppe Petracca, Yuqiong Sun, Trent Jaeger, and Ahmad Atamli. AuDroid: Preventing Attacks on Audio Channels in Mobile Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC), Los Angeles, CA, USA, December 7-11, 2015*, 2015.
- [45] Nirupam Roy, Sheng Shen, Haitham Hassanieh, and Romit Roy Choudhury. Inaudible Voice Commands: The Long-Range Attack and Defense. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Renton, WA, USA, April 9-11, 2018*, 2018.
- [46] 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 Workshop on Offensive Technologies (WOOT), Washington, DC, USA, August 10-11, 2015*, 2015.
- [47] Zhi Xu and Sencun Zhu. Abusing Notification Services on Smartphones for Phishing and Spamming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT), Bellevue, WA, USA, August 6-7, 2012*, 2012.
- [48] 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 *Proceedings of the 27th USENIX Security Symposium (USENIX-SEC), Baltimore, MD, USA, August 15-17, 2018*, 2018.
- [49] 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), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.
- [50] 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), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.
- [51] Yury Zhauniarovich and Olga Gadyatskaya. Small Changes, Big Changes: An Updated View on the Android Permission System. In *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, 2016.

Appendix

Here we give ten additional examples of accessibility service descriptions.

1. com.yahora.ioslocker15 ⇒ “Turn it on to receive notifications such as unread messages, unread mail, missed

call and so on, it doesn't collect any data except receive notifications."

2. *arun.com.chromer ⇒ "This service will be used by Chromer to scan text links on your screen and load them in the background. No other information is processed by Chromer. You can still use Chromer in light mode if you choose to not grant this. For more information, launch Chromer app."*
3. *com.parentsware.ourpact.child ⇒ "OurPact Jr. requires accessibility permissions in order to set healthy screen time limits for your children and avoid excessive or compulsive device usage. Please ENABLE accessibility permissions, then press Back twice to return to set up. Note: No personal information is collected through permissions."*
4. *pl.damianpiwowarski.navbarapps ⇒ "Navbar Apps will use this service to detect active running app, which can be used to color navigation bar by active app color. This can be useful for users with disabilities as it makes Navigation Bar more distinguishable and visible."*
5. *com.cootek.smartinputv5 ⇒ "Turning on Accessibility makes sending GIFs easier. Flip the switch to enable GIF Keyboard."*
6. *com.lenovo.anyshare.cloneit ⇒ "Open CLONEit install (Accessibility), help you click on the button when install applications from old phone. Open CLONEit install service in accessibility, confirmation dialog will pop up. Cloneit agreement, this feature is only available*
7. *com.companionlink.clusbsync ⇒ "Enables DejaOffice to respond to various voice commands."*
8. *com.joaomgcd.touchlesschat ⇒ "Touchless Chat allows users with disabilities to interact with several chat apps to automatically send and reply to messages without ever needing to touch the device. This can be of tremendous help for people who have trouble handling their devices with their hands. Permissions: Observe your actions: this permission is requested by default by all accessibility services. Touchless Chat doesn't need it. Retrieve window content: Touchless Chat needs to know what's on the screen so it can find text fields to paste written messages on behalf of the disabled user."*
9. *com.ace.cleaner ⇒ "Turn on the above button to activate Ace Cleaner Power Mode for maximum acceleration! Ace Cleaner use accessibility features to help stopping notusing apps. Please don't worry if you see the privacy risk reminder, that's just a regular informative warning for any accessibility service. We promise NOT to collect ANY information."*
10. *com.callpod.android ⇒ "KeeperFill allows you to securely and quickly fill your login credentials on websites and mobile apps. On the next screen, enable the KeeperFill keyboard."*

as install application. No other permissions, Android 4.1 or later."

Automatic Generation of Non-intrusive Updates for Third-Party Libraries in Android Applications

Yue Duan¹, Lian Gao², Jie Hu², and Heng Yin²

¹Cornell University ²University of California, Riverside
¹yd375@cornell.edu ²{lgao027, jhu066}@ucr.edu ²heng@cs.ucr.edu

Abstract

Third-Party libraries, which are ubiquitous in Android apps, have exposed great security threats to end users as they rarely get timely updates from the app developers, leaving many security vulnerabilities unpatched. This issue is due to the fact that manually updating libraries can be technically non-trivial and time-consuming for app developers. In this paper, we propose a technique that performs automatic generation of non-intrusive updates for third-party libraries in Android apps. Given an Android app with an outdated library and a newer version of the library, we automatically update the old library in a way that is guaranteed to be fully backward compatible and imposes *zero* impact to the library's interactions with other components. To understand the potential impact of code changes, we propose a novel *Value-sensitive Differential Slicing* algorithm that leverages the diffing information between two versions of a library. The new slicing algorithm greatly reduces the over-conservativeness of the traditional slicing while still preserving the soundness with respect to update generation. We have implemented a prototype called LIBBANDAID. We further evaluated its efficacy on 9 popular libraries with 173 security commits across 83 different versions and 100 real-world open-source apps. The experimental results show that LIBBANDAID can achieve a high average successful updating rate of 80.6% for security vulnerabilities and an even higher rate of 94.07% when further combined with potentially patchable vulnerabilities.

1 Introduction

Third-party libraries (TPL) have been used extensively in Android to provide rich complementary functionalities for Android apps and ease the app development. This trend becomes more obvious as Android apps get increasingly complicated. Prior research has shown that every app contains 8.6 distinct TPLs on average [59], and 42.9% of apps even have more code in TPLs than in their real logic [31].

*This work was conducted while Yue Duan was a PhD student at University of California, Riverside, advised by Prof. Heng Yin.

Despite the benefits, TPLs can bring serious security problems for Android app. It has been revealed [15] that 70.40% of Android apps include at least one outdated TPL and 77% of the app developers only update at most a strict subset of their included TPLs, leaving many known and easy-to-exploit security vulnerabilities unpatched. In fact, updating TPLs in Android apps can be so time-consuming and tedious that developers are often forced to leave them outdated. First, updating libraries to the latest version is likely to involve considerable manual efforts to solve backward incompatibility issues [23]. Second, although 97.8% of actively used library versions with a known vulnerability could be fixed via a drop-in replacement with a specific version [23], it is impractical for app developers to manually find suitable versions for every TPL.

Existing Research. Prior efforts have been made to study and mitigate the problems with TPLs in Android apps. A variety of library detection techniques are proposed [15, 21, 23, 31, 32, 38, 50, 59] to detect TPLs in apps and conduct measurement study. Further, techniques are proposed to isolate TPLs from the Android app. TPLs can be transformed into new processes [47, 56], new apps [27, 49], or new services [41]. Other works enforce in-app privilege separations [46, 51] in order to keep the apps' privileges from TPLs. However, these techniques do not fix security issues per se but merely limit the harmfulness of potential problems in TPLs from the apps.

To alleviate the issues, Android patching techniques are proposed to prevent component hijacking attacks [54], detect information leakage [36, 55], fix cryptographic-misuses [37] and detect runtime crashes [14]. Nonetheless, these techniques only aim to fix specific types of security issues and do not deal with the outdatedness problem on TPLs. Hence, no existing patching techniques on Android can keep TPLs updated and fix security issues in a generic fashion.

Our Approach. To solve the problem, we aim to automatically generate updates for TPLs in Android apps such that it does not require any code modification on the app side and more importantly, introduces no impact to the library

interactions with other components locally and remotely as we call it *non-intrusive*. The advantages of *non-intrusiveness* are two-fold: 1). it requires zero change to the code for the given Android app so that the full backward compatibility and maintainability of the apps are ensured; 2). the internal state consistency of the app is secured since the updates guarantee no impact to the program logic of the updated library.

To achieve this goal, we need to understand the impact of the code changes between the outdated libraries and the latest versions. LIBBANDAID utilizes forward program slicing algorithm to perform Impact Analysis [18]. Traditional slicing algorithm [52] is extremely conservative and often generates unwieldy slices [17, 45]. In our case, these slices will very likely to violate the *non-intrusiveness*. Techniques [44, 48, 58] have been proposed to prune the slices. However, they either consider only data-flow [48] or calculate relevance scores [44, 58] and remove the less relevant codes. Obviously, none of them can meet our need of soundness. As a result, we propose a novel slicing algorithm called *Value-sensitive Differential Slicing* that fully leverages the diffing information between the two versions and eliminates the over-conservativeness of the traditional slicing by keeping track of value set changes for all variables. Then, we are able to produce much smaller slices while still preserving the soundness for the purpose of updates generation.

We implement a prototype called LIBBANDAID. Our system first extracts the outdated libraries from a given Android app, compares each outdated library with its latest version counterpart and generates diffing information that precisely characterizes the code changes at code statement level. Then it uses our new slicing algorithm to analyze the impact of each code change and group related changes together to form a set of candidate updates based on control and data dependencies. Finally, our system carries out a selective updating process to apply only the *non-intrusive* updates to the Android app.

We then conduct a comprehensive evaluation on LIBBANDAID by collecting 9 popular TPLs with 173 security related commits across 83 versions and 100 real world apps. The results show that LIBBANDAID can effectively patch the security vulnerabilities with a high success rate.

Contributions. In summary, this paper has made the following contributions:

- We propose an automatic non-intrusive patch generation technique and implement a prototype system called LIBBANDAID, which is the first of its kind to solve the outdatedness problem for TPLs in Android apps.
- A novel slicing algorithm called *Value-sensitive Differential Slicing* is proposed to utilize the diffing information between old and new versions of the code and reduce the over-conservativeness of the traditional forward slicing while still preserving the soundness.
- We evaluate LIBBANDAID with 9 popular TPLs with 173 security related commits across 83 different versions and

100 real world apps. The experimental results show that LIBBANDAID can effectively fix security vulnerabilities with an average success rate of 80.6% and even higher rate of 94.07% when combined with potentially patchable vulnerabilities. We demonstrate the correctness of the updated apps with automatic program testing.

2 Problem Statement

Deployment Model. Our proposed technique is anticipated to be deployed as a service for Android app developers (other than app markets or end users). Developers can feed their app that contains an outdated TPL as well as the latest version of that TPL into LIBBANDAID. It will automatically generate and apply non-intrusive updates to the TPL within the submitted app without any modification to the app's code. Our approach is designed to be conservative to guarantee a maximal updating in a non-intrusive manner. As a result, security related updates as well as other updates (e.g., new features and optimizations) can be applied to the outdated library.

It is noteworthy that the trade-off for *non-intrusiveness* is the completeness. LIBBANDAID avoids applying updates that could change the interactions among the TPL and other components. As a result, our approach makes a reasonable underlying assumption so that LIBBANDAID is designed to cover most of the security related updates.

Assumption. LIBBANDAID updates the outdated TPLs as much as possible with a high coverage for security related updates without violating the *non-intrusiveness*. The underlying assumption is that a security patch (e.g., insert a new condition check) is unlikely to introduce backward incompatibility or change how the TPL interacts with other components locally (e.g., with the app) and remotely (e.g., with TPL server). Hence, most of the security related issues can be fixed by our technique as they are very unlikely to be filtered out by the pre-defined rules that are designed to ensure the *non-intrusiveness*. This assumption is demonstrated by our evaluation with 9 most popular TPLs in Section 7.

Design Goals. LIBBANDAID achieves the following goals:

- **No source code required.** Our technique does not require any source code from Android app or the included TPLs. This is important because TPLs can be closed-source.
- **High coverage for security patches.** LIBBANDAID aims for a high coverage in updating security related issues in outdated TPLs.
- **Non-intrusiveness.** The generated updates do not change how the original app interacts with other components nor do they break the correctness of the app.

3 System Overview

In this section, we present a running example and use it to explain the work-flow of LIBBANDAID. Note that our approach

works at byte-code level, source code is presented here only for ease of understanding.

3.1 Running Example

The example is based on Dropbox library [3], one of the most popular third-party libraries. Assuming that a given Android app is using Dropbox library version 3.0.3 (released in May 2017). There exist 50 commits from version 3.0.3 to the latest version 3.0.6 (released in Jan 2018), including 16 code commits¹. Listing 1 displays two commits. Lines with colors show the code changes: lime indicates code insertions while pink and yellow specify code modifications.

The first commit is a new security feature that adds a field `accountId` in the class `DbxAuthFinish` to identify Dropbox users instead of using `userId` in older versions. The second commit is a vulnerability fix that adds a `body` field and calls `close()` function of the `body` in a callback function `onFailure()`. When Internet access is cut off, the callback function `onFailure()` will be invoked to close `body` so that potential system hang is avoided.

3.2 Overview of LIBBANDAID

Figure 1 delineates the overview of LIBBANDAID. There are four major components in LIBBANDAID: preprocessing, diffing analysis, update generation and selective updating.

Preprocessing. This step is to filter out the unchanged functions and generate function pairs that are modified across the two versions. Preprocessing component takes as inputs an app with outdated library and a latest version of the library, and outputs a set of function pairs. More specifically, it extracts the outdated library within the given app, analyzes all classes in the two versions of the library and performs function level byte-by-byte comparisons.

As shown in Figure 2, LIBBANDAID pulls out all the functions in the class and performs byte-by-byte comparisons for each function in old library with the functions in the new library as long as they share the same function name. Note that we use function name other than function signature to tolerate changes of modifier, parameter or return type. For example, `DbxAuthFinish()` in the old library is compared with `DbxAuthFinish()` and `DbxAuthFinish(String, String, Body)` in the new library. When the byte-by-byte comparison fails (two functions are not identical), we put them in the potential function mapping list and send it to diffing analysis for further analysis. This list signifies the functions in which the code changes between old and new versions reside.

Diffing Analysis. Diffing analysis in LIBBANDAID is to perform function level matching with a granularity of code statement so as to comprehend the exact code changes between old and new versions of a given library. To achieve this goal, we leverage the Tracelet Execution [22] and use 3-tracelet

to perform code matching at code statement level. Given the output of preprocessing, 3-tracelets are generated to capture partial flow information by breaking down the control-flow graphs for each function pair. Then, the distance between tracelets are calculated to match code statements.

Listing 1: Running example

```
1 public class DbxAuthFinish implements Callback {
2     private String userId;
3     + private String accountId;
4     + private PipedRequestBody body;
5
6     - public DbxAuthFinish(String uid) {
7     + public DbxAuthFinish(String uid, String aid, Body body) {
8         this.userId = uid;
9         + this.accountId = aid;
10        + this.body = body;
11    }
12    public DbxAuthFinish() {
13        + this.body = null;
14        + this.accountId = null;
15        this.userId = null;
16    }
17    public void onFailure (IOException ex) {
18        this.error = ex;
19        + if(body) this.body.close();
20        notifyAll();
21    }
22    public DbxAuthFinish read() {
23        + String accountId = null;
24        String userId = null;
25
26        while(getCurrentToken()) {
27            if(n.equals("uid"))
28                userId = readField();
29            + else if(n.equal("accountId"))
30                + accountId = readField();
31
32            + if(accountId == null)
33                + throw JsonReadexception;
34        }
35        - return new DbxAuthFinish(userId);
36        + return new DbxAuthFinish(userId, accountId, body);
37    }
38    + public String getAccountId() {
39        + return accountId;
40    }
```

For LIBBANDAID, we need to further match the functions that have multiple candidates. For example, in Figure 2, `DbxAuthFinish()` in the old library can be matched to either `DbxAuthFinish()` or `DbxAuthFinish(String, String, Body)` in the new library. To understand the real change, LIBBANDAID leverages the distance information to further match the functions. Particularly, we consider it as a linear assignment problem and use Hungarian Algorithm [29] to find the optimal matching. Tracelet technique has demonstrated a 0.99 accuracy in comparing functions in binary code [22]. In our case, byte-code matching is easier than binary code since it is more semantic-rich. Therefore, we observe no false positive during evaluation.

¹Other non-code commits include changes in README, build file, tutorial and tests.

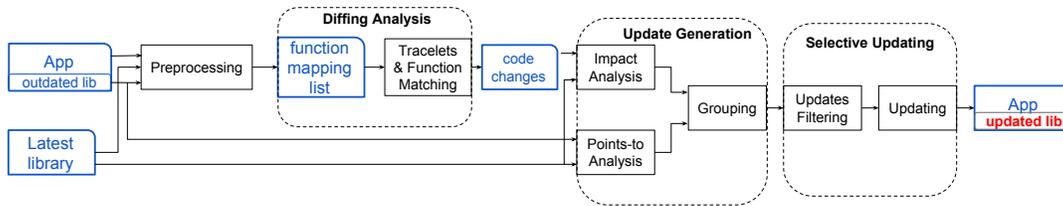


Figure 1: Architecture Overview.

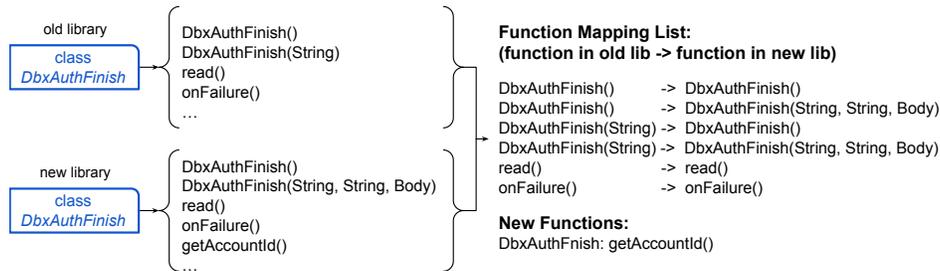


Figure 2: Preprocessing.

`DbxAuthFinish()` and `DbxAuthFinish(String)` in the old library are then matched to `DbxAuthFinish()` and `DbxAuthFinish(String, String, Body)` in the new library respectively. The output of diffing analysis is the real mapping of the functions as well as a set of code changes (pairs of code statements) that precisely characterize the changes between the old and new versions of the third-party library. For our running example, the produced code changes are the same as the colored lines in Listing 1.

Update Generation. Once LIBBANDAID identifies all the code changes between the old and new versions, it starts the update generation process. The whole process takes three inputs: 1). code changes generated by diffing analysis; 2). the old version of the library; and 3). the new version of the library, and generates one output (a set of updates). It first generates system dependence graphs (SDGs) for new and old library, and then generates a slice for each code change by performing impact analysis. Finally, it performs grouping based on the alias information gathered from points-to analysis to produce updates.

The purpose of this indispensable step is two-fold. First, since many code changes have control and data dependencies with each other, LIBBANDAID should always put them together and perform updating collectively. For example, in Listing 1, Ln.10 and 13 assign values to a newly added class field `body` (defined in Ln.4). Ln.19 further calls a member function `close()` of the field. These code changes should be put into one `group` as they are the definition and usages of a same variable `body`. Second, to fulfill the non-intrusiveness design goal as described in Section 2, LIBBANDAID performs impact analysis, combines code changes with all the potentially affected code and further associates the `group` into one update so that our system can apply them as a whole if the

update is indeed *non-intrusive*. As for our running example, after this step, the code changes in Listing 1 will be grouped precisely into two updates, one for each commit. More details are presented in Section 4 and 5.

Selective Updating. The last component of LIBBANDAID is selective updating. It takes the updates generated in the previous step, performs filtering to discard the ones that could potentially break the non-intrusiveness, and eventually updates the old library to generate a new app with an updated library. The core part of this step is to systematically devise a set of pre-defined rules for filtering so that the non-intrusiveness of our generated updates can be preserved. As for the running example, two updates are generated and fed into selective updating. The one related to `accountId` can potentially be filtered out since it will change an interface `DbxAuthFinish(String)` and may cause incompatibility issue. More detailed information is presented in Section 6.

4 Update Generation

In this section, we describe how LIBBANDAID performs update generation by presenting the three major steps: impact analysis, points-to analysis and grouping.

4.1 Impact Analysis

Impact Analysis is to understand the impact (affected codes) of the code changes generated from diffing analysis. Once the impact of the code changes is known, LIBBANDAID groups code changes into updates and performs filtering to remove the ones that violate the *non-intrusiveness*.

Starting from a subset of a program's behavior, program slicing technique reduces the program to a minimal form that still produces that behavior [53]. If we start slicing from a specific code change, it will conservatively includes all the

codes that can potentially be affected by the change. However, traditional slicing is too conservative to be practical and tends to generate gigantic slices. The larger a slice is, the more codes it contains, hence, the bigger chance it will violate the *non-intrusiveness* and get filtered out (more in Section 6). To solve this problem, a new slicing algorithm is desired to perform a sound impact analysis with respect to our definition of *impact* while greatly reducing the over-conservativeness. We discuss the slicing in detail in Section 5.

4.2 Points-to Analysis and Grouping

After the impact analysis, LIBBANDAID performs points-to analysis to extract alias information and further groups code changes into updates. This step is to group slices that are accessing the same global variables or have overlapping code statements. We rely on the existing points-to analysis in Soot [1] to extract alias information.

5 Value-Sensitive Differential Slicing

In this section, we first introduce some important definitions and then describe how our slicing algorithm works in detail.

5.1 Formal Definitions

We formally define the *impact* of a code change and then lay out our definitions on the relationships between program behaviors and variable value sets, upon which the soundness of our slicing algorithm is built.

Definition 1. We denote *impact* of a code change on a code statement as $I(d, c)$, where

- d represents a code change in the new library;
- c represents a code statement that has not changed from the old to the new version of the library;

Therefore, $I(d, c) \neq \emptyset$ means that a code change d has *impact* on code statement c . Intuitively, $I(d, c) = \emptyset$ means that a code change d has no *impact* on c . We then define a code change that has no *impact* on a code statement as:

Definition 2. $I(d, c) = \emptyset \iff B_c^d \subseteq B_c$, where

- B_c^d is a set of behaviors representing all possible program behaviors of c **with** d applied;
- B_c is a set of behaviors representing all possible program behaviors of c **without** applying d ;

Here, the *impact* of a code change to a certain code statement is represented by the change of *program behaviors* for that code statement. If and only if all the possible program behaviors of a code statement c with the code change d applied are still within the original behavior set, we can say d has no impact on c .

We then have following definition on the relationship between value set [16] of all the variables within one code statement and the program behaviors of that code statement:

Definition 3. $VS^d(I, c) \subseteq VS(I, c) \Rightarrow B_c^d \subseteq B_c$, where

- $VS^d(I, c)$ denotes the value set of all the variables I (global and local) and their combinations used in a code statement c **with** d applied;
- $VS(I, c)$ denotes the value set of all the variables I (global and local) and their combinations used in a code statement c **without** applying d ;

Essentially, this definition shows that if the value sets of all variables and their combinations used in a code statement are unchanged or a subset of the original value sets, then the program behaviors of that code statement must stay unchanged or a subset of the original ones. It gives a strong mapping from value sets of all variables in a code statement to the program behaviors of that statement. Together with Definition 2, we can draw a link between value sets of all variables in a code statement and the impact of a code change to that code statement. Specifically, our impact analysis can remove the over-conservativeness by examining the value set changes of all variables in a code statement between old and new versions of the library. If the value sets are unchanged or a subset of the original set for a statement before and after applying a code change, that means the code change has no impact on the statement and our algorithm can safely stop further slicing.

This may seem to be counter-intuitive at first glance. For example, if after applying code change d , statement c has only one behavior in its behavior set while the original behavior set has 100 behaviors, d would still be considered as having **no** impact on c as long as the one behavior is within the original behavior set. In our case, we can safely stop slicing since we know the original code c can correctly handle d and its affected behavior (it is within the original behavior set and introduces no unexpected behavior).

5.2 Basic Scheme

The core idea is to take into account the value changes of all variables between old and new versions of the code and leverage this info to reduce the over-conservativeness of the traditional slicing.

Intuitively, the basic scheme starts from a code change and performs whole library-wise context- and flow-sensitive value-set analysis (VSA) [16] on all variables and their combinations for each code statement that has dependency (control or data) with the code change. Then it compares the value sets for the variables within these code statements between two versions of the library. If there exists no change in the value sets, which means the code change has no impact on the current code statement, then our algorithm does not include that code statement in the slice. Since many values cannot be statically determined, we compute value formulas in a context- and flow-sensitive fashion as the value-set for non-constant variables.

Theoretically, this analysis is sound with respect to the definition of *impact* and could remove the over-conservativeness of traditional slicing. However, it clearly introduces a huge

performance overhead for the whole library-wise context- and flow-sensitive VSA on all variables and their combinations on every control or data dependent code statement for a single code change (there could be thousands of code changes between two versions), rendering the algorithm impractical.

Consequently, we present two optimizations to this basic scheme to improve the runtime performance as well as to further reduce the over-conservativeness. Again, source code is listed just for ease of presentation while LIBBANDAID works on byte-code.

5.3 Slice-wise VSA

To reduce the complexity, we propose an optimization to narrow down the search space to the current slice which begins from the code change.

Listing 2: Slice-wise VSA

```

1 void postSingleEvent (Obj event) {
2     subscriptions = subscriptionsByEventType.
3     get();
4     if (subscriptions != null
5         + && !subscriptions.isEmpty()) {
6         for (Subscription sc : subscriptions)
7             {
8                 postToSubscription(sc, event);
9             }
10        subscriptionFound = true;
11    }
12    ...
13 void postToSubscription (Subscription s, Obj
14    event) {
15    switch (s.threadMode) {
16    case PostThread:
17        invokeSubscriber(s, event);
18    ...

```

Listing 2 shows a real-world security commit from a popular library EventBus [4]. At Ln.4, a condition check `!subscriptions.isEmpty()` is added in the new version. The traditional forward slicing will start from the code change and include every single line from Ln.4 to Ln.23 and even more codes in functions like `invokeSubscriber()` since they all have dependency with the code change. However, by manual investigation, we know the code change does not actually introduce any new behavior to `postToSubscription()`.

For the basic scheme, we compute value sets for all variables and their combinations in every code statement that is data-dependent on the code change. For instance, for code at Ln.6, we calculate value sets for variables `sc` and `event` as well as their combinations (say, `sc = 1` only if `event == 0`). This calculation can only be done in a whole library-wise context-sensitive fashion since the value of `event` is from the caller function `postSingleEvent()`.

To accelerate the process, we can perform VSA only within the slice instead of the whole program. This is because our analysis is to include all code statements that can be affected by the starting of the slice (a code change). That is, as long as the code change (Ln.4) does not affect the value sets of `sc` or `event` or their combinations, we could

stop VSA and keep our slicing from further propagating into `postToSubscription()`. This analysis can be done much faster within the current slice other than the whole library. As a result, a much smaller slice (Ln.4-8) will be produced in a very lightweight fashion.

This optimization is an approximation to the basic scheme algorithm. It sacrifices precision of the whole library-wise VSA but greatly improves the performance. Consequently, it is more conservative than the basic scheme. For example, in a case where an assignment `a = 1` is inserted in a new library, every code that uses the variable `a` will be included under our optimization. However, a library-wise VSA may tell us that `a = 1` is still within the original value-set. Therefore, we do not need to include the code statements that are data-dependent on the newly inserted assignment.

5.4 Intra-procedural VSA

As discussed, the first optimization that searches only within the slice may bring over-conservativeness. As a result, we propose a second optimization to relax the search scope of VSA to the beginning of the function that contains the code change.

Listing 3: Intra-procedural VSA

```

1 void onResume () {
2     if (hasDropboxApp (officialAuthIntent))
3         startActivity (officialAuthIntent);
4     else
5         startWebAuth (state);
6 }
7 boolean hasDropboxApp () {
8     for (Signature sig : packInfo.sigs) {
9         - for (String dbSig : DROPBOX_SIGS)
10            - if (dbSig.equals (signature))
11                - return true;
12    }
13    + if (!DROPBOX_SIGS.contains (sig))
14        + return false;
15 }
16 ...

```

Listing 3 shows another real-world security commit that fixes Android Fake ID vulnerability from Dropbox library. Code statements at Ln.9-11 in the old version are updated to codes at Ln.13-14 in the new version. Statement `return true` (Ln.11) has now become `return false` (Ln.14). Apparently, the value set of variable in the return statement has changed. According to the first optimization, our slicing algorithm will continue flowing into the call site of `hasDropboxApp()` at Ln.2, further propagate to Ln.2-5 and eventually include almost every line of code in the example.

In fact, a closer look will tell us that the code changes within `hasDropboxApp()` does not really expose any *impact* on its caller `onResume()`. Although the return value is modified, both the old and new versions of the function bear the same function-wise return value set: `{true, false}`. In order to capture this information, our algorithm needs to perform intra-procedural VSA beyond the scope of a slice but still within

`hasDropboxApp()`, which is the function that contains the code changes. As a result, our algorithm will stop slicing and generate a much smaller slice.

From the description above, we can see that this optimization sits between the basic scheme (whole library-wise context- and flow-sensitive analysis) and the first optimization (pure slice-wise analysis). Therefore, by applying this optimization to all the variables, our slicing will be more accurate while maintaining the similar performance gain from the first optimization with negligible overhead.

5.5 Value-sensitive Differential Slicing

We now present the details of our slicing algorithm in Algorithm 1, which is a dependence graph based slicing algorithm as [26]. It takes three inputs and generates slice for that code change as output.

Algorithm 1 Value-sensitive Differential Slicing

```

1: input1:  $diff \leftarrow \{stmt_o, stmt_n\}$ 
2: input2:  $SDG_n \leftarrow \{\text{SDG of the new library.}\}$ 
3: input3:  $SDG_o \leftarrow \{\text{SDG of the old library.}\}$ 
4: procedure  $V\_Slicing(diff, SDG_n, SDG_o)$ 
5:    $slice \leftarrow \emptyset$ 
6:    $f_n \leftarrow Locate(stmt_n, SDG_n)$ ;  $f_o \leftarrow Locate(stmt_o, SDG_o)$ 
7:    $workingSet \leftarrow workingSet \cup stmt_n$ 
8:    $slice \leftarrow slice \cup stmt_n$ 
9:   while  $workingSet \neq \emptyset$  do
10:      $stmt \leftarrow workingSet.remove()$ 
11:      $Set_{succs} \leftarrow ImmediateSuccessors(stmt, SDG_n)$ 
12:     for  $succ \in Set_{succs}$  do
13:       if  $succ$  contains new invocation then
14:          $slice \cup \leftarrow Forward\_Slicing(succ, SDG_n)$ 
15:       else if  $succ$  is another  $diff'$  then
16:          $slice \cup \leftarrow V\_Slicing(diff', SDG_n, SDG_o)$ 
17:       else if  $succ$  is control-dependent on  $stmt$  then
18:          $slice \leftarrow slice \cup succ$ 
19:          $workingSet \leftarrow workingSet \cup succ$ 
20:       else if  $succ$  is return statement then
21:         if  $\neg (RetVS(f_o) \subseteq RetVS(f_n))$  then
22:            $slice \leftarrow slice \cup succ$ 
23:            $workingSet \leftarrow workingSet \cup succ$ 
24:         end if
25:       else if  $succ$  is only data-dependent on  $stmt$  then
26:          $vf_n \leftarrow VSA(succ, slice, SDG_n)$ 
27:          $vf_o \leftarrow VSA(succ', slice, SDG_o)$ 
28:         if  $\neg (vf_n \subseteq vf_o)$  then
29:            $slice \leftarrow slice \cup succ$ 
30:            $workingSet \leftarrow workingSet \cup succ$ 
31:         end if
32:       end if
33:     end for
34:   end while
35:   Return  $slice$ 
36: end procedure

```

The algorithm first locates the $diff$ in two $SDGs$ (Ln.6) and adds $stmt_n$ into a $workingSet$ (Ln.7) to start the iterative process. The algorithm will continue running as long as the $workingSet$ is not empty (Ln.9). For every statement in the working set, we extract its immediate successors in SDG (Ln.11). For every immediate successor $succ$, the algorithm

checks if it is another code change. There exist two cases under this scenario. First, if $succ$ is a code change that contains a new function invocation, our algorithm needs to leverage traditional slicing by calling $Forward_Slicing()$ to keep track of the new function call (Ln.13-14) as all its codes are new codes compared to the old version. Second, if $succ$ is a normal code change, we consider it as another input to a recursive function call for $V_Slicing()$ (Ln.15-16).

When $succ$ is not a code change, we add it into the $workingSet$ as well as the $slice$ if it is only control-dependent on $stmt$ (Ln.17-19). When $succ$ is a return statement, we apply the second optimization discussed in Section 5.4 by performing function-wise VSA for all return statements to improve the accuracy (Ln.20-23). When $succ$ is data-dependent on $stmt$, we calculate and compare the value-sets by calling $VSA()$ to extract value formulas at the scope discussed in the second optimization for both old and new versions and only add $succ$ when $stmt$ has impact on it (Ln.25-30). Eventually, it produces a slice by returning $slice$ (Ln.35).

6 Selective Updating

This component takes the generated updates from the previous step, performs filtering and applies the updates to eventually produce an updated TPL, as depicted in Figure 1.

6.1 Filtering

In this step, LIBBANDAID relies on a set of pre-defined rules to filter out the generated updates that may affect the interactions between the library and other components in order to achieve the *non-intrusiveness* goal as explained in Section 2. These rules are defined to be conservative and can guarantee that all satisfying updates will not change how the library interacts with other components. To this end, we investigate into how TPLs work and propose four categories of interactions.

Interaction with the given app. The first category is listed in the first row in Table 1. It defines the rules for interactions with the given app. When TPLs get updated by LIBBANDAID, we guarantee the interactions with the app will not be affected.

Since the interactions are always through library APIs, we need to make sure the used APIs will stay the same in terms of function names, return types, parameters and thrown exceptions. To this end, LIBBANDAID performs static program analysis to collect the library APIs used within the app and filters the updates that could change these APIs. Additionally, LIBBANDAID collects exception information and discards the updates that introduce new exceptions.

It is noteworthy that the interaction with the given app is the only category that relies on program analysis due to two reasons. First, we need to perform program analysis on the two versions of the library to understand which APIs are changed. Second, even if some APIs are indeed changed in the newer version, we may still safely update as long as the Android app does not directly call them.

Table 1: Pre-defined Rules for Filtering

Categories	Representative Behaviors		Rules
Interaction with the given app	API changes	public API signature change (return type, parameter, etc)	depend on analysis
		exception thrown change (new exception type)	depend on analysis
Interaction with server	protocol changes	incoming message change	F
		outgoing message change	F
Interaction with Android system	new Android API usages	no permission change	T
		new permission needed	F
	file manipulation	new file creation	T
		file access that modifies file pointer	F
		new file write	F
		kernel object change	thread/process creation
Interaction with other apps	communication to other components	new intent	F
		intent modification	F
	services	start/bind/unbind services	F

Interaction with server. Another important interaction for a TPL is to communicate with its server. For example, Dropbox library communicates with Dropbox server to access files. Therefore, our system needs to make sure that the protocol between server and client stays the same. To do so, LIBBANDAID scans over each update and checks if it contains code that performs network communication (incoming or outgoing). As long as such code exists, our system will conservatively choose to ignore this update. For example, if one update contains API calls such as `URLConnection.getResponseMessage()`, LIBBANDAID will filter it out.

Interaction with system. We then consider the interactions between a TPL and the underlying Android system.

First, our update may interact with the Android framework by calling a new Android API that was not called in the old version. We rely on PScout [13] to check if the new Android API requires new Android permission. If it does, LIBBANDAID will discard the update. Second, we examine if an update performs any file manipulation in the Android system. Particularly, LIBBANDAID checks if the update affects the current system state, such as creating a new file or writing into a file. The tricky part is the file read. Our system only prevents the library from modifying the file pointer while reading a file (e.g., a call to `RandomAccessFile.seek()`). Third, library may create new kernel objects such as thread and process. LIBBANDAID allows this kind of interactions since they do not affect the execution of Android apps.

Interaction with other apps. The last category of interaction is the interaction with other apps in the Android system. Apps within an Android system could communicate with each other via Binder. LIBBANDAID disallows any update to change the communication either by creating a new intent or by changing any of the existing intent. Also, an update that starts, binds or unbinds services in the system is discarded.

6.2 Updating

After filtering out the unsatisfying updates based on our rules, LIBBANDAID applies the satisfying ones to the outdated library. This step is done at Jimple IR level by using byte-code

rewriting capability in Soot [1]. After the rewriting, we convert the updated Jimple IR into Dalvik byte-code, repackage the DEX file with other resource files and eventually create a new Android app (APK file) with updated library.

7 Evaluation

7.1 Dataset and Configuration

We collect 9 popular Android third-party libraries [15] including Butterknife [2], Dropbox [3], EventBus [4], Glide [6], Gson [7], Leakcanary [8], Okhttp [9], Picasso [10] and Retrofit [11], with a total of 173 security commits over 83 different versions to evaluate our system. Table 2 shows the library names, total number of security commits as well as the associated library versions.

We first collect ground truth based on commit information in Github repositories to gather the vulnerability information for all the 173 security commits. Vulnerability types proposed in prior research [35] to these security related commits are presented in Table 3. As shown, our representative dataset covers a wide range of different types of vulnerabilities.

Then, we compile libraries into a number of testing versions with two requirements: 1). each testing version contains at least one security commit; 2). these testing versions cover all the security commits and version numbers that are listed in Table 2. Finally, we develop Android apps that utilize these testing versions. For each testing version other than the latest one, we feed the Android apps with these versions along with the latest version of each library into LIBBANDAID for evaluation. For instance, Butterknife library has 6 security commits from version 7.0.1 to 8.0.1. We compile 6 testing versions v1 to v6 to guarantee each one will contain at least 1 commit. Then we develop 5 Android apps a1 to a5 that use testing versions v1 to v5 and feed (a1,v6), (a2,v6),..., (a5,v6) into LIBBANDAID for experiments.

Furthermore, we collect 100 real-world Android apps from F-Droid [5] to demonstrate LIBBANDAID in practice. On average, the size of these apps is 4.1MB and they contain 7.1 TPLs per app. We handpick these apps since they all contain

Table 2: Overview of TPLs in Evaluation

Library	# of Security Commits	# of Testing Versions	Versions
Butterknife	6	6	7.0.1 - 8.0.1
Dropbox	11	10	3.0.0 - 3.0.6
EventBus	15	10	2.1.0 - 3.1.0
Glide	22	10	4.4.0 - 4.6.1
Gson	13	10	2.2.4 - 2.8.2
Leakcanary	42	7	1.3.1 - 1.5.4
Okhttp	26	10	3.7.0 - 3.10.0
Picasso	19	10	1.5.3 - 3.0.0
Retrofit	19	10	2.0.0 - 2.4.0

at least one of the 9 libraries described above. Therefore, we can use the latest versions of these TPLs to update the apps.

7.2 Effectiveness of LIBBANDAID

As discussed, we feed each Android app that contains an older version library along with the latest version into LIBBANDAID, and then manually investigate the updated libraries to see if the commits have been updated.

Security commits can be divided into three categories: 1). 'patched'; 2). 'fail to patch'; and 3). 'potentially patchable'. 'patched' means our system can successfully update the library with the commit. 'fail to patch' shows the commits that are filtered out because of to the violation of our pre-defined rules. 'potentially patchable' indicates the commits that change the APIs of the library. LIBBANDAID may still update the 'potentially patchable' ones as long as the analyzed Android apps do not directly invoke the changed APIs.

By Absolute Numbers. Figure 4 gives the results in absolute numbers for the 9 libraries. The x-axis shows each execution of LIBBANDAID while y-axis is the absolute number of vulnerabilities. For example, the x-axis in Figure 4b gives the 9 executions from (a1,v10) to (a9,v10) for Dropbox library and the y-axis shows the number of security commits updated for each run. By looking at the first bar in the figure, we can see that there are total of 11 vulnerabilities between the old and new versions of the library. LIBBANDAID is able to fix 7 of them but fails in 2. Moreover, there are 2 security commits that change the APIs, so we mark them as 'potentially patchable'.

From the 9 figures, 2 libraries (Butterknife and Picasso) are shown to have no 'fail to patch' commit (no yellow bar) for all the versions. And for the rest 7 libraries, 'fail to patch' commits only take up a small average portion of total numbers across all executions. (a9,v10) execution in Okhttp (Figure 4g) is the worst case in our evaluation in which it has 1 'fail to patch' commit out of 3. Further investigation shows these commits could potentially lead to protocol changes due to the fact that Okhttp is an HTTP client and performs considerable amount of network communications. A more interesting observation is that the 'fail to patch' commits will disappear in many libraries when the outdated library becomes more recent and closer to the latest version. For Gson library in

Figure 4e, starting from (a5,v10), the 'fail to patch' commit is gone.

From the experiments, LIBBANDAID could achieve an average success rate of 80.6% for updating security commits and even a higher rate of 94.07% when combining with the 'potentially patchable'.

By Vulnerability Categories. We then examine the categories of vulnerabilities that LIBBANDAID fails to update. The results are exhibited in Table 4. It shows the breakdown of vulnerabilities and the number of failures for that security commit if LIBBANDAID fails to update in all executions.

We find that among all kinds of security vulnerabilities, Info Leak is most likely to fail (1 failed in 3 total commits). In general, vulnerabilities that are related to IO exceptions and information processing (e.g., input validation, data handling) also bear relatively high failure rates. This result is expected since the updates to these vulnerabilities are most likely to affect the interactions between the library and the system or the server. Therefore, the filtering process in LIBBANDAID is triggered.

Observations. Two observations can be made from the above experimental results. First, our assumption made in Section 2 (security patches usually do not introduce backward incompatibility or change how the TPL interacts with other components) holds in practice. Second, LIBBANDAID performs better in updating relatively newer version of the library. This is because the newer the library is, the less code changes it has compared to the latest version. As a result, fewer and smaller slices will be generated and they are less likely to be filtered out by LIBBANDAID.

7.3 Correctness of LIBBANDAID

The correctness of LIBBANDAID is demonstrated by performing random testing as well as manual investigation for the updated apps. To this end, we first use LIBBANDAID to update TPLs within the 100 real-world apps from F-Droid [5]. Then, we collect apps with updated TPLs for testing.

For random testing, we run Monkey, which is a popular UI/Application testing tool developed by Google, on every app with an updated library for 2 hours. Although we did observe some crashes, we have confirmed that they are bugs in the original apps. No new crash is introduced by LIBBANDAID. The results demonstrate that the updated library can function normally and pass the random testing successfully. Due to the code coverage issue for random testing, we augment it with manual investigation to try out all the combinations of UI components. Combined with Monkey, our testing achieves an average code coverage of 25.7% for all the updated libraries. A closer look shows that our testing covers 30.1% of the functions that are actually updated. Admittedly, the code coverage is still far from complete, however, the correctness of LIBBANDAID can still be demonstrated to

Table 3: Security Fixes Distribution

Vulnerability \ Library	Butterknife	Dropbox	EventBus	Glide	Gson	Leakcanary	Okhttp	Picasso	Retrofit
Improper Input Validation	1	3	3	6	5	2	7	6	1
Data Handling Error	4	4	5	3	3	3	7	1	6
Uncaught Exception	1	1	3	4	1	2	7	2	7
Memory Leak			1		1	32	1	3	
Info Leak						2			1
Race Condition			3						
Improper Access Control				2					
Uncontrolled Resource Consumption				1					
System Hang		1		1		1	2		
Uncheck Return Value				5					2
Illegal Reflective Access					1				
Stack Overflow					2			5	
Heap Access Error							1	1	1
Missing Initialization							1		1
Integer Overflow								1	
Fake ID		1							
New Security Feature		1							
Total	6	11	15	22	13	42	26	19	19

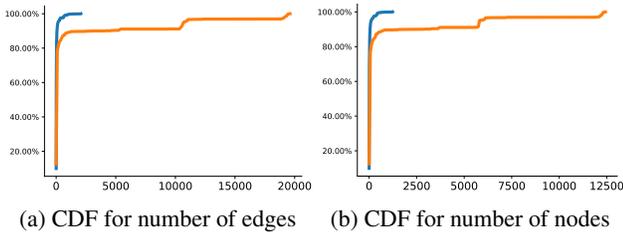


Figure 3: Effectiveness of New Slicing Algorithm

gether with our manual investigation showed in the previous Section 7.2.

7.4 Effectiveness of new slicing

Finally, we evaluate the effectiveness of *Value-sensitive Differential Slicing* by comparing it with the traditional slicing algorithm. We seek to evaluate the algorithm by answering the two following questions:

- 1). How well it performs in terms of over-conservativeness reduction?
- 2). Can it help LIBBANDAID achieve better results?

Over-conservativeness Reduction. We evaluate the effectiveness of *Value-sensitive Differential Slicing* by examining how much it could reduce the over-conservativeness across the 9 testing libraries. Figure 3 displays the cumulative distributions of the sizes of generated slices for traditional slicing as well as the new slicing with respect to the numbers of edges and nodes. The blue line indicates the new slicing algorithm while the yellow line represents the traditional slicing.

From the figures, we can see that *Value-sensitive Differential Slicing* could effectively reduce the number of edges as well as nodes by at least one order of magnitude. For example, 100% of the generated slices by *Value-sensitive Differential Slicing* have less than 2,500 edges and 2,000 nodes. On the contrary, traditional slicing generates way larger slices up to 20,000 edges and 12,500 nodes. This information gives us a

clear view for the advantage of our slicing over the traditional slicing in terms of over-conservativeness reduction.

Updating Improvements. We further evaluate our slicing by examining the updating results improvements. The results in Section 7.2 shows LIBBANDAID could achieve a high successful updating rate for security commits when leveraging our new slicing. To evaluate, we run the experiments again with traditional slicing and compare the differences. The results show that LIBBANDAID could only achieve an updating rate of 61.84% with a rate of 74.95% when combined with the potentially patchable commits. In contrast, with the help of new slicing, our system could perform much better at rates of 80.6% and 94.07% when combined with potentially patchables as reported in Section 7.2. Detailed information is presented in Figure 6.

8 Discussion

Soundness. The soundness of our approach results from that of diffing analysis, update generation and patching respectively.

For diffing analysis, we leverage Tracelet Execution [22] technique, which has demonstrated a 0.99 accuracy in its evaluation, to compare TPLs at statement level. In our case, false positive (statements that are not code changes to be considered as changes) is impossible since we match the exact strings to confirm. Theoretically, false negatives are possible. However, we argue that false negative can only lower the successful patching rate but not bring any correctness or compatibility issue.

For update generation, the soundness of our impact analysis inherits from the soundness of traditional slicing. The basic scheme strictly follows the definition of *impact* in Section 2. However, due to the two optimizations, our slicing is still sound with respect to the definition of *impact* but may contain over-conservativeness for performance gain.

Based on the soundness analysis of our slicing, the correctness of updating is ensured by virtue of two reasons. First, LIBBANDAID introduces absolutely no code changes other than the ones from the new library itself. We assume the library developers have already tested their code before committing. Second, the completeness of each generated update is guaranteed by our slicing algorithm.

Correctness. In certain extreme cases, LIBBANDAID may affect the correctness of the updated apps in practice. For instance, a TPL function originally returns 0 only on a very rare failure, but now returns 0 for all kinds of failure after a patch. The Android app that uses the old version may simply ignore the case of returning 0 since it is so rare that developers could never make it happen during testing. However, the app may break after using LIBBANDAID.

We argue that it is the app developers' responsibility to fully test their apps in a complete fashion. But in practice, LIBBANDAID could use some lightweight sampling process such as fuzzing [60] to estimate the satisfying space for return values of a function and choose whether to perform the update. We leave this as a future work.

Limitations. To begin with, LIBBANDAID can only handle Java libraries and Java codes, and cannot update native libraries in Android apps. Moreover, non-code changes could also bring issues. For example, a version number may be stored in a file and used to communicate with server as part of the protocol. In this case, LIBBANDAID may change the protocol and introduce incompatibility. To solve the problem, we need to consider the accesses to the same file as data dependency. We also leave this as a future work.

Second, our analysis technique cannot handle obfuscated code. Recently, there is a growing trend for Android apps to use different obfuscation and packing techniques [24] to hide real logic. We argue that this is not a big problem for LIBBANDAID as it is designed for App developers who should possess unobfuscated code. Also, most of the popular TPLs [15] in Android apps are not obfuscated.

Third, our slicing relies on an accurate data dependency analysis that in turn depends on a complete modeling of Java and Android APIs. We manually write models for more than 500 most popular APIs but they still can be incomplete. This incompleteness may thwart the soundness of our analysis.

Fourth, we handle the diffing analysis as a code matching problem and leverage existing research [22] to perform analysis. We argue that this problem is orthogonal to our major focus of updating the TPLs in Android apps. We can definitely make use of the advance in code matching techniques to improve the performance of LIBBANDAID.

Finally, although LIBBANDAID analyzes the library API to collect new exception information, the analysis results in theory can be incomplete. For example, a code change in a TPL's API can call other function outside the library that

eventually rises an exception. In this case, we may miss it, jeopardizing the *non-intrusiveness*.

9 Related Work

Change Impact Analysis. Change Impact Analysis [18] studies how code changes in one place could affect codes in other places of the program. Techniques have been proposed [12, 28, 30, 33, 40, 42–44, 48, 58] to improve the change impact analysis. Some of them utilize call graph analysis to study the impact of code change [12, 42, 43]. The limitation is that call graphs by nature can only provide a coarse-grained information usually at method level. Another set of research [30, 40] utilizes dynamic analysis to understand the impact of code changes. However, dynamic analysis often falls short of code coverage.

Program slicing [52] becomes a promising technique to grasp a comprehensive understanding of the impact for code changes. A series of research [28, 33, 34, 44, 48, 58] has been done towards this direction. TAILER [34] computes a tailored program that comprises the statements in all possible execution paths passing through a given statement sequence. GRACE [28] performs forward slicing to capture all potentially affected codes. To deal with the conservativeness, Sridharan et.al. [48] propose a new slicing algorithm called thin slicing that only considers value-flow. P-slicing [44] and PRIOSLICE [58] augment the forward slicing with relevance scores that indicate how likely a code statement can be affected by the change.

Android Program Patching. Automatic Program Patching in the context of Android falls into two categories: Android system patching and Android app patching. Many works have been done [19, 20, 39, 57] to perform patching on Android system and kernel. PatchDroid [39] uses in-memory patching techniques to address vulnerabilities. KARMA [20] is proposed as an adaptive live patching system for Android kernels by featuring a multi-level adaptive patching model. Embroidery [57] only targets the binary code in Android kernels by using binary rewriting techniques. It transplants official patches of known vulnerabilities to different devices by adopting heuristic matching strategies. InstaGuard [19] adopts hot-patching to patch the system programs in Android by enforcing updatable rules that contain no code to block exploits of unpatched vulnerabilities.

Android application patching techniques, on the other hand, are also proposed to mitigate security problems in Android apps. AppSealer [54], which is the most similar work with ours, performs automatic patching for preventing component hijacking attacks in Android apps. Duan et.al. [25] uses Android rewriting technique to perform privacy-preserving offloading of Android apps to the public cloud. Capper [55] and Liu et.al. [36] rewrite the Android apps to keep track of private information flow and detect privacy leakage at runtime. CDRep [37] fixes cryptographic-misuses in Android with

similar byte-code rewriting technique. Azim et.al. [14] detect crashes dynamically and use byte-code rewriting technique to avoid such crashes in the future.

10 Conclusion

In this paper, we developed a novel technique named LIBBANDAID to solve the outdatedness problem for TPLs in Android apps by automatically generating non-intrusive updates. Our system extracts the outdated library within apps, compares it to the latest version of the library and generates diffing information that precisely characterizes the code changes at code statement level. Then, it analyzes the impact of each code change and generates updates. To do so, we propose a novel slicing algorithm named *Value-sensitive Differential Slicing* to reduce the over-conservativeness of the traditional slicing algorithm while still preserving the soundness. LIBBANDAID further performs selective updating by filtering out the updates that can potentially change the interactions between the library and other components. Our evaluation on 9 real-world popular third-party libraries and 100 real-world Android apps demonstrates that LIBBANDAID could effectively patch the security vulnerabilities within libraries with an average of 80.6% success rate and an even higher 94.07% when combined with potentially patchable vulnerabilities.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by DARPA under grant FA8750-16-C-0044.

References

- [1] Soot: a Java Optimization Framework.
- [2] Butterknife: Bind Android views and callbacks to fields and methods. <https://github.com/JakeWharton/butterknife>, 2018.
- [3] Dropbox: A Java library for the Dropbox Core API. <https://github.com/dropbox/dropbox-sdk-java/>, 2018.
- [4] EventBus. <https://github.com/greenrobot/EventBus>, 2018.
- [5] F-Droid - Free and Open Source Android App Repository. <https://f-droid.org/en/>, 2018.
- [6] Glide: An image loading and caching library. <https://github.com/bumptech/glide>, 2018.
- [7] Gson: Java serialization library. <https://github.com/google/gson>, 2018.
- [8] Leakcanary: A memory leak detection library for Android and Java. <https://github.com/square/leakcanary>, 2018.
- [9] Okhttp: An HTTP+HTTP/2 client for Android and Java applications. <https://github.com/square/okhttp>, 2018.
- [10] Picasso: A powerful image downloading and caching library for Android. <https://github.com/square/picasso>, 2018.
- [11] Retrofit: Type-safe HTTP client. <https://github.com/square/retrofit>, 2018.
- [12] Taweessup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th international conference on Software engineering*, pages 432–441. ACM, 2005.
- [13] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.
- [14] Md Tanzirul Azim, Iulian Neamtii, and Lisa M Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 623–628. ACM, 2014.
- [15] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.
- [16] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *International Conference on Compiler Construction*, pages 250–254. Springer, 2005.
- [17] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):8, 2007.
- [18] Shawn Anthony Bohner. A graph traceability approach for software change impact analysis. 1996.
- [19] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.
- [20] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, 2017.
- [21] Mike Chi. *LibDetector: Version Identification of Libraries in Android Applications*. Rochester Institute of Technology, 2016.
- [22] Yaniv David and Eran Yahav. Tracelet-based code search in executables. *ACM SIGPLAN Notices*, 49(6):349–360, 2014.
- [23] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and

- Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200. ACM, 2017.
- [24] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. Things you may not know about android (un) packers: A systematic study based on whole-system emulation. In *NDSS*, 2018.
- [25] Yue Duan, Mu Zhang, Heng Yin, and Yuzhe Tang. Privacy-preserving offloading of mobile app to the public cloud. In *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.
- [26] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.
- [27] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1037–1049. ACM, 2017.
- [28] Jaakko Korpi and Jussi Koskinen. Supporting impact analysis by program dependence graph based forward slicing. In *Advances and innovations in systems, computing sciences and software engineering*, pages 197–202. Springer, 2007.
- [29] Harold W Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 2(1-2):83–97, 1955.
- [30] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society, 2003.
- [31] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 403–414. IEEE, 2016.
- [32] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in android markets. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 335–346. IEEE, 2017.
- [33] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. *IEEE Transactions on Software Engineering*, 44(2):182–201, 2018.
- [34] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- [35] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 2–13. IEEE, 2017.
- [36] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. Fixing resource leaks in android apps with light-weight static analysis and low-overhead instrumentation. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 342–352. IEEE, 2016.
- [37] Siqi Ma, David Lo, Teng Li, and Robert H Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 711–722. ACM, 2016.
- [38] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 653–656. ACM, 2016.
- [39] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 259–268. ACM, 2013.
- [40] Alessandro Orso, Taweewat Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 128–137. ACM, 2003.
- [41] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. ACM, 2012.
- [42] Xiaoxia Ren, Barbara G Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings of the 27th international conference on Software engineering*, pages 664–665. ACM, 2005.
- [43] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.
- [44] Raul Santelices and Mary Jean Harrold. Probabilistic slicing for predictive impact analysis. Technical report,

Georgia Institute of Technology, 2010.

- [45] Raul Santelices, Mary Jean Harrold, and Alessandro Orso. Precisely detecting runtime change interactions for evolving software. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 429–438. IEEE, 2010.
- [46] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.
- [47] Shashi Shekhar, Michael Dietz, and Dan S Wallach. Ad-split: Separating smartphone advertising from applications. In *USENIX Security Symposium*, volume 2012, 2012.
- [48] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. In *ACM SIGPLAN Notices*, volume 42, pages 112–122. ACM, 2007.
- [49] Mengtao Sun and Gang Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 165–176. ACM, 2014.
- [50] Haoyu Wang and Yao Guo. Understanding third-party libraries in mobile app analysis. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 515–516. IEEE, 2017.
- [51] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 25–36. ACM, 2014.
- [52] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.
- [53] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
- [54] Mu Zhang and Heng Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, February 2014.
- [55] Mu Zhang and Heng Yin. Efficient, Context-aware Privacy Leakage Confinement for Android Applications Without Firmware Modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS'14)*, 2014.
- [56] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 9–18. ACM, 2013.
- [57] Xuewen Zhang, Yuanyuan Zhang, Juanru Li, Yikun Hu, Huayi Li, and Dawu Gu. Embroidery: Patching vulnerable binary code of fragmented android devices. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 47–57. IEEE, 2017.
- [58] Yiji Zhang and Raul Santelices. Prioritized static slicing for effective fault localization in the absence of runtime information. Technical report, Technical Report TR 2013-06, CSE, U. of Notre Dame, 2013.
- [59] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152. IEEE, 2018.
- [60] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

Table 4: Effectiveness Results By Vulnerability Category

Vulnerabilities	Total	Failures	Failure Rate
Race Condition	3	0	0%
Improper Access Control	2	0	0%
Uncontrolled Resource Consumption	1	0	0%
System Hang	5	0	0%
Illegal Reflective Access	1	0	0%
Stack Overflow	7	0	0%
Heap Access Error	3	0	0%
Missing Initialization	2	0	0%
Integer Overflow	1	0	0%
Fake ID	1	0	0%
New Security Feature	1	0	0%
Memory Leak	38	1	2.63%
Uncaught Exception	28	2	7.14%
Data Handling Error	36	3	8.33%
Uncheck Return Value	7	1	14.28%
Improper Input Validation	34	5	14.7%
Info Leak	3	1	33.33%

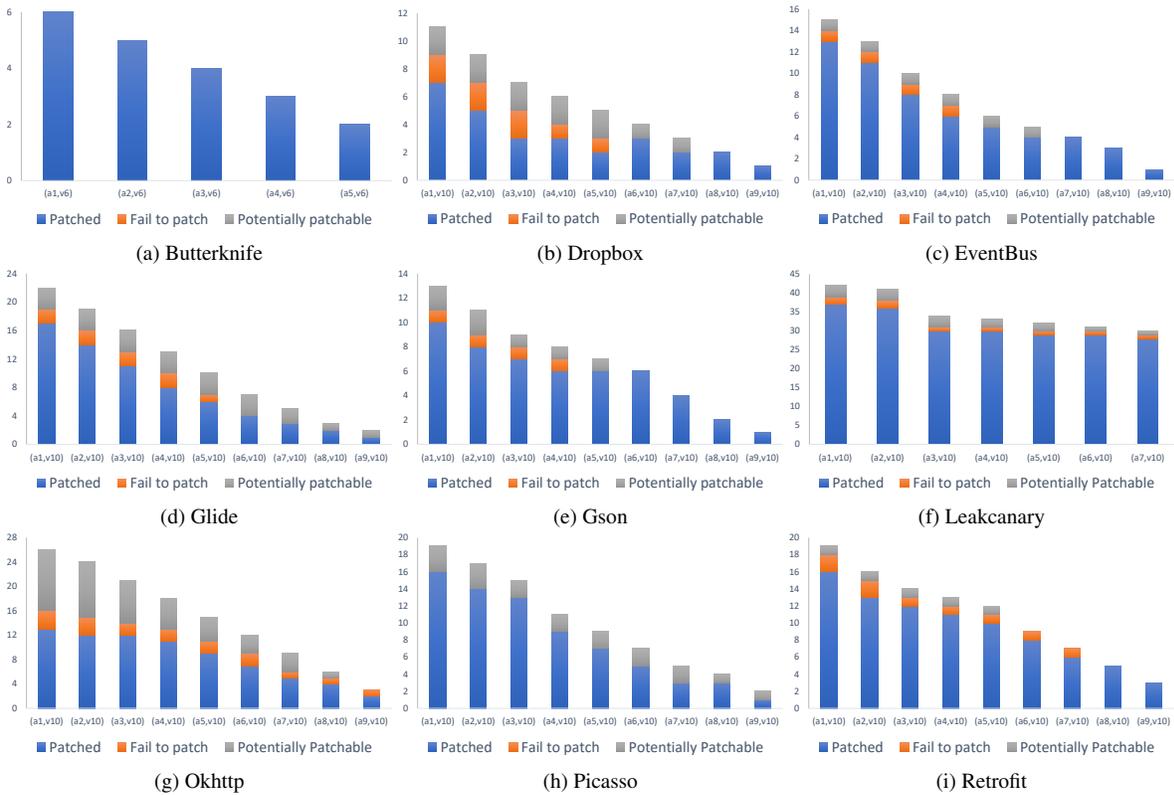


Figure 4: Effectiveness Results By Numbers

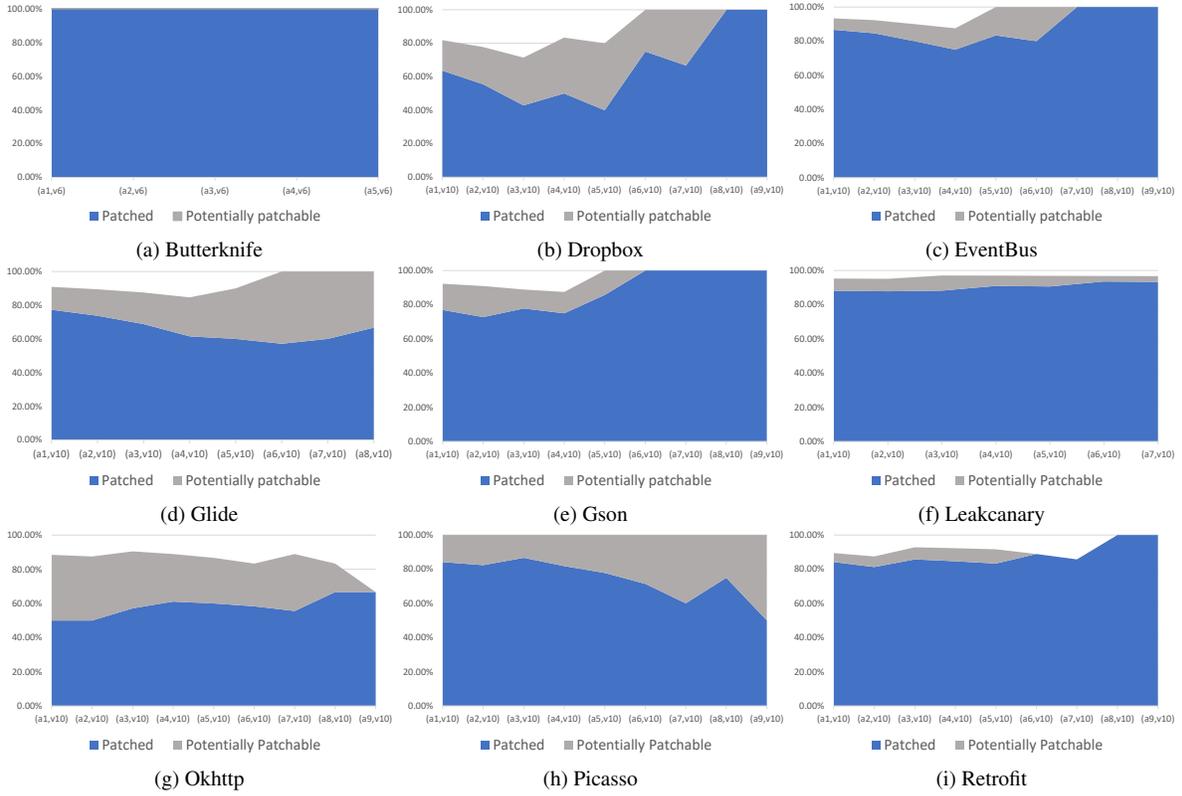


Figure 5: Effectiveness Results by Percentage

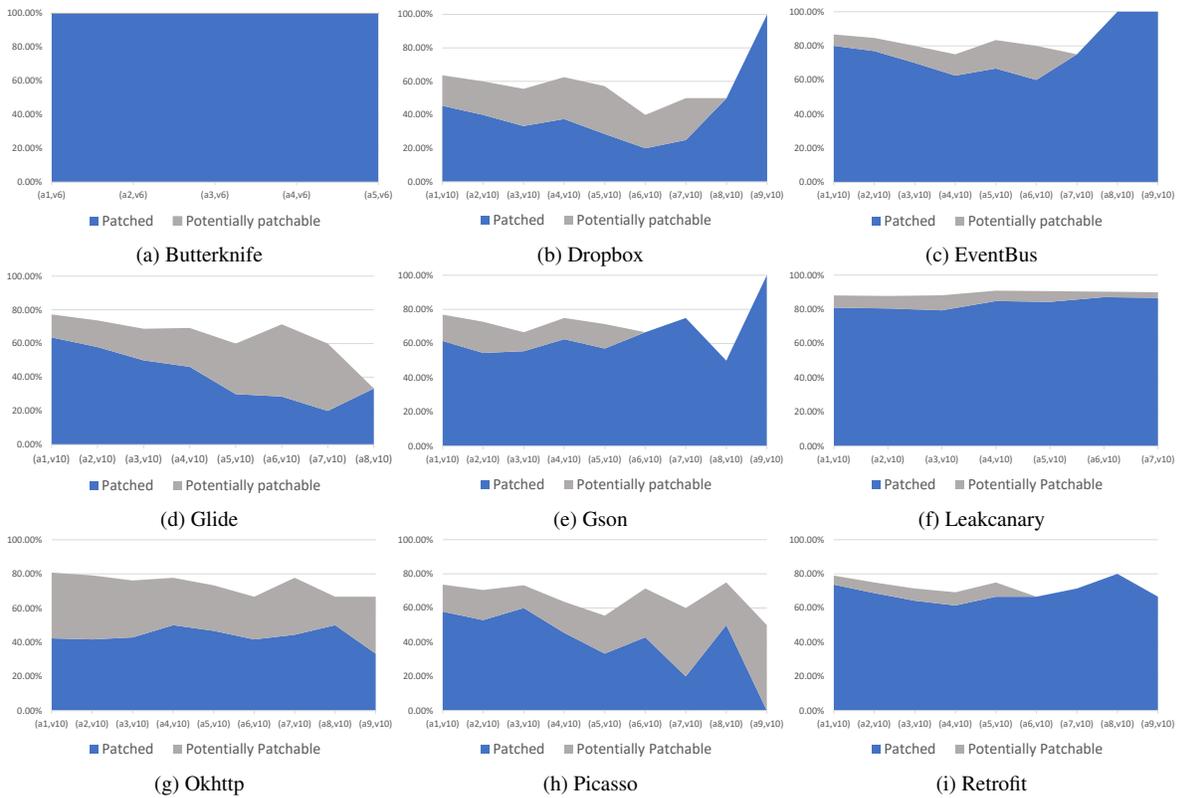


Figure 6: Effectiveness Results with Traditional Slicing

Exploiting the Inherent Limitation of L_0 Adversarial Examples

Fei Zuo
University of South Carolina
fzuo@email.sc.edu

Bokai Yang
University of South Carolina
bokai@email.sc.edu

Xiaopeng Li
University of South Carolina
xl4@email.sc.edu

Lannan Luo
University of South Carolina
lluo@cse.sc.edu

Qiang Zeng
University of South Carolina
zeng1@cse.sc.edu

Abstract

Despite the great achievements made by neural networks on tasks such as image classification, they are brittle and vulnerable to adversarial example (AE) attacks, which are crafted by adding human-imperceptible perturbations to inputs in order that a neural-network-based classifier incorrectly labels them. In particular, L_0 AEs are a category of widely discussed threats where adversaries are restricted in the number of pixels that they can corrupt. However, our observation is that, while L_0 attacks modify as few pixels as possible, they tend to cause large-amplitude perturbations to the modified pixels. We consider this as an inherent limitation of L_0 AEs, and thwart such attacks by both detecting and rectifying them. The main novelty of the proposed detector is that we convert the *AE detection problem* into a *comparison problem* by exploiting the inherent limitation of L_0 attacks. More concretely, given an image I , it is pre-processed to obtain another image I' . A Siamese network, which is known to be effective in comparison, takes I and I' as the input pair to determine whether I is an AE. A trained Siamese network automatically and precisely captures the discrepancies between I and I' to detect L_0 perturbations. In addition, we show that the pre-processing technique, *inpainting*, used for detection can also work as an effective defense, which has a high probability of removing the adversarial influence of L_0 perturbations. Thus, our system, called AEPecker, demonstrates not only high AE detection accuracies, but also a notable capability to correct the classification results.

1 Introduction

Recent years have witnessed tremendous success of neural networks in a variety of fields, such as object detection [39], motion tracking [44], face recognition [36,45], and code analysis [26,38,49]. Despite these great achievements, they are vulnerable to adversarial examples (AEs). Szegedy et al. [41] analyze the robustness of neural networks when facing adversarial attacks, and show that deep learning systems are

sensitive to small adversarial perturbations. A neural-network-based classifier thus can be misled by AEs and generate incorrect classification results. Many image AE generation methods have been proposed and multiple off-the-shelf tools are available [8,13,22,34].

The adversarial perturbations in an image AE are usually subtle in order to be human-imperceptible. To quantitatively describe such perturbations, L_p norms are usually used to measure the discrepancy between an original benign image I_0 and its corresponding AE I_a . According to the value of p , the mainstream AE generation algorithms can be categorized into three families, i.e., L_0 , L_2 and L_∞ attacks. Informally, L_0 measures the number of modified pixels, L_2 the Euclidean distance between the two images, and L_∞ the largest modification among the pixels. Note that our work focuses on L_0 AEs, a category of attacks widely considered by previous works [8,27,34,47].

To defeat attacks based on AEs, both detection and defensive techniques attract the research community's attention. Given an input image, the *detection* system outputs whether it is an AE, so that the target neural network can reject those adversarial inputs. A *defense* technique, given an AE, helps the target neural network make correct prediction by either rectifying the AE or fortifying the classifier itself.

Many AE detection methods [3,24,32] and defense techniques [11,25,46] have been proposed. However, prior methods either are not very effective in handling L_0 AEs or omit discussing them. For example, feature squeezing [47] is capable of detecting L_0 AEs. However, He et al. [17] have shown that feature squeezers, either single or joint, are not resilient to adaptive attacks. Previous work even argues that it is challenging to recover the correct classification of L_0 AEs by input transformation, as “it is very difficult to properly reduce the effect of the heavy perturbation” [24].

We identify two characteristics of L_0 AEs. By exploiting the two characteristics, we build a detector based on a very simple architecture that achieves a high detection accuracy. Moreover, a pre-processor based on these observations can effectively rectify L_0 AEs to recover the correct classifications.

The first characteristic is that it limits the number of modified pixels, but not the amplitude of pixels. Thus, L_0 attacks tend to introduce large-amplitude perturbations, especially for targeted attacks that aim to achieve an attacker-desired output from a neural network. Second, as L_0 attacks try to modify as few pixels as possible, the optimization-based AE generation process tends to result in altered pixels that scatter in the image. In other words, those corrupted parts are mostly small and isolated regions. Both characteristics are verified by our experiments.

We accordingly propose a novel AE detection method. The main novelty is that we convert the *AE detection problem* into a *comparison problem*. Specifically, the architecture of the detector uses a Siamese network [6], which is known to be powerful in comparison. Given an image I , it is processed by a pre-processor to obtain another image I' . The Siamese network takes I and I' as the inputs and outputs whether I is an AE. The advantage of the design is that the Siamese network is able to automatically and precisely capture the discrepancies between the two inputs for AE detection.

Another advantage is that the pre-processor used for AE detection can also work as an effective defense by removing the influence of the adversarial perturbations. Specifically, we propose an *inpainting*-based algorithm to process images, where *inpainting* refers to the process of *reconstructing* the lost or corrupted parts of an image. The inpainting techniques are a fruitful sub-field in the area of digital image processings [29, 40, 42], which have been widely used in practice. As we will show in Section 4.3, inpainting is more effective at eliminating the heavy perturbations created by L_0 attacks than previous defenses.

We implement a system AEPECKER to demonstrate the advantages aforementioned and the weakness of L_0 attacks. The system architecture is shown in Figure 1. After inputting an image I to a pre-processor \mathcal{P} , we obtain another image I' . Then, the Siamese network predicts whether I is adversarial by taking $\langle I, I' \rangle$ as the input pair. If I is detected as an L_0 AE, then we regard I' as a rectified image and use it to replace I in subsequent image classification for the defense purpose.

We have evaluated our system in terms of its detection and defense capabilities using the popular image datasets CIFAR-10 and MNIST. Two leading L_0 AE generation methods, JSMA [34] and CW- L_0 [8], are both considered in the evaluation. In the case of CIFAR-10 (we have similar results for MNIST), the evaluation results show that (1) the detection rate on the CW- L_0 and JSMA attack is 97.1% and 99.7% respectively, both with a low false positive rate; (2) the proposed system has outstanding *transferability*, as a detector trained only with JSMA AEs can detect CW- L_0 AEs with a high detection rate (99.4%), and vice versa; (3) the detection is also *attack-target-model agnostic* (model agnostic, for short), since in the aforementioned experiments CW- L_0 AEs and JSMA AEs actually target different image classification models; and (4) our defense method recovers the classifica-

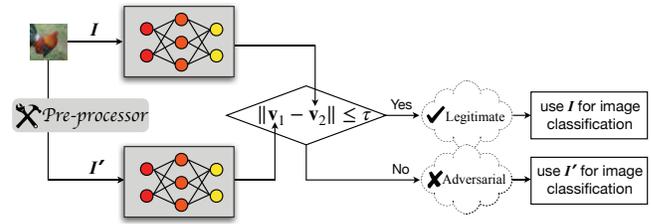


Figure 1: An architecture of AEPECKER. If I is detected as an L_0 AE, I' is used for image classification as a defense.

tion accuracy from 0% (when classifying those successful AEs) to 87.3% for CW- L_0 , and from 0% to 96.1% for JSMA, and meanwhile, has a very small impact on benign images.

Moreover, in order to illustrate the effectiveness of the Siamese network in detecting AEs, we experiment to use a preprocessing technique, *bit depth reduction*, that is known to be weak. Feature squeezing [47] used it as one of the pre-processors and obtained an AE detection rate 4.1%. In contrast, the Siamese network plus the weak preprocessing technique achieves 99.6%, which demonstrates the unique advantage of the Siamese architecture in detecting AEs.

The key contributions of our work include:

- We point out the inherent characteristics of L_0 AEs, which typically contain high-amplitude perturbations to very few and isolated pixels, and propose to exploit them to develop detection and defense techniques.
- We convert the L_0 AE detection problem into an image comparison problem, and propose to use a Siamese network to automatically extract the subtle discrepancies of the input pair as features for the AE detection. The detector demonstrates multiple prominent strengths, such as transferability across attacks and being attack-target-model agnostic (so the detector keeps effective across attack methods and target classifiers).
- We propose an effective *inpainting*-based defense against L_0 perturbations, which can recover the correct classification at a high probability. To the best of our knowledge, this defense method achieves the highest accuracy when dealing with L_0 AEs.
- Adaptive attacks that try to bypass our detection are considered and evaluated. The evaluation results show that our system is resilient to them.

The rest of the paper is organized as follows. First, we briefly introduce some background about L_0 AE generation methods in Section 2. Section 3 describes our system architecture and design. Then, we present the evaluation design and results in Section 4. We also empirically evaluate the resilience of the proposed technique under adaptive attacks in Section 5. The related work is then reviewed in Section 6.

We finally discuss the limitations of our work and draw conclusions in Section 7 and 8, respectively.

2 Adversarial Example Generation

Adversarial examples are carefully crafted inputs intended to fool artificial intelligence systems to output incorrect labels. The term *adversarial example* can be formally defined as following. For a pre-trained neural network f , let x be an original image. An adversarial example x^{adv} is such an intentionally designed input by attackers which can guide the model f to make an incorrect prediction. Moreover, to hide the adversarial perturbation, the generation of x^{adv} is equivalent to solve the following constrained optimization problem:

$$\begin{aligned} \min_{x^{adv}} & \|x^{adv} - x\|_p \\ \text{s.t. } & \bar{y} = f(x^{adv}) \\ & y = f(x) \\ & y \neq \bar{y} \end{aligned}$$

where y and \bar{y} are respectively the prediction results of feeding x and x^{adv} to f , and $\|\cdot\|_p$ denotes the L_p -norm.

Based on the value of p , three metrics exist, i.e., L_0 , L_2 , and L_∞ , which are usually used to measure human's perception of visual difference. Specifically, L_0 measures how many pixels are modified at the corresponding positions in the resulting image; L_2 represents the Euclidean distance between the two images; and L_∞ measures the maximum difference for all pixels at the corresponding positions in the two images.

Depending on the manner of how \bar{y} misleads a pre-trained classifier, adversarial attacks to neural networks can be categorized as either targeted or non-targeted. The aim of non-targeted attacks is to make the image be classified as any arbitrary class except the true one. By contrast, in targeted attacks the prediction result will be misguided to a specific class different from the correct one and desired by the attacker.

In this study, we focus on the discussion of L_0 AE attacks, where JSMA and CW- L_0 are two widely used and representative L_0 AE generation methods. Next, we will describe these AE generation methods briefly.

2.1 Jacobian Saliency Map Attack (JSMA)

The JSMA is a targeted attack based on a greedy iterative idea proposed by Papernot et al. [34]. It takes L_0 distance minimization as the optimization target, that is, the number of pixels that can be updated in the original image is bounded. To determine which pixels will be manipulated, the authors introduce the concept of *saliency map* which provides an adversarial saliency score for each pixel. One single pixel that possesses a higher adversarial saliency score usually has more impact on misleading the target model to predict a specific label desired by attackers. Thus, the attacker only manipulates

those pixels that have high adversarial saliency scores in each iterative step based on a greedy strategy. The adversarial saliency score for each pixel is calculated as:

$$x_{i,t}^{adv} = x_{i,t} + \begin{cases} 0, & \text{if } \frac{\partial f_t(x)}{\partial x_i} < 0 \text{ or } \sum_{j \neq t} \frac{\partial f_j(x)}{\partial x_i} > 0 \\ \frac{\partial f_t(x)}{\partial x_i} \left| \sum_{j \neq t} \frac{\partial f_j(x)}{\partial x_i} \right|, & \text{otherwise} \end{cases}$$

where i denotes the i th pixel in the image, and f_j is the prediction value of the neuron j in the target model's output layer.

2.2 Carlini & Wagner Attack (CW)

CWs are a group of targeted AE generation methods developed by Carlini and Wagner [8]. There are three types of CW attacks that use distance metrics L_0 -, L_2 - and L_∞ -norm, respectively. In this work, we focus on the first type of CW attacks, where L_0 -norm is used as the distance metric during the construction of AEs. In the following presentation, We refer to such a CW attack as CW- L_0 .

Some notable features are developed by the authors which make CW attacks very effective. First, to compute the loss in gradient descent, the algorithm does not directly use final prediction given by the target model; instead, a logit function is used which plays a key role in the resilience improvement of the attack against defensive distillation [35]. Second, this algorithm maps the target variable to a space of the inverse trigonometric function; as a result, the optimization problem is suitable to be computed by a modern solver such as Adam [20]. Finally, a particular constant is designed to adjust the relative importance between perturbations and the misclassification; through this, a fine-grained trade-off is enabled. These techniques ensure that the CW method can generate superior adversarial examples with minimized perturbations.

3 System Design

The proposed system consists of a Siamese network (Section 3.2) which determines whether an input image is an L_0 AE, and a pre-processor (Section 3.1) which also can be used as a defense component to correct the classification under the existence of L_0 AEs. Note that the pre-processor has a very small impact on benign images; thus it can be used as a defense component independently without relying on detection.

3.1 Pre-processor

The pre-processor adopted in our system is designed to reduce adversarial noises while preserving the features in images to reduce false positives. From this perspective, the proposed pre-processor can also be deployed as a defense against L_0 attacks.

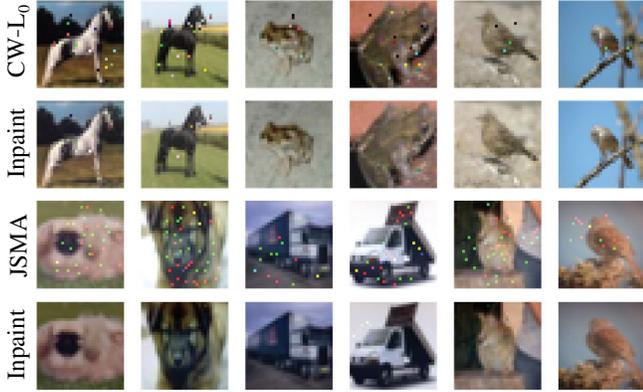


Figure 2: Defense based on inpainting. The first and third rows show the CW- L_0 and JSMA attack applied to CIFAR-10 images, respectively. The second and fourth rows show the corresponding resulting images after inpainting.

Intuitively, failing to limit the amplitude of those altered pixels in the images will result in outlier pixels. Previous work [24] emphasizes that it is challenging to get rid of the effect of those heavy perturbations. However, we argue the outlier pixels can be fixed by applying a processor based on *inpainting*. In image processing, the term “inpainting” refers to the process of reconstructing lost or corrupted regions of image data (or to remove small defects). Our idea is to treat those outliers as small corrupted regions, and the inpainting technique exactly meets the need for eliminating the L_0 noise.

In detail, we observe that those L_0 perturbations manifest themselves visually as salient noises. A mask to determine which pixels should be reconstructed can help identify these cases. When inspecting the pixel intensity in different color channels (e.g., the R, G, B channels for color images), for an altered pixel, it is highly possible that one *extreme value* can be observed in at least one channel. For example, an original pixel is represented as an intensity vector [0.32, 0.56, 0.62], where all the values are normalized. After corrupting by the L_0 attack, it becomes [0.33, 0.55, 0.96], whose B channel has an *extreme value* 0.96. We define a value as *extreme* if it is either smaller than an upper bound α or larger than a lower bound β . Thus, to obtain such a mask, we first locate all pixels of which the intensity are exceptional at least one channel. Meanwhile, we noticed that such pixels that achieve *extreme values* in all of the three channels are often the bright parts such as the sky in a natural image. Therefore, we use a parameter γ to help filter out such pixels in color images. According to our observation, we choose $\gamma = 0.7$ as an empirical value. Lines 4-10 in Algorithm 1 show the procedures to initially create the mask.

In addition, considering that the number of altered pixels only occupies a small portion of the image, the possibility that most of the altered pixels will assemble to form a connected region is very low. Consequently, to further exclude

Algorithm 1: The Pre-processor based on Inpainting.

Input: A color image I ;
two bounds α and β used to find extreme values;
a parameter γ to describe *bright pixels* in natural images;
a *structuring element* \mathcal{E} of the specified size and shape.
Output: A processed color image, denoted by \mathcal{S} .

- 1 Normalize $I \leftarrow [\min(I) - I] / [\min(I) - \max(I)]$;
- 2 Extract three channels (I^R, I^G, I^B) from I ;
- 3 Initialize the masks $\mathcal{M}^R, \mathcal{M}^G, \mathcal{M}^B \leftarrow \{0\}$;
- 4 **for** each pixel $I_i \in I$, **do**
- 5 **if** ($I_i^R < \alpha$) \vee [$(I_i^R > \beta) \wedge (I_i^G \leq \gamma \vee I_i^B \leq \gamma)$] **then**
- 6 $\mathcal{M}_i^R \leftarrow 1$;
- 7 **if** ($I_i^G < \alpha$) \vee [$(I_i^G > \beta) \wedge (I_i^R \leq \gamma \vee I_i^B \leq \gamma)$] **then**
- 8 $\mathcal{M}_i^G \leftarrow 1$;
- 9 **if** ($I_i^B < \alpha$) \vee [$(I_i^B > \beta) \wedge (I_i^G \leq \gamma \vee I_i^R \leq \gamma)$] **then**
- 10 $\mathcal{M}_i^B \leftarrow 1$;
- 11 **for** each pixel $\mathcal{M}_i^\chi \in \mathcal{M}^\chi$, where $\chi := R, G, B$, **do**
- 12 **if** $\exists N(\mathcal{M}_i^\chi) > \mathcal{E}$, s.t. $\mathcal{M}_j^\chi = 1 \wedge \mathcal{M}_j^\chi \in N(\mathcal{M}_i^\chi)$ **then**
- 13 $\mathcal{M}_j^\chi \leftarrow 0$;
- 14 **for** each $I^\chi := I^R, I^G, I^B$, **do**
- 15 $\mathcal{S}^\chi \leftarrow$ Inpainting I^χ according to \mathcal{M}^χ ;
- 16 Reconstruct \mathcal{S} with $\mathcal{S}^R, \mathcal{S}^G$ and \mathcal{S}^B ;
- 17 **return** \mathcal{S} .

those unlikely candidates, we will remove those relatively large connected regions from the mask. Specifically, we use a *structuring element* \mathcal{E} to describe a connected region with the specified size and shape. If a connected region is larger than \mathcal{E} , we will exclude such region from the mask, as Lines 11-13 in Algorithm 1 show, where $N(\cdot)$ denotes a connected neighborhood.

We thus independently produce an inpainting mask for each channel of a color image. We then take advantage of the inpainting method proposed in [42] to restore those deteriorated pixels for each channel, as Lines 14-15 in Algorithm 1 show. Figure 2 displays some concrete examples applying Algorithm 1 with $\alpha = 0.2$, $\beta = 0.8$ on CIFAR-10. The resultant images in the even numbered row show that the adversarial perturbations are almost completely eliminated. We will provide more detailed experimental results to demonstrate how our defense influences the effectiveness of L_0 attacks in Section 4.

The algorithm for gray images is very similar to Algorithm 1, but we only need to consider one channel rather than three. Thus, we can consider the algorithm for gray images as a special case of the algorithm for color images.

Parameters selection. At beginning, our algorithm normalizes the value of all input pixels, such that their values are in the range of [0, 1]. (1) α is the upper bound of extremely small values; thus, the value of α should be small (e.g. less

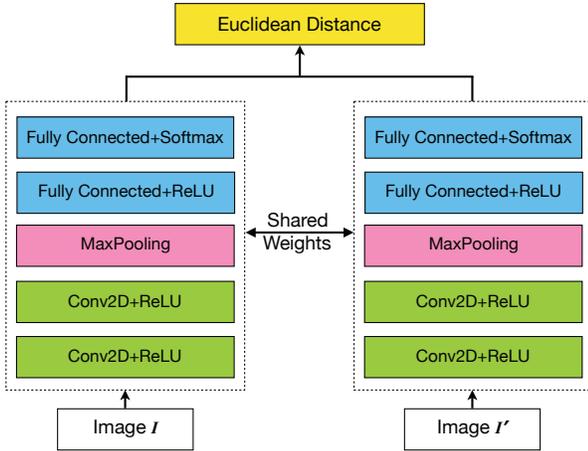


Figure 3: The architecture of a Siamese network which is used as our AE detector.

than 0.2). (2) β is the lower bound of extremely large values; thus, it should be relatively large (e.g. 0.7 at least). Different parameters settings slightly affect the effectiveness of rectifying AEs. We show the experiment results in Section 4.3. (3) In addition, as aforementioned, we use a parameter, γ , to help filter out the normal bright parts in a natural image. The term *atmospheric light* refers to those pixels, which has been discussed in detailed in the field of image processing [19]. Based on our experience, in our experiment (Section 4), the value of γ is set to 0.7. (4) Finally, the structuring element \mathcal{E} is closely related to the restoration capability of the inpainting algorithm and the size of input images. The corrupted region that can be restored by the widely used inpainting algorithms is not only a single pixel but also a small patch [29, 40, 42]. However, as the size of patch increases, the restoration effect usually degrades. A recommendation size of \mathcal{E} given by [42] ranges from three to ten pixels. Note that the performance of the pre-processor has little impact on the detection accuracy of AEPECKER, as demonstrated in our evaluation (see Section 4.4.3).

3.2 Siamese Network-Based Detector

As a classic category of neural network architecture, Siamese networks [6] are widely applied among those tasks that involve detecting similarities or other relationships between two or more comparable things [49]. In general, a Siamese network consists of two sub-networks which share one identical architecture with the same weights.

Given an input image I , when pre-processing is adopted, the input image I and the pre-processed one I' may be very different even if I is benign. On the other hand, the discrepancy between the two images, I and I' , may not be simply described using a single value and compared with a threshold,

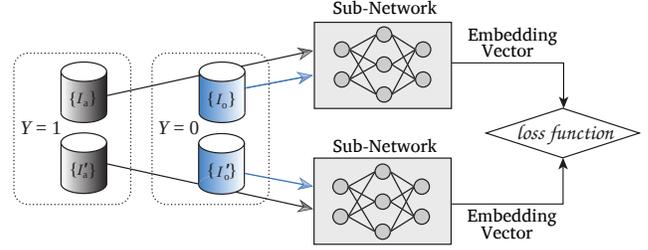


Figure 4: Training phrase of the AE detector based on a Siamese network.

as adopted by *feature squeezing* [47]. These are the main challenges in devising an accurate detection technique.

We propose a Siamese-based L_0 AE detector with the help of a *pre-processor*, which converts the AE detection problem into an image comparison problem. Once the model with fine tuned weights is established (via training), the discrepancy between I and I' can be extracted by the Siamese network. Taking the discrepancies as features, the model can predict whether the input image is adversarial or not.

Figure 3 illustrates the architecture of our Siamese network-based AE detector. In particular, we learn from the classical AlexNet [21] to design our CNN-based sub-networks but only use a shallow network. The purpose is to explore how well the AE detector performs even when it only uses a simple network design. The details of the sub-network employed by each twin in the Siamese network are as follows:

$$\begin{aligned}
 CNN : & \rightarrow conv(3 \times 3, 64) \rightarrow ReLU \\
 & \rightarrow conv(3 \times 3, 64) \rightarrow ReLU \\
 & \rightarrow maxpool(2 \times 2) \rightarrow dropout(0.3) \\
 & \rightarrow Flatten \\
 & \rightarrow linear(_, 128) \rightarrow ReLU \rightarrow dropout(0.5) \\
 & \rightarrow linear(128, 10) \rightarrow softmax.
 \end{aligned}$$

Figure 4 elaborates the training phrase of the proposed detector based on a Siamese network. Given an image I and its pre-processed version I' , the Siamese network takes $\langle I, I' \rangle$ as inputs, where the label is 0 if I is not an AE (denoted as I_o), or 1 if I is an AE (denoted as I_a). Although it is difficult to use a formula to describe the discrepancy between the input pair $\langle I_a, I'_a \rangle$ and the consistency between $\langle I_o, I'_o \rangle$, the Siamese network is effective in learning such relationship. Moreover, the consistency and discrepancy can be learned even when a non-powerful pre-processor is adopted, such as a bit depth reducer (see Section 4.4.3). The result of the last layer of each of the two sub-networks is fed to a contrastive loss function [15]:

$$(1 - Y) \frac{1}{2} (D_W)^2 + (Y) \frac{1}{2} \{(\max(0, m - D_W))\}^2$$

where D_W is defined as the Euclidean distance between the outputs of the two sub-networks, Y is a binary label assigned to the input pair, and $m > 0$ is a margin used to define a radius

around the output of one of the sub-networks. Finally, once the model is successfully trained, the Siamese network can be used to determine whether I is an AE.

Our evaluation shows that, even with a relatively small training dataset and a network with very few layers, our detector can still achieve a very high accuracy.

4 Evaluation

In this section, we evaluate our system on its detection and defense capability. We first describe the experimental settings and implementation (Section 4.1) and discuss the datasets used in our evaluation (Section 4.2). We then evaluate the effect of our pre-processing method as a defense alone (Section 4.3). Next we evaluate the accuracy of our system on detecting AEs generated by JSMA and CW- L_0 (Section 4.4), and the efficiency in terms of training and testing (Section 4.5). The resilience of our system against an adaptive attack is presented in Section 5.

It is worth noting that our proposed method can not only detect adversarial examples but also rectify the classification results. Thus, Section 4.3 shows that our pre-processor as a defense can *individually* and functionally rectify the classification results of L_0 AEs. Section 4.4 demonstrates that the detector (i.e., pre-processor plus the Siamese architecture) can distinguish AEs from benign images.

4.1 Experimental Settings

Threat model. We assume attackers have full knowledge on a trained target image classification model, but no ability to influence that model. Thus, given a trained target model, attackers can use the L_0 attacks including JSMA and CW- L_0 to generate AEs that will be misclassified by the target model.

Target models. We use two popular datasets for the image classification task: MNIST and CIFAR-10. For each dataset, we build up two individual models for the two types of L_0 attacks. Specifically, for MNIST, we set up a CNN-based classifier [18] for JSMA, and reuse the model structure provided in [8]—which we denote as Carlini $_M$ —for CW- L_0 . For CIFAR-10, we select the 32-layered ResNet model based on a residual learning framework [16] for JSMA, and reuse the model structure given in [8]—which we denote as Carlini $_C$ —for CW- L_0 . All the target models are trained from scratch.

Table 1 summarizes the classification accuracy on the testing data of each model. The accuracy of Carlini $_M$ and the CNN target model for MNIST is 99.26% and 99.52%, respectively; and the accuracy of Carlini $_C$ and the ResNet model for CIFAR-10 is 78.86% and 91.96%, respectively. Note that only those images which can be *correctly classified* by the corresponding target models are used to generate AEs in the following experiments.

Dataset	Target Model	Accuracy
MNIST	Carlini $_M$ [8]	99.26%
	CNN [18]	99.52%
CIFAR-10	Carlini $_C$ [8]	78.86%
	ResNet [16]	91.96%

Table 1: Classification accuracy of the target models.

Attacks. For the target models Carlini $_M$ and Carlini $_C$, we reuse the code provided in [8] to generate CW- L_0 AEs. The default parameters settings suggested by Carlini and Wagner [8] are as follows: the number of maximum iterations is 1000, the initial constant is 0.001, and the largest constant is 2^6 . To compare with the state-of-the-art works [27, 47], we follow these parameters settings. Furthermore, for the target CNN and ResNet model, we generate AEs with JSMA by leveraging the Adversarial Robustness Toolbox (ART) [33]. We used the same parameters settings as [27, 47], i.e., $\theta = 1, \gamma = 0.1$. As both JSMA and CW- L_0 are targeted attacks, we designate the *next* class as the target class.

Table 2 reports the results of the AEs. The *success rate* is defined as the probability that an adversary achieves their goal. For a targeted attack, it is only considered a success if the model predicts the target class. Note that we only use the AEs that can *successfully attack the target models* to evaluate the performance of our system on detecting AEs.

Implementation. We implement our Siamese-based detector in Python using the Keras [9] platform with TensorFlow [1] as backend. Keras provides a large number of high-level neural network APIs and can run on top of TensorFlow. The Tealea’s inpainting algorithm [42] is implemented based on Open Source Computer Vision Library (OpenCV) [5].

The experiments were performed on a computer running the Ubuntu 18.04 operating system with a 64-bit 3.6 GHz Intel® Core™ i7 CPU, 16 GB RAM and GeForce GTX 1070 GPU.

4.2 Data Preparation

We generate AEs based on two image datasets, i.e., CIFAR-10 and MNIST.

CIFAR-10 contains 60,000 color images; each is assigned to one of ten different classes, such as dog, frog and ship. CIFAR-10 is split into the training and testing dataset, which contains 50,000 and 10,000 images, respectively.

We first filter out those images that cannot be correctly classified by the corresponding target model. We then use the CW- L_0 algorithm to generate AEs that can *successfully attack* the Carlini $_C$ model [8], and create two disjoint datasets, denoted as $\mathcal{D}_{C-CWL0-Train}$ and $\mathcal{D}_{C-CWL0-Test}$. In detail, $\mathcal{D}_{C-CWL0-Train}$ contains 10,000 legitimate images and 10,000 AEs. $\mathcal{D}_{C-CWL0-Test}$ contains 1,000 benign images and 1,000 AEs. Next, we follow the similar method on

Dataset	Attack	Success rate
MNIST	CW- L_0 [8]	100%
	JSMA [34]	81.6%
CIFAR-10	CW- L_0 [8]	100%
	JSMA [34]	99.8%

Table 2: Evaluation of the L_0 attacks

Loss ratio	60%	40%	20%
JSMA	27.4%	17.9%	5.4%
CW- L_0	44.5%	35.1%	21.4%

Table 5: The classification accuracy on testing datasets after applying SVD compression.

CIFAR-10 but instead using JSMA to generate AEs based on ResNet classifier [16]. As a result, we obtain two dis-joint datasets, denoted as $\mathcal{D}_C\text{-JSMA-Train}$ and $\mathcal{D}_C\text{-JSMA-Test}$. There are 10,000 legitimate images and 10,000 AEs in training set. There are 1,000 legitimate images and 1,000 AEs in testing set.

MNIST contains 70,000 8-bit grayscale images of handwritten digits. Each image is assigned a label from 0 to 9. MNIST is split into the training and testing dataset, which contains 60,000 and 10,000 images, respectively. We carry out similar procedures on MNIST to create a training and testing set but using different target models. As a result, we have $\mathcal{D}_M\text{-CWL0-Train}$ and $\mathcal{D}_M\text{-CWL0-Test}$ based on Carlini_M model [8], as well as $\mathcal{D}_M\text{-JSMA-Train}$ and $\mathcal{D}_M\text{-JSMA-Test}$ based on CNN [18] model. The sizes of these datasets are the same as their counterparts in CIFAR-10. Considering that CIFAR-10 is a more challenging dataset compared with MNIST, we will spend more space on explaining the results for CIFAR-10 in the following experiments.

Note that all the aforementioned legitimate images can be classified correctly by the target model, and all the AEs can successfully fool the corresponding target model.

4.3 Effectiveness of Pre-processor as Defense

To mislead a classifier to predict a specific target class, the adversarial perturbations produced by an L_0 attack such as JSMA or CW- L_0 are introduced intentionally instead of randomly. Moreover, the adversarial strength of an L_0 attack limits the number of pixels that can be manipulated; and as a result, the manipulated pixels need to have significant changes. The proposed inpainting-based pre-processor is to eliminate the possible adversarial pixels while preserving the benign ones. Therefore, the inpainting-based pre-processor can also be considered as a defense against L_0 attacks.

$\beta \backslash \alpha$	0.0	0.1	0.2
0.6	81.3%	86.9%	84.2%
0.7	80.5%	87.3%	86.5%
0.8	76.0%	86.2%	86.5%

Table 3: The classification accuracy on AEs in $\mathcal{D}_C\text{-CWL0-Test}$ after using inpainting-based pre-processors.

$\beta \backslash \alpha$	0.0	0.1	0.2
0.6	90.0%	81.2%	63.2%
0.7	94.1%	88.8%	74.5%
0.8	96.1%	91.2%	77.3%

Table 4: The classification accuracy on AEs in $\mathcal{D}_C\text{-JSMA-Test}$ after using inpainting-based pre-processors.

Dataset	Attack	Accuracy	Precision	Recall	F1 Score	FPR
CIFAR-10	JSMA	99.85%	100.0%	99.70%	99.85%	0.0%
	CW- L_0	95.80%	94.64%	97.10%	95.85%	5.5%
MNIST	JSMA	99.80%	99.90%	99.70%	99.80%	0.1%
	CW- L_0	99.40%	99.30%	99.50%	99.40%	0.7%

Table 6: The detection performance of the proposed system.

Inpainting-based pre-processor for color images. We first evaluate the effectiveness of the inpainting-based pre-processor as a defense against L_0 attack on CIFAR-10. The inpainting-based algorithm has two parameters: the threshold α extracts the pixels whose values tend to be very small, and β is used to screen all pixels whose values tend to be extremely large. We use 1,000 AEs in $\mathcal{D}_C\text{-CWL0-Test}$ to evaluate the effectiveness of the inpainting-based pre-processor as a defense against CW- L_0 attacks and examine its performance with varying values of α and β . Without pre-processing, these AEs result in 0% classification accuracy when using the Carlini_C model [8]. After pre-processing, each recovered AE is analyzed by the model to predict a class label. Table 3 shows the results. When $\alpha = 0.1$ and $\beta = 0.7$, the performance is the best—the classification accuracy on these AEs is increased from 0% to 87.3%.

We then use 1,000 AEs in $\mathcal{D}_C\text{-JSMA-Test}$ to evaluate the effectiveness of the inpainting-based pre-processor as a defense against JSMA attack. Without pre-processing, these AEs result in 0% classification accuracy of the ResNet model [16]. After applying inpainting-based pre-processor to rectify those AEs, each recovered AE is analyzed by the ResNet model to predict a class label. Table 4 shows the results. We can see that when $\alpha = 0.0$ and $\beta = 0.8$, the performance is the best—the classification accuracy on these AEs is increased from 0% to 96.1%. This classification accuracy is higher than 87.3% given by the previous Carlini_C model. Moreover, the ResNet model is more robust against the benign perturbations introduced by the inpainting procedure.

As a comparison, we examine the impact of both SVD compression and median filter on AEs generated by L_0 attacks. First, as a low-pass filter, SVD compression is usually used to reduce noise in images. As shown in Table 5, when varying the loss ratio of SVD compression, the classification accuracy on the processed AEs is very low—at most 44.5% and 27.4% for CW- L_0 AEs and JSMA AEs, respectively. Note that we

only use those images which can be correctly classified by the target model to generate AEs; thus the maximum classification accuracy given by the target model here is 100%. Therefore, the experiment suggests that the perceptible perturbations introduced by an L_0 attack are very difficult to be reduced when only using the frequency domain filters. Alternatively, [47] and [14] claim that median filter is particularly effective in mitigating adversarial examples generated by an L_0 attack because such perturbations are very similar to salt-and-pepper noises. In our experiments, after applying the median filter to process AEs in $\mathcal{D}_C\text{-CWL0-Test}$ and $\mathcal{D}_C\text{-JSMA-Test}$, the classification accuracy given by Carlini_C and ResNet is 79.8% and 85.3%, respectively; both are lower than the proposed defense.

Inpainting-based pre-processor for gray images. We can observe similar results on MNIST when taking advantage of the inpainting-based pre-processor as a defense against an L_0 attack. Our experiment shows that processing the 1,000 AEs in $\mathcal{D}_M\text{-CWL0-Test}$ with the proposed inpainting-based method results in a significant increase of the classification accuracy on the Carlini_M model [8]—from 0% to 88.2%. Similarly, after using the proposed inpainting-based method on the 1,000 AEs in $\mathcal{D}_M\text{-JSMA-Test}$, the recovered AEs result in the classification accuracy on the CNN model [18] to increase from 0% to 86.1%. All of the results above are obtained when $\alpha = 0.1$ and $\beta = 0.8$. When varying the value of α from 0.1 to 0.2 and the value of β from 0.7 to 0.8, the classification accuracy increases slowly (84.9% at least).

As a comparison, Bafna et al. [2] independently focus on the L_0 attacks, and propose a defense based on Fourier transform. But our approach outperforms theirs—after applying their defense algorithm against CW- L_0 , their classification accuracy on the MNIST testing set is only 72.8%. Note that they did not conduct experiments on standard color-image datasets such as CIFAR-10.

Impact on benign images. To investigate the impact of the defense methods on benign images, we first carry out an experiment on the 1,000 benign images from $\mathcal{D}_C\text{-JSMA-Test}$. Specifically, we use the ResNet model [16] to classify each color image after the inpainting-based defense is applied. The classification accuracy on these processed images only decreases from 100% to 95.6%. Next, we conduct a similar experiment on the 1,000 benign images from $\mathcal{D}_M\text{-JSMA-Test}$. We use the CNN model [18] to classify each gray image after the inpainting-based defense is applied. As a result, the classification accuracy on these processed images only decreases from 100% to 99.7%. The results show that a very small impact is imposed on classifying benign images.

Summary. Therefore, the proposed inpainting-based algorithm is effective in defending against L_0 attacks such as CW- L_0 and JSMA. Moreover, our defense methods have a very small impact on benign images, which implies it can be directly applied without relying on detection.

4.4 Detecting L_0 Adversarial Inputs

We next evaluate the effectiveness of our system on detecting AEs generated by L_0 attacks.

4.4.1 Detection Efficacy

We evaluate the detection performance of the proposed scheme against CW- L_0 and JSMA attack. The inpainting-based pre-processor is used to create input pairs to the Siamese network.

Color images. The two training datasets, $\mathcal{D}_C\text{-CWL0-Train}$ and $\mathcal{D}_C\text{-JSMA-Train}$, are used to train our system individually for 200 epochs using early stopping configured with a minimum accuracy change of 0.001 and 50 patience steps. If an accuracy change is less than 0.001, we consider that there is no improvement of the model performance; after 50 epochs with no improvement, the training is stopped. We save the resulting models as the base models.

We now evaluate the detection accuracy of the base models against CW- L_0 and JSMA attack on $\mathcal{D}_C\text{-CWL0-Test}$ and $\mathcal{D}_C\text{-JSMA-Test}$, respectively. Each dataset contains 1,000 benign images and 1,000 AEs. We plot the ROC (receiver operating characteristic) curves, which are showed in Figure 5(a). We can achieve the AUC values of 98.69% and 99.94% for the two L_0 attacks. Table 6 shows more detailed results evaluated on the testing set.

We consider the adversarial images as positive, and the benign images as negative. Thus, the recall value (i.e., the detection rate) is the ratio of the number of successfully detected AEs to the total number of AEs; and False Positive Rate (FPR) is the fraction of the negative testing data (i.e., benign images) that is misclassified as positive. In practice, the distribution of adversarial and benign images are not balanced—most of the images should be benign. Thus, FPR is a very important metric to evaluate the model performance; a lower FPR indicates that the system makes fewer mistakes for benign images.

As shown in Table 6, when analyzing AEs generated by the CW- L_0 attack, the detection rate of $\mathcal{D}_C\text{-CWL0-Test}$ is 97.1% and the FPR is 5.5%. When analyzing AEs generated by the JSMA attack, the detection rate of $\mathcal{D}_C\text{-JSMA-Test}$ is 99.7% and the FPR is as low as 0.0%.

Gray images. We follow the same configurations to conduct an experiment on MNIST. Because gray images only have one channel, training a Siamese network on MNIST is simpler than that on CIFAR-10. We thus only train the detector for 100 epochs using an early stopping with 30 patience steps.

We evaluate the detection accuracy of the base models against CW- L_0 and JSMA attacks on $\mathcal{D}_M\text{-CWL0-Test}$ and $\mathcal{D}_M\text{-JSMA-Test}$, respectively. We plot the ROC curves as the Figure 5 (b) shows. The AUC value can achieve 99.84% and 99.93%. In Table 6, we can observe similar results as the experiments given on CIFAR-10. When facing CW- L_0 attacks, the detection rate for the AEs from $\mathcal{D}_M\text{-CWL0-Test}$ is

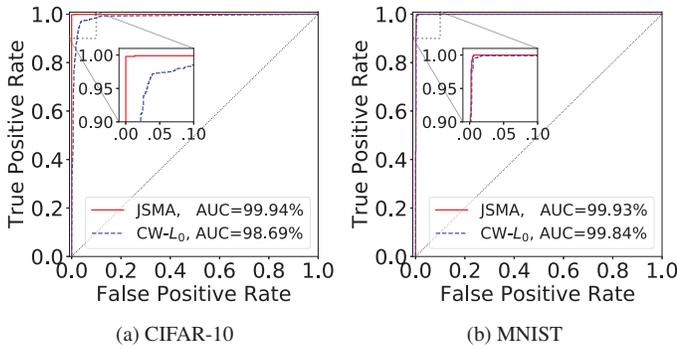


Figure 5: ROC curves for different datasets.

99.5%, and the FPR is 0.7%. When facing JSMA attacks, the detection rate for the AEs from $\mathcal{D}_{\mathcal{M}}\text{-JSMA-Test}$ can achieve 99.7%, and the FPR is as low as 0.1%.

Comparison. We compare the proposed system with the state-of-the-art AE detectors, including feature squeezing [47] and NIC [27]; both of them show that their systems are able to effectively detect L_0 AEs. Moreover, feature squeezing [47] uses multiple feature squeezers, and we only compare our system with the *best* results of their work. To this end, we train two comprehensive models for color images and gray images, respectively. Specifically, for color images, we train the detector using both $\mathcal{D}_{\mathcal{C}}\text{-CWL0-Train}$ and $\mathcal{D}_{\mathcal{C}}\text{-JSMA-Train}$. For gray images, we train another detector using $\mathcal{D}_{\mathcal{M}}\text{-CWL0-Train}$ and $\mathcal{D}_{\mathcal{M}}\text{-JSMA-Train}$. We summarize the adversarial detection rate and FPR in Table 7. For CIFAR-10, the detection rate of feature squeezing [47] on CW- L_0 and JSMA attacks is 98.1% and 83.7%, respectively, and its FPR—the percentage of the benign images among all the testing benign images that is misclassified as positive—is 4.9%. NIC [27] can achieve the detection rate of 98.0% and 94.0% on CW- L_0 and JSMA attacks respectively, and its FPR is 3.8%. Our AEPECKER outperforms theirs—we can achieve the detection rate of 98.4% and 99.5% for the two types of L_0 attacks and our FPR is only 2.0%. With respect to MNIST, the detection rate of our model is comparable with feature squeezing [47] and NIC [27]. Moreover, AEPECKER achieves the lowest FPR for both CIFAR-10 and MNIST. Therefore, our proposed detector outperforms the two state-of-the-art detectors.

4.4.2 Transferability

This experiment is to evaluate the transferability of our system: whether our system trained on one type of L_0 AEs can be directly applied to detect another type of L_0 AEs that have not been seen during training without any adaptation. To this end, we train our system using $\mathcal{D}_{\mathcal{C}}\text{-JSMA-Train}$ and use $\mathcal{D}_{\mathcal{C}}\text{-CWL0-Test}$ to test the detector. The result shows that the detection rate is as high as 99.4%. Similarly, we train our

Dataset	Detector	FPR	CW- L_0	JSMA
CIFAR-10	AEPECKER	2.0%	98.4%	99.5%
	FS [47]	4.9%	98.1%	83.7%
	NIC [27]	3.8%	98.0%	94.0%
MNIST	AEPECKER	0.4%	99.1%	99.3%
	FS [47]	4.0%	91.1%	100%
	NIC [27]	3.7%	100%	100%

Table 7: Comparison with state-of-the-art detectors in terms of FPR and detection rate.

system using $\mathcal{D}_{\mathcal{C}}\text{-CWL0-Train}$ and use $\mathcal{D}_{\mathcal{C}}\text{-JSMA-Test}$ to test the detector. The result shows that the detection rate is as high as 98.7%.

The similar results are obtained for MNIST: if we use $\mathcal{D}_{\mathcal{M}}\text{-JSMA-Train}$ to train the Siamese network and use $\mathcal{D}_{\mathcal{M}}\text{-CWL0-Test}$ to test the detector, the detection rate is 96.3%; if our system is trained using $\mathcal{D}_{\mathcal{M}}\text{-CWL0-Train}$ and tested on $\mathcal{D}_{\mathcal{M}}\text{-JSMA-Test}$, the detection rate can achieve 95.4%.

Summary. Therefore, our system has good transferability; our system trained on AEs generated by one L_0 attack can be directly applied to detect AEs generated by another L_0 attack without any adaptation.

4.4.3 Pre-processor Study

We next conduct an experiment to examine the impact of the pre-processor; specifically, we would like to see what the detection accuracy will be if a weak pre-processor is adopted. The *weak* here means the manipulated AEs through such a pre-processor still cannot be classified correctly by the target model with a high possibility. Through this, we will show that even with a weak pre-process, our system can still achieve a high detection accuracy—this means that a perfect pre-processor is unnecessary for our Siamese-based detector to achieve a high success rate of detection.

Without loss of generality, we use color images as an example for the following discussion. For color images such as CIFAR-10, each channel of RGB is encoded by 8 bits. As Figure 6 shows, we can reduce the original 8 bits to fewer bits without influencing the image recognizability for human eyes. Figure 6 also shows that it is very difficult to remove those striking adversarial perturbations introduced by L_0 attacks only with such an approach. Moreover, the original L_0 AEs can mislead the target neural networks to a classification accuracy of 0%. After applying bit depth reduction, the classification accuracy for AEs in the testing datasets is calculated. The experiment results are shown in Table 8, which suggest that processing the AEs generated by JSMA and CW- L_0 with

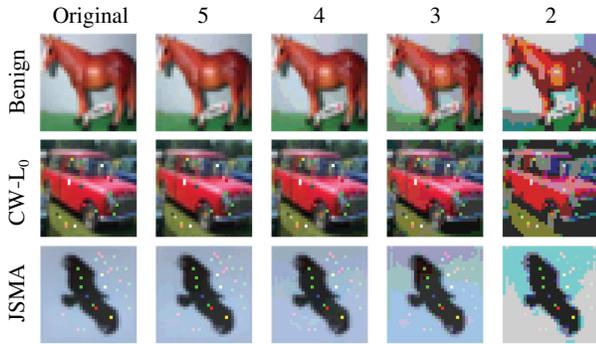


Figure 6: Image examples from CIFAR-10 after applying bit depth reduction. Given the different numbers of bit depth, the first row displays a benign image and its processed versions; the second row displays an AE generated by $CW-L_0$ and its corresponding processed images; the third row displays an AE generated by JSMA and its corresponding processed images.

bit depth reduction cannot increase the classification accuracy of the target model. Therefore, the bit depth reduction approach only has a very limited capability to defend against L_0 attacks.

We thus choose the bit depth reduction as a weak pre-processor for color images. The experiment results show that even when a weak pre-processor (such as bit depth reduction) is applied, our Siamese-based detector still can achieve a very good performance. The detection rates for AEs generated by JSMA and $CW-L_0$ are 99.6% and 99.4%, respectively; and the FPR is 2.1%. Xu et al. also use bit depth reduction as a pre-processor [47]; however, the best detection rates provided by their system for AEs generated by JSMA and $CW-L_0$ are 4.1% and 36.5% respectively, and its FPR is 5%.

Summary. Therefore, our proposed Siamese-based detector outperforms the state-of-the-art method when using the same weak pre-processor. The result also demonstrates that the good performance of our detector does not rely on a perfect pre-processor, but is due to the Siamese network design.

4.5 Efficiency

Training time. It is widely known that neural networks usually require a large amount of data and time for training. However, as our sub-networks employed within the Siamese architecture are quite simple and shallow, the training is very efficient. For example, for $\mathcal{D}_M\text{-JSMA-Train}$ and $\mathcal{D}_C\text{-JSMA-Train}$, each epoch with 20,000 images (10,000 benign images and 10,000 AEs) only takes 5 and 7 seconds, respectively. On the other hand, due to the simple and shallow sub-networks, with a relatively small training set, our Siamese neural-network-based AEPeCKER can still achieve high detection accuracies (Section 4.4). Moreover, the training time

Datasets	Bit Depth			
	2-bit	3-bit	4-bit	5-bit
$\mathcal{D}_C\text{-JSMA-Test}$	22.2%	27.1%	21.2%	12.0%
$\mathcal{D}_C\text{-CWL0-Test}$	51.2%	56.6%	55.1%	51.5%

Table 8: The classification accuracy for AEs in testing datasets after applying bit depth reduction.

is linear with respect to the number of epochs and the number of training samples for each epoch.

Our experiment results show that our system trained on CIFAR-10 and MNIST can converge very quickly and achieve high accuracy within 100 and 200 epochs, respectively—thus, the training only requires around 8 minutes and 23 minutes, respectively.

Testing time. The trained detector can detect an AE very fast. For example, AEPeCKER only takes approximately 0.5ms on average to detect whether an image from CIFAR-10 is adversarial or not.

5 Adaptive Attack

An adversary who knows the details of AEPeCKER will try to adapt the attacks. Thus, we seek to understand the resilience of AEPeCKER to adaptive attacks by answering the following questions. **(Q1)** What is the percentage of the high-amplitude altered pixels in AEs generated using non-adaptive L_0 attacks? Exploration of this question not only helps us understand L_0 attacks and why the proposed detection and defense techniques work well, but also guides the adversary to adapt L_0 attacks. **(Q2)** How difficult is it for an adversary to adaptively generate L_0 AEs that bypass our detection?

To answer these questions, we launch an adaptive L_0 attack by adopting a similar method described in [17], which has successfully demonstrated a capability to impede *feature squeezing* [47]. Our implementation is based on the source code given by [8]. To generate L_0 AEs, after each step of stochastic gradient descent (SGD), an intermediate distorted image is generated as a resolution of the optimizer. Each time the optimizer runs, the process tries to minimize the number of altered pixels and, in the meanwhile, keep the targeted attack successful.

Answer to Q1. Let $N_{\mathcal{A}}$ be the number of altered pixels and $N_{\mathcal{E}}$ the number of such altered pixels that possess *extreme values* (i.e., values smaller than α or larger than β). We consider the ratio $\rho = N_{\mathcal{E}}/N_{\mathcal{A}}$ as an indicator showing the percentage of pixels with large-amplitude perturbations, and want to understand how this ratio changes in the AE generation process. As an empirical analysis, we carry out SGD step by step on 100 randomly selected images from CIFAR-10. For each of the last 10 steps, we calculate an average ratio $\bar{\rho}$ value of the 100 intermediate images, as shown in Figure 7. We observe that the average ratio $\bar{\rho}$ goes higher and higher as $N_{\mathcal{A}}$ de-

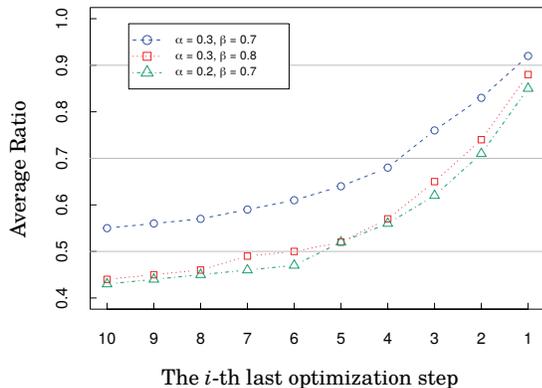


Figure 7: L_0 attacks are launched on 100 randomly selected images from CIFAR-10. For each of the last 10 optimization steps, we examine the average ratio $\bar{\rho}$ of the 100 intermediate distorted images.

creases. Finally, when the optimal resolution is found, around 90% of the altered pixels by average possess extreme values. This helps understand why the proposed technique works well, since it is designed to deal with such large-amplitude perturbations by recovering these pixels.

An adversary who is aware of the details of the proposed technique thus should try to control the amplitude of those altered pixels while satisfying the L_0 optimization target (i.e., minimizing $N_{\bar{a}}$). Thus, given an image, we run the SGD multiple times; once the value of ρ is over 80% (note this value finally can reach 90% by average), we explore different optimization paths. The result shows that only 5% of the cases succeed to control the ratio ρ under 80%. Therefore, it is difficult to control the amplitude of the altered pixels while satisfying the L_0 optimization target.

Answer to Q2. We follow the procedures described in [17] to adaptively search potential L_0 AEs. Our design of the adaptive L_0 attack is as follows. Since inpainting is used in both detection and defense, the adversary integrates it into the AE generation; during the AE generation, the intermediate image at each step of the optimization procedure is processed using our inpainting pre-processor. Next, we check whether the resulting image is a successful attack. If that it cannot successfully fool the neural network, we iteratively run SGD multiple times (10 in our experiments) until a resolution is found. We randomly select 100 color images from CIFAR-10. Our experiments show that the final number of altered pixels only takes up less than 2% of the total number of the pixels in images from CIFAR-10, which means that they achieve the L_0 optimization target. The result finally shows that only 7% of cases can generate successful AEs to evade our detection. In contrast, [17] shows that adaptive attacks using a similar method can bypass *feature squeezing* [47] at 100%. Therefore, our method is much more resilient than prior work.

Summary. Based on these explorations, we conclude that L_0 attacks have an inherent limitation, and it is difficult for adaptive attacks to overcome the limitation to bypass our detection.

6 Related Work

Generally, the protection strategies against AE attacks fall into two groups, i.e., detection and defense. In this section, we will briefly review them both.

6.1 Detecting Adversarial Examples

An AE detector is a binary classifier which is designed to distinguish an adversarial sample from a legitimate one. There are two strategies which are often used to design AE detectors, i.e., adversarial training and predication mismatch.

6.1.1 Detector Training

Some techniques use both AEs and legitimate images to train a detector. For example, Li et al. [23] extract PCA features after inner convolutional layers of the neural network, then use a cascade classifier to detect AEs. Metzen et al. [31] use both adversarial and benign samples to train a CNN-based auxiliary network. This light-weight sub-network works with the target model to detect AEs. They usually require a large number of samples to train the model while we only need a relatively small dataset. More importantly, our detector achieves a high detection rate but low FPR for handling L_0 AEs.

6.1.2 Prediction Mismatch

Some techniques use the prediction mismatch strategy. For example, Bagnall et al. [3] train an ensemble of multiple models to use a rank voting mechanism to combine those outputs. In this way, an ensemble disagreement can be used to detect adversarial examples. *Bi-model* [32] firstly employs two pre-trained distinct models to generate features, then feeds the concatenated features to an additional binary classifier.

Other works [24, 30, 43, 47] apply pre-processors on an input image. For example, Meng et al. [30] train an auto-encoder as the image filter. Tian et al. [43] pre-process the images with randomly rotation and shifting since adversarial examples are usually sensitive to such transformation operations. Liang et al. [24] implement an adaptive denoiser based on image entropy as the filter. Their methods then feed the original image and the processed one to the same neural network—if the predictions of the two images fail to match, the input is adversarial. In addition, Xu et al. [47] propose *feature squeezing* to detect AEs by comparing the prediction on original inputs with that on the squeezed ones. However, the proposed detector outperforms *feature squeezing* for handling L_0 attacks. Note that the performance of their approach heavily

relies on the effectiveness of the feature squeezing methods. On the contrary, our Siamese-based detector does not rely on powerful pre-processing (see Section 4.4.3).

Our detection technique seems close to the approaches in the second category but is actually very different. Rather than using a simple mismatch or a distance value to describe the discrepancy between an AE and its manipulated image, our technique uses a Siamese network to automatically extract the discrepancy between the two as features for detection.

6.2 Defense

The primary task of defensive techniques is to alleviate or eliminate the influences of AEs. In general, the current defensive techniques can be grouped into two major categories, that is, model enhancement and input transformation.

6.2.1 Model Enhancement

The first category improves the resilience of neural networks by including AEs in the process of model training, i.e., *adversarial training* [13, 28]. However, this type of defense is usually less effective against black-box attacks than white-box attacks considering the training only focuses on one certain neural network. Also, Xu et al. claim that this kind of technique suffers high cost because of iterative re-training with both adversarial and normal examples [47]. Alternatively, *defensive distillation* is proposed, and can obstruct the neural networks from fitting too tightly to the data [35]. However, the prior work [8] demonstrates that the approach can be easily circumvented with a minimal modified attack such as a CW- L_0 . *Shield* [11] enhances a model by re-training it with multiple levels of compressed images based upon JPEG. However, this method is still ineffective against L_0 attacks.

6.2.2 Input Transformation

For the second category of defenses, researchers have averted their eyes from neural networks to the adversarial inputs themselves. In short, pre-processing the inputs before feeding them to networks is helpful for increasing the prediction accuracy even when facing adversarial examples. The reason why adversarial examples could successfully fool the deep learning model without being perceived is that attackers take advantage of the information redundancy of images to add adversarial noise. Consequently, well designed filters or denoisers can be considered a cure for adversarial images by removing unwanted noise. For instance, Liao et al. [25] propose higher-level guided denoisers to remove the adversarial noise from inputs; however, their approach is computationally expensive and their work does not show its effectiveness on L_0 attacks. Some other methods adopt compression techniques, such as PCA [4] and JPEG [10, 12, 14, 37], to filter out the information redundancy which may provide living space for adversarial perturbations in images; however, these approaches

are not suitable for L_0 attacks. Furthermore, Bafna et al. [2] independently propose a defense against L_0 attacks; but their Fourier-transform-based approach is not as effective as ours (see Section 4.3).

There exist some approaches that do not fall in either category. For example, MVP-Ears [48] borrows the idea of multi-version programming from software engineering and applies it to audio AE detection. It deploys multiple diverse automatic speech recognition systems in parallel, and detects audio AEs by comparing their recognition results. However, the idea will probably fail in handling image AEs, which are known to have good transferability [13].

7 Discussion

First, the proposed technique is not a panacea for detecting or defending against all possible attacks. As future work, we are to explore whether the Siamese network-based detector can be generalized to detect other types of attack, such as L_∞ .

Second, our adaptive attack (following the procedures in [17]) is based on the exploration of different optimization paths. There exist some other alternative white-box attacks, such as the method proposed in [7] which attempts to create new AEs by modifying the loss functions to bypass detectors. Whether other different adaptive attacks, such as [7], can bypass our detector is interesting, and we plan to investigate it in the future.

Third, how to prevent the over-fitting problem is still an open question in the field of machine learning. Although we took over-fitting into consideration when designing the experiments (e.g., the testing dataset and training dataset are completely disjoint), the over-fitting problem is still possible. Specifically, our evaluation only examined base classifiers ResNet and CNN. Future work will consider other classifiers with different architectures.

Fourth, we are also interested in whether the proposed technique is scalable to handle a larger data set such as ImageNet. We leave this evaluation as the future work.

Lastly, this work shows that only controlling the number of altered pixels without limiting the resulting amplitude weakens the power of the generated AEs. Thus, how to make a good trade-off between the number of altered pixels and their amplitude becomes critical when designing new AE generation algorithms.

8 Conclusions

In the setting of classic L_0 AE attacks, a bounded number of pixels can be corrupted without limiting the amplitude. These large-amplitude perturbations in L_0 AEs are considered as a challenge by many previous works, since the effect of such corruptions is difficult to eliminate. Considering the threats caused by L_0 AEs, a highly accurate detection technique and

an effective defense that can rectify the classification under L_0 perturbations are urgently needed. By identifying and exploiting the inherent characteristics of L_0 AEs, we develop AEPECKER that thwarts this type of attacks. Its novel Siamese network based design shows very high accuracies in detecting L_0 AEs, and its inpainting-based preprocessing technique can effectively rectify those AEs and thus correct the classification results. Plus, it is resilient to adaptive attacks that bypass prior approaches.

Acknowledgments

We would like to thank our shepherd, Dr. Zachary Weinberg, and the anonymous reviewers for their constructive suggestions and comments. This project was supported by NSF CNS-1815144, NSF CNS-1856380, and NSF CNS-1850278.

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, et al. Tensorflow: a system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [2] Mitali Bafna, Jack Murtagh, and Nikhil Vyas. Thwarting adversarial examples: An L_0 -robust sparse Fourier transform. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.
- [3] Alexander Bagnall, Razvan Bunescu, and Gordon Stewart. Training ensembles to detect adversarial examples. *arXiv preprint arXiv:1712.04006*, 2017.
- [4] Arjun Nitin Bhagoji, Daniel Cullina, Chawin Sitawarin, and Prateek Mittal. Enhancing robustness of machine learning systems via data transformations. In *IEEE Conference on Information Sciences and Systems*, 2018.
- [5] Gary Bradski et al. OpenCV. <https://opencv.org>, 2017.
- [6] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "Siamese" time delay neural network. In *Advances in Neural Information Processing Systems (NeurIPS)*, 1994.
- [7] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *ACM Workshop on Artificial Intelligence and Security*, 2017.
- [8] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (Oakland)*, 2017.
- [9] François Chollet et al. Keras. <https://keras.io>, 2015.
- [10] Nilaksh Das, Madhuri Shanbhogue, Shang-Tse Chen, Fred Hohman, Li Chen, Michael E Kounavis, and Duen Horng Chau. Keeping the bad guys out: Protecting and vaccinating deep learning with JPEG compression. *arXiv preprint arXiv:1705.02900*, 2017.
- [11] Nilaksh Das, Madhuri Shanbhogue, Shang-Tse Chen, Fred Hohman, Siwei Li, Li Chen, Michael E Kounavis, and Duen Horng Chau. Shield: Fast, practical defense and vaccination for deep learning using JPEG compression. In *SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2018.
- [12] Gintare Karolina Dziugaite, Zoubin Ghahramani, and Daniel M Roy. A study of the effect of JPG compression on adversarial images. *arXiv preprint arXiv:1608.00853*, 2016.
- [13] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2015.
- [14] Chuan Guo, Mayank Rana, Moustapha Cisse, and Laurens van der Maaten. Countering adversarial images using input transformations. In *International Conference on Learning Representations (ICLR)*, 2018.
- [15] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2006.
- [16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [17] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial example defenses: Ensembles of weak defenses are not strong. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.
- [18] Yash Katariya. Applying convolutional neural network on the MNIST dataset. <https://github.com/yashk2810>, 2017.
- [19] Jin-Hwan Kim, Won-Dong Jang, Jae-Young Sim, and Chang-Su Kim. Optimized contrast enhancement for real-time image and video dehazing. *Journal of Visual Communication and Image Representation*, 24(3), 2013.

- [20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.
- [21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.
- [22] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *ICLR Workshop*, 2017.
- [23] Xin Li and Fuxin Li. Adversarial examples detection in deep networks with convolutional filter statistics. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.
- [24] Bin Liang, Hongcheng Li, Miaoqiang Su, Xirong Li, Wenchang Shi, and Xiaofeng Wang. Detecting adversarial image examples in deep neural networks with adaptive noise reduction. *IEEE Transactions on Dependable and Secure Computing*, 2018.
- [25] Fangzhou Liao, Ming Liang, Yinpeng Dong, Tianyu Pang, Jun Zhu, and Xiaolin Hu. Defense against adversarial attacks using high-level representation guided denoiser. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.
- [26] Lannan Luo and Qiang Zeng. Solminer: mining distinct solutions in programs. In *IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*, 2016.
- [27] Shiqing Ma, Yingqi Liu, Guanhong Tao, Wen-Chuan Lee, and Xiangyu Zhang. NIC: Detecting adversarial samples with neural network invariant checking. In *Network and Distributed System Security Symposium (NDSS)*, 2019.
- [28] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.
- [29] Julien Mairal, Michael Elad, and Guillermo Sapiro. Sparse representation for color image restoration. *IEEE Transactions on Image Processing*, 17(1), 2007.
- [30] Dongyu Meng and Hao Chen. MagNet: a two-pronged defense against adversarial examples. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.
- [31] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. On detecting adversarial perturbations. In *International Conference on Learning Representations (ICLR)*, 2017.
- [32] João Monteiro, Zahid Akhtar, and Tiago H Falk. Generalizable adversarial examples detection based on Bi-model decision mismatch. *arXiv preprint arXiv:1802.07770*, 2018.
- [33] Maria-Irina Nicolae, Mathieu Sinn, Minh Ngoc Tran, Ambrish Rawat, Martin Wistuba, Valentina Zantedeschi, Nathalie Baracaldo, et al. Adversarial robustness toolbox v0.3.0. *arXiv preprint arXiv:1807.01069*, 2018.
- [34] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.
- [35] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Symposium on Security and Privacy (Oakland)*, 2016.
- [36] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. In *British Machine Vision Conference (BMVC)*, 2015.
- [37] Aaditya Prakash, Nick Moran, Solomon Garber, Antonella DiLillo, and James Storer. Protecting JPEG images against adversarial attacks. In *IEEE Data Compression Conference (DCC)*, 2018.
- [38] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. In *NDSS Workshop on Binary Analysis Research*, 2019.
- [39] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.
- [40] Jianhong Shen and Tony F Chan. Mathematical models for local nontexture inpaintings. *SIAM Journal on Applied Mathematics*, 62(3), 2002.
- [41] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations (ICLR)*, 2014.
- [42] Alexandru Telea. An image inpainting technique based on the fast marching method. *Journal of Graphics Tools*, 9(1), 2004.

- [43] Shixin Tian, Guolei Yang, and Ying Cai. Detecting adversarial examples through image transformation. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.
- [44] Naiyan Wang and Dit-Yan Yeung. Learning a deep compact image representation for visual tracking. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2013.
- [45] Yandong Wen, Kaipeng Zhang, Zhifeng Li, and Yu Qiao. A discriminative feature learning approach for deep face recognition. In *European Conference on Computer Vision (ECCV)*, 2016.
- [46] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. Mitigating adversarial effects through randomization. In *International Conference on Learning Representations (ICLR)*, 2018.
- [47] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [48] Qiang Zeng, Jianhai Su, Chenglong Fu, Golam Kayas, Lannan Luo, Xiaojiang Du, Chiu C. Tan, and Jie Wu. A multiversion programming inspired approach to detecting audio adversarial examples. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.
- [49] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

NLP-EYE: Detecting Memory Corruptions via Semantic-Aware Memory Operation Function Identification

Jianqiang Wang
wjq.sec@gmail.com
Shanghai Jiao Tong University

Siqi Ma (✉)
siqi.ma@csiro.au
CSIRO DATA61

Yuanyuan Zhang (✉)
yyjess@sjtu.edu.cn
Shanghai Jiao Tong University

Juanru Li (✉)
jarod@sjtu.edu.cn
Shanghai Jiao Tong University

Zheyu Ma
zheyuma@mail.nwpu.edu.cn
Northwestern Polytechnical University

Long Mai
root_mx@sjtu.edu.cn
Shanghai Jiao Tong University

Tiancheng Chen
sec_mxgc@sjtu.edu.cn
Shanghai Jiao Tong University

Dawu Gu
dwgu@sjtu.edu.cn
Shanghai Jiao Tong University

Abstract

Memory corruption vulnerabilities are serious threats to software security, which is often triggered by improper use of memory operation functions. The detection of memory corruptions relies on identifying memory operation functions and examining how it manipulates the memory. Distinguishing memory operation functions is challenging because they usually come in various forms in real-world software. In this paper, we propose NLP-EYE, an NLP-based memory corruption detection system. NLP-EYE is able to identify memory operation functions through a semantic-aware source code analysis automatically. It first creates a programming language friendly corpus in order to parse function prototypes. Based on the similarity comparison by utilizing both semantic and syntax information, NLP-EYE identifies and labels both standard and customized memory operation functions. It uses symbolic execution at last to check whether a memory operation causes incorrect memory usage.

Instead of analyzing data dependencies of the entire source code, NLP-EYE only focuses on memory operation parts. We evaluated the performance of NLP-EYE by using seven real-world libraries and programs, including Vim, Git, CPython, etc. NLP-EYE successfully identifies 27 null pointer dereference, two double-free and three use-after-free that are not discovered before in the latest versions of analysis targets.

1 Introduction

The memory-unsafe programming languages, such as C and C++, provide memory operation functions in the standard library (e.g., `malloc` and `free`) to allow manipulating the memories. During the development process, developers could implement dynamic memory operation functions by their own memory management policies to achieve higher performance, or by wrapping the standard memory operation functions with additional operations to fulfill other purposes (e.g., print debugging information).

Mistakes made by misusing the memory operations lead to well-seen memory corruption vulnerabilities such as buffer-

overflow and double-free in real-world software and their number is steadily increasing. For customized memory operation functions, some private memory operation functions are poorly implemented and thus carry some memory vulnerabilities at birth. On the other hand, developers keep making common mistakes, such as using the memory after it has been released (i.e., the use-after-free vulnerability), during the development process. Both cases aggravate the emerging of memory corruption vulnerabilities, which endow the attackers higher chance of compromising a computer system. A recent report of Microsoft demonstrated that around 70 percent of vulnerabilities in their products are memory safety issues [14].

To identify memory corruptions, various analysis methods using different kinds of techniques have been proposed. For instance, code similarity detection and information flow analysis are proposed to identify memory safety issues in source code [29] [44] [41]. Some tools such as AddressSanitizer [40], Dr. Memory [22] can also detect memory corruptions in binary code by instrumentation. These analyses require to abstract the usage of memory, and then extract certain patterns that are related to memory corruption. Otherwise, analyzing a program with millions of lines of code is inefficient and error-prone.

Customized memory operations could not help to decrease the chance of memory corruption at all, and moreover, the customized functions cause great difficulty in memory corruption analysis. Previous works, such as CRED [44], Pinpoint [41] and Dr. Memory [22], only consider the memory operation functions defined in the standard library. They are unable to identify customized memory operation functions, and thus disregard vulnerabilities caused by customized functions. Manual efforts can be involved to identify and label those functions, but it is exhausted and time consuming.

To address the above problems, we propose NLP-EYE, a source code-based security analysis system that adopts natural language processing (NLP) to detect memory corruptions. NLP-EYE will only parse the function prototypes instead of analyzing implementation of the functions. It then applies

symbolic execution to check whether the corresponding memory usages are correct. Unlike the other tools [1], the accuracy of NLP-EYE in memory operation function identification helps reduce the time cost by only analyzing partial code snippets and facilitate a better detection performance.

NLP-EYE reports typical memory corruption vulnerabilities, i.e., null pointer de-reference, double-free and user-after-free in seven open source software, such as Vim and Git. NLP-EYE has found 49 unknown vulnerabilities from their latest versions. For source code with more than 60 thousand of function prototypes, NLP-EYE is able to parse every ten thousand functions in one minute and finish the memory operation checking within an hour.

Contributions. Major contributions of this paper include:

- We proposed a source code-based analysis system that detects vulnerabilities by only analyzing a few function implementations, i.e., function prototypes and comments. Since these information are usually available, it is helpful for analysts and developers to build secure software with limited details.
- We implemented a vulnerability detection tool, NLP-EYE, that discovers memory corruption vulnerabilities effectively and efficiently. By combining NLP and symbolic execution, NLP-EYE labels both standard and customized memory operation functions and records states of the corresponding memory regions.
- We analyzed the latest versions of seven libraries and programs with NLP-EYE, and identified 49 unknown memory corruption vulnerabilities with 32 of them caused by customized memory operation functions. It demonstrates that the semantic-aware identification of NLP-EYE helps find new vulnerabilities that are unseen before.

Structure. The rest of the paper is organized as below: Section 2 lists the challenges of identifying memory corruptions caused by customized memory operation functions, and provide corresponding insights to solve these challenges. Section 3 details the design of NLP-EYE. In Section 4, we reported new vulnerabilities found by NLP-EYE, and illustrated the experiment results covering both vulnerability detection accuracy and performance comparison with the other tools. Section 5 discusses related works. We conclude this paper in Section 6.

2 Background

We give a concrete example of memory corruption vulnerability in Figure 1. Followed by that, we point out some challenges that hinders the detection of such vulnerabilities, and give corresponding insights to address those challenges.

2.1 Running Example

Detecting a memory corruption vulnerability (e.g., use-after-free) requires three significant steps: 1) identifying memory

```

1 //functions are provided by TTL module to operate dynamic memory
2 void TTLreleaseMem2Pool(Pool *pool, MemRegion p)
3 {
4     return pool->destroy_func(p);
5 }
6 MemRegion TTLretrieveMemFromPool(Pool *pool, size_t len)
7 {
8     return pool->alloc_func(len);
9 }
10
11 //memory pool used to provide dynamic memory region manipulation
12 extern Pool globalPool;
13
14 int main(int argc, char **argv, char **env)
15 {
16     char content[100];
17     scanf("%s", content);
18     char* buf = (char*)TTLretrieveMemFromPool(&globalPool, 1000);
19     int ret = processContent(content, buf);
20     if(!ret)
21     {
22         err("error occurs during process content!");
23         TTLreleaseMem2Pool(&globalPool, (MemRegion)buf);
24         goto clean;
25     }
26     ...
27     clean:
28         TTLreleaseMem2Pool(&globalPool, (MemRegion)buf);
29 }

```

Figure 1: Double-free vulnerability caused by the customized memory operation functions

operation functions and labeling dynamically allocated memory regions; 2) tracing the allocated memory regions to understand how they are operated; and 3) detecting incorrect operations on allocated memory regions. However, existing vulnerability detection techniques barely consider *customized memory operation functions*, and thus fail to detect vulnerabilities triggered by them.

The customized memory operation functions has caused the memory corruption vulnerability in Figure 1. Instead of using the standard memory operation functions provided by C standard library, functions `TTLretrieveMemFromPool` and `TTLreleaseMem2Pool` are used to allocate a dynamic memory (Line 18) and release the corresponding allocated memory (Line 23), respectively. While executing, `TTLreleaseMem2Pool` releases the memory if the function `processContent` returns a null value (Line 20); then, a duplicate release (Line 28) causes a double-free vulnerability. Consider this double-free vulnerability, it cannot be detected by simply analyzing standard memory operation procedures because of the customized memory operation functions (i.e., `TTLreleaseMem2Pool` and `TTLretrieveMemFromPool`).¹

Generally, whether a function is a memory operation function, we can observe whether it calls C standard library memory operating functions, or compare the similarity with other memory operation function implementations. In either case, it requires the function implementation which is usually not

¹Actually we have applied typical tools such as `Cppcheck` [2] and `VisualCodeGrepper` [18] to detect the vulnerability in this sample and found that none of them could detect this vulnerability.

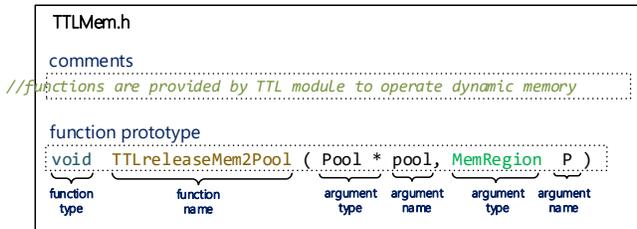


Figure 2: A function prototype with comments

available. For example, the declared memory operation function, `alloc_func()` (Line 8) might be implemented externally and only its binary is available. Under such circumstance, the semantic information in a function prototype (i.e., function declaration) becomes the only reference for the memory operation function identification.

As Figure 2 depicts, a function prototype consists of a function type, a function name, argument types for arguments, and (optionally) names of arguments. While defining a function prototype, developers prefer to use meaningful function name and proper data types for this function. Besides, developers may add comments to describe in more details.

In most cases, function prototypes and comments help us to determine the semantics without knowing function implementations. Therefore, we can analyze prototype structures to retrieve meanings of those words.

2.2 Challenges

Most challenges lie in understanding the function semantics and identify memory operation functions accurately.

Challenge I: Irregular Representations. Searching for specific words in the source code is the common strategy to identify functions, such as locating the keyword *memory* to identify memory operation functions. While plenty of abbreviations and informal terms are used in function prototypes, it is difficult to extract the semantic information effectively by only applying a keyword-based searching strategy.

Consider the function prototype `TTLreleaseMem2Pool` in Figure 1. An abbreviation `Mem2` represents *memory to* in an informal way. The abbreviation `Mem` is unable to be located by using the word *memory*, and the number `2` makes it harder to understand the semantics of the phrase.

Challenge II: Ambiguous Word Explanations. Since the context collected from function prototypes is insufficient, it makes the semantics extraction more challenging. Although some function prototypes may use the same word, the actual function semantics can be different because of their various naming formats.

Considering two function names, `PyObject_Malloc` and `_PyObject_DebugMallocStats`, in the source code of CPython², the former function is for allocating a dynamic memory while the latter one is for outputting debugging information of memory allocator.

²CPython is the reference implementation of Python.

Even though the lexical analysis with a specific dictionary can help split the word *malloc* from those two function names, the corresponding function semantics cannot be inferred precisely. For the function `PyObject_Malloc`, the word *malloc* does represent memory allocation; however in function `PyObject_DebugMallocStats`, *malloc* is used to qualify the object, that is *Stats*, to illustrate the status of the memory allocator. Therefore, we need not only analyze the meaning but also the format of the word to construct the function prototype.

Challenge III: Diverse Type Declaration. Diversified data types declaration in C/C++ programming makes it harder to compare two function prototypes. For instance, both `short` and `unsigned short int`, are used to represent the Integer type. Besides, C/C++ has provided a type re-define feature (i.e., `typedef`) that programmers can shorten the name of a complex type.

2.3 Insights

Fortunately, function prototypes are constructed by following some certain formats in programming. We utilize these formats to extract the semantic information.

Adaptive Lexical Analysis. Irregular representations make the function prototype segmentation even harder. A natural language corpus is not suitable for word segmentation in computer programming. Thus, we construct an adaptive corpus to address the problem of the lexical analysis in Challenge I. The corpus consists of natural language used in computer science, common keywords in the programming language (e.g., `proc`, `ttl`) and comments in the source code. Common keywords reveal the words that are often used in programming, and comments in the source code suggest some semantic information of a function.

Grammar-free Comparison. By examining the function prototypes manually, we observe that developers do not usually follow English grammars when naming a function. However, they still use similar words (e.g., *get*, *acquire*, *alloc*) with similar grammatical order (i.e., the order of words), such as `AcquireVirtualMemory` and `getMemfromPool`. We then propose a grammar-free analysis, which performs an NLP-based comparison, to solve Challenge II.

To identify the semantic information of each function prototype, we create a set of reference functions (e.g., standard memory operation functions), whose semantics are known. Then, we compare the function name and argument names of each function prototype with the corresponding names in reference functions. If the similarity between a function prototype and a reference function is higher than a threshold, we label this function prototype as a potential memory operation function, and proceed with the type comparison to confirm.

Various Types Clustering. NLP-based comparison only helps decide whether a function prototype is a potential memory operation function. We design a type comparison scheme to handle its declared return type and argument types. Because of the diversity of function types, we first normalize

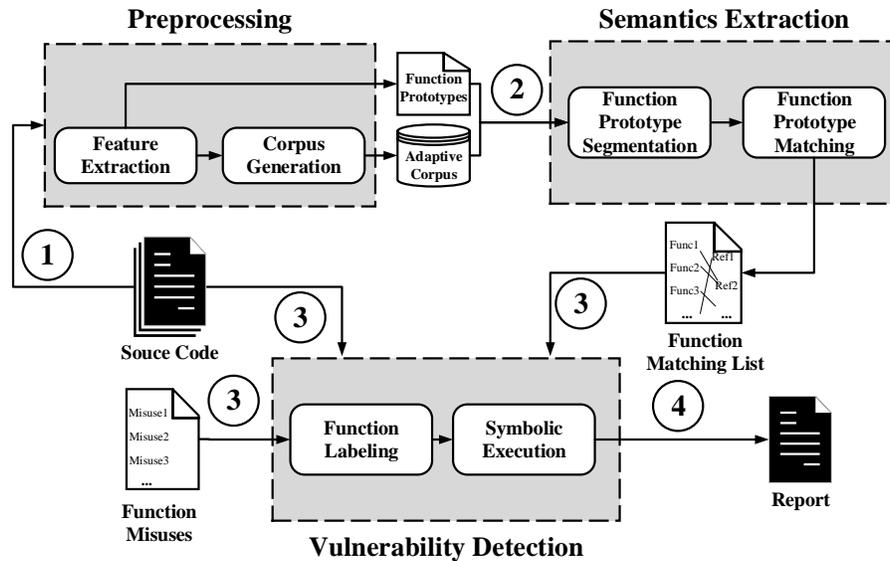


Figure 3: System overview of NLP-EYE

those types in aliases (e.g., types defined by `typedef`) by using their original forms, which solve Challenge III. Having the pair of a function prototype and its matched reference function, we then compare their return types and argument types. We assume a function prototype as a memory operation function if both names and types are matched.

3 Design of NLP-EYE

We propose NLP-EYE, a source code analysis system that utilizes NLP to retrofit the process of the memory corruption vulnerability detection. There are three phases: **preprocessing**, **semantics extraction**, and **vulnerability detection** in NLP-EYE. Figure 3 illustrates the overview of NLP-EYE. It takes source code files as inputs, i.e., the analysis target. The preprocessing phase extracts function prototypes and comments to generate an adaptive corpus. The semantics extraction phase uses the adaptive corpus to build a matching list by collecting all the possible memory operation functions in the analysis target. Vulnerability detection phase labels memory operation functions in the target and feeds it to the symbolic execution to facilitate the vulnerability detection. We introduce the working details of each phase below.

3.1 Preprocessing

NLP-EYE takes a batch of source code as inputs and generates *function prototypes* and an *adaptive corpus* to perform adaptive lexical analysis. First, NLP-EYE extracts function prototypes and comments from source code. Then, it comprises comments with the other two corpuses to construct an adaptive corpus. Details are presented below.

3.1.1 Feature Extraction

Feature extraction component of NLP-EYE is built on top of Clang Static Analyzer plugin [1], which provides an interface for users to scan the declaration of each function. Given the source code, NLP-EYE uses this plugin to extract all function prototypes in the format of "Type@Name", including those functions that are imported from other libraries. For comments from source code, NLP-EYE uses regular expressions to match comment symbols in C language.

3.1.2 Corpus Generation

After collecting those comments, NLP-EYE constructs an adaptive corpus to perform adaptive lexical analysis. The adaptive corpus includes three parts, that are Google Web Trillion Word Corpus (GWTWC) [7], MSDN library API names [21] [20], and comments from source code.

The GWTWC is a popular corpus created by Google, containing more than one trillion words extracted from public web pages. It can be applied to identify common words used in natural languages. With the help of MSDN library API names and comments, NLP-EYE can process programming languages. The MSDN library provides normalized APIs in Camel-Case format. Therefore, it is easy to divide each function name into words/abbreviations through capital letters. For example, function `GetProcAddress` can be divided into ["Get", "Proc", "Address"]. While processing comments from source code, NLP-EYE first filters the symbol characters (e.g., `#%!`), and then splits text by applying regular expressions. Numbers and words appeared in GWTWC are excluded.

Since abbreviations are commonly used in programming, we set the appearance frequency of MSDN APIs to be higher than the appearance frequency of comments, to provide them a higher priority. We further assume that a word, who is a

substring of another word in MSDN API names, should have a lower frequency than its parent word. For example, `arm` in function `mallocWithAlarm` is a substring of `Alarm`, obviously `Alarm` is to be regarded as a whole; then we assign a lower frequency for `arm` than for `Alarm`.

3.2 Semantics Extraction

NLP-EYE compares each function prototype with a set of *reference functions* (e.g., `malloc`, `free`), and generates a *function matching list*. When a match was found in the function matching list, we can infer the semantics from the function prototype that it has the similar semantics with the reference function.

NLP-EYE processes the data type and the function name, arguments name separately to identify memory operation functions in two steps. First, it divides the function name and arguments name into a serial of words. Next, it performs NLP-based comparison to select the potential function prototypes with memory operation functionalities and confirm the results by applying type comparison.

3.2.1 Function Prototype Segmentation

To proceed function prototype segmentation, NLP-EYE applies Segaran *et al.*'s word segmentation algorithm [39] to select the *segmentation list* with the highest *list frequency*.

Given a function prototype (*FP*), with n letters, NLP-EYE first creates 2^{n-1} possible combinations of these letters and constructs 2^{n-1} segmentation lists. Each segmentation list reserves the original order of these letters appeared in the *FP*. NLP-EYE then computes the list frequency for each segmentation list. It compares each word (w_i) in a segmentation list (*SL*) with words in the adaptive corpus, and returns the following list frequency (*LF*):

$$LF = \frac{|SL|}{\prod_{i=1} freq(w_i)}$$

where $|SL|$ represents how many words are contained in *SL*, and $freq(w_i)$ is the frequency of w_i in the adaptive corpus. Finally, NLP-EYE considers the segmentation list with the highest list frequency as its segmentation result.

3.2.2 Function Prototype Matching

Due to the diversity of type declaration, NLP-EYE processes names (i.e., function names and argument names) and types (i.e., return types and argument types) separately. It performs NLP-based comparison to identify those names that are related to memory operation functionalities. NLP-EYE then applies type comparison to determine memory operation functions and generates a function matching list.

NLP-based Comparison. Natural language processing (NLP) has been widely used to identify the connection between two words for semantic similarity matching. To measure the word similarity, a *context corpus* is required to extract

the taxonomy information. Words in the context corpus are then represented by sets of vectors in Word2vec [34] model. The cosine distance between two words positively related to their semantic similarity, and a higher cosine distance represents a higher similarity between two words.

To extract the semantic meaning of an unknown name, we generates a set of reference functions manually, which contains standard memory operation functions provided by C/C++ and other known memory operation functions. Having those reference functions, NLP-EYE compares the name of an unknown function with the names of the reference functions and calculate their similarity scores. If a similarity score is higher than a *threshold*, NLP-EYE labels this unknown function as *similar* to the reference function, that is, the corresponding function is a potential memory operation function.

We address function names and argument names individually, since the comparison results of function names and argument names may interfere each other while applying the NLP-based comparison. Consider a function with only abbreviations for function names, but complete words for argument names, its similarity score may not achieve the threshold. Although the similarity score of the argument names is the highest, the total similarity score will be impacted by the low similarity of function names. Therefore, we set different similarity threshold, *fn-similarity* and *arg-similarity*, as the threshold of function names and argument names, respectively. Only when *fn-similarity* and *arg-similarity* are both satisfied, NLP-EYE will label the function. For function arguments, NLP-EYE compares each argument of the reference function with every argument of the target function and generates similarity score. Then NLP-EYE chooses the most similar one as the corresponding arguments regardless of the number of arguments.

Type Comparison. Given the potential memory operation functions and their matched reference functions, NLP-EYE compares their data types correspondingly. We use Clang Static Analyzer to classify the data types into several categories to address the type diversity.

First, NLP-EYE normalizes data types. Some data types are re-defined as aliases by `typedef`. Thus, NLP-EYE uses the original data types to replace those aliases. Second, we define some coarse grained categories based on the basic data types in C programming. NLP-EYE finally suggests the correct category for each data type. For example, `unsigned int` and `signed short` are assigned to the category of `Integer`. `void *` and `char *` belong to the category of `pointer`.

We compare the return type and corresponding argument types of the potential memory operation function with data types of the matched reference function. If their types are assigned to the same category, the unknown function is a memory operation function, and it is assumed to have the same semantics as the corresponding reference function. Each pair of a function prototype and its matched reference function is inserted to the function matching list.

3.3 Vulnerability Detection

NLP-EYE creates a vulnerability report for each source code by comparing the usages of memory operation functions with the pre-defined *function misuses*. NLP-EYE first labels memory operation functions in the source code; then, NLP-EYE checks whether there exists any function misuse.

3.3.1 Function Labeling

NLP-EYE takes the function matching list as an inputs to identify memory operation functions. It compares functions in the source code with the functions in the function matching list. If a function appears in the function matching list, NLP-EYE labels this function as a memory operation function.

3.3.2 Symbolic Execution

The code, that can be compiled independently, is regarded as an unit. NLP-EYE first generates the call graph for each unit and then executes each unit from top to bottom one by one by adopting symbolic execution.

The output of semantics extraction is a function matching list which maps the standard memory operation functions and its corresponding customized memory operation functions. Given this function matching list, NLP-EYE dynamically instruments stubs before function calls memory operation and memory access points in advance to record and revise memory region states. NLP-EYE identifies memory operation function calls by simply comparing the called function name and the function names in the function matching list. The stubs are extra code snippets that are executed before the symbolic execution engine measuring the instrumented statements. We manually made up a coarse function misuse list which contains general function misuse implementations, such as a memory region can not be released more than once and a memory region can not be accessed after being released. Given this list, once symbolic execution reaches any memory access point or any function call site of a memory operation function, NLP-EYE executes the instrumented stub and checks misuses. If it meets a misuse, NLP-EYE will report this misuse as a vulnerability. Otherwise, the corresponding memory state will be updated (i.e., allocated or released) based on the function call of the memory operation function. For instance, the source code in figure 1, NLP-EYE instruments before line 18, 23 and 28 since the called functions are identified as memory operation functions. Then during symbolic execution, NLP-EYE records that a memory region is allocated in line 18 and released in line 23. When symbolic execution reaches line 28, it recognizes that a memory region (i.e., buf) is to be released twice which is one of the given function misuses, therefore NLP-EYE reports a double-free vulnerability.

	Lines of code	# of functions	# of memory operation functions
Vim-8.1 [17]	468,133	16,012	73
ImageMagick-7.0.8-15 [9]	514,472	14,636	79
CPython-3.8.0a0 [3]	556,950	12,000	66
Git-2.21.0 [4]	289,532	8,788	32
GraphicsMagick-1.3.31 [8]	369,569	7,406	29
GnuTLS-3.6.5 [6]	488,654	5,433	11
LibTIFF-4.0.10 [11]	85,791	1,326	4
Total	2,773,101	65,601	294

Table 1: Lines of code, number of functions and number of memory operation functions collected from each library/program.

4 Evaluation

In this section, we report the results of four experiments. The first experiment assesses the performance of function prototype segmentation. The second demonstrates the accuracy of NLP-EYE while identifying memory operation functions, and whether the context corpus has any impact on the identification accuracy. The third experiment looks into the vulnerability detection ability of NLP-EYE, and the last experiment discusses its runtime performance.

4.1 Experiment Setup

Dataset. We collected the latest version of seven popular open source libraries and programs written in C/C++ programming language with a total of 65,601 functions by December 2018 (see Table 1 for more details).

Due to the lack of open source labeled memory operation functions, we created our benchmarks. For identifying memory operation functions, we asked a team of annotators (3 programmers), all with more than seven years of programming experience in C/C++ to examine the implementations of memory operation functions. We first required team members to label memory operation functions independently, and then all members checked the results together. If there were any function with different labels, team members would discuss an agreement to label this function before it could be included in the dataset. In this procedure, we found 294 memory operation functions in total.

Implementation. We evaluated NLP-EYE on a Ubuntu 16.04 x64 workstation with an Intel Core i7-6700 CPU (four cores, 3.40 GHz) and 64 GB RAM. For the function prototype segmentation, we used NLTK [32], a natural language processing toolkit, to create the adaptive corpus for segmentation. We used the WordSegment [15] module in Python to split function prototypes. Gensim [37] is set up for NLP-based comparison, which conducts the similarity comparison based on the context corpus. Finally, we adopted Clang Static Analyzer [1] to perform type comparison and symbolic execution. Clang Static Analyzer is a source code analysis tool which adopts symbolic execution to analyze each translation unit. It provides a framework that developers can intercept the symbolic execution process at specific points such as function call and memory access. In addition, Clang Static Analyzer provide

useful programming interfaces that can be used by developers to interact with the data type.

4.1.1 Experiment Design

To evaluate the effectiveness and efficiency of NLP-EYE, we present the designed four experiments in details below.

EX1 (Prototype Segmentation). To evaluate the effectiveness of prototype segmentation, we measured the Levenshtein-inspired distance [45] [38] of the segmentation results as our evaluation metrics. The distance between two segmentation lists i and j of string s is given by d_{ij} , which can be calculated as:

$$d_{ij} = \sum_{k=1}^{|s|-1} (vec_i[k] \ xor \ vec_j[k])$$

where $|s|$ represents the length of string s . Segmentation lists i and j are converted into vectors, vec_i and vec_j . In each vector, zero is regarded as “without split”, and one is “split”.

For the Levenshtein-inspired distance, a lower distance with the correct one indicates that the segmentation list requires fewer edit operations (i.e., split and merge) to be adjusted to the correct one. Thus, a lower distance specifies a better segmentation result.

EX2 (Memory Operation Function Identification). NLP-EYE identifies memory operation functions by using NLP-based comparison and type comparison. We evaluated the function identification performance by using precision, recall and F-measure as the evaluation metrics.

EX3 (Vulnerability Detection). We targeted on typical memory corruption vulnerabilities in this paper, i.e., double-free, use-after-free, and null pointer de-reference against real world software products such as Vim and CPython. To evaluate the effectiveness of NLP-EYE, we further compared it with the other four vulnerability detection tools (MallocChecker [13], Cppcheck [2], Infer [10] and SVF [42]), and counted the number of vulnerabilities that are correctly detected.

EX4 (Runtime Performance). We evaluated the average time cost of each phase in NLP-EYE, including preprocessing, semantics extraction, and vulnerability detection.

4.2 Ex1: Prototype Segmentation

Before we start, we manually split the function names we collected as the ground truth. We counted the number of function names that are correctly segmented, and then calculated the Levenshtein-inspired distance to evaluate the performance of each segmentation. Further, we compared the segmentation results that are generated by the adaptive corpus of NLP-EYE with the corresponding results generated by Google Web Trillion Word Corpus (GWTWC). It assesses the result accuracy while applying the adaptive corpus.

NLP-EYE correctly segments 230 out of 350 function names. Levenshtein-inspired distances of those function names are zero. Figure 4 demonstrated the average distance of each library and program by using the adaptive

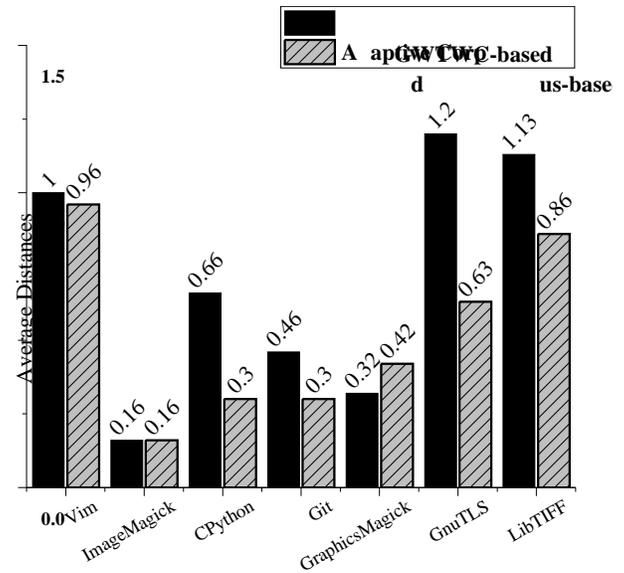


Figure 4: Segmentation results of function names by using NLP-EYE and GWTWC

corpus of NLP-EYE and GWTWC. NLP-EYE segments function names of Vim, ImageMagick, CPython, Git, GraphicsMagick, GnuTLS and LibTIFF accurately with the Levenshtein-inspired distance as 0.96, 0.16, 0.3, 0.3, 0.42, 0.63 and 0.86 respectively. The results for LibTIFF and Vim are worse than the others, because lots of function names involve single letters, and NLP-EYE cannot distinguish those letters from a word.

Except for GraphicsMagick and ImageMagick, the adaptive corpus-based segmentation performs better than the GWTWC-based segmentation. According to our manual inspection, we found that GWTWC is not a programming language-based corpus and it cannot proceed programming abbreviations. Thus, most of its segmentation results are worse than the results of the adaptive corpus-based segmentation. However, this conclusion is not satisfied on GraphicsMagick, because some function names are incorrectly divided into abbreviations by the adaptive corpus. Taken a function name preview as an example, it is divided into [“pre”, “view”] instead of “preview”, because the frequencies of those two abbreviations are higher in comments. For the ImageMagick, most function names are declared in normalized words, which are easy for GWTWC and the adaptive corpus to distinguish each word.

4.3 Ex2: Memory Operation Function Identification

We counted the number of memory operation functions that are correctly detected by NLP-EYE and computed the precision, recall, and F-measure on the entire dataset. To conduct this experiment, we separately set the thresholds (fn-similarity and arg-similarity) as (0.3, 0.4, 0.5) for function names and argument names, and found that NLP-EYE performs the best when fn-similarity and arg-similarity are 0.4 and 0.5, respectively.

	# of identified functions	# of correctly identified functions	Precision	Recall	F-measure
Vim	304	42	13%	57%	21%
ImageMagick	137	44	32%	55%	40%
CPython	131	48	36%	72%	48%
Git	46	8	17%	25%	20%
GraphicsMagick	69	16	23%	55%	32%
GnuTLS	74	4	5%	36%	8%
LibTIFF	8	0	0	0	0
Total	769	162	21%	55%	30%

Table 2: Memory operation function identification results of NLP-EYE

	NPD		DF		UAF	
	Detected	Confirmed	Detected	Confirmed	Detected	Confirmed
Vim	17	17	2	1	8	2
CPython	10	4	1	1	8	1
Git	1	1	0	0	0	0
GraphicsMagick	6	5	0	0	0	0
Total	34	27	3	2	16	3

Table 3: Detection results of null pointer de-reference (NPD), double-free (DF) and use-after-free (UAF). Note that this result only shows the vulnerabilities caused by customized memory operation functions.

Function Identification Results. We applied the StackOverflow corpus for NLP-based comparison. All the posts from the StackOverflow forum [16] are included in the StackOverflow corpus. Table 2 shows the best identification result with the number of identified functions and the number of memory operation functions that are correctly identified. We also computed precision, recall, and F-measure of NLP-EYE. NLP-EYE correctly identifies 162 memory operation functions out of the 769 identified functions, with precision, recall, F-measure value of 21%, 55%, and 30%, respectively. For LibTIFF, NLP-EYE cannot detect any memory operation functions because many single letters are used to name a function argument. For example, “s” is commonly used to express “size” that causes the recognition of memory operation functions even harder if the thresholds are too high. We then determine a balance between the thresholds (i.e., fn-similarity and arg-similarity) and the identification accuracy.

Within millions of functions, NLP-EYE narrows down the number of functions that need to be analyzed, and the total number of functions for manual analysis is acceptable. Furthermore, the false positive and the false negative are reasonable.

Context Corpus Selection. We further applied NLP-based comparison on two extra context corpuses (i.e., Wikipedia corpus, and customized corpus) to assess the identification performance. The Wikipedia corpus contains all webpages from Wikipedia [19]. Alternatively, the customized corpus consists of: 1) Linux man pages [12]; 2) Part of GNU Manuals [5]; and 3) two programming tutorials, i.e., C++ Primer [31] and C Primer Plus [36].

Based on the Wikipedia corpus, NLP-EYE only identifies no more than ten memory operation functions in each library and program with a precision value of 7%, and a worse recall value. While using customized corpus as the context corpus, the precision and recall of NLP-EYE are 42% and 19%, respectively. Although its precision is acceptable, it still causes too many false negatives. By manually analyzing the results, we found that Wikipedia corpus is insensitive to the programming language, and most identified functions are unrelated to memory operation. For the customized corpus, it fails to identify functions that use abbreviations, which cause exceptions if words are not found in the corpus.

4.4 Ex3: Vulnerability Detection

We tested NLP-EYE on the seven libraries and programs to examine whether there is any unknown memory corruption vulnerability. Note that the seven collected libraries and programs are the latest versions (collected in December 2018).

Vulnerabilities Detected by NLP-EYE. NLP-EYE detects 49 vulnerabilities from these libraries and programs in total. While only considering vulnerabilities caused by customized memory operation functions, four libraries and programs are involved. The detection result is shown in Table 3. By manually verifying these results, NLP-EYE successfully detects 32 vulnerabilities, including 27 null pointer de-reference, two double-free, and three use-after-free, existed in customized memory operation functions. To further verify the correctness of our results, we reported the manual-confirmed vulnerabilities to developers, and they have confirmed and patched ten

	NLP-EYE	MallocChecker	Cppcheck	Infer	SVF
Vim	3.82	2.77	18.90	51.28	50.92
ImageMagick	6.16	5.00	28.00	64.25	0.25
Cpython	8.31	7.70	1.47	23.43	0.26
Git	3.11	2.80	0.88	13.52	2.36
GraphicsMagick	2.08	1.45	11.83	8.75	0.15
GntTLS	2.75	2.33	9.65	11.13	0.11
LibTIFF	0.91	0.87	0.93	3.55	0.04
Total	27.14	22.92	71.66	175.91	54.09

Table 4: Runtime performance comparison (minutes)

null pointer de-reference and all the double-free, use-after-free vulnerabilities. Each customized memory operation function may cause vulnerabilities, since NLP-EYE failed to identify a part of them, this may lead to a false negative of vulnerability detection result. Besides the successfully detected vulnerabilities, NLP-EYE made false positive as well listed in Table 3. However, after we manually inspected the false positive, we found that none of them are caused by the wrong identification result.

There are two reasons that cause the false positive: 1) symbolic execution engine proceeds the expression with indexes in a loop as a static expression. For instance, the engine may report a double free on an array with different index in a loop since the engine regard the array element with different index as the same value; 2) While processing a conditional statement with a complex logic, the symbolic execution engine executes every path without considering the constraints defined in the conditional statements.

Detection Effectiveness Assessment. To assess the detection effectiveness of NLP-EYE, we applied four detection tools, MallocChecker, Cppcheck, Infer and SVF, to the entire dataset for comparison. MallocChecker and Infer claim to detect all three kinds of vulnerabilities. Cppcheck and SVF are designed to detect vulnerabilities of use-after-free and double-free. For the null pointer de-reference vulnerability, MallocChecker and Infer correctly reported 11 and 30 vulnerabilities, respectively. However, they can only report those misuses caused by standard memory allocation functions, while NLP-EYE can detect both standard and customized memory allocation functions. Even worse, none of these tools can detect vulnerabilities of use-after-free and double-free correctly.

We analyzed false positives caused by these tools. Similar to NLP-EYE, the symbolic execution engine of MallocChecker cannot identify the index of an array in a loop. Although Cppcheck can detect use-after-free vulnerabilities, it became inaccurate when lots of variables are declared to operate dynamic memories. Infer checks all returned pointers, which cause many false positives. It even reported a use-after-free vulnerability existed in an integer statement. SVF performed the worst by reporting hundreds of double-free vulnerabilities, which causes lots of errors.

4.5 Ex4: Runtime Performance

We evaluated the time cost of each phase (i.e., preprocessing, semantics extraction, and vulnerability detection) of NLP-EYE. Additionally, we tested the runtime of the other detection tools to assess the efficiency of NLP-EYE.

Before vulnerability detection, we collected all the posts on StackOverflow forum with the size of 17GB to create the context corpus, and it costs 56 hours to generate the model file. This step processes only once because we can repeatedly use the context corpus in further analysis.

Table 4 shows the total runtime cost of NLP-EYE and the other tools while analyzing our dataset. NLP-EYE preprocesses each library and program, and constructs the corresponding adaptive corpus within one seconds. It further spends 36.601s on average to identify memory operation functions in each library and program. NLP-EYE spends 70.917s on ImageMagick, but no more than 6s on LibTIFF, because ImageMagick has 14,636 functions and LibTIFF only includes 1,326 functions.

By comparing with the other tools, the runtime performance of NLP-EYE and MallocChecker are similar, since they use the same symbolic execution engine. SVF sacrifices the detection accuracy to achieve a higher runtime performance. Unfortunately, it is unhelpful for programmers to pinpoint vulnerabilities. Cppcheck and Infer analyze the entire source code to ensure a complete coverage, which costs much time.

4.6 Limitations

NLP-EYE successfully detects some memory corruption vulnerabilities other tools cannot detect. The results of function identification and vulnerability detection indicate that NLP-EYE understands the function semantics well with only limited information. However, we still have the following limitations that cause detection failures.

1. When a function implementation is complex, the symbolic execution engine in NLP-EYE cannot correctly analyze the data flow and control flow.
2. NLP-EYE cannot handle single letters involved in the function prototypes which may causes false positive and false negative.

```

File1: GraphicsMagick/magick/memory.c
1 MagickExport void * MagickMalloc(const size_t size){
2   if (size == 0)
3     return ((void *) NULL);
4   MEMORY_LIMIT_CHECK(GetCurrentFunction(),size);
5   return (MallocFunc)(size);
6 }

File2: GraphicsMagick/coders/pdb.c
1 static Image *ReadPDBImage(const ImageInfo
2 *image_info,ExceptionInfo *exception){
3   ...
4   comment=MagickAllocateMemory(char *,length+1);
5   p=comment;
6   p[0]='\0';
7   ...
8 }

```

Figure 5: A null pointer de-reference vulnerability in GraphicsMagick

4.7 Case Study

We discuss two representative vulnerabilities found in the GraphicsMagick library and the CPython interpreter, respectively.

GraphicsMagick is a library that was derived from the ImageMagick image processing utility in November 2002. GraphicsMagick is securely designed and implemented after being tested by Memcheck and Helgrind³. Also, AddressSanitizer (ASAN) [40], the most mature redzone-based memory error detector, proves it to be secure against memory errors. Nevertheless, NLP-EYE detects six null pointer de-reference vulnerabilities from its latest version. An example is presented in Figure 5. The function `MagickAllocateMemory` is declared to allocate memories. If the dynamic memory is insufficient and a null pointer is returned by this function (Line 4 of File2), a segmentation fault will be triggered (Line 6 of File2).

To detect this vulnerability, a detector should recognize the customized memory allocation function `MagickAllocateMemory`, which is a macro definition of the `MagickMalloc` function. For `MagickMalloc`, its implementation is defined in File1, and a customized memory allocation function `MallocFunc` is declared in this function. Besides analyzing the standard memory operation functions, NLP-EYE first identifies the macro definition, `MagickAllocateMemory` in Line4 of File2, and uses its original function `MagicMalloc` in File1 to replace it. By proceeding the preprocessing and semantics extraction phases, NLP-EYE labels those functions as memory operation functions, and finally locates function misuses. In comparison, other detection tools (e.g., `MallocChecker`) cannot distinguish those customized functions (i.e., `MallocFunc`, `MagicMalloc`, and `MagickAllocateMemory`), and thus fail to detect the flaw.

³Memcheck is a memory error detector for C and C++ programs. Helgrind is a tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives. They are all based on Valgrind [35]

```

File1: CPython/Objects/obmalloc.c
1 void PyMem_Free(void *ptr){
2   _PyMem.free(_PyMem.ctx, ptr);
3 }

File2: CPython/Modules/_randommodule.c
1 static PyObject *
2 random_seed(RandomObject *self, PyObject *args){
3   ...
4   res = _PyLong_AsByteArray((PyLongObject *)n,
5                             (unsigned char *)key,
6                             keyused * 4,
7                             PY_LITTLE_ENDIAN,
8                             0); /* unsigned */
9   if (res == -1) {
10      PyMem_Free(key);
11      goto Done;
12   }
13   ...
14   Done:
15      PyMem_Free(key);
16      return result;
17 }

```

Figure 6: A double-free vulnerability in CPython

Another sample code snippet with double-free vulnerability is shown in Figure 6, which is detected from CPython interpreter. Apparently, function `PyMem_Free` in File1 is a memory de-allocation function. If the variable `res` is -1, the variable `key` will be freed twice (Line 10 and 15 of File2, respectively). To our surprise, this simple vulnerability was found neither by manual audit nor automated source code analysis. According to the feedback of CPython developers, the corresponding host function has been tested for many times, but the vulnerability still exists. Based on this feedback, we would say that identifying customized memory operation functions is suitable to memory corruption detection. NLP-EYE is very helpful in this scenario.

5 Related Work

There are prior efforts of vulnerability detection, in this section, we introduce these works based on their analysis approaches, i.e., source code-based analysis and binary code-based analysis.

5.1 Source Code-based Analysis

Previous studies detect vulnerabilities by applying program analysis on source code to extract pointer information [41] [44] and data dependencies [33], [24], [29], [28].

To analyze C programming source code, CRED [44] detects use-after-free vulnerabilities in C programs. It extracts pointer information by applying a path-sensitive demand-driven approach. To decrease false alarms, it uses spatio-temporal context reduction technique to construct use-after-free pairs

precisely. However, the pairing part is time consuming that every path in the source code is required to be analyzed and memorized. Instead of analyzing the entire source code, Pinpoint [41] applies sparse value-flow analysis to identify vulnerabilities in C programs, such as use-after-free, double-free. To reduce the cost of data dependency analysis, Pinpoint analyzes local data dependence first and then performs symbolic execution to memorize the non-local data dependency and path conditions.

Similar to the above, some other tools detect vulnerabilities by compare data-flows with some pre-defined rules/violations. CBMC [28] is a C bounded model checker, which examines safety of the assertions under a given bound. It translates assertions and loops into a formula. If this formula satisfies any pre-defined violations, then a violated assertion will be identified. Coccinelle [29] finds specific bug by comparing the code with a given pattern written in Semantic Patch Language (SmPL).

Source code-based analysis has also been applied to Linux kernel. Due to the large amount of kernel code in Linux, DR. CHECKER [33] and K-Miner [24] are designed to be more effective and efficiency. DR. CHECKER employs a soundy approach based on program analysis. It is capable of conducting large-scale analysis and detecting numerous classes of bugs in Linux kernel drivers. K-Miner finds vulnerabilities by setting up a virtual kernel environment and processing syscalls separately.

Those proposed tools perform well to detect vulnerabilities implemented under standard programming styles, such as calling standard library APIs, designing standard implementation steps. They cannot proceed those customized functions just like how NLP-EYE does.

Instead of applying program analysis, both VulPecker [30] and VUDDY [27] detects vulnerabilities based on the code similarity. VulPecker builds a vulnerability database by using diff hunk features collected from each vulnerable code and its corresponding patch code. VUDDY proceeds each vulnerable function as an unit, and then abstracts and normalizes vulnerable functions to ensure that they are able to detect clones with modifications. However, similarity-based techniques require a massive database that can be learnt from.

5.2 Binary Code-based Analysis

Instead of analyzing source code, binary code can also be adopted to identify memory corruption vulnerabilities on stacks and allocated memories [22], [35], [40], [26], [43], [25], [23].

Memory shadowing helps to track the memory status at runtime. It also causes large memory consumption. Dr.Memory [22] conducts memory checking on Windows and Linux. It uses memory shadowing to track the memory status and identifies stack usage within heap memory. Dr.Memory is flexible and lightweight by using an encoding for callstacks to reduce memory consumption. AddressSanitizer [40] minimizes the memory consumption by creating a compact shadow mem-

ory, which achieves a 128-to-1 mapping. By implementing a specialized memory allocator and code instrumentation in the compiler, AddressSanitizer analyzes the vulnerabilities on stack, heap, global variables. HOPTracer [26] discovers heap overflow vulnerabilities by examining whether a heap access operation can be controlled by an attacker. HOPTracer finds vulnerabilities by giving an accurate definition to buffer overflow and it uses a heuristic method to find memory allocation functions. HOPTracer is able to identify memory allocation functions with a higher accuracy, and several unknown overflow vulnerabilities are detected.

Unfortunately, detecting memory corruptions through binary code-based analysis requires proper inputs, that can precisely trigger the corresponding memory operation. It might cause some false negatives because of the incomplete code coverage.

6 Conclusion

We propose an NLP-based automated approach to detect memory corruption vulnerabilities. A detection tool, NLP-EYE, is developed to identify vulnerabilities of null pointer dereference, use-after-free, double free. The novelty of our approach is that we retrieve the function semantics accurately based on a little function information, i.e., function prototypes and comments, instead of using the entire function implementations. With the help of NLP-based and type-based analyses, NLP-EYE identifies memory operation functions accurately. Our approach is also adaptable since NLP-EYE generates an adaptive corpus for different dataset by extracting their comments from source code and various programming styles.

In this work, we only focused on memory corruption vulnerabilities. We plan to extend NLP-EYE in future with additional reference functions to identify the other vulnerabilities. We also open source NLP-EYE to help analysts and developers to improve software security.

7 Acknowledgments

The authors would like to thank the anonymous reviewers for their feedback and our shepherd, Dongpeng Xu, for his valuable comments to help improving this paper.

This work was supported by the General Program of National Natural Science Foundation of China (Grant No.61872237), the Key Program of National Natural Science Foundation of China (Grant No.U1636217) and the National Key Research and Development Program of China (Grant No.2016YFB0801200).

We especially thank Huawei Technologies, Inc. for the research grant that supported this work, Ant Financial Services Group for the support of this research within the *SJTU-AntFinancial joint Institute of FinTech Security*, and Nanjing Turing Artificial Intelligence Institute with the internship program.

References

- [1] Clang Static Analyzer. <http://clang-analyzer.lvm.org>.
- [2] Cppcheck. <http://cppcheck.sourceforge.net>.
- [3] CPython. <https://www.python.org>.
- [4] Git. <https://git-scm.com>.
- [5] GNU Manuals Online. <https://www.gnu.org/manual/manual.en.html>.
- [6] GnuTLS. <https://www.gnutls.org>.
- [7] Google Web Trillion Word Corpus. <http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html>.
- [8] GraphicsMagick. <http://www.graphicsmagick.org>.
- [9] ImageMagick. <https://www.imagemagick.org>.
- [10] Infer. <https://fbinfer.com>.
- [11] LibTIFF. <http://www.libtiff.org>.
- [12] Linux man pages online. <http://man7.org/linux/man-pages/index.html>.
- [13] MallocChecker. <https://clang-analyzer.lvm.org/>.
- [14] Microsoft: 70 percent of all security bugs are memory safety issues. <https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/>.
- [15] Python Wordsegment. <https://pypi.org/project/wordsegment/>.
- [16] Stackoverflow. <https://stackoverflow.com>.
- [17] Vim. <https://www.vim.org>.
- [18] VisualCodeGrepper. <https://github.com/nccgroup/VCG>.
- [19] Wikipedia. <https://www.wikipedia.org>.
- [20] Windows 8 APIs References. <https://docs.microsoft.com/en-us/windows/desktop/apiindex/windows-8-api-sets>.
- [21] Windows Driver API references. <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/>.
- [22] Bruening, Derek and Zhao, Qin. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2011.
- [23] Dinakar Dhurjati and Vikram Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2006.
- [24] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering Memory Corruption in Linux. In *Proceedings of 2018 Network and Distributed System Security Symposium (NDSS)*, 2018.
- [25] Niranjana Hasabnis, Ashish Misra, and R Sekar. Lightweight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, 2012.
- [26] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. Towards Efficient Heap Overflow Discovery. In *Proceedings of 26th USENIX Security Symposium USENIX Security (USENIX Security)*, 2017.
- [27] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of 2017 IEEE Symposium on Security and Privacy (SP)*, 2017.
- [28] Daniel Kroening and Michael Tautschnig. CBMC—C Bounded Model Checker. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014.
- [29] Julia Lawall, Ben Laurie, Ren’e Rydhof Hansen, Nicolas Palix, and Gilles Muller. Finding Error Handling Bugs in Openssl Using Coccinelle. In *Proceedings of 2010 European Dependable Computing Conference (EDCC)*, 2010.
- [30] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. VulPecker: an Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, 2016.
- [31] Stanley B. Lippman. *C++ Primer*. 2012.
- [32] Edward Loper and Steven Bird. NLTK: the Natural Language Toolkit. *arXiv preprint cs/0205028*, 2002.
- [33] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR.CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *Proceedings of 26th USENIX Security Symposium USENIX Security (USENIX Security)*, 2017.
- [34] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of word representations in Vector Space. *arXiv preprint arXiv:1301.3781*, 2013.
- [35] Nicholas Nethercote and Julian Seward. Valgrind: a Framework for Heavyweight Dynamic binary instrumentation. In *Proceedings of ACM Sigplan notices*, 2007.
- [36] Stephen Prata. *C Primer Plus*. 2014.

- [37] Radim Rehurek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks (LREC)*, 2010.
- [38] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.
- [39] Toby Segaran and Jeff Hammerbacher. *Beautiful Data: the Stories Behind Elegant Data Solutions*. 2009.
- [40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*, 2012.
- [41] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.
- [42] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction (CC)*, 2016.
- [43] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangan: Scalable use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.
- [44] 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 *Proceedings of 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.
- [45] Li Yujian and Liu Bo. A Normalized Levenshtein Distance Metric. *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, 2007.

Robust Optimization-Based Watermarking Scheme for Sequential Data

Erman Ayday

Case Western Reserve University

Cleveland, OH, USA

and

Bilkent University, Turkey

Emre Yilmaz

Case Western Reserve University

Cleveland, OH, USA

Arif Yilmaz

Bilkent University, Turkey

Abstract

In this work, we address the liability issues that may arise due to unauthorized sharing of personal data. We consider a scenario in which an individual shares their sequential data (such as genomic data or location patterns) with several service providers (SPs). In such a scenario, if their data is shared with other third parties without their consent, the individual wants to determine the service provider that is responsible for this unauthorized sharing. To provide this functionality, we propose a novel optimization-based watermarking scheme for sharing of sequential data. The proposed scheme guarantees with a high probability that (i) the malicious SP that receives the data cannot understand the watermarked data points, (ii) when more than one malicious SPs aggregate their data, they still cannot determine the watermarked data points, (iii) even if the unauthorized sharing involves only a portion of the original data or modified data (to damage the watermark), the corresponding malicious SP can be kept responsible for the leakage, and (iv) the added watermark is compliant with the nature of the corresponding data. That is, if there are inherent correlations in the data, the added watermark still preserves such correlations. The proposed scheme also minimizes the utility loss due to changing certain parts of the data while it provides the aforementioned security guarantees. Furthermore, we conduct a case study of the proposed scheme on genomic data and show the security and utility guarantees of the proposed scheme.

1 Introduction

Sequential data includes time-series data such as location patterns, stock market data, speech, or ordered data such as genomic data. Individuals share different types of sequential data for several purposes, typically to receive personalized services from online service providers (SPs). Data collected and processed by these SPs may reveal privacy sensitive information about individuals. Thus, the way these SPs handle the collected data poses a threat to individuals' privacy and it is crucial for individuals to have control on how their data is collected and handled by the SPs.

When an individual shares their personal data with an SP for a particular purpose, they want to make sure that their data will not be observed by other third parties. Privacy leakage occurs when personal data of individuals is further shared by an SP with other third parties (e.g., for financial benefit). To deter the SPs from such unauthorized sharing, it is required to develop technical solutions that would keep them liable for such unauthorized sharing (e.g., by connecting the unauthorized sharing to its source). One well-known tool for such scenarios is watermarking. An individual may add a unique watermark into their data before sharing it with each SP, and if their data is further shared without their authorization, they can associate the unauthorized sharing to the corresponding SP.

Watermarking is a well-known technique to address the liability issues especially for multimedia data [18]. Using the high amount of redundancy in the data and the fact that human eye cannot differentiate slight differences between the pixel values, watermark is inserted into multimedia data by changing some pixel values. However, watermarking is not a straightforward technique for sequential data such as location patterns or genomic data. To insert watermark into sequential data, original data should be modified according to the watermark which reduces the quality of service provided by the SPs. Thus, watermarking sequential data while preserving data utility has unique challenges.

Another challenge for watermarking sequential data is the identifiability (or robustness) of the watermark. An individual cannot identify the SP that is responsible for the data leakage if the SP finds the watermark inserted data points and removes (or tampers with) the watermark before the unauthorized sharing. Furthermore, an SP may partially share the data (rather than sharing the whole data of the individual) or modify the data (to damage the watermark). These make it harder for the individual to identify the source of the leakage.

An SP may utilize different types of auxiliary information in order to determine (and hence tamper with) the watermark in the data. The most common type of such auxiliary information may be the inherent correlations in the data. Location

patterns are correlated in both time and space [9]. Similarly, genomic data carries inherent correlations (referred as linkage disequilibrium) inside. A malicious SP may also use external auxiliary information that is correlated with the data (e.g., phenotype or kinship information for genomic data or inter-dependent check-in information for location data). Thus, an SP can identify the watermarked data points by identifying the points that violate the expected correlations in the data. One other type of auxiliary information is the data shared by the individual with other SPs. Multiple SPs may collect the same sequential data from the same individual (with different watermark patterns) and they may compare their collected data in order to identify the watermarked points with higher probability. Thus, a watermarking algorithm for sequential data should be robust against these kinds of threats.

To address these robustness and utility challenges, we propose a novel watermarking scheme to share sequential data. Initially, we assume the data has no correlations and propose an algorithm to determine the data points to be watermarked by solving a non-linear optimization problem. This algorithm is developed to be robust against collusion of malicious SPs. Then, we explain how to deal with correlated data in the proposed algorithm. Hence, we minimize the risk of correlation attacks by malicious SPs who know the pairwise correlations between data points. We evaluate the security (robustness) and the utility of the proposed algorithm on a genomic dataset. The main motivations to choose genomic data sharing as the use case are as follows: (i) genomic data includes privacy-sensitive information such as predisposition to diseases [19], (ii) it is not revokable, and hence it is crucial to make sure that it is not leaked, and (iii) it has inherent correlations that makes watermarking even more challenging.

The main contributions of the proposed work are summarized as follows:

- We propose a novel collusion-secure watermarking scheme for sequential data. The proposed scheme minimizes the probability for the identifiability of the watermark by the SPs. We show that even when multiple SPs join their data together or they use the knowledge of inherent correlations in the data the watermark cannot be identified (with a high probability).
- We show that the SPs that are responsible for the unauthorized sharing can be detected with a high probability even when they share a portion of the data or when they modify the data in order to damage the watermark. We also show relationship between the probabilistic limits of this detection and the shared portion of data.
- While providing these security (or robustness) guarantees, the proposed system also minimizes the utility loss in the sequential data due to watermarking.
- We also implement and evaluate the proposed scheme for genomic data sharing.

The rest of the paper is organized as follows. In the next section, we discuss the related work. In Section 3, we introduce the data model, the system model, and the threat model. In Section 4, we provide the details of the proposed solution. In Section 5, we evaluate the security of the proposed watermarking algorithm. In Section 6, we discuss potential extensions of the proposed scheme and possible future research directions. Finally, in Section 7, we conclude the paper.

2 Related Work

Digital watermarking is the act of hiding a message related to a digital signal (e.g., an image, song, or video) within the signal itself [7]. While digital watermarks are typically used for copyright and copy protection [4, 6, 17, 18], they are also used in different applications such as broadcast monitoring [15], transaction tracking [8], and content authentication [27]. Digital watermarks are prominently used for copy protection and copy deterrence on multimedia content. Multimedia watermarking schemes [14] benefit from the high redundancy in the data and they do not consider sophisticated attacks against the watermarking scheme. Since the amount of redundancy is not typically high in non-media data, it is more challenging to add watermark into such data. The watermarking techniques proposed for non-media such as text [12] and graphs [26] cannot be applied to sequential data because they do not consider robustness of the watermark against various types of attacks (that are discussed in Section 3.3).

Several works proposed watermarking techniques for sequential data such as time-series data and spatiotemporal data. Kozat et al. proposed a technique for hiding sensitive metadata such as SSN or date-of-birth into electrocardiograms (ECG) in order to authenticate the ownership of data without distorting important ECG characteristics [13]. In addition, Panah et al. [24] introduced a low complexity watermarking scheme for tamper-proofing of ECG signals at the sensory nodes. Watermarking spatiotemporal data is also challenging due to low redundancy of data and works in this area are mostly focused on watermarking trajectory datasets that include trajectories of multiple objects [11, 16, 25]. However, we consider the case in which an individual wants to share their individual data with multiple SPs after watermarking. This is a more challenging problem since the redundancy in the shared data is much lower. In general, neither of these schemes consider the correlations in data nor the possibility of colluding SPs.

Boneh and Shaw proposed a general fingerprinting (watermarking) solution that is robust against collusion [5]. Their scheme constructs fingerprints in such a way that no coalition of attackers can find a fingerprint. However, there are still some practical drawbacks of this scheme. First, fingerprint length may be very long to guarantee robustness against collusion, which reduce the utility of the data. Furthermore, the scheme does not consider complex attacks against the watermarking algorithms such as the ones using auxiliary

x_1, \dots, x_ℓ	Set of ordered data points
d_1, \dots, d_m	Possible values (states) of a data point
I_i	Index set of the data points that are shared with the SP i
D_{I_i}	Set of data points in I_i
W_{I_i}	Set of data points in I_i after watermarking
Z_{I_i}	Set of watermarked data points in W_{I_i}

Table 1: Frequently used symbols and notations.

information or the ones tampering the watermark and it does not consider the correlations in the data. We address these drawbacks in our proposed scheme.

3 Problem Definition

Here, we describe the data model, system model, and the threat model. Frequently used symbols and notations are presented in Table 1.

3.1 Data Model

Sequential data consists of ordered data points x_1, \dots, x_ℓ , where ℓ is the length of the data. The value of a data point x_i can be in different states from the set $\{d_1, \dots, d_m\}$ according to the type of the data. For instance, x_i can be coordinate pairs in terms of latitude and longitude for location data, it can be location semantics (e.g., cafe or restaurant) for check-in data, or it can be the value of a nucleotide or point mutation for genomic data.

We approach the problem for two general sequential data types: (i) sequential data with no correlations in which data points are independent and identically distributed. In this type, value of a data point cannot be predicted using the values of other data points. Sparse check-in data might be a good example for this type. And, (ii) sequential data with correlations between the data points. Correlation between data points may vary based on the type of data. For example, consecutive data points that are collected with small differences in time may be correlated in location patterns. That is, an individual’s location at time t can be estimated if their locations at time $(t - 1)$ and/or $(t + 1)$ are known. In genomic data, point mutations (e.g., single nucleotide polymorphisms or SNPs¹) may have pairwise correlations between each other. Such pairwise correlations are referred as linkage disequilibrium [23] and they are not necessarily between consecutive data points. The correlation value may differ based on the state of each data point and correlation between the data points is typically asymmetric. Furthermore, it has been shown that correlations in human genome can also be of higher order [22]. For the clarity of the presentation, we first build our solution for uncorrelated sequential data and then extend it for correlated data.

¹We provide a brief background on genomics in Section 5.

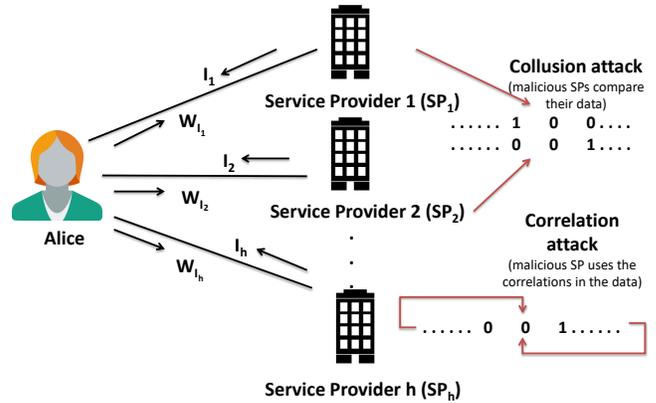


Figure 1: Overview of the system and threat models.

3.2 System Model

We consider a system between a data owner (Alice) and multiple service providers (SPs) as shown in Figure 1. For genomic data, the SP can be a medical institution, a genetic researcher, or direct-to-customer service provider. For location data, the SP can be any location-based service provider. In the description of the scheme, for clarity, we explain some parts of the algorithm on binary data but the proposed scheme can be extended for non-binary data. In fact, for the evaluation of the proposed scheme, we focus on the point mutations in genomic data that may have values from $\{0, 1, 2\}$. Alice shares parts of her data with the SPs to receive different types of services. Note that the part Alice shares with each SP may be different and we do not need same data to be shared with each SP. When we talk about the collusion attack (as will be detailed in the next section), we consider the intersection of the data parts owned by all malicious SPs.

On one hand, when Alice shares her data with an SP, she wants to make sure that her data will not be shared with other third parties by the corresponding SP. In the case of further unauthorized sharing, she wants to know the SP that is responsible from this leak. Therefore, whenever Alice shares her data with a different SP, she inserts a unique watermark into it. On the other hand, an SP may share Alice’s data with third parties without the consent of Alice. While doing so, to avoid being detected, the SP wants to detect and remove the watermark from the data. Instead of sharing the whole data with a third party, an SP may also share a certain portion of Alice’s data to reduce the risk of detection (but compromising from the shared data amount). Similarly, malicious SP (or SPs) may try to damage the watermark by modifying the data. Furthermore, two or more SPs may join their data to detect the watermarked points. Security of the watermarking scheme increases (against the attacks discussed in the next section) as the length of the watermark increases. However, a long watermark causes significant modification on the original data, and hence decreases the utility of the shared data. In our proposed scheme, utility loss in the data is minimized while

the watermarking scheme is still robust against the potential attacks with high probability.

3.3 Threat Model

Different types of attacks are defined for image or text watermarking such as elimination attack, collusion attack, masking attack, insertion/deletion attack, and reordering attack [12,21]. We consider the following attacks on the proposed watermarking scheme by adapting previously defined attacks to sequential data and defining new attacks for sequential data such as correlation attack.

Single SP attack on uncorrelated data. Assume that Alice shares her (uncorrelated) sequential data of length ℓ with an SP and she includes a watermark of length w into this data. Since data is uncorrelated, each data point is independent from other, and hence for each data point, the SP infers the probability of being watermarked as w/ℓ . Instead of trying to detect the watermark, the malicious SP may also tamper with the data in order to damage the watermark.

Correlation attack. If an SP has correlated data points and it also knows the corresponding correlation values, it may identify the watermarked points with higher probability. To be general, we assume pairwise, asymmetric correlations between different states of data points. The proposed scheme can be extended to other scenarios (e.g., higher order correlations or symmetric correlations) similarly. For instance, if d_α state of x_i (i.e., $x_i = d_\alpha$) is correlated with d_β state of x_j (i.e., $x_j = d_\beta$), then $Pr(x_i = d_\alpha | x_j = d_\beta)$ is high, but the opposite does not need to hold. Note that d_α state of x_i may be in pairwise correlation with other data points as well. We consider all possible pairwise correlations between different states of all data points in our analysis. Following this example, assume the SP has one of the correlated data points as $x_j = d_\beta$, but $x_i = d_\gamma$ (where $d_\gamma \neq d_\alpha$). Then, the SP can conclude that x_i is watermarked with probability $p(x_i^w) = Pr(x_i = d_\alpha | x_j = d_\beta)$. If d_α state of x_i is also correlated with other data points (that the SP can observe), then the SP computes the watermark probability on x_i as the maximum of these probabilities. Similarly, d_γ state of x_i may also be correlated with other data points. Since $x_i = d_\gamma$, such correlations imply that data point x_i is not watermarked. Using such correlations, the SP also computes the probability that x_i is not watermarked, $p(x_i^f)$. Eventually, the SP computes the probability of data point x_i being watermarked as $(p(x_i^w) - p(x_i^f))$. Once the SP determines the probability of being watermarked for each data point, it sorts them based on the computed probabilities, and identifies the w watermarked data points as the ones with the highest probabilities. We assume that the SP knows the watermarking algorithm, and hence the length of the watermark (w). Thus, the SP may choose w data points corresponding to the w highest probabilities to infer the watermarked data points in the shared data.

Collusion attack. Multiple SPs that receive the same data portion (from the same data owner) with different watermark

patterns may join their data to identify the watermarked points with higher probability. In such a scenario, when the SPs vertically align their data points, they will observe some data points with different states. Such data points will definitely be marked as watermarked data points by the SPs. Collusion attack may also benefit from the correlation attack. Malicious SPs may first run the collusion attack to identify some watermarked points and then they may individually run the correlation attack to infer the watermark pattern. We also evaluate the robustness of the proposed scheme against such an attack. Malicious SPs may also try to modify the data in order to damage the watermark.

3.4 Watermark Robustness

“Robustness” and “security” terms have been used interchangeably for watermarking schemes in different works. For text watermarking, robustness of a watermarking scheme is defined as strength of the technique to resist attacks that aim to retrieve or modify hidden data [12]. For image watermarking, Nyeem et al. define robustness as the ability to withstand any distortions and they define security as the ability to resist any hostile attacks that try to circumvent the system or to destroy the watermark’s purpose(s) [20]. Same authors formalize the robustness for image watermarking by defining three levels of robustness such as robust, fragile, and semi-fragile by considering detection ability after distortion [21]. Adelsbach et al. provide formal definitions for watermark robustness [3]. Different from our work, in [3], authors consider watermarking mechanisms that use a secret embedding key (that is used when adding watermark to the data). They define watermark robustness as the information of the watermark that is revealed to the adversary and watermark security as the information revealed about the secret embedding key. Inspired from [3], we come up with the following robustness definitions for watermarking sequential data.

Robustness against watermark inference. This property states that watermark should not be inferred by the malicious SP (or SPs) via the aforementioned attack models. In the proposed scheme, inferring the watermark does not rely on a computationally hard problem; malicious SP (or SPs) probabilistically infer the watermark. Thus, we evaluate the proposed scheme for this property in terms of malicious SPs’ (or SP’s) inference probability for the added watermark. We provide the following definition to evaluate the robustness of a watermarking scheme against watermark inference.

Definition 1 *p-robustness against f-watermark inference.* A watermarking scheme is *p-robust against f-watermark inference* if probability of inferring at least *f* fraction of the watermark pattern ($0 \leq f \leq 1$) is smaller than *p*.

Robustness against watermark modification. This property states that the malicious SP (or SPs) should not be able to modify the watermark in such a way that the watermark detection algorithm of the data owner misclassifies the source

of the unauthorized data leakage. We evaluate the proposed scheme for this attribute in terms of precision and recall of the data owner to detect the malicious SP (or SPs) that leak their data. For this, we define “false positive” as watermark detection algorithm classifying a non-malicious SP as a malicious one and “false negative” as watermark detection algorithm classifying a malicious SP as a non-malicious one. We provide the following definition to evaluate the robustness of a watermarking scheme against watermark modification.

Definition 2 ρ/ζ -robustness against watermark modification. A watermarking scheme is ρ/ζ -robust against watermark modification if malicious SP (or SPs), by modifying the watermark, cannot decrease the precision and recall of the watermark detection algorithm below ρ and ζ , respectively.

For all the aforementioned attack models, we evaluate the proposed watermarking scheme based on its robustness. In Section 5, we show the limits of the proposed scheme for these definitions considering different variables.

4 Proposed Solution

Here, first we present an overview of the proposed protocol and then describe the details of the proposed watermarking algorithm. When Alice wants to share her data with an SP i , they engage in the following protocol. The SP i sends the indices of Alice’s data it requests, denoted by I_i . Alice generates $D_{I_i} = \bigcup_{i \in I_i} x_i$. Alice finds the data points to be watermarked considering her previous sharings of her data. This part is done using our proposed watermarking algorithm as described in detail in this section. Alice inserts watermark into the data points in D_{I_i} and generates the watermarked data W_{I_i} . Alice stores the ID of the SP and Z_{I_i} (watermark pattern for the SP i). Alice sends W_{I_i} to SP i .

The proposed watermarking algorithm describes the selection of data points to be watermarked in the sequential data so that the watermark will be secure against the attacks discussed in Section 3.3. We insert watermark into a data point by changing this data point’s state. For instance, if data is binary, this change is from 0 to 1, or vice versa. If each data point can have states from the set $\{d_1, \dots, d_m\}$, the change is from the current state to some other state d_j^* . In Section 4.1, since we assume there is no correlation in data, a data point x_i is changed to a state d_j^* uniformly at random. However, due to the correlation in data, the new state d_j^* of a data point x_i is determined to minimize the probability of correlation attack in Section 4.2. In the following, we first detail our solution for sequential data that has no correlations (data points are independent from each other) and then, we describe how to extend our solution for correlated sequential data.

4.1 Watermarking Sequential Data without Correlations

Before giving the details of the proposed algorithm, we first provide the following notations to facilitate the discussion.

- n_i^h : number of data points that are watermarked i times when the whole data is shared with h SPs.
- \hat{y}_i^h : number of data points that are watermarked i times when the whole data is shared h times and will not be watermarked in the $(h + 1)$ -th sharing.
- y_i^h : number of data points that are watermarked i times when the whole data is shared h times and will be watermarked in the $(h + 1)$ -th sharing.

When Alice shares her data with a new SP, first, watermark locations in the data are determined for the new request according to the watermark patterns in previously shared data and with the goal of minimizing the success of the collusion attack. From these definitions, it is obvious that $n_i^h = \hat{y}_i^h + y_i^h$, which means that among the n_i^h data points that are watermarked i times after h sharings, y_i^h of them will be shared in the $(h + 1)$ -th sharing and the remaining \hat{y}_i^h of them will not be shared in the $(h + 1)$ -th sharing. Therefore, the proposed algorithm computes y_i^h and \hat{y}_i^h values to minimize the probability of collusion attack when Alice shares her data with $(h + 1)$ -th SP. After computing these values any y_i^h of n_i^h data points that are watermarked i times can be selected to insert the watermark since data points are not correlated.

As discussed, a malicious SP may increase its probability to find the watermark inserted data points by colluding with other malicious SPs that received the same data from Alice with different watermark patterns. For simplicity, assume that each data point’s state can be either 0 or 1 and h malicious SPs have the same data portion (belonging to Alice) with different watermark patterns. They vertically align their data portions, compare their data, and find the differences. For instance, for a data point x_i , they observe k 0s and $(h - k)$ 1s (where $0 \leq k \leq h$) and they conclude that the corresponding data point has been watermarked either k or $(h - k)$ times. We assume that the proposed watermarking algorithm is also known by the malicious SPs. Therefore, these h SPs may run our proposed algorithm (as discussed next) and find n_k^h and n_{h-k}^h values. Once they have these values, they may conclude that (i) the corresponding data point has been watermarked k times with probability $n_k^h / (n_k^h + n_{h-k}^h)$, and (ii) $(h - k)$ times with probability $n_{h-k}^h / (n_k^h + n_{h-k}^h)$. In our algorithm, watermarks are inserted into the watermark locations that minimizes the probability of identifying the whole watermark patterns of all malicious SPs when they collude. To do so, we propose solving an optimization problem to determine the data points to be watermarked at each data sharing instance of Alice. The objective function of this problem for the $(h + 1)$ -th sharing can be formulated as follows:

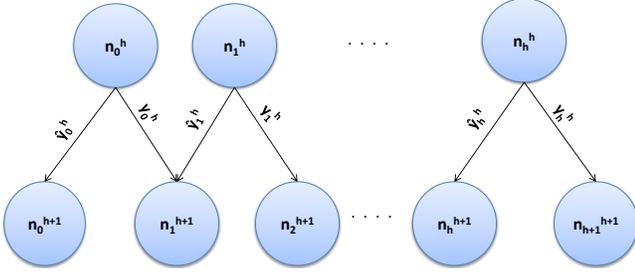


Figure 2: Relationship between n_i^h , n_i^{h+1} , y_i^h , and \hat{y}_i^h values in the watermark insertion scheme.

$$\min \prod_{i=0}^{h+1} \left(\frac{n_i^{h+1}}{n_i^{h+1} + n_{h-i+1}^{h+1}} \right)^{n_i^{h+1}} \quad (1)$$

with the following constraints: (i) $\sum_{i=0}^{h+1} y_i^h = w$, (ii) $n_0^{h+1} = \hat{y}_0^h$, (iii) $n_{h+1}^{h+1} = y_h^h$, (iv) $n_i^{h+1} = y_{i-1}^h + \hat{y}_i^h$ for $i = 1, \dots, h$, (v) $\hat{y}_i^h + y_i^h = n_i^h$, (vi) $y_i^h, \hat{y}_i^h \geq 0$, and (vii) $y_0^h > 0$. Here, constraint (i) determines the number of data points that we watermark. That is, the algorithm does not modify more data points than the limit defined in this constraint. Constraints (ii), (iii), (iv), and (v) denote the relationship between n_i^h , n_i^{h+1} , y_i^h , and \hat{y}_i^h . In Figure 2, we show this relationship. Constraint (vi) is used to prevent negative y_i^h and \hat{y}_i^h values. Finally, constraint (vii) is to make sure that each SP has a unique watermark pattern. As the solution of this optimization problem, we obtain the y_i^h and \hat{y}_i^h values. As mentioned before, for the $(h+1)$ -th sharing, the algorithm selects any y_i^h of the data points that are watermarked i times after h sharings.

4.2 Addressing Correlations in the Data

By solving the optimization problem in Section 4.1, we obtain the y_i^h and \hat{y}_i^h values. Since this time data is correlated, watermarks should be inserted in such a way that no malicious SP can understand the watermark inserted data points by checking the validity of the correlations. To guarantee this, if a data point x_i 's state is changed from d_α to d_β (due to added watermark), the states of other data points that are correlated with d_β state of x_i should be also changed (since we assume asymmetric correlations). Assume data has been shared for h times before. Watermark insertion algorithm for the $(h+1)$ -th sharing of the data with SP ψ is summarized as follows.

From the solution of the optimization problem, we know the number of data points which are watermarked i times and will be watermarked in the current sharing (y_i^h). Since a data point could be watermarked between 0 and h times, we have the solution set of the optimization problem as $Y = \{y_0^h, y_1^h, \dots, y_h^h\}$. Data points to be shared with SP ψ are $D_{T_\psi} = \{x_1, \dots, x_\ell\}$ and the states of a data point are from the set $\{d_1, d_2, \dots, d_m\}$. To add watermarks into data points that are watermarked for t times ($t = 0, 1, \dots, h$) in the previous h sharings, we find the set of t times watermarked data points (T_t) and sort them

in ascending order according to their presence probabilities. Presence probability can be found as follows. Assume d_j state of data point x_j is correlated with the set of data points in $C = \{x_{i_0} = d_{i_0}, \dots, x_{i_n} = d_{i_n}\}$. Then, the presence probability for $(x_j = d_j)$ is computed as $\prod_{i=0}^n Pr(x_j = d_j | x_{i_i} = d_{i_i})$.

Then, for each t value from 0 to h , we get the data point with minimum presence probability (x_j) in T_t , determine the state (d_j^*) that maximizes its presence probability, and change the state of x_j . This way, we choose the most likely state value for x_j according to the whole data. If the state of x_j is already d_j^* , we skip this data point and continue with the next data point with minimum presence probability. Since we change a data point that is watermarked for t times, we also decrement the value of $Y[t]$ (y_t^h) by 1. After the state of x_j is changed to d_j^* , we find the data points that are correlated with d_j^* state of x_j . That is, we construct a set C with data points that satisfy $Pr(x_i | x_j = d_j^*) > \tau$ and change the states of the data points in C . For each data point in C , we find its "desired state" (i.e., correlated state with d_j^* state of x_j) and change it. During this process, if we change a data point that is watermarked for t^* times, we also decrement the value of $Y[t^*]$ ($y_{t^*}^h$) by 1. We continue this process until we add w watermarks to the data.

In this algorithm, we consider pairwise correlations between the data points. When correlations between the data points are more complex (e.g., higher order), we can still use a similar algorithm to handle them. We assume that malicious SPs also have the same resources we use in this algorithm to use the correlations (in order to detect the watermarked points) and evaluate the scheme accordingly in Section 5.

5 Evaluation

We implemented the proposed watermarking scheme on genomic data and evaluated its security (robustness) and utility guarantees. To solve the proposed non-linear optimization problem, we used the APMonitor Optimization Suite [10]. In this section, we provide the details of the data model we used in our evaluation and our results.

5.1 Data Model

For the evaluation, we used single-nucleotide polymorphism (SNP) data on the DNA. The human genome consists of approximately three billion letters (A, T, C, or G). Even though more than 99% of these letters are identical in any two individuals, there are differences between us due to genetic variations. SNP is the most common DNA variation in human population. A SNP is a position in the genome holding a nucleotide, which varies between individuals [2]. For example, in Figure 3, two sequenced DNA fragments from two different individuals contain a single different nucleotide at a particular SNP position. In general, there are two types of alleles (nucleotides) observed at a given SNP position: (i) the major allele is the most frequently observed nucleotide, and (ii) the minor allele is the rare nucleotide. For instance, the two alleles for the SNP position in Figure 3 are C and T (G

and A in the figure are the complementary nucleotides for C and T, respectively). Almost all common SNPs have only two alleles, and everyone inherits one allele of every SNP position from each of their parents. If an individual receives the same allele from both parents, they are said to be homozygous for that SNP position. If however, they inherit a different allele from each parent (one minor and one major), they are called heterozygous. Depending on the alleles the individual inherits from their parents, the state (or value) of a SNP position can be simply represented as the number of minor alleles it possesses, i.e., 0, 1, or 2. SNPs may have pairwise correlations between each other. Such pairwise correlations are referred as linkage disequilibrium [23].

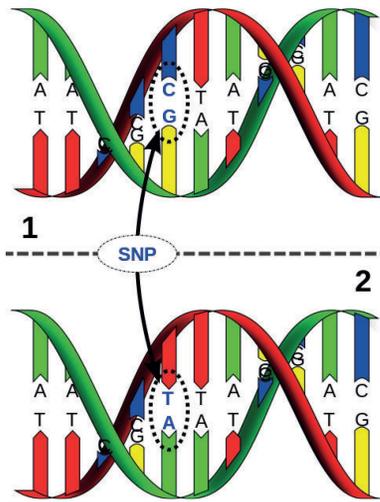


Figure 3: Single nucleotide polymorphism (SNP) with alleles C and T (©David Hall, License: Creative Commons).

We obtained SNP data of 99 individuals from 1000 Genomes Project [1]. In the obtained dataset, each individual has 7690 SNP values meaning that we have a 99 by 7690 matrix and elements of matrix are either 0, 1, or 2. We used this dataset to learn the statistics (e.g., correlations between the SNPs) that are used in the proposed algorithm. Thus, the size of this dataset is not an indicator for the scalability of the proposed algorithm.

5.2 Experimental Results

We evaluated the proposed watermarking scheme in various aspects. In particular, we evaluated its security (robustness) against watermark inference and watermark modification (Section 3.4). Robustness against watermark inference is evaluated by running collision and correlation attacks (as discussed in Section 3.3). In all collusion attack scenarios, we assume that Alice shares the same data portion with the SPs. This assumption provides the maximum amount of information to the malicious SPs. If different set of data points are shared with the SPs, malicious SPs can use the intersection of

these data points for the collusion attack. Robustness against watermark modification is evaluated under various attacks in terms of the (watermark) detection performance of the data owner. The results also include evaluation of the loss in data utility due to watermark addition. We ran all experiments for 1000 times and report the average values. We denote the fraction of watermarked data (or watermark ratio) as $r = w/\ell$. Watermark ratio r also represents the utility loss in the shared data due to the added watermark.

5.2.1 Robustness against watermark inference

Here, we evaluate the robustness of the proposed scheme against watermark inference under collision and correlation attacks.

Collusion attack: First, we evaluated the probability of identifying the whole watermarked points in the collusion attack (when correlations in data are not considered). We considered the worst case scenario and assumed that all the SPs that has Alice’s data are malicious, and hence they exactly know how many times Alice has shared her data to compute the exact probabilities for the attack (as discussed in Section 4.1). In Figure 4, we show the logarithm of this inference probability when data is shared with h SPs and they are all malicious (where $h = (1, 2, \dots, 10)$) and when different fractions of data is watermarked. Overall, we observed that the probability to completely identify the watermark via the collusion attack is significantly low when the proposed technique is used for watermarking the data. Following our definition of robustness against watermark inference (in Section 3.4), under this attack model, the proposed scheme is p -robust against f -watermark inference for $f = 1$ and $p \leq 10^{-2}$ when h is as high as 10 and data utility is as high as 97% (i.e., r is as small as 0.025). As expected, we observed that the inference probability of the malicious SPs increases with decreasing r and increasing h values. That is, as data is shared with more malicious SPs, the probability to identify the watermarked data increases due to the collusion attack. Also note that even for significantly low values of r (that corresponds to high data utility), the proposed scheme provides high resiliency against collusion attacks.

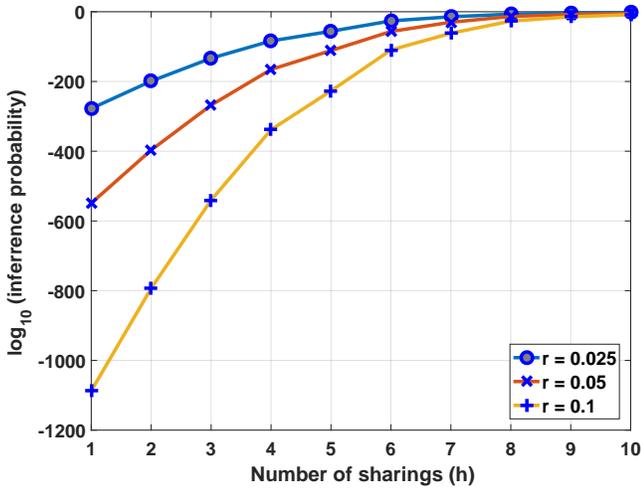


Figure 4: Probability of identifying the whole watermarked points in the collusion attack when h malicious SPs collude. r represents the fraction of watermarked data.

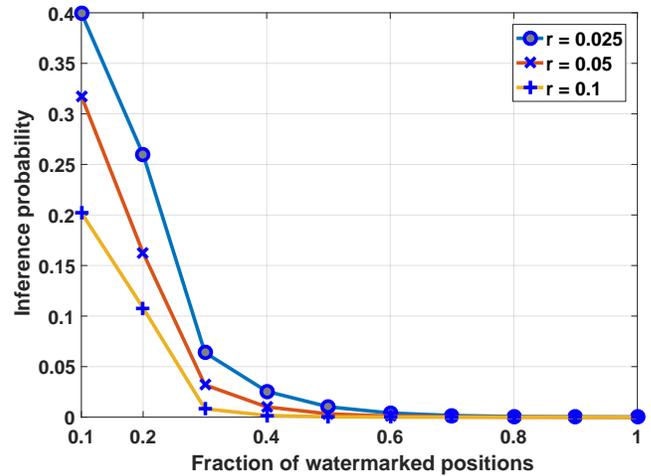
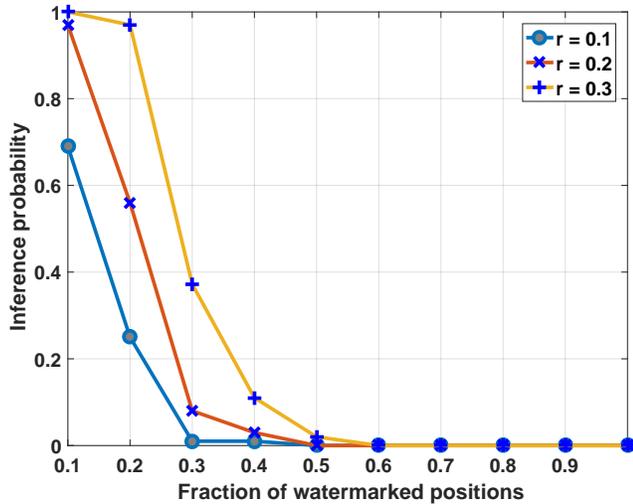


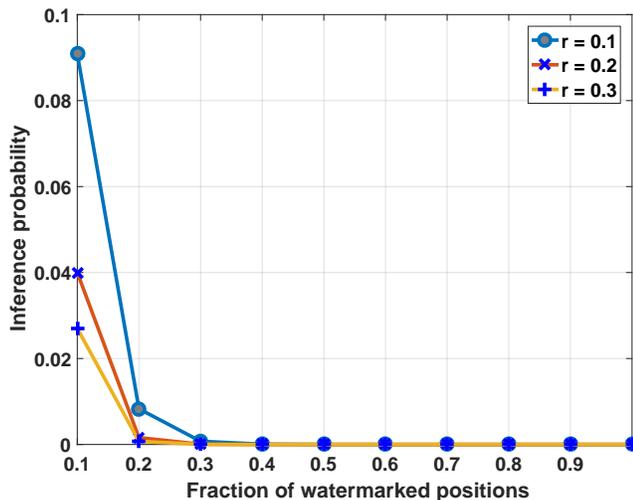
Figure 5: Inference probability to identify different fractions of the watermarked positions in the collusion attack when the number of colluding malicious SPs $h = 6$. r represents the fraction of watermarked data.

We also ran the same experiment to observe the probability of malicious SPs to identify different fractions of the watermarked positions. In Figure 5, we show this inference probability. For this experiment, we assume that the malicious SPs initially try to identify the watermark positions that has higher probability to be watermarked. Since we assume that the watermarking algorithm is publicly known by the malicious SPs, once they observe vertically aligned data points, they can compute the probability of being watermarked for each data position (as discussed in Section 4.1) and initially try to identify high probability watermark positions. We also set the number of colluding malicious SPs $h = 6$ and watermarked different fractions of the whole data (i.e., varied the r value). We observed that colluding SPs can identify small portion of watermark locations with small probabilities and this probability rapidly decreases with increasing fraction of watermarked data (r). Also, the probability to identify more than 30% of the watermarked locations is significantly low even when the malicious SPs collude. Notably, we show that when $r = 0.025$ (which means 200 watermarked data points on a data of size 7690, and hence preserves more than 97% of data utility), even when 6 malicious SPs collude, the probability to recover more than 30% of the watermark locations is very small. In other words, under this attack model, when $r = 0.025$, the proposed scheme is p -robust against f -watermark inference for $f = 0.3$ and $p \leq 10^{-1}$.

Correlation attack: To evaluate the security of the proposed scheme against the correlations in the data, we compared two techniques presented in Sections 4.1 and 4.2. In this analysis, we focused on a data length (ℓ) of 100 in our dataset. We find each pairwise correlation $Pr(x_i = \alpha | x_j = \beta)$ between these 100 data points, where $\alpha, \beta \in \{0, 1, 2\}$. To consider only strong correlations (and to avoid the noise that arise due to weak correlations), we only consider the ones above a threshold τ (we selected $\tau = 0.9$). Note that the correlations in the data are not symmetric. That is, $Pr(x_i = d_i | x_j = d_j)$ being high does not mean that $Pr(x_j = d_j | x_i = d_i)$ is also high.



(a) Correlations in the data are not considered when selecting the data points to be watermarked (i.e., technique proposed in Section 4.1 is used for watermarking).



(b) Correlations in the data are considered using the proposed algorithm when selecting the data points to be watermarked (i.e., technique proposed in Section 4.2 is used for watermarking).

Figure 6: Inference probability to identify different fractions of the watermarked positions in the single SP correlation attack. r represents the fraction of watermarked data.

First, we compared two schemes for a single SP attack in terms of the probability of the malicious SP to identify different fractions of the watermarked positions. Note that in this attack, the malicious SP also utilizes its knowledge of correlations in the data.² In Figure 6 we show this comparison for different r values. We observed in Figure 6a that as r increases, the inference probability of the malicious SP increases for the technique presented in Section 4.1. This is expected since (i) if correlations are not considered while

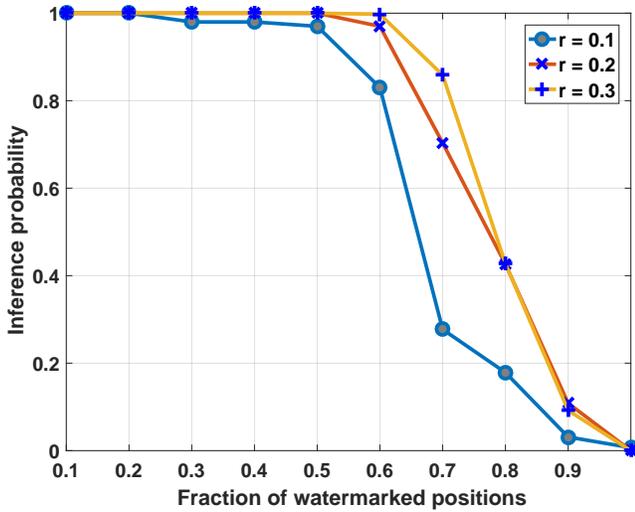
²We assume that knowledge of the malicious SP about the correlations is the same as the knowledge we utilized while adding the watermark in Section 4.2.

selecting the watermarked positions, the probability of the attacker to identify the watermarked positions also increases, and (ii) as more data points are watermarked in this way, the attacker can identify more watermarked position. However, when we consider the correlations in the data when selecting the watermark locations, the inference probability of the malicious SP significantly decreases as shown in Figure 6b. Also, in this scenario, inference probability decreases with increasing r value as expected. For instance, when $r = 0.3$, the watermarking scheme is p -robust against f -watermark inference for $f = 0.2$ and $p \simeq 1$ when the correlations in the data are not considered. When we consider the correlations in the data using the proposed watermarking algorithm, it becomes p -robust against f -watermark inference for $f = 0.2$ and $p \simeq 0$.

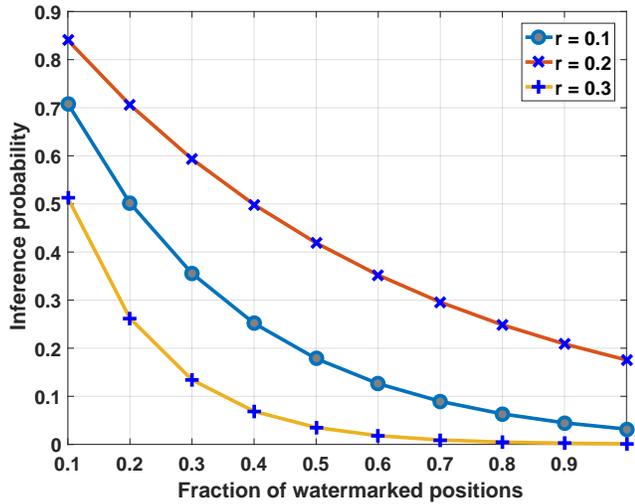
Collusion and correlation attack: We also compared two techniques presented in Sections 4.1 and 4.2 to show the resiliency of the proposed watermarking scheme against both collusion and correlation attacks at the same time. In this attack, each malicious SP first runs the correlation attack independently. As a result of this part, each malicious SP detects a number of watermarked points. For the advantage of the malicious SPs (and to consider the worst case scenario), we consider the outcome of the malicious SP with the highest number of correct detections. Let the number of watermarks detected by this malicious SP be m as a result of the first part.³ Then, to detect the remaining $w - m$ watermarked points, malicious SPs run the collusion attack.

In Figure 7 we show this comparison for different r values when the number of colluding malicious SPs $h = 6$ (and data has been shared for 6 times). We observed that when the correlations are not considered in the watermarking algorithm, malicious SPs can identify more than half of the watermarked data locations with high probability as shown in Figure 7a. However, when we consider the correlations to select the data points to be watermarked, the inference probability of the malicious SPs significantly decreases (as in Figure 7b). For instance, when $r = 0.3$, the watermarking scheme is p -robust against f -watermark inference for $f = 0.5$ and $p \simeq 1$ when the correlations in the data are not considered. When we consider the correlations in the data using the proposed watermarking algorithm, it becomes p -robust against f -watermark inference for $f = 0.5$ and $p \leq 0.1$. This shows that the proposed watermarking scheme provides security guarantees against both collusion and correlation attacks with high probabilities even when all the SPs that receive the data are malicious and colluding (as in this experiment). Note that in Figure 7b, the reason inference probabilities for $r = 0.2$ is larger than the ones for $r = 0.1$ is due to the result of the optimization problem.

³As discussed, malicious SPs may detect less than w watermarked points as a result of the correlation attack.



(a) Correlations in the data are not considered when selecting the data points to be watermarked (i.e., technique proposed in Section 4.1 is used for watermarking).

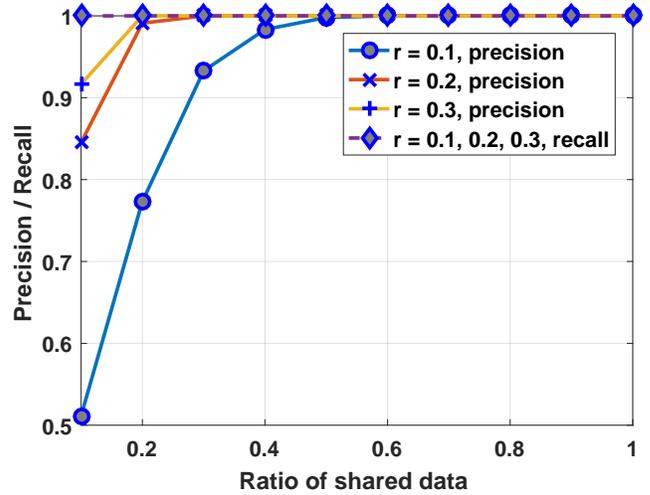


(b) Correlations in the data are considered using the proposed algorithm when selecting the data points to be watermarked (i.e., technique proposed in Section 4.2 is used for watermarking).

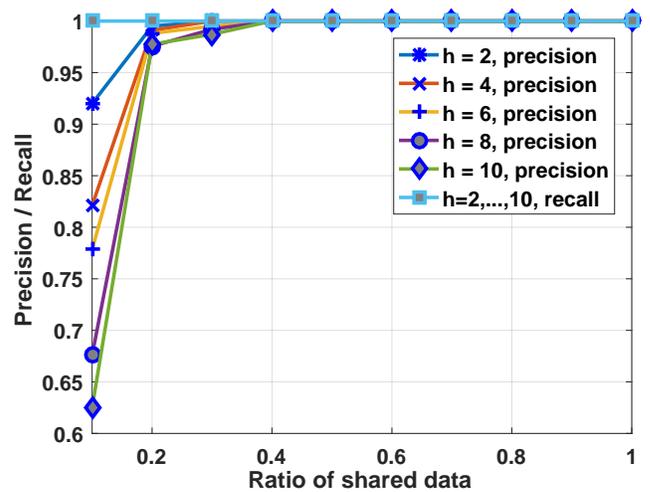
Figure 7: Inference probability to identify different fractions of the watermarked positions in collusion attack (when $h = 6$) in which the malicious SPs also use the correlations in the data. r represents the fraction of watermarked data.

5.2.2 Robustness against watermark modification

Here, we evaluate the robustness of the proposed scheme against watermark modification.



(a) Different fractions of watermarked data (r) when data has been shared with $h = 4$ SPs.



(b) Alice shares her data with h SPs when fraction of watermarked data $r = 0.2$.

Figure 8: Precision and recall values for the data owner to detect the malicious SP when the malicious SP partially shares Alice's data.

Partial sharing: We evaluated the detection performance (and robustness against watermark modification) of the proposed watermarking scheme when a malicious SP partially shares Alice's data. In this scenario, we assume that Alice has shared her data (same data portion at each sharing) with h SPs (SP_1, \dots, SP_h). The malicious SP, rather than sharing the whole data with a third party without Alice's authorization, shares different fractions of the data to avoid being detected by Alice. As we have shown in previous experiments, the probability for a malicious SP to detect the watermarked data points is significantly low for our proposed scheme (even in the existence of collusion attack). Thus, we assume that the malicious SP randomly selects different fractions of data points to share with the third party. Here, we assume the mali-

cious SP does not further modify Alice’s data before sharing it with a third party as such modification would degrade the credibility of the data (as we discuss in Section 6). We also consider and extensively study the impact of such modification to the detection performance later in this section.

We quantify the robustness against watermark modification under this attack using precision and recall metrics. Alice constructs a set \mathbf{S} that includes the malicious SPs detected by her. We define true positive as a malicious SP that is in set \mathbf{S} , false positive as a non-malicious SP that is in \mathbf{S} , true negative as a non-malicious SP that is not in \mathbf{S} , and false negative as a malicious SP that is not in \mathbf{S} . In Figure 8, we show the precision and recall values for varying ratio of shared data by the malicious SP and for different r and h values. Following our definition of robustness against watermark modification (in Section 3.4), under this attack model, the proposed scheme is ρ/ζ -robust against watermark modification with $\zeta = 1$ for all considered values of h and r . Furthermore, when $r \geq 0.2$, $h \leq 10$, and the ratio of shared data by the malicious SP is more than 0.2, the proposed scheme is ρ/ζ -robust against watermark modification with $\rho \simeq 0.97$ and $\zeta = 1$. We conclude that the data owner can associate the source of the leakage to the corresponding SP with high probability in most of the cases, except when the malicious SP shares very small portion of user’s data with a third party. However, this particular case would also reduce the benefit of the malicious SP (due to the unauthorized sharing) significantly. Furthermore, such partial sharing may degrade the credibility of data.

Watermark modification: Finally, we studied a stronger attack in which malicious SP (or SPs) modify the data in order to damage the watermark (and hence, it becomes harder for the data owner to detect the source of the data leak). Note that in practice, such modification of data not only reduces data utility (as we show in our experiments), but it also degrades data credibility while the malicious SPs share the data with a third party. Here, malicious SPs (or SP) try to remove or damage the watermark by (i) changing the states of data points that are different when they aggregate their data (i.e., when they detect a data point with multiple states in the aggregate data, they change its state to the majority of the observed states), and (ii) adding noise to other data points (i.e., changing states of other random data points). Eventually, data leaked by the malicious SPs has a watermark pattern represented as Z_α . Using Z_α and unique watermark patterns of the SPs (that previously received the data), Alice constructs the set \mathbf{S} that includes the malicious SPs detected by her. As before, we evaluate the success of the detection via precision and recall metrics. For all following experiments we set the watermark ratio (r) to 0.05.

First, we consider the single SP attack in which data has been shared with h SPs and there is a single malicious SP. Watermark length (w) is known by the malicious SP and the malicious SP randomly changes $(\pi \times w)$ data points in the data and shares it. For each SP i that received her data, Alice

computes $g_i = |Z_\alpha \cap Z_{I_i}|$ (Z_{I_i} is the watermark pattern of SP i) and identifies the malicious SP as the one with the highest g_i value. In Figure 9, we show the precision and recall when the data owner knows that there is a single malicious SP and for different π and h values. In this scenario, both the precision and recall values are high even when the malicious SP significantly damages the watermark. Under this attack, the proposed scheme is ρ/ζ -robust against watermark modification with $\rho = \zeta \simeq 1$ when $\pi < 13$ and $h \leq 20$ ($\pi = 13$ means a utility loss of 65%).

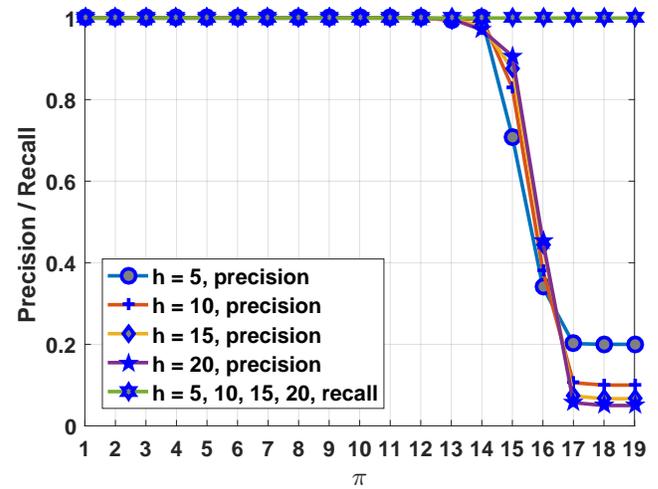
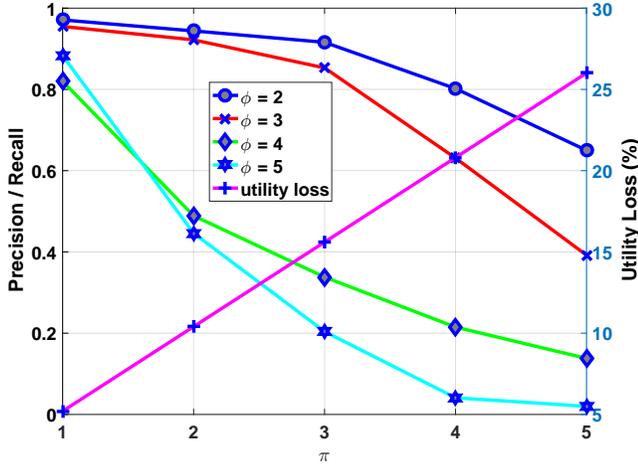
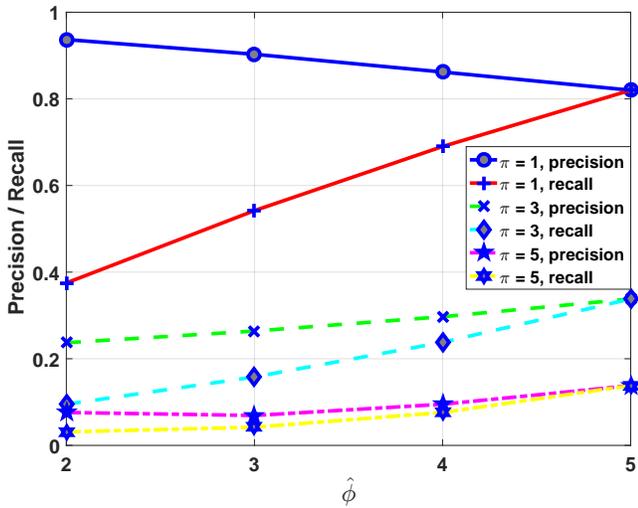


Figure 9: Precision and recall values for the data owner to detect the malicious SP in the single SP attack in which data has been shared with h SPs. Malicious SP randomly changes $\pi \times w$ data points to damage the watermark.

We also considered the case in which colluding malicious SPs compare their aggregated data and change the states of data points that are different, as discussed before. Colluding malicious SPs also add random noise in addition to changing the states of data points that are different in the aggregate data. We assume data has been shared with h SPs and colluding malicious SPs randomly change $(\pi \times w)$ data points in the data before they leak it. The data owner Alice may or may not know the number of malicious SPs. Let the actual number of malicious SPs be ϕ and the prediction of Alice for the number of malicious SPs be $\hat{\phi}$ which can be any number from 1 to h . Alice first generates all combinations of h with $\hat{\phi}$. Then, she eliminates the combinations for which the union of the watermarked points of the SPs (in that particular combination) does not contain the watermark pattern in the leaked data (Z_α). Next, for each non-eliminated combination c_i , she computes $g_i = \sum_{j \in c_i} |Z_\alpha \cap Z_{I_j}|$. That is, she computes the sum of intersections of watermarked data points for each SP in the corresponding combination c_i with Z_α . Finally, she selects the set \mathbf{S} as the most likely combination with the highest g_i value and concludes that the SP (or SPs) in the corresponding combination are malicious.



(a) Data owner knows the number of malicious SPs ($\phi = \hat{\phi}$).



(b) Data owner predicts the number of malicious SPs as $\hat{\phi}$ when the actual number of malicious SPs $\phi = 5$ and for varying π values.

Figure 10: Precision and recall values for the data owner to detect the malicious SPs in the collusion attack in which data has been shared with $h = 10$ SPs. Malicious SPs both change the states of data points that are different in the aggregated data and they randomly change $\pi \times w$ data points to damage the watermark. In (a), precision and recall curves for different ϕ values overlap. Also, in (a), we show the percentage of utility loss due to addition of extra noise by the malicious SPs.

In Figure 10, we show the precision and recall when $h = 10$, $\hat{\phi} = \phi$, and when the data owner does not know ϕ , respectively. In Figure 10a, we also show the percentage of utility loss in the data due to the noise addition by the malicious SPs (to damage the watermark). Here, the utility loss is shown when $r = 0.05$ (i.e., when 5% of original data is watermarked). As r value increases, the loss in utility (due to extra noise addition by the malicious SPs) also increases linearly. For instance when $r = 0.1$, to decrease the precision and recall values

down to 0.2, half of the SPs that received the data should be malicious and they need to add noise to 50% of the original data to damage the watermark. As shown in Figure 10a, if the data owner knows the number of malicious SPs, both precision and recall of detection performance are high up to 30% of the SPs that received the data are malicious (and colluding) and up to a utility loss of 15%. That is, the proposed scheme is ρ/ζ -robust against watermark modification with $\zeta = \rho \simeq 0.9$ up to $\phi = 3$ and $\pi = 3$. Beyond this, we observed a decrease in both precision and recall with increasing π and ϕ values. This behavior gives some idea about the practical limits of our proposed scheme. When data owner predicts the number of malicious SPs (Figure 10b), we observed two cases: (i) when the added noise by the malicious SPs is less than 3 times the watermark length, the proposed scheme includes the actual malicious SPs in set \mathbf{S} with a high probability. That is, the proposed scheme is ρ/ζ -robust against watermark modification with $\rho \simeq 0.7$ up to $\pi = 3$ and for all $\hat{\phi}$ values. When the added noise by malicious SPs is beyond this value, both precision and recall values start decreasing. However, adding noise beyond this value significantly reduces data utility as discussed before.

6 Discussion

Here, we discuss the potential use of our proposed scheme in real-life, its potential extensions, and future research directions.

Usability and Scalability. The proposed system detects the malicious SPs if data is leaked without the data owner's consent and if the data owner observes this leakage. Similarly, the SP that buys the data may keep the malicious SPs liable from this unauthorized sharing (with the cooperation of the data owner). It may be practically infeasible for a data owner to notice their data is leaked. Instead, this can be outsourced to a third party that continuously analyzes publicly available datasets that are made available by SPs that collect personal information.

The data owner can share their data with numerous SPs. The main constraint of the algorithm described in Section 4 is that the watermark pattern given to each SP should be unique. Thus, it is sufficient to change as less as one watermarked data point between two sharings of the same data with two SPs. However, as the overlap between watermark patterns increase, the precision and recall of the data owner to detect the malicious SP(s) decrease (as discussed in Section 5). We will further study this trade-off between scalability and watermark robustness in future work.

It is also important to note that the robustness guarantees of the proposed scheme may vary over time depending on the data type. For instance, via new discoveries in genomics, things that are non-sensitive today may turn out to be sensitive in the future. Similarly, new discoveries may result in new correlation models in the data. Thus, the evaluations we have

shown in Section 5 represent the robustness guarantees we can provide with today’s knowledge.

Data utility. We may include w (i.e., watermark length) as one of the objectives of the optimization problem and put a limit on it. When we do so, the problem becomes a multi-objective optimization problem. Solution of a multi-objective optimization problem is non-trivial and many proposed techniques suggest converting the multi-objective problem into a single-objective one. Thus, we transform this multi-objective problem into single objective problem.

In this new formulation, there are two additions to the optimization problem introduced in Section 4.1. First, the objective function is changed as follows:

$$\min\{\beta \cdot \prod_{i=0}^{h+1} \left(\frac{n_i^{h+1}}{n_i^{h+1} + n_{h-i+1}^{h+1}} \right)^{n_i^{h+1}} + (1 - \beta) \cdot w\}$$

We use the weighted sum of the watermark length and the inference probability as the new objective function. The weight (β) determines the tradeoff between the inference probability and the watermark length (i.e., data utility). Second, we add a new constraint as $w < w_m$, where w_m is the maximum allowed watermark length. This new constraint puts a threshold to the maximum number of watermark points. This new optimization problem guarantees the minimum weighted sum of inference probability and watermark length.

Depending on the data type, other utility constraints may also be included in the proposed algorithm. For instance, if adding watermark to two consecutive data points significantly reduces data utility, once y_i^h and y_{i+1}^h values are determined as a result of the optimization problem, watermark addition algorithm in Section 4.1 (or Section 4.2) can be tailored to take this constraint into account while adding the watermarks.

Other applications. The proposed watermarking algorithm can be applied for any type of sequential data (we describe the general framework for sequential data in Section 4). However, implementation for different data types is non-trivial. For instance, correlations in other types of data may be more complex. Furthermore, auxiliary information about the data owner may help a malicious SP to infer the watermarked positions with higher probability. To address some of these challenges, we will work on the application of the proposed scheme for location patterns as future work.

7 Conclusion and Future Work

In this work, we have proposed a scheme to share sequential data while addressing the liability issues in case of unauthorized sharing. The proposed scheme is between a data owner and one or more service providers. We have shown that the proposed watermarking scheme provides high security against collusion and correlation attacks. That is, with high probability, malicious service providers cannot identify the watermark on the data even if they collude or try to use the inherent correlations in the data. We have also shown that the

proposed scheme does not degrade the utility of data while it provides the aforementioned security guarantees. We believe that the proposed work will deter the service providers from unauthorized sharing of personal data with third parties. The algorithm proposed in this paper does not consider if malicious SPs share statistics (e.g., average or median) about the data without the authorization of the data owner. Such statistics can also be shared by aggregating multiple data owners’s data. In future work, we will also consider this and work to develop algorithm that also identify the unauthorized sharing in such scenarios.

References

- [1] 1000gp phase 3 haplotypes. https://mathgen.stats.ox.ac.uk/impute/1000GP_Phase3.html, 2017.
- [2] Single-nucleotide polymorphism. https://isogg.org/wiki/Single-nucleotide_polymorphism, 2017.
- [3] André Adelsbach, Stefan Katzenbeisser, and Ahmad-Reza Sadeghi. A computational model for watermark robustness. *Proceedings of the 8th International Conference on Information Hiding*, pages 145–160, 2007.
- [4] J.A. Bloom, I.J. Cox, T. Kalker, J.-P.M.G. Linnartz, M.L. Miller, and C.B.S. Traw. Copy protection for DVD video. *Proceedings of the IEEE*, 87(7):1267–1276, Jul. 1999.
- [5] Dan Boneh and James Shaw. Collusion-secure fingerprinting for digital data. *IEEE Transactions on Information Theory*, 44(5):1897–1905, 1998.
- [6] Tae-Yun Chung, Min-Suk Hong, Young-Nam Oh, Dong-Ho Shin, and Sang-Hui Park. Digital watermarking for copyright protection of mpeg2 compressed video. *IEEE Transactions on Consumer Electronics*, 44(3):895–901, 1998.
- [7] Ingemar J Cox, Matthew L Miller, Jeffrey Adam Bloom, and Chris Honsinger. *Digital watermarking*. Springer, 2002.
- [8] S Emmanuel, AP Vinod, D Rajan, and CK Heng. An authentication watermarking scheme with transaction tracking enabled. In *Digital EcoSystems and Technologies Conference*, pages 481–486. IEEE, 2007.
- [9] Huiji Gao, Jiliang Tang, and Huan Liu. gscorr: modeling geo-social correlations for new check-ins on location-based social networks. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, pages 1582–1586. ACM, 2012.

- [10] John D. Hedengren, Reza Asgharzadeh Shishavan, Kody M. Powell, and Thomas F. Edgar. Nonlinear modeling, estimation and predictive control in APMonitor. *Computers & Chemical Engineering*, 70:133–148, 2014.
- [11] Xiaoming Jin, Zhihao Zhang, Jianmin Wang, and Deyi Li. Watermarking spatial trajectory database. In *International Conference on Database Systems for Advanced Applications*, pages 56–67. Springer, 2005.
- [12] Nurul Shamimi Kamaruddin, Amirrudin Kamsin, Lip Yee Por, and Hameedur Rahman. A review of text watermarking: Theory, methods, and applications. *IEEE Access*, 6:8011–8028, 2018.
- [13] Suleyman S Kozat, Michail Vlachos, Claudio Lucchese, Helga Van Herle, and S Yu Philip. Embedding and retrieving private metadata in electrocardiograms. *Journal of Medical Systems*, 33(4):241–259, 2009.
- [14] Sin-Joo Lee and Sung-Hwan Jung. A survey of watermarking techniques applied to multimedia. In *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No. 01TH8570)*, volume 1, pages 272–277. IEEE, 2001.
- [15] Li Liu and Xiaoju Li. Watermarking protocol for broadcast monitoring. In *Proceedings of International Conference on E-Business and E-Government*, pages 1634–1637. IEEE, 2010.
- [16] Claudio Lucchese, Michail Vlachos, Deepak Rajan, and Philip S Yu. Rights protection of trajectory datasets with nearest-neighbor preservation. *The VLDB Journal—The International Journal on Very Large Data Bases*, 19(4):531–556, 2010.
- [17] Maurice Maes, Ton Kalker, J-PMG Linnartz, Joop Talstra, FG Depovere, and Jaap Haitzma. Digital watermarking for DVD video copy protection. *IEEE Signal Processing Magazine*, 17(5):47–57, 2000.
- [18] N. Memon and Ping Wah Wong. A buyer-seller watermarking protocol. *IEEE Transactions on Image Processing*, 10(4):643–649, Apr. 2001.
- [19] M. Naveed, E. Ayday, E. W. Clayton, J. Fellay, C. A. Gunter, J.-P. Hubaux, B. A. Malin, and X. Wang. Privacy in the genomic era. *ACM Computing Surveys*, 48(1), Sep. 2015.
- [20] Hussain Nyeem, Wageeh Boles, and Colin Boyd. On the robustness and security of digital image watermarking. In *Informatics, Electronics & Vision (ICIEV), 2012 International Conference on*, pages 1136–1141. IEEE, 2012.
- [21] Hussain Nyeem, Wageeh Boles, and Colin Boyd. Digital image watermarking: its formal model, fundamental properties and possible attacks. *EURASIP Journal on Advances in Signal Processing*, 2014(1):135, 2014.
- [22] S. S. Samani, Z. Huang, E. Ayday, M. Elliot, J. Fellay, J.-P. Hubaux, and Z. Kutalik. Quantifying genomic privacy via inference attack with high-order SNV correlations. *Proceedings of Workshop on Genome Privacy and Security*, 2015.
- [23] Montgomery Slatkin. Linkage disequilibrium — understanding the evolutionary past and mapping the medical future. *Nature Reviews Genetics*, 9(6), 2008.
- [24] Arezou Soltani Panah and Ron Van Schyndel. A lightweight high capacity ecg watermark with protection against data loss. In *Proceedings of the 8th International Conference on Pervasive Computing Technologies for Healthcare*, pages 93–100. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.
- [25] Michail Vlachos, Johannes Schneider, and Vassilios G Vassiliadis. On data publishing with clustering preservation. *ACM Transactions on Knowledge Discovery from Data*, 9(3):23, 2015.
- [26] Xiaohan Zhao, Qingyun Liu, Haitao Zheng, and Ben Y Zhao. Towards graph watermarks. In *Proceedings of the 2015 ACM on Conference on Online Social Networks*, pages 101–112. ACM, 2015.
- [27] Jianpeng Zhu, Qing Wei, Jun Xiao, and Ying Wang. A fragile software watermarking algorithm for content authentication. In *IEEE Youth Conference on Information, Computing and Telecommunication*, pages 391–394. IEEE, 2009.

Smart Malware that Uses Leaked Control Data of Robotic Applications: The Case of Raven-II Surgical Robots

Keywhan Chung¹, Xiao Li¹, Peicheng Tang², Zeran Zhu¹, Zbigniew T. Kalbarczyk¹, Ravishankar K. Iyer¹,
and Thenkurussi Kesavadas¹

¹University of Illinois at Urbana-Champaign

²Rose-Hulman Institute of Technology

Abstract

In this paper, we demonstrate a new type of threat that leverages machine learning techniques to maximize its impact. We use the Raven-II surgical robot and its haptic feedback rendering algorithm as an application. We exploit ROS vulnerabilities and implement smart self-learning malware that can track the movements of the robot's arms and trigger the attack payload when the robot is in a critical stage of a (hypothetical) surgical procedure. By keeping the learning procedure internal to the malicious node that runs outside the physical components of the robotic application, an adversary can hide most of the malicious activities from security monitors that might be deployed in the system. Also, if an attack payload mimics an accidental failure, it is likely that the system administrator will fail to identify the malicious intention and will treat the attack as an accidental failure. After demonstrating the security threats, we devise methods (i.e., a safety engine) to protect the robotic system against the identified risk.

1 Introduction

A number of attempts have been made to leverage machine learning (ML) techniques to realize malicious intentions. For instance, adversarial learning was used to effectively deceive data-driven models by strategically injecting malicious input [8, 25, 45, 47] to identify the target of an attack [28], or to infer information hidden behind encrypted data [26, 33]. In this context, smart malware that employs ML techniques represents a new concept for the implementation of sophisticated attack strategies. Such malware can infer attack strategies based on live operational data and trigger an attack at the most opportune time so as to maximize the impact.

For threats that use self-learning malware, robotic applications turn out to be a fascinating target. Like other cyber-physical systems (CPSes), robotic applications incorporate sensors and actuators that are connected through a network that passes around data. Their (i) *relatively weak security* [5, 13, 31], (ii) *abundance of data that can be used to infer actionable intelligence* [4, 9, 54], and (iii) *close proximity to*

and direct interactions with humans (such that a successful attack could have a life-threatening impact) [14, 22, 23] make robotic applications a tempting target for advanced threats.

To demonstrate the feasibility of smart malware, we have built an injection module as a prototype. Our prototype smart malware eavesdrops on the communication between the robot components of a near-real-time system (as an input for the smart malware), uses the leaked data to infer intelligence on when to trigger the payload (or take control over the robot), and executes the payload at the most opportune time (i.e., the output of our smart malware) so that it can maximize the impact. While our attack model applies to any robotic system, in this paper, we use the Raven-II surgical robot [3] and its haptic feedback rendering algorithm as a target application.

Raven-II is driven by the Robot Operating System (ROS) [44], an open-source framework that has been widely deployed across various robotic applications (i.e., more than 125 applications [38]), and its resiliency is critical to varying domains (e.g., robotic surgery, aviation, and manufacturing). However, the most commonly used ROS contains vulnerabilities [15] that leak data (e.g., robot state) transmitted within the application, and those data can become the basis from which smart malware can *learn* about the system behavior and use this information to decide when to trigger an attack. We exploited ROS vulnerabilities and implemented *smart malware* that tracks the movement of the robot's arms and triggers the attack payload when the robot is in a critical state of a (hypothetical) surgical procedure. After demonstrating the security threats, we discuss the methods (i.e., a safety module) that we devised to protect the robotic system against the identified risk.

What makes our malware stealthy is the invisibility of its learning process to security monitoring systems. Unlike common malware, which is installed *in* a victim system, our malware runs outside the physical components of the robotic application. The ROS allows any new node/process to register with a master (core) node; hence, an attacker can register its malicious node to the robotic application without being noticed. By keeping the learning procedure internal to the

malicious node, an adversary can hide all malicious activities (except for its network activities with genuine nodes of the target application) from security monitors that might be deployed. Hence, only the impact of the attack, which mimics accidental failures, is observable to the system administrator. As a result, it is likely that malicious faults will be seen as accidental failures (especially if the network traffic is not being monitored). Note that additional network traffic introduced by our malware prototype is negligible (about 0.24% of the volume of the genuine traffic).

The contributions of this paper are the following:

- We show the possibility of a *real attack on a surgical robot* that exploits known vulnerabilities in the underlying runtime environment, ROS. The vulnerabilities allow a malicious entity to operate as a man-in-the-middle (MITM), with the ability to eavesdrop (i.e., leak robot control data) and overwrite communication among the robot components (i.e., effectively take control of the robot).
- We demonstrate *smart malware logic* that can infer the most opportune time of attack from the information obtained through exploitation of the vulnerabilities in ROS. Our experiment with three use cases that mimic (hypothetical) surgical operations of different levels of complexity shows that the ML algorithm (DBSCAN) used by our malware can determine the position of the robot end-effector with respect to the target object and use this information to trigger the execution of the payload. Specifically, the DBSCAN algorithm triggers the injection of the attack payload (i.e., corruption of data used to control the robot) when the robot arm is in close proximity to the target object (i.e., there is less than 10 mm distance between the robot end-effector and the target object).
- We present a set of *unique faults* that, when used as an attack payload, can threaten the integrity of the surgical operation. The faults consist of realistic scenarios that can be disguised as accidental failures, such as network packet drop, data corruption, or bugs in the control software.
- We implement a *ROS-generic safety module* that can detect abnormalities introduced by attacks and can bring the robot to a safe state. Specifically, the safety module detects the shutdown signal generated by the *roscore* in the case of a name conflict (i.e., a new node registers itself with an already existing name). If that happens, the safety module terminates the new node, takes over the control of the robot, and returns it to a predefined safe state to prevent further impact.

While we demonstrate the feasibility of the advanced threat in the context of Raven-II and its underlying framework (i.e., ROS), the design of the smart malware is sufficiently generic that it can be used on other robotic systems that generate a stream of sensor data from input sensors and robot control data to the physical robots. As summarized in [1], an attacker can intrude into a robotic system through various entry points (e.g., third-party networks, vulnerable workstations, and vul-

nerable or incorrectly configured firewalls or gateways). Once smart malware has established an MITM attack (i.e., it can listen to and overwrite control data), its smart injection module can infer the critical time of the operation of any robot. The attack payloads (i.e., the faults to be injected), on the other hand, are specific to the robotic application.

2 Motivation for smart malware

Machine learning techniques have been applied in different domains (e.g., image processing and natural language processing) to derive intelligence from data. Researchers and engineers in cyber security have also deployed ML-based techniques as part of an effort to advance methods for detecting malicious activities. However, not much work has considered the possibility that adversaries could take advantage of machine learning algorithms to devise attack strategies. More specifically, a few studies have investigated the potential impact of attacks that are supported by machine learning algorithms [40, 41]. In this paper, we define smart malware as malicious software that can, by itself, derive intelligence from data obtained from the victim system.

Smart malware is available only at a cost (i.e., high computation workload). However, we find reasons that might justify the overhead: access to rich data and an ability to achieve high impact with minimized remote interaction between the malware (software) and the attacker (human). (That is, to a certain extent, machine learning algorithms can replace human-driven analysis in designing/customizing malware.) Notably, long and unusual remote connections often lead to exposure of attackers. Furthermore, the computational load imposed by the execution of the smart malware can be obfuscated with techniques such as the “low and slow” approach, whereby attackers intentionally reduce the computation workload despite having to tolerate a longer time of execution.

For machine-learning-driven threats, cyber-physical systems (especially robotic applications) turn out to be tempting targets. In cyber-physical systems, sensors and monitors are deployed across the system to gather information (e.g., on images, sounds, temperature, and flows). Data collected from input sensors are sent to controllers or computation units that derive control variables or decisions, which are passed to the actuator to update the state of the system. While traditional robots were contained within a single physical system, the new concept of distributed robotics (or collaborative robotics) is expanding the boundary of robotic systems. (E.g., with remote surgery, a physician can perform surgery from a remote location.) A key enabler for this new mode of attacking the system is a protocol for sharing data across a network. However, if the protocol is not properly designed for security, it can introduce vulnerabilities that eventually exploited by smart malware.

For instance, a publish-subscribe model is a common messaging pattern in which the information is shared between

the publisher and the subscriber. Its advantages include scalability and loose coupling between the publishers and the subscribers. However, such advantages introduce side effects that impact the security of the system. Without authentication and encryption, unauthorized entities can read messages and leak data. Such data become a baseline for learning, from which malicious entities can derive actionable intelligence. In this paper, we demonstrate the threat by using the Raven-II surgical robot (running on top of ROS) as the target for such an attack strategy.

3 Background: Robots, ROS, and Raven-II

Robots have been adopted across different application domains. For example, in manufacturing, robot manipulators assist human workers; drones are deployed in agriculture, entertainment, and military operations; and surgical robots support surgeons in performing medical procedures. For such applications, robots play a critical role. A robot's failure to make a correct and timely movement can lead to catastrophic consequences, such as injuring people near the robots in factories or risking a patient's life during surgery. This study focuses on the resiliency of a surgical robot against malicious attacks. We use the Raven-II surgical robot [3] and its haptic rendering algorithm as an application to demonstrate the security threat, and suggest methods to cope with the risk.

Robot Operating System (ROS). The Robot Operating System (ROS) is an open-source framework for programming robots [44], and is commonly used by various robotic applications. According to its official website, ROS is widely deployed across more than 125 different robots, including mobile robots, drones, manipulators, and humanoid [38]. The framework is being developed to support collaborative development by experts from different domains (e.g., computer vision or motion planning) and provides hardware abstraction, device drivers, libraries, and a communication interface [44]. For instance, the OpenCV library [6] provides interfaces that can be used to add vision to robotics applications, and the OpenNI library [35] focuses on integrating 3D sensors into robots. As ROS provides the core underlying runtime environment, the security of ROS is critical in ensuring the correct operation of the robot.

As shown in Fig. 1a, a ROS-based application consists of multiple *ROS nodes*. They can be running on a single physical machine or can be distributed across multiple machines (i.e., *Computers A* and *B* in the figure), as long as they share the *ROS core*, which is deployed on the computer declared as the *ROS master*. Each node communicates over the network, and the ROS core serves as the central server for all nodes. Data are exchanged in the context of a *topic*, where a *topic* is a data structure defined to deliver a specific context type; e.g., the image sensor data are exchanged in the form of multidimensional arrays, which consist of the RGB color codes for all pixels captured by the image sensor. A node, either a *pub-*

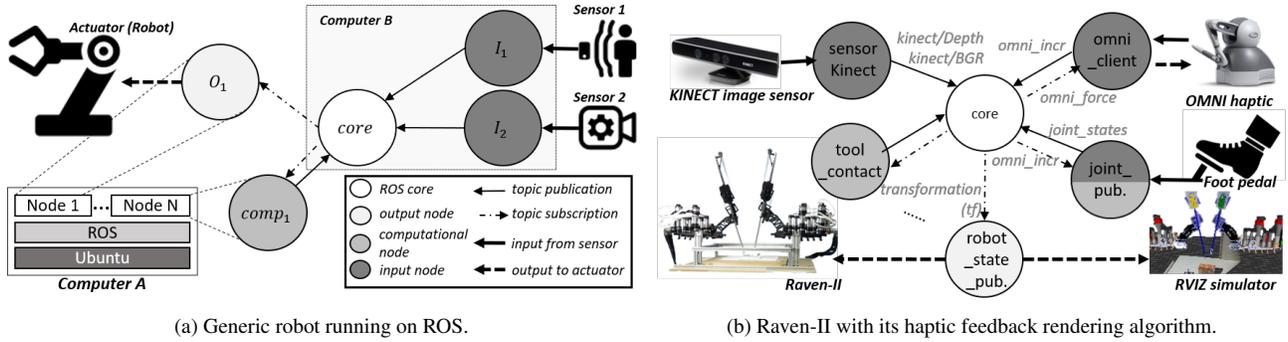
lisher or a *subscriber*, registers itself to the ROS core for the topic that the node is about to publish (or subscribe to). The ROS core then passes the information (i.e., the IP address) of the publisher to the subscriber waiting for the topic, so that the subscriber can establish a TCP connection with the publisher. After a handshaking protocol and transmission of the metadata that include the structure of the topic message, the two entities start passing the message by using a ROS-specific protocol.

The ROS nodes can be classified into three types: input nodes, output nodes, and computational nodes. An *input node* is a node connected to a piece of hardware (e.g., an image sensor or a haptic device) that provides input to the robot application. The input node, using the device driver provided by (or interfaced with) ROS, collects the data and converts the data into a ROS message as defined for the topic. Once the message is ready, the input node publishes it. The input node should have declared itself to the ROS core as a publisher for a topic. A *computational node* is a node that takes the input data to produce the output (e.g., a command to a robot actuator); it subscribes from the input node and publishes to the output node. Finally, an *output node* is a node connected to an actuator (i.e., the robot). The values (i.e., robot states and control commands) subscribed from the computational node are converted into a form that the hardware can interpret, and passed to the hardware. The output data determine the (joint) state of the robot.

The ROS framework comes with the *RVIZ* software package, which allows users to test and visualize the operation of the robot applications in a virtualized environment. *RVIZ* takes the physical specifications of the robot and displays the mesh of the robot; it is heavily used to test robot designs without using a physical robot.

Raven-II and haptic force feedback rendering engine. In this paper, we study the resiliency of a ROS application in the context of a surgical robot (i.e., Raven-II) and its haptic feedback rendering engine. Leveraging the open-architecture surgical robot, the authors of [30] present a hardware-in-the-loop simulator for training surgeons in telerobotic surgery. The simulator, in addition to having all the features of the Raven surgical robot, introduces a novel algorithm to provide haptic feedback to the operator and, hence, offer a touch sensation to surgeons. Unfortunately, commercially available surgical systems¹ (e.g., da Vinci by Intuitive [52]) do not provide haptic feedback to the operator. The traditional approach for haptic feedback uses physical force sensors to determine the force applied to the robot. Since the instruments (on which the force sensors are installed) are disposable, that approach turns out to be costly. Instead, the authors of [30] proposed an indirect haptic feedback rendering approach that does not rely

¹In 2017, the FDA approved a new surgical system [49] with haptic force feedback. However, we do not have sufficient information to understand the underlying technology and, hence, the capability of the system.



(a) Generic robot running on ROS. (b) Raven-II with its haptic feedback rendering algorithm.

Figure 1: Software architecture of robotic applications.

on force sensor measurement, but instead uses image sensor data to derive the force feedback.

In our study, the haptic feedback rendering algorithm, as implemented in the augmented Raven-II simulator, utilizes information from a depth map (a matrix of distances from the image sensor to each pixel of a hard surface) to derive the distance from the object (e.g., a patient’s tissues) to the robot arm. Using the current position of the arm and the measured distance to the object, the algorithm returns an interactive force value that generates resistance in the haptic device.

Figure 1b provides an overview of the software architecture of Raven-II and its haptic feedback rendering algorithm. The haptic algorithm takes input from the Kinect image sensor and the OMNI haptic device to control the Raven-II robot (or its virtual representation in RVIZ). The `sensor_kinect` node parses the image data (as BGR and depth) from the Kinect image sensor, packages the data into ROS messages, and publishes the messages to the ROS core as topics (`kinect/BGR` and `kinect/Depth`). The `omni_client` node is connected to the OMNI haptic device for user input. The `omni_client` node shares the processed operator input as a topic (`omni_incr`). A set of nodes, dedicated to running the algorithm, subscribe to the topics from the ROS core and derive the force feedback, which the `omni_client` sends to the haptic device.

The `kinect/Depth` topic from `sensor_kinect` is used to derive the distance from the robot arm to the object. However, in deriving the distance, the algorithm needs a reference frame. It leverages the ArUco library [21, 46], which is an Open-Source library commonly used for camera pose estimation. With the ArUco marker (i.e., a squared marker) location fixed and used as a reference point, we can derive the location of the robot arm(s) relative to the marker. Using that information, the algorithm can derive the distance from the robot arm to the object by using (i) the transformation from the marker to the robot, (ii) the transformation from the image sensor to the marker, and (iii) the transformation from the image sensor to the object. Because the transformation from the robot arm to the object is evaluated in near-real-time, the algorithm can provide timely haptic force feedback to the OMNI device.

4 Approach

Cyber security is often referred to as a “cat and mouse” game. In this paper, we are considering potential advances in cyber threats, assuming that adversaries will eventually take advantage of machine learning techniques (if they are not already doing so). In this section, we present our approach for corrupting a robotic application with self-learning malware. We developed this approach to raise awareness of the potential threats, and to promote preparation for responding to this threat. The methodologies included in our approach can be used for (i) preemptive identification of vulnerabilities, (ii) hardening of robotic applications against potential threats, and (iii) design of detection/mitigation methods.

Threat model. In our threat model, we assume:

- The attacker can penetrate into the control network of the robot. As presented in [13], ROS applications are often connected to a public network without proper protection. One can provide a level of protection by virtually isolating the control network (i.e., by deploying a VLAN). However, it would be possible to intrude into the virtual network either with stolen credentials or by exploiting a weak link (i.e., a vulnerable computer that has access to the VLAN). In the context of attacks on surgical robots, a survey on potential entry points of a hospital network can be found in [1].
- The attacker understands the operation of ROS, and has access to ROS-provided APIs (which are easily obtainable online). With remote access to the ROS master, one can execute ROS commands.
- The target robot runs on top of ROS 1. Our attack model is designed for ROS 1 (e.g., *Kinetic* and *Melodic*), which is still the most commonly deployed version despite the release of ROS 2 (discussed in detail in Section 7.1) in 2015. Software patches have been issued to fix the vulnerabilities in ROS, but, as we discuss in Section 7.1, the patches merely require attackers to take another step to neutralize them. Hence, in describing our attack model, we assume the default setting of the most commonly used ROS.

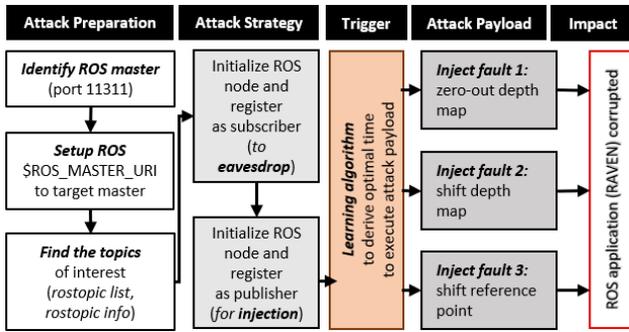


Figure 2: Approach overview, from attack preparation to impact to ROS application.

Approach overview. Vulnerabilities present in the ROS framework allow unauthorized entities to eavesdrop on messages passed across ROS nodes. Utilizing the obtained data, a malicious entity can identify an ideal time to trigger an attack and corrupt the operation of the robot by injecting faulty input or output commands. In Figure 2, we present an overview of our approach. During the **attack preparation** phase, we identify the victim (i.e., a ROS master that is remotely accessible) and its critical components. Once they are identified, the **attack strategy** can be applied to perform a man-in-the-middle (MITM) attack. As the malware eavesdrops on the sensor and control data of the robot, the smart malware runs a learning algorithm to infer the location of the target object. When the object is identified (i.e., the algorithm returns a cluster) and the robot reaches the predicted location of the target object, a **trigger** is raised to initiate the **attack payload**. In the following, we describe the details of our approach.

4.1 Attack preparation

To deploy the attack, the first step is to identify machines that are running ROS as a master (core) node. Using a network scanning tool, we scan for the default port for ROS masters (i.e., 11311, a well-known port for ROS masters) [13]. Once the master and its IP address are known, we set up ROS on our machine (which mimics a remote attacker) and update the ROS master’s Uniform Resource Identifier (URI) variable to that of the identified master. Using the ROS APIs, we search for the topics of interest (i.e., the topics registered to the ROS master are used as a signature for identifying the ROS application).

4.2 Attack strategy: ROS-specific MITM

In corrupting a ROS application, we take advantage of the vulnerabilities in ROS and execute a ROS-specific man-in-the-middle attack. As described in Section 3, ROS provides a set of interfaces (*publish/subscribe*) that ROS nodes can use to communicate; the ROS core serves as the arbitrator.

While the communication might include sensitive data, the ROS 1 framework does not provide options for authenticating or validating the ROS entities. (I.e., any ROS node that can access the master can register itself as a publisher to write messages or as a subscriber to read messages.) After configuring the ROS setup to connect to the victim ROS master (attack preparation; see Section 4.1), our malware can initiate a subscriber that eavesdrops on the network communications. To take control of the robot, it kicks out a genuine node and publishes malicious data while masquerading as the original publisher. Without noticing the change in the publisher, the robotic application takes the malicious data as an input (or command) and updates the state of the robot accordingly.

4.3 Trigger: Inference of critical time to initiate the malicious payload

Most security attacks are detected when the attack payload is executed [50]. Once detected, the attacker (or the malware that the attacker had installed) is removed from the system. Consequently, in many cases, the attacker may have one chance to execute the payload before being detected. As a result, it is realistic to consider the case in which an attacker tries to identify the ideal time to execute the attack payload (in our case, to inject a fault) in order to maximize the chances of success. A common approach is to embed a trigger function into the malware, which checks for a condition and executes the payload only when the condition is satisfied.

In [2], Alemzadeh et al. presented an attack model that is triggered by a prediction of the robot state derived by a side-channel attack. In the model, the attacker installs malware on the robot control system, eavesdrops on a USB packet, and infers the state of the robot (i.e., either “engaged” when the surgeon’s input is updating the position of the robot, or “disengaged” when the position of the robot is not being updated). The robot state (which is controlled by pedal input from the surgeon) is an effective indicator in determining when malicious input would be fed into the robot. However, in such an approach, it is hard to accurately determine the time window during which the robot is performing critical activities; e.g., it is more critical when the robot is cutting tissue than when it is transitioning towards the target object.

In this study, we present an approach that leverages a well-studied learning technique to infer the *critical time* to trigger the attack payload, so as to maximize the impact.

Inference of object location. During a surgical operation, the robot usually moves within a limited range defined by the nature of the surgical procedure. Hence, the precision in identifying ‘the time when the robot is touching (or maneuvering close to) the target object’ can help in triggering the attack at the most opportune time so as to maximize the impact. For instance, when the robot is moving from its idle location to the patient on the operating table, the robot is operating in an open space without obstacles. Hence, visual input is sufficient

to allow the surgeon to operate. Furthermore, the surgeon will not even notice whether the haptic feedback rendering algorithm is operational, as there is no surface that the robot would touch (i.e., there is zero force feedback). On the other hand, when the robot is inside the abdomen of the patient, it is operating in limited space packed with obstacles (e.g., organs) and with blind spots that the image sensor cannot monitor. In that situation, correct operation of the rendering algorithm is critical. Also, the shorter the distance from the robot (at the point of the trigger) to the target object, the less time it takes the surgeon to respond² upon discovering the failure of the rendering algorithm (which can be determined only by noticing the lack of force feedback when a surface is touched). In this paper, *we analyze the spatial density of the robot end-effector position throughout the operation to infer a time when the robot (i.e., the surgical instrument) is near the object.*

Algorithm. We use unsupervised machine learning to determine the location of the target object with respect to the position of the robot’s end-effector(s). Specifically, we adopted the density-based spatial-clustering algorithm with noise (DBSCAN) [20, 48] to accomplish this task. The DBSCAN algorithm takes two parameters, ϵ and *numMinPoints*. The maximum distance parameter (ϵ) defines the maximum distance between neighboring data points. Iterating over all data points, the algorithm checks for neighbors whose distance from a data point is less than ϵ . If the number of neighbors is less than *numMinPoints*, the data point is considered noise (i.e., the data point is not part of a cluster). Otherwise, the algorithm checks whether the neighbors form a cluster. Any clusters already formed by a neighbor are merged into the current cluster. Although the DBSCAN algorithm is known for its sensitivity to the choice of parameters, the attacker does not have much information from which to derive the right parameters. Based on the data subscription frequency (i.e., 80 Hz for our eavesdropper) and our conservative assumption that at least 10% of the overall operation time corresponds to the critical procedures³ (i.e., 10 seconds for our data from ≈ 100 seconds of robot operation), we set *numMinPoints* to 800 (i.e., *subscription_frequency* \times *seconds_of_stay*). Also, we consider points (corresponding to the robot’s end-effector position) within 1 cm of each other to be “close,” and define $\epsilon=10$. With a goal of demonstrating the feasibility of self-learning malware (not of presenting the best algorithm or parameters for a clustering problem), we find the parameter pair (ϵ , *numMinPoints*) to be accurate enough for our study. The optimization of the parameters is outside the scope of this paper.

²Similar to the concept of braking distance when driving a car.

³From a set of medical studies, we find that the mean of the total operation time was 178.2 minutes [12] and that the mean time for a critical procedure was 22.2 minutes [19] (i.e., 12.4% of the mean procedure time).

4.4 Attack payload: Fault injection

While the attack strategy in Section 4.2 is generic to the ROS framework, the payload is specific to the ROS application under study (i.e., Raven-II and its haptic feedback rendering algorithm). As part of assessing the resiliency of the haptic feedback rendering engine, we designed a set of faults that can be injected on-the-fly. The faults were designed through a careful study of Raven-II’s operation and its rendering algorithm. In this paper, we present three fault models that cause the haptic feedback rendering engine to fail to prevent the operator from penetrating the surface of the object under operation. The three fault models are representative in mimicking realistic cases of (i) loss of information during transmission of data, (ii) data corruption, (iii) a glitch in sensors, and/or (iv) a bug in the software algorithm. None of the faults are specific to the environment (i.e., the faults are not affected by custom settings of the robot in a certain environment). Hence, understanding of the application (without needing to understand certain custom configurations) is sufficient for designing effective faults.

Fault 1: Loss of granularity in the depth map. As discussed in Section 3, the haptic feedback rendering algorithm relies heavily on image sensor data. Our first fault model demonstrates a case in which the quality of the image from the sensor is degraded. More specifically, we consider a case in which the granularity of the depth map has become sparse due to (i) hardware problems in the image sensor or (ii) loss of data during transmission of the depth data. In this fault model, we randomly choose a certain percentage of the pixels, for which we neutralize the depth information (i.e., set it to zero, which can be interpreted as setting the distance to that of the ground surface). By carefully choosing the rate at which pixels are dropped, we can disguise an attack as natural noise, despite its critical impact.

Fault 2: Shifted depth map. The second fault model considers a case in which an entity with malicious intent manipulates a ROS message to obfuscate the visual data provided to the operator. Just as we dropped the depth information from the image sensor in Fault 1, we can overwrite the depth map message with shifted values, causing the rendering algorithm to provide incorrect haptic feedback that the operator will rely on. In our experiment, we shifted the depth map of the object under operation by 50 pixels to the right. As the ROS message that contained the BGR information remained untouched, the 3D rendered image of the object was incomplete. Similarly, an attacker can shift the data in the BGR message and deliver the malicious visual image to the Raven operator.

Fault 3: Corrupted reference frame. As part of rendering haptic feedback, the application needs to derive the distance from the object (under operation) to the robot arms, as the location of the object can change on every run or during the operation of the robot. An attacker can corrupt the coordinates of the reference frame and make the rendered feedback

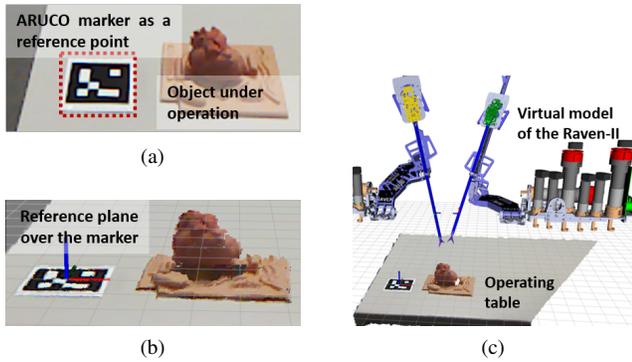


Figure 3: Experimental setup with (a) BGR image of the operating table; (b) *RVIZ* representation of the operating table in 3D; and (c) *RVIZ* representation of the operating room, including Raven-II.

become invalid. In this fault model we modify the reference frame during the transmission of the coordinates from the image sensor node to the computational node. Note that the ArUco detection-based reference frame is applicable in experimental settings such as ours. In commercial surgical robots, the known position of trocars (i.e., pen-shaped instruments used to create an opening into the body [7, 29]) are used as the reference frame, and the fault model would need to be modified accordingly.

5 Experiment design

Experimental setup. In order to mimic the settings of a surgical operation, we set up a mock-up of a heart (the object under operation in Figure 3a) on an operating table. We placed an ArUco marker to calibrate the configuration of the object (i.e., the mock-up of the heart). Once the image sensor passed the information to the Raven-II simulator, the operating table and the target object were rendered in 3D, as shown in Figure 3b. Note that the 3D axes were added to the image upon the algorithm’s detection of the marker. Using the predetermined transformation from the marker to the robot, *RVIZ* can place the virtual representation of the robot over the operating table (Figure 3c). For the demonstration of the smart malware, the mock-up heart was replaced with a simpler shape, i.e., a cuboid (see Figures 5 and 10–12). However, the performance measurements were not affected by the simplification of the shape of the object.

Systems setup. To demonstrate the smart malware, as depicted in Figure 4, we set up three Linux (Ubuntu 16.04) machines running ROS (i.e., Kinetic, one of the most recent versions of ROS). While the nodes for Raven-II and the rendering algorithm can be distributed across any combination of machines, we simplified the configuration by distributing the application across two machines (*tchaikovsky* and *pachelbel*, shown in Figure 4). The third machine (*cloud7*) is owned

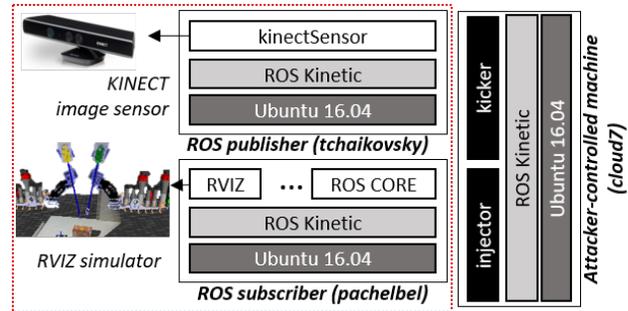


Figure 4: Overview of the system setup for the experiment.

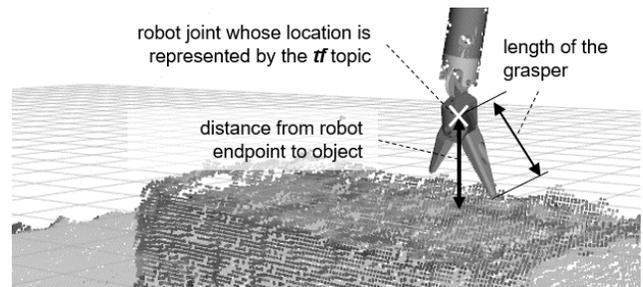


Figure 5: Evaluation of the distance from the robot arm to the object.

by the attacker. The two machines, both running Raven-II, reside in the same (virtual) network (marked with the dotted box in Figure 4), whereas *cloud7* resides in the same network only when executing the attack strategy (see Section 4.2). The rendering algorithm is designed to run with the (physical) Raven-II robot. However, we limited the experiment to a simulated environment to protect the physical robot from potential damage. Although the experiment was limited to a simulated environment, the faults and their impact still apply to the physical robot.

Data. With the MITM established (as described in Section 4), we were able to eavesdrop on all messages transmitted between the ROS nodes. Such messages included ones communicating the robot’s joints’ state (i.e., the angle of each joint), which determined the robot’s end-effector position. We took advantage of the ROS-provided API (i.e., `TransformListener()` [37]) to derive the position of each joint. Using the API, we (in the shoes of an attacker) could collect data on the x , y , and z coordinates of the robot end-effector in three-dimensional space. We considered three scenarios to mimic surgical operations of different levels of complexity: (i) a surgeon focuses on a single region of the target object; (ii) a surgeon operates on two regions of the target object; and (iii) a surgeon operates on three regions of the target object. Note that the movements of the robot arms (manipulated by the surgeon) follow a trajectory specific to a given surgical procedure. Because of our lack of real data on such trajectories, we imitated rather complex routes to challenge our algorithm.

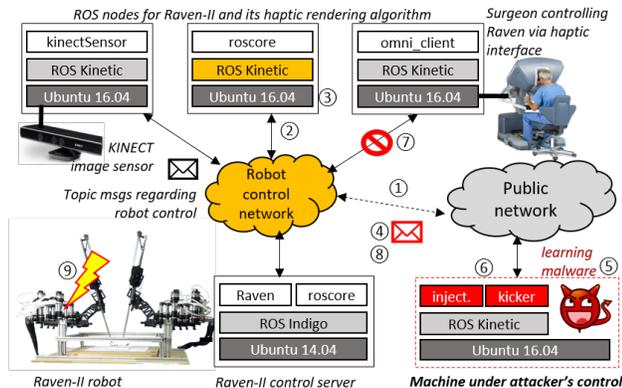


Figure 6: Overview of the steps taken by the malware.

Evaluation metrics. The goal of an attacker is to trigger the execution of the attack payload at the most opportune time so as to maximize the damage, e.g., hurt a patient or damage the robot. In order to achieve that objective, the attacker must precisely determine when the robot is operating near (if not in contact with) the target object. The clustering algorithm (in Section 4) indirectly derives the decision by monitoring the trace density of the robot arm in 3D space. To evaluate the effectiveness of the decision, we measured the distance from the robot arm to the object (see Figure 5). That would not be possible for a real attacker, as the location of the object would remain unknown. We evaluate the predictions by using a threshold (i.e., 10 mm) that defines “close to object.” Applying the definition, we derive the number of predicted instances that would lead to a successful attack. (We consider an attack to be “successful” if the execution is triggered when the distance from the robot arm to the object is less than the threshold).

Automated malware execution. In Figure 6, we show how an attack using our smart malware would proceed. The attacker starts by getting access to the control network of the robot (1). This step could be accomplished by scanning for the target ROS application connected to the public network [13], or stealing the credentials of a legitimate user in the control network (via social engineering or phishing attacks). With access to the control network, the attacker scans the network for the 11311 port to find the ROS master (2). In 3, the attacker checks the version of ROS and disables all patches that remediate the vulnerabilities of ROS 1. Next, the attacker can deploy the smart self-learning malware. First, the malware subscribes (4) to topics of interest (e.g., tf, a ROS-generic topic for the x, y, z coordinates of the robot end-point). By running the DBSCAN algorithm, the malware can label each point (i.e., member of a cluster or noise) (5). When the robot arm position is classified as a cluster, the malware triggers the payload execution (6), which registers the malicious publisher with the name of a genuine publisher. Because of the name conflict, in 7, the ROS master shuts

down the genuine publisher (i.e., `omni_client`) and the malicious topic (i.e., faulty data) is passed to the ROS application (8). Because of the faulty data, the operation of Raven is corrupted, which puts the patient at risk (9).

6 Results

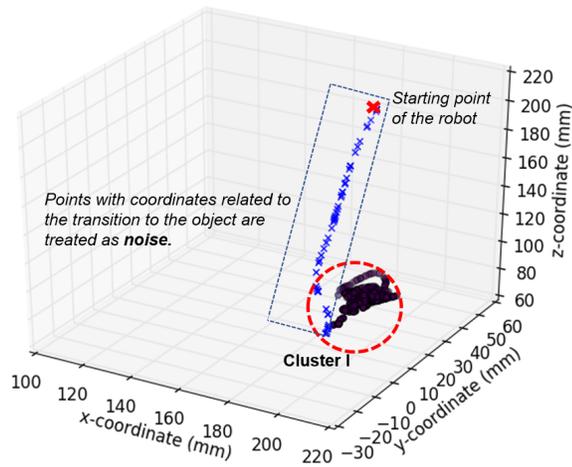
In this section, we present our results from inferring the time to trigger the attack payload and injecting realistic faults.

6.1 Determining attack triggers

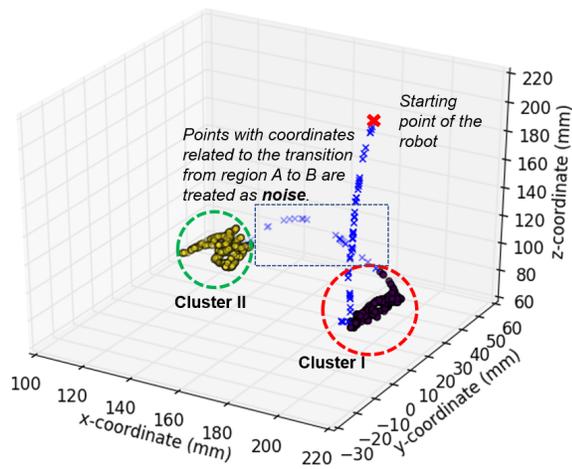
In this section, we evaluate our accuracy in determining the robot’s end-effector position with respect to the target object. In Figure 7, we present the results of the clustering algorithm (based on DBSCAN) for the three scenarios: (i) a surgeon operates on a single region of the target object; (ii) a surgeon operates on two regions of the target object; and (iii) a surgeon operates on three regions of the target object. Note that in Figures 7a–7c, an “x” indicates that the point is considered noise, and a circle indicates that the point belongs to a cluster. (Different colors are used to differentiate clusters.) Also, in Figure 5, we compare the clustering results with those from a pedal-detection-based approach (*pedal*) [2].

Case 1: Single region of operation. Figure 7a depicts the trajectory of the robot arm for the case in which a surgeon is operating at a single region of the object. The algorithm effectively identifies the data points that correspond to the region of operation and successfully filters out the data points related to the transition of the robot arm from the starting point of the robot arm to the region of operation. In Figure 8a, we present a cumulative distribution of the distance from the clustered points (robot’s joint positions) to the target object. While all points of the DBSCAN-derived clusters had a distance of less than 1 cm from the target object, the *pedal-detection-based algorithm* included points related to transition of the robot, which resulted in reduction of the probability of a successful attack. Also, as depicted in Figure 9, the algorithm effectively clusters the instances in which the robot is closer to the object (“clustered1” in Figure 9), as opposed to the points that were labeled as transitions to the object (“transition1” in Figure 9). For our algorithm, the distance (from the robot arm to the object) varied from 0.0 mm to 7.5 mm, and our clustering algorithm was able to filter out the points that corresponded to transitions from the starting point of the robot arm to operational regions (*cluster 1* in Figure 7a). The *pedal-detection-based approach* includes the starting point of the robot arm as a potential trigger for an attack. (Note that the starting point is 121 mm from the object.). As shown in Figure 8a, 99.9% of the DBSCAN-predicted triggers were within 7.1 mm of the object. However, for the *pedal* detection-based approach, only 80.3% of the predicted points were within 7.1 mm.

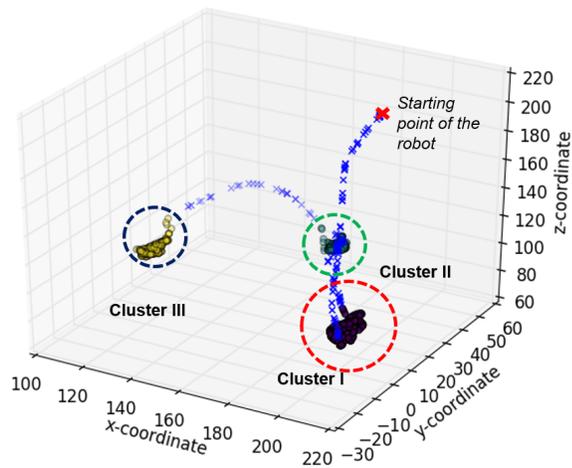
Case 2: Two regions of operation. As shown in Figure 7b, the algorithm successfully captured the two regions despite



(a) Single region of operation.

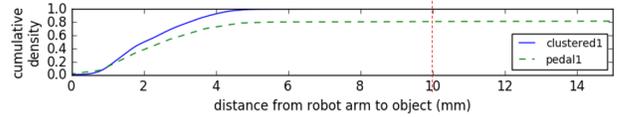


(b) Two regions of operation.

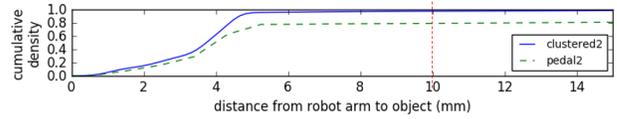


(c) Three regions of operation.

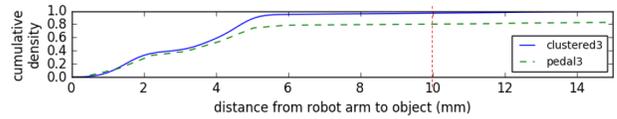
Figure 7: Results of tracing of the robot’s arm movements in three hypothetical surgical procedures.



(a) Single region of operation.



(b) Two regions of operation.



(c) Three regions of operation.

Figure 8: Cumulative distribution of the distance from the robot arm to the object.

the complexity added to the operation. While we have added an intermediate transition between the regions of operation, the algorithm successfully filtered out such transitions, and distinguished the two regions. As depicted in Figure 9 and Figure 8b, the clustering algorithm was able to find a subset that contained the majority of the points that were closest to the object (i.e., $0.26 \text{ mm} \leq \text{distance} \leq 20.44 \text{ mm}$).

Case 3: Three regions of operation. In Figure 7c, we present the case in which the surgery takes place in three adjacent regions. The algorithm successfully detected all three regions. Also, all points clustered by our algorithm turned out to be within 19.8 mm of the object (i.e., all points in the clusters had a $\text{distance} \leq 19.82 \text{ mm}$). For the pedal-detection-based algorithm, 19.2% of the points triggered unsuccessful attacks while with our DBSCAN-based approach, 3.23% would have been unsuccessful.

Discussion. Triggering when the instrument is in close proximity to the target object is essential to increasing the likelihood of success. As demonstrated in the experiments, our DBSCAN-based approach effectively predicts points that are close to the target object. As discussed in [53], the success of the DBSCAN algorithm is sensitive to the choice of the two parameters (i.e., ϵ and n). In this paper, we have taken a trial-and-error approach, which would not be feasible for an attacker with limited information. (I.e., the attacker cannot confirm whether the resulting cluster truly represents the region of interest.) Instead, the attacker can tune the learning algorithm (i.e., find the optimal parameters, ϵ and n) offline and install the malware with the parameters embedded.

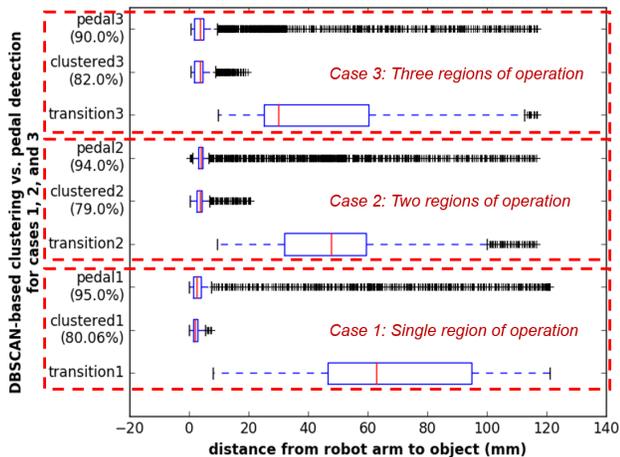


Figure 9: Distribution of the distances from the predicted clusters (predicted either by DBSCAN or by pedal detection) to the object. The labels “clustered” and “transition” indicate coordinates that were predicted as “surgical operation” and “transition of the robot”, while “pedal” is for the coordinates filtered by the approach in [2].

6.2 Impact of attacks on the Raven-II haptic feedback rendering algorithm

This section presents the impacts of executing the three attack payloads: (i) loss of granularity in the depth map, (ii) shifted depth map, and (iii) corrupted reference point.

In Figure 10, we present the result of dropping 90% of the pixels from the depth map. (Note that to maximize the visibility of the fault’s impact, we have chosen an extreme case and neutralized an unrealistically large portion of pixels.) As a result, the robot arm tip penetrated the surface of the object (Figure 10b), whereas the algorithm should have blocked it from doing so (as seen in Figure 10a). In reality, incorrect rendering of the force feedback can damage or endanger the underlying surface and make the robot suffer a heavy load.

In Figure 11, we show the impact of shifting depth map information during transmission of the information from the publisher to the subscriber. The figure shows that because of the shifted distance measure, the 3D rendering of the left half of the box is flattened (and indeed would be hard to differentiate from the surface, were it not for the colors), and the original surface on the right side of the box has gained volume. As shown in the figure, this fault model can lead to penetration of the object by the robot that would have been prevented by the non-corrupted image.

The last fault we studied was corruption of the derived reference point (the ArUco marker). The object in Figure 12a has the reference point set on the ArUco marker, whereas the reference point in Figure 12b has been shifted upward. The distance between the object and the robot arm (marked with a double-headed arrow) has been updated accordingly. (I.e., the

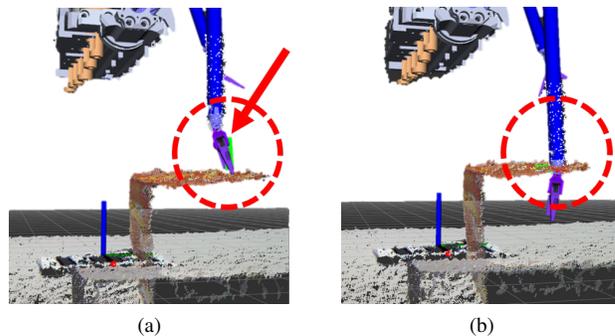


Figure 10: Simulated Raven operation with (a) uncorrupted depth map and (b) corrupted depth map. Note the difference between the dotted circles.

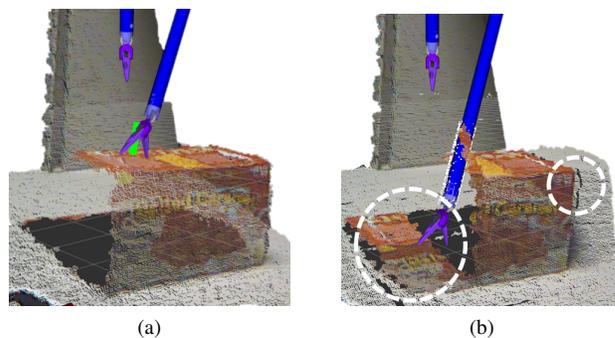


Figure 11: Simulated Raven operation with (a) uncorrupted depth map, and (b) shifted depth map. Note the problems inside the contents of dotted circles.

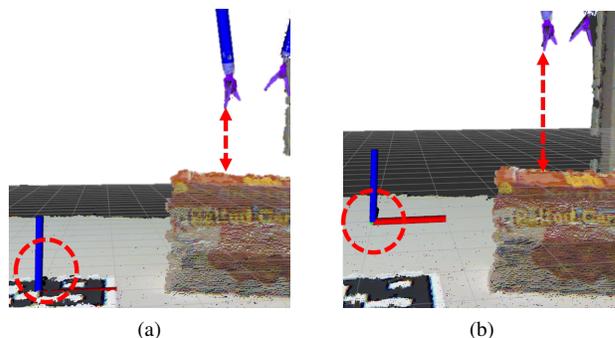


Figure 12: Simulated Raven operation with (a) uncorrupted camMsg, and (b) shifted camMsg. Note the difference between the reference points (inside dotted circles).

distance in (b) is larger than in (a), while (a) depicts the actual setup of the robot and the object.) For the corrupted distance, the rendered feedback is no longer valid. In the experiment shown in Figure 12, the haptic device did not receive haptic

feedback upon touching the surface of the object, and the robot penetrated the object.

7 Discussion

Generalization. As demonstrated in this paper, the ROS 1 is vulnerable to variations of MITM attacks. We show that our prototype smart malware can utilize the leaked data to trigger an attack at the most opportune time so as to maximize the attack's impact. The inference of the opportune time range for execution of the attack payload reduces the chances of exposure, which helps the malware disguise the attack as an accidental failure. (Recall that the faults were designed to represent accidental failures.) Without the smart triggering, frequent and likely unsuccessful injections of the fault could make the system administrator aware of the malicious intent behind the sequence of failures. To make things worse, our DBSCAN-based approach does not require extensive prior knowledge of the target robotic application. Alemzadeh et al. [2] also introduce a triggering algorithm (i.e., a side-channel attack that predicts the state of the robot from the byte values of specific packets), but we find that our approach is more intuitive and effective⁴. (I.e., our approach simply searches for a dense region (corresponding to a high density of critical activities), and that does not require background knowledge on the design/implementation details of the target robot.) However, despite the smartness of the malware, our attack is limited in its payload. Unlike the fault in [2], which tackles the time gap between a safety check of the input and its execution (i.e., faults are injected after the safety check), our faults (or faulty scenarios) are injected through corruption of the raw data, and that corruption might be detected by safety checks (if such checks were implemented as part of the robotic application).

While we demonstrated the feasibility of a smart malware attack in the context of the Raven-II surgical robot and its haptic feedback rendering algorithm, our threat model exploits vulnerabilities in the underlying framework (ROS). Hence, a robotic application running on the most common version of ROS, ROS 1, is vulnerable to MITM attacks and to smart malware that exploits the leaked data. Furthermore, the generic idea of malware driven by ML algorithms can be expanded to any computing infrastructure that generates a stream of data if the data contain actionable intelligence that smart malware can infer and the system has vulnerabilities that allow malicious entities to access data. While we leverage vulnerabilities in the ROS, as discussed in [1], various entry points exist through which malicious entities could intrude into robotic applications. By leveraging such vulnerabilities, our design for smart malware can be revised to target robotic applications in general.

⁴Please note that the goals of the two approaches were different. The goal of the approach in [2] is to infer the time when the corrupted data will be passed to the robot and update its state.

The work performed in this study is not intended to support hackers, but to proactively assess the resiliency of robotic applications, identify vulnerabilities in their design, and drive development of methods to harden robotic systems. For instance, the sensitivity of the haptic feedback rendering algorithm to its input data (as identified during our experiment) requires hardening of the data validation process. That hardening can be done by validating the publisher (to maintain the integrity of the data) or by deploying redundancy in the sensors.

7.1 Protection

The leakage of control data being transmitted between components of a robot can lead to inference of sensitive information that can threaten the operation of the robot. As a result, it is critical to secure the robot by (i) assuring that only authorized entities can control robot operation, (ii) securing the communications between the components of the application, and (iii) closely monitoring the robot for anomalies. In this section, we discuss technologies that can be used to secure the application, and their limitations. Also, we introduce our safety module, which detects abnormal circumstances and brings the robot to a predefined *safe state*.

In terms of computer security, MITM attacks have been well-studied, and a number of protection and detection methods have been introduced [11, 24, 32]. For instance, to prevent ARP poisoning (which is a critical step in performing an attack), each machine can have its ARP table set to be static, to prevent unknown entities from updating the table. Also, authentication of the nodes can prevent unauthorized entities from hijacking a session. (I.e., unauthorized ROS nodes should not be able to register to the ROS core, and entries that can publish/subscribe a topic should be defined.)

Security enhancements for ROS (SROS). To better secure the communications within the robotics applications, SROS [39, 55] provides TLS support in the socket-level transport. However, the current distribution of SROS is limited to TCPROS (not UDPROS) and robotic applications written in Python (not C++ or Java). As an alternative, one can add a layer of authentication by running all ROS nodes within a VPN (which would require authentication), and that approach is already common. However, it is not rare for malicious entities to intrude into a protected network by using weak or stolen credentials.

Secure ROS. Unlike the well-studied TCP MITM attack, our ROS-specific attack model has not been well-investigated. Fortunately, a “fork” of the core ROS packages was released to enable secure communication in the ROS applications [51]. The “Secure ROS” introduced a new configuration file, which specifies the configuration of the application. Furthermore, by utilizing IPsec, Secure ROS ensures that the IP packets cannot be tampered with or spoofed. While Secure ROS enhances the security of ROS applications, the neutralization of the patch

Packet bytes	Bytes in ASCII
3e 0a 3c 6d 65 74 68 6f 64 4e 61 6d 65 3e 73 68	>.<metho dName>sh
75 74 64 6f 77 6e 3c 2f 6d 65 74 68 6f 64 4e 61	utdown</ methodNa
6d 65 3e 0a 3c 70 61 72 61 6d 73 3e 0a 3c 70 61	me>.<par am>-<pa
72 61 6d 3e 0a 3c 76 61 6c 75 65 3e 3c 73 74 72	ram>.<va lue><str
69 6e 67 3e 2f 6d 61 73 74 65 72 3c 2f 73 74 72	ing>/mas ter</str
69 6e 67 3e 3c 2f 76 61 6c 75 65 3e 0a 3c 2f 70	ing></va lue>.</p
61 72 61 6d 3e 0a 3c 70 61 72 61 6d 3e 0a 3c 76	aram>.<sp _aram>.<v
61 6c 75 65 3e 3c 73 74 72 69 6e 67 3e 6e 65 77	alue><st ring>new
20 6e 6f 64 65 20 72 65 67 69 73 74 65 72 65 64	node re gistered
20 77 69 74 68 20 73 61 6d 65 20 6e 61 6d 65 3c	with sa me name<
2f 73 74 72 69 6e 67 3e 3c 2f 76 61 6c 75 65 3e	/string> </value>

Figure 13: Packet capture of the network packet carrying the “shutdown” command from the roscore, triggered by a conflict in node names.

is not particularly difficult. As the package is an addition to an already existing ROS installation in the system, it has a single parameter (defined in a bash script) that enables the patch. As a result, an attacker can overwrite the parameter to disable the entire security patch, thus returning the system back to the vulnerable ROS.

ROS 2. ROS 2 (released in 2015) [36] is a major update from the original ROS. On top of introducing new features to address challenges that arose from extended usage of the framework (e.g., with real-time systems, or groups of robots), the upgrade also covers the vulnerabilities discussed in this paper. The ROS 2 uses the data distribution service (DDS) [34] for publish-subscribe transport. The security enhancements provided by the Secure ROS and SROS patch can be embedded in ROS 2 using DDS. Hence, the vulnerabilities exploited in this paper can be eliminated by using the upgraded framework. However, ROS 2 is not backward-compatible. As a result, the existing applications must be rewritten to take advantage of the new features in ROS 2, and such rewriting is not trivial. Even if the developer manages to re-program the robotic application with the new interface, configuring of the DDS to support the security needs is left to the robot’s programmers. Such configuration requires thorough understanding of encryption, certification, and access control. Without security in mind, if the programmer decides not to enable the security features of DDS (e.g., using the eProxima Fast RTPS middleware [18]) or if the programmer makes a mistake in configuring the DDS [16], the vulnerabilities discussed in this paper remain current. As discussed in [27], the DDS and its implementation have limitations (e.g., lack of forward security) and vulnerabilities (e.g., skipping of variable initialization) that attackers can exploit.

7.2 Safety module

The limitations of existing technologies mean that the threats described in this paper (especially that specific to ROS applications) remain current and, hence, can occur in any of the 125+ existing robots. As a result, we find a need for a safety module that can (i) detect abnormalities in the robot and (ii) take control of the robot and bring it to a safe state under such circumstances. As discussed in Section 4, when

the ROS master detects a name conflict (due to a new node’s registering of itself with a name already in use), it terminates the original node and registers the new node with the name in conflict. Our safety module detects the *shutdown* signal transmitted over the network (see Figure 13). Upon detecting the shutdown signal due to a name conflict (see Figure 13), our safety module (operating as another ROS node) terminates the node that publishes the state of the robot joints. By terminating the node, the safety module can prevent the malicious entity from taking control of the robot. Furthermore, if needed, the safety module can bring the robot to a predefined safe state (in our experiment, the reset position of the robot). As the safety module runs on the ROS master with privilege, we can be assured that the shutdown packet will be detected, with minimal risk that the attacker can corrupt the detection. Also, as the detection relies on an unusual signature, we can minimize false positives. (I.e., name conflict is a rare event and when there are multiple nodes with identical names by design, the programmer enables the *anonymous* mode, which pads a hash to the name to avoid the conflict.)

8 Related work

Attacks with learning features. In [41], an open-source hacking AI, DeepHack, was presented. Powered with a neural network, the tool learns how to intrude into web applications. DeepLocker [28], on the other hand, takes advantage of a deep neural network for target detection. Until the malware detects the target, the malware disguises itself as benign software. Furthermore, the malware encrypts the payload to conceal the malicious intent, which makes reverse-engineering challenging. In [10], the authors leverage a learning technique to infer an attack payload from CPS operational data. The malware in [10] predicts failure-causing abnormalities in the CPS operational data, and injects abnormalities into the control data to corrupt the operation of the CPS.

Attacks against ROS. Some vulnerabilities of ROS were discussed in [42, 43]. Using the STOP surveillance system, the authors demonstrate an attack that changes the route of the patrol robot. They proposed the use of IPSec, which was indeed incorporated in the upgrade to ROS 2. The whitelist method proposed by Dóczy et al. [17] has also become part of the new ROS framework. Similarly, [15] discusses a method for preventing malicious publishers and subscribers from interfering with a given ROS node network. The authors ensured broadcast encryption by whitelisting nodes in an authentication server and by requiring any new publisher to run an authentication to certify itself as a legitimate new publisher. Despite the efforts to secure ROS applications, as demonstrated in [13], a significant number of ROS applications that are connected to networks are vulnerable. The authors of [13], by scanning over the whole IPV4 address space, identified more than 100 hosts running as ROS masters. Also, in [5], the authors demonstrate that ROS applications can be vulnerable

to attacks that modify the instructions of a Raven operator (e.g., by manipulating packets to cause loss, reordering, or delay of commands) and to session-hijacking attacks.

9 Conclusions

In this paper, we studied the impact of security attacks that exploit security vulnerabilities in ROS to attack robotic applications. More specifically, we demonstrated (i) the possibility of neutralizing the force feedback engine in Raven-II by corrupting a message passed across ROS nodes over the network, and (ii) the possibility of misleading the robot operator by providing incorrect feedback. Our study of ROS and observations on the impact of security attacks reveal a need for advanced security APIs to be provided by the framework. We suggest that the applications be secured in the implementation phase, and be enforced by the framework.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 18-16673 and 15-45069. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- [1] Homa Alemzadeh. *Data-driven Resiliency Assessment of Medical Cyber-physical Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2016.
- [2] Homa Alemzadeh, Daniel Chen, Xiao Li, Thenkurussi Kesavadas, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Targeted Attacks on Teleoperated Surgical Robots: Dynamic Model-based Detection and Mitigation. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 395–406, 2016.
- [3] Applied Dexterity Website. Accessed May 1, 2019. <http://applieddexterity.com>.
- [4] Rachad Atat, Lingjia Liu, Jinsong Wu, Guangyu Li, Chunxuan Ye, and Yang Yi. Big Data Meet Cyber-Physical Systems: A Panoramic Survey. *CoRR*, abs/1810.12399, 2018.
- [5] Tamara Bonaci, Jeffrey Herron, Tariq Yusuf, Junjie Yan, Tadayoshi Kohno, and Howard Jay Chizeck. To Make a Robot Secure: An Experimental Analysis of Cyber Security Threats against Teleoperated Surgical Robots. *arXiv preprint arXiv:1504.04339*, 2015. <https://arxiv.org/abs/1504.04339>.
- [6] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O’Reilly Media, Inc., 2008.
- [7] Charles Chang, Zoe Steinberg, Anup Shah, and Mohan S. Gundeti. Patient Positioning and Port Placement for Robot-assisted Surgery. *Journal of Endourology*, 28(6):631–638, 2014.
- [8] Yizheng Chen, Yacin Nadji, Athanasios Kountouras, Fabian Monrose, Roberto Perdisci, Manos Antonakakis, and Nikolaos Vasiloglou. Practical Attacks against Graph-based Clustering. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1125–1142. ACM, 2017.
- [9] Xiuzhen Cheng, Yunchuan Sun, Antonio Jara, Houbing Song, and Yingjie Tian. Big Data and Knowledge Extraction for Cyber-Physical Systems. *International Journal of Distributed Sensor Networks*, 11(9):231527, 2015.
- [10] Keywhan Chung, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Availability Attacks on Computing Systems through Alteration of Environmental Control: Smart Malware Approach. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 1–12. ACM, 2019.
- [11] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A Survey of Man in the Middle Attacks. *IEEE Communications Surveys & Tutorials*, 18(3):2027–2051, 2016.
- [12] Altair da Silva Costa Jr. Assessment of Operative Times of Multiple Surgical Specialties in a Public University Hospital. *Einstein (São Paulo)*, 15(2):200–205, 2017.
- [13] Nicholas DeMarinis, Stefanie Tellex, Vasileios Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the Internet for ROS: A View of Security in Robotics Research. *arXiv preprint arXiv:1808.03322*, 2018. <https://arxiv.org/abs/1808.03322>.
- [14] Department of Homeland Security. Cyber Physical Systems Security, Feb. 2019. <https://www.dhs.gov/science-and-technology/csd-cpssec>.
- [15] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. Application-level Security for ROS-based Applications. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4477–4482, 2016.
- [16] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1272–1289, New York, NY, USA, 2018. ACM.

- [17] Roland Dóczy, Ferenc Kis, Balázs Sütő, Valéria Póser, Gernot Kronreif, Eszter Jósmai, and Miklos Kozlovsky. Increasing ROS 1.x Communication Security for Medical Surgery Robot. In *Proc. IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 4444–4449, 2016.
- [18] eProsima. eProsima Fast RTPS, Mar. 2019. <https://www.eprosima.com/index.php/products-all/eprosima-fast-rtps>.
- [19] Richard H. Epstein and Franklin Dexter. Influence of Supervision Ratios by Anesthesiologists on First-case Starts and Critical Portions of Anesthetics. *Anesthesiology: The Journal of the American Society of Anesthesiologists*, 116(3):683–691, 2012.
- [20] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.
- [21] Sergio Garrido-Jurado, Rafael Muñoz Salinas, Francisco J. Madrid-Cuevas, and Rafael Medina-Carnicer. Generation of Fiducial Marker Dictionaries Using Mixed Integer Linear Programming. *Pattern Recognition*, 51(C):481–491, March 2016.
- [22] Martin Giles. Triton is the World’s Most Murderous Malware, and It’s Spreading. *MIT Technology Review*, March 5, 2019. <https://www.technologyreview.com/s/613054/cybersecurity-critical-infrastructure-triton-malware/>.
- [23] Megan Henney. Amazon Bear Repellent Accident this Week Wasn’t Its First. *FOX Business*, December 7, 2018. <https://www.foxbusiness.com/retail/amazon-bear-repellent-accident-this-week-wasnt-its-first>.
- [24] Brian Hernacki and William E. Sobel. Detecting Man-in-the-middle Attacks via Security Transitions, October 15, 2013. US Patent 8,561,181.
- [25] 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*, pages 43–58. ACM, 2011.
- [26] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Proceedings of the Annual Network and Distributed System Security Symposium*, 2012. <https://www.ndss-symposium.org/ndss2012/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation>.
- [27] Jongkil Kim, Jonathon M. Smereka, Calvin Cheung, Surya Nepal, and Marthie Grobler. Security and Performance Considerations in ROS 2: A Balancing Act. *CoRR*, abs/1809.09566, 2018. <http://arxiv.org/abs/1809.09566>.
- [28] Dhilung Kirat, Jiyoung Jang, and Marc Ph. Stoecklin. DeepLocker: Concealing Targeted Attacks with AI Locksmithing. Black Hat USA, <https://i.blackhat.com/us-18/Thu-August-9/us-18-Kirat-DeepLocker-Concealing-Targeted-Attacks-with-AI-Locksmithing.pdf>, 2018.
- [29] Laparoscopic.MD. Laparoscopic Trocars, Feb. 2019. <https://www.laparoscopic.md/surgery/instruments/trocar>.
- [30] Xiao Li and Thenkurussi Kesavadas. Surgical Robot with Environment Reconstruction and Force Feedback. In *Proceedings of the 40th Annual International Conference of the IEEE Medicine and Biology Society*, pages 1861–1866, July 2018.
- [31] Santiago Morante, Juan G. Victores, and Carlos Balaguer. Cryptobotics: Why Robots Need Cyber Safety. *Frontiers in Robotics and AI*, 2:23, 2015. <https://www.frontiersin.org/article/10.3389/frobt.2015.00023>.
- [32] Seung Yeob Nam, Dongwon Kim, and Jeongeun Kim. Enhanced ARP: Preventing ARP Poisoning-based Man-in-the-middle Attacks. *IEEE Communications Letters*, 14(2):187–189, 2010.
- [33] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.
- [34] Object Management Group. DDS Portal - Data Distribution Services, 2018. <https://www.omgwiki.org/dds/>.
- [35] Occipital. OpenNI 2 Downloads and Documentation, Mar. 2019. <https://structure.io/openni>.
- [36] Open Robotics. ROS2 Overview, 2019. <https://index.ros.org/doc/ros2/>.
- [37] Open Source Robotics Foundation. APIs - ROS Wiki, 2016. <http://wiki.ros.org/APIs>.

- [38] Open Source Robotics Foundation. Robots That You Can Use with ROS, 2018. <https://robots.ros.org/>.
- [39] Open Source Robotics Foundation. SROS, 2018. <http://wiki.ros.org/SROS>.
- [40] Sean Palka and Damon McCoy. Fuzzing E-mail Filters with Generative Grammars and N-Gram Analysis. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., 2015. USENIX Association. <https://www.usenix.org/conference/woot15/workshop-program/presentation/palka>.
- [41] Dan Petro and Ben Morris. Weaponizing Machine Learning: Humanity Was Overrated Anyway. DEF CON, <https://www.defcon.org/html/defcon-25/dc-25-speakers.html#Petro>, 2017.
- [42] David Portugal, Samuel Pereira, and Micael Couceiro. The Role of Security in Human-robot Shared Environments: A Case Study in ROS-based Surveillance Robots. In *Proc. 26th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 981–986, Aug. 2017.
- [43] David Portugal, Miguel Santos, Samuel Pereira, and Micael Couceiro. On the Security of Robotic Applications using ROS. In Roman V. Yampolskiy, editor, *Artificial Intelligence Safety and Security*, pages 279–289. Taylor & Francis, 2018.
- [44] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: An Open-source Robot Operating System. In *Proceedings of the ICRA Workshop on Open Source Software*. IEEE, 2009.
- [45] Erwin Quiring and Konrad Rieck. Adversarial Machine Learning against Digital Watermarking. In *Proceedings of the 26th European Signal Processing Conference (EUSIPCO)*, pages 519–523, Sep. 2018.
- [46] Francisco J. Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. Speeded Up Detection of Squared Fiducial Markers. *Image and Vision Computing*, 76:38–47, 2018.
- [47] 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. <http://arxiv.org/abs/1808.05665>.
- [48] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Systems*, 42(3):19:1–19:21, July 2017.
- [49] Senhance. The Senhance Surgical System, 2018. <https://www.senhance.com/us/digital-laparoscopy>.
- [50] Aashish Sharma, Zbigniew Kalbarczyk, James Barlow, and Ravishankar Iyer. Analysis of Security Data from a Large Computing Organization. In *Proc. 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 506–517. IEEE, 2011.
- [51] Aravind Sundaresan, Leonard Gerard, and Minyoung Kim. Secure ROS. Computer Science Laboratory, SRI International, <http://secure-ros.csl.sri.com/>, 2017.
- [52] Intuitive Surgical. Da Vinci by Intuitive: Enabling Surgical Care to Get Patients back to What Matters, 2019. <https://www.intuitive.com/en-us/products-and-services/da-vinci/>.
- [53] Thanh N. Tran, Klaudia Drab, and Michal Daszykowski. Revised DBSCAN Algorithm to Cluster Data with Dense Adjacent Clusters. *Chemometrics and Intelligent Laboratory Systems*, 120:92–96, 2013.
- [54] Jeffrey Voas, Rick Kuhn, Constantinos Kolias, Angelos Stavrou, and Georgios Kambourakis. Cybertrust in the IoT Age. *Computer*, 51(7):12–15, July 2018.
- [55] Ruffin White, Henrik I. Christensen, and Morgan Quigley. SROS: Securing ROS over the Wire, in the Graph, and Through the Kernel. *CoRR*, abs/1611.07060, 2016. <http://arxiv.org/abs/1611.07060>.

SGXJail: Defeating Enclave Malware via Confinement

Samuel Weiser, Luca Mayr, Michael Schwarz, Daniel Gruss
Graz University of Technology

Abstract

Trusted execution environments, such as Intel SGX, allow executing enclaves shielded from the rest of the system. This fosters new application scenarios not only in cloud settings but also for securing various types of end-user applications. However, with these technologies new threats emerged. Due to the strong isolation guarantees of SGX, enclaves can effectively hide malicious payload from antivirus software. Were these scenarios already outlined years ago, we are evidencing functional attacks in the recent past. Unfortunately, no reasonable defense against enclave malware has been proposed.

In this work, we present the first practical defense mechanism protecting against various types of enclave misbehavior. By studying known and future attack vectors we identified the root cause for the enclave malware threat as a too permissive host interface for SGX enclaves, leading to a dangerous asymmetry between enclaves and applications. To overcome this asymmetry, we design SGXJail, an enclave compartmentalization mechanism making use of flexible memory access policies. SGXJail effectively defeats a wide range of enclave malware threats while at the same time being compatible with existing enclave infrastructure. Our proof-of-concept software implementation confirms the efficiency of SGXJail on commodity systems. We furthermore present slight extensions to the SGX specification, which allow for even more efficient enclave compartmentalization by leveraging Intel memory protection keys. Apart from defeating enclave malware, SGXJail enables new use cases beyond the original SGX threat model. We envision SGXJail not only for site isolation in modern browsers, *i.e.*, confining different browser tabs but also for third-party plugin or library management.

1 Introduction

Isolation is an essential element of modern computer systems. Traditionally, the operating system was responsible for isolating processes. With the emergence of various novel use cases, further isolation became necessary. For instance, executing untrusted JavaScript code demands isolation from the

browser via sandboxing. Also, mutually untrusted services in the cloud, e.g., from different tenants, run in different containers or virtual machines. In any case, it is still necessary to trust system administrators, operating systems, and hypervisors.

Intel addressed this problem with SGX. Intel SGX can be used to isolate software modules via hardware protected enclaves from a compromised or malicious administrator, operating system, or hypervisor. The trust anchor in SGX is only the processor. Even if any other system part is manipulated or compromised, SGX maintains its security guarantees. This enables new use cases, such as trusted cloud computing, where tenants do not only distrust the other tenants, but also the cloud provider and its hardware and software infrastructure [3, 17, 50]. A similar distrust also exists when protecting copyrighted material [2] or cryptographic or security-critical secrets [30, 35, 42] on a compromised user PC or server.

While isolation techniques such as SGX can be an excellent tool for security, they can also be misused for hiding malicious activity inside an enclave. In the recent past, we have seen not only enclave malware exploiting side channels [54] but also enclave ransomware and shellcode [38], however, with the help of a colluding host application. Recent research showed that enclaves can effectively hijack and impersonate any benign host application [53], opening up enclaves for various types of userspace malware. This confirms what researchers already suspected years ago [13, 16, 48]. Having witnessed first proof-of-concept attacks [38, 53], we can expect that more sophisticated and real-world attacks will appear in the future. Hence, it is necessary to providently explore the defense space, before real-world attacks are discovered.

Unfortunately, little is known about how to address this emerging threat properly. While conventional programs can be scanned for misbehavior by anti-virus technology, SGX is a complete game changer when it comes to enclave analysis. On the one hand, SGX prevents runtime inspection of enclaves. On the other hand, SGX allows lazy loading of malicious enclave content at runtime. Thus, malware infection can be completely decoupled from enclave distribution and installation, which renders all static analysis techniques on

the enclave void. In other words, SGX is a viable alternative to malware obfuscation and analysis evasion techniques. If Intel chose to allow certified anti-virus software to inspect enclaves, this would undermine essential security guarantees and is in fundamental conflict with the very goal SGX has [48]. Others proposed to detect enclave malware via their I/O behavior [13, 16], which is prone to both false positives and false negatives. Moreover, tracing and analyzing all enclave I/O behavior is believed infeasible in practice [38]. Others proposed to embed malware analysis code within the enclave itself, which raises several questions regarding its practicality [13]. Consequently,

“[...] the release and adoption of SGX-protected enclaves is likely to require a completely new approach to protecting our machines from the very malware SGX was designed to prevent.” [16]

So far, no practical defense against enclave malware exists.

In this work, we propose the first practical defense mechanism against enclave malware. To do so, we analyze enclave primitives and their resulting attack vectors and identify the root cause for the enclave malware threat as a too permissive feature set available to enclaves, forcing applications to trust any enclaves they host blindly. Consequently, a proper defense mechanism should give applications means to confine enclave operation to a clearly specified interface. To that end, we propose SGXJail, a lightweight yet effective measure to establish mutual distrust between enclaves and its host application. SGXJail does so by confining enclave operation to a clearly defined set of memory pages. This mitigates entire classes of runtime attacks (ROP, JOP, DOP, etc.) from the enclave to the host and enables reasoning about enclave misbehavior purely based on the legitimate communication interface. We instantiate SGXJail using process sandboxing and syscall filters and demonstrate its efficiency. Furthermore, we propose HSGXJail, a minimal hardware extension to the SGX specification making use of Intel memory protection keys to confine enclave execution, which is even more efficient. (H)SGXJail is opt-in, works on unmodified enclaves and can be easily integrated with the SGX software development kit¹.

With SGXJail, we expand possible SGX use cases beyond isolated execution. We envision modern software which is additionally hardened using SGXJail against potentially malicious or misbehaving third-party code. This is, for example, vital for all software enabling third-party plugins and add-ons, such as browsers, mail clients, or password managers.

Contributions. We summarize our contributions as follows.

1. We systematically break down the enclave malware threat and identify a number of enclave malware primitives.
2. We devise SGXJail, the first practical defense against enclave malware.
3. We implement and evaluate SGXJail in software.
4. We propose highly efficient HSGXJail via minimal hardware changes.

¹The code is available at <https://github.com/IAIK/sgxjail>

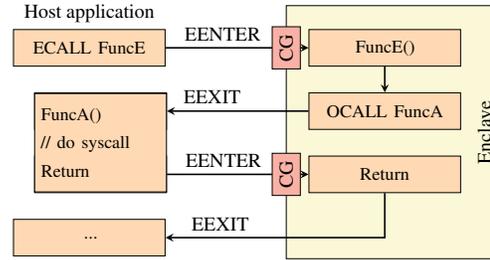


Figure 1: SGX enclaves are tightly integrated in a host application. The application can invoke the enclave via ECALLs while the enclave can perform OCALLs. Enclaves can only be entered via the EENTER instruction at certain call gates (CG) and can only be left via EEXIT.

The rest of the paper is organized as follows. Section 2 provides background information. Section 3 describes the threat model. Section 4 analyzes various enclave primitives and attack vectors. Section 5 presents (H)SGXJail. Section 6 discusses related work. We summarize our discussion of enclave malware in Section 7 and conclude in Section 8.

2 Background

In this section, we provide background on Intel SGX as well as runtime attacks.

2.1 Intel SGX

Intel Software Guard Extensions (SGX) are an instruction-set extension introduced with the Skylake microarchitecture [26]. SGX allows creating so-called *enclaves* running trusted code isolated from the remaining system.

Enclaves are hosted by an ordinary application process. Although the enclave and the host application reside in the same virtual address space, the address range of the enclave is inaccessible to the host application. Only the enclave itself can access its memory while the hardware prevents any other access to enclave memory. However, the enclave can access the entire virtual address space of the host application, allowing to share data between the enclave and the host application. This asymmetry in access permissions fits the original threat model of SGX but gives rise to enclave malware.

The host application is responsible for loading the enclave into the current address space and providing an interface through which the enclave communicates with the outer world. The CPU measures the loading process to ensure the integrity of the loaded enclave. The enclave is only executed if the resulting measurement matches a developer-specified value.

Figure 1 shows the process of invoking an enclave. The enclave defines secure functions denoted as ECALLs, which the application can call with the EENTER instruction. Call gates (CG) restrict enclave invocation to valid entry points.

Enclaves can request OS services such as syscalls via so-called OCALLs. To leave the enclave, an enclave can issue the `EXIT` instruction. Enclave developers need to specify the ECALL/OCALL interface via a so-called Enclave Definition Language (EDL). Each enclave is shipped with its own EDL file. An EDL file roughly contains function signatures of the enclave's ECALLs and OCALLs, augmented with additional security attributes (e.g., `in`, `out`). Intel provides developers with an SDK [25] that automatically generates glue code from the EDL file with appropriate parameter validation and buffer copying inside the enclave.

SGX assumes that all non-enclave code (*i.e.*, operating system and host application) is untrusted. SGX provides no means to protect applications from misbehaving enclaves. Instead, an enclave can access all application memory and divert control flow to arbitrary application code via `EXIT`.

2.2 Runtime Attacks

While it is typically not possible to directly inject or modify code at runtime, e.g., via a buffer overflow, an attacker can often manipulate control data and thus change the control flow of an application. By overwriting a code pointer, an attacker can divert the control flow to existing code snippets, resulting in so-called control-flow hijacking attacks. One of the most generic and powerful attacks is return-oriented programming (ROP) [55] which overwrites return addresses to create arbitrary attack payloads. Similar attacks exist for overwriting function pointers [5, 9, 10, 18, 32, 51] or signal handlers [6].

Some widely deployed techniques against code-reuse attacks are address-space layout randomization (ASLR) [45], stack canaries [14, 46] and shadow stacks [12]. While stronger control-flow integrity (CFI) [1, 31] can eradicate control-flow attacks, they leave data-only attacks [8, 28] unaddressed.

3 Threat Model

In this section, we first outline various application scenarios of SGX and argue why the original SGX threat model does not properly address enclave misbehavior. We then present our extended SGX threat model addressing enclave malware.

Scenario A. In the near future, SGX technology will likely permeate consumer systems and create diverse and many-faceted trust relations. Multiple independent software vendors (ISV) can use SGX for mutually protecting their proprietary library code (e.g., multimedia codecs, classification algorithms) or sensitive customer data (e.g., user passwords, encryption keys or bitcoin wallets) inside third-party enclaves. Applications can embed such third-party enclaves to leverage their functionality.

In this scenario, an attacker develops innocent-looking enclave malware (e.g., disguised as browser plugins) and distributes it as a third-party enclave via existing software stores or repositories. A user installs those third-party enclaves

alongside other applications. The attacker defers installation of malicious payload to runtime via a generic loader [48]. Hence, neither the maintainers of software repositories nor the user can detect this malware before it is actually triggered.

This might not only be invasive malware like ransomware, bots, or rootkits. A malicious enclave can also stealthily collect data about the user and host system without the user knowing, and with plausible deniability for the developer. An enclave developer can then monetize this data, e.g., by selling it to advertising agencies.

Scenario B. As more software is moved into enclaves, chances increase for exploitable vulnerabilities within enclave code. Enclaves are equipped with increasingly complex software, such as fully-fledged TLS stacks [24]. Thus, it is just a matter of time for bugs in the trusted code of enclaves, enabling well-known memory corruption attacks [56] inside enclaves. In fact, it has already been shown that enclaves are prone to such attacks [33, 52, 60]. This can be used to infiltrate trusted enclaves with a malicious payload.

A Holistic Threat Model. The original threat model of Intel SGX considers all non-enclave code as untrusted, including application code hosting enclaves (cf. Section 2). This model might be well-suited from an enclave's perspective. However, it does not fit more advanced application scenarios outlined before, leaving applications completely unprotected against misbehaving third-party enclaves. This creates a dangerous asymmetry, as also outlined by Schwarz et al. [53].

In this work, we introduce a more holistic threat model which does not violate the original threat model of SGX but augments it to explicitly address misbehaving enclaves. We consider a commodity system running software from various independent software vendors. On the one hand, third-party library vendors protect their secret data (e.g., cryptographic keys or intellectual property) inside enclaves. On the other hand, application developers include third-party enclaves in their applications for implementing various tasks. However, they want some form of assurance that third-party enclaves are well-behaving, for the reasons outlined before. While from an enclave vendor's perspective SGX provides strong protection against other enclaves as well as compromised systems, application developers have no means to assure themselves of proper behavior of (third-party) enclaves they use.

From a user's perspective, the computer (including the operating system and certain applications) are trusted. A mechanism is needed to protect applications (and, subsequently the computer) from potential enclave misbehavior, even if such enclaves are fully controlled by a dedicated attacker (e.g., enclave malware). In particular, an application needs protection against any inspection or alteration of its state (memory, CPU registers) by enclaves, apart from what it is exposing to the enclave via the ECALL/OCALL interface. SGXJail does not prevent API attacks, exploiting too permissive OCALLs or badly designed interfaces (e.g., avoiding Iago attacks and confused deputy attacks), which is a separate, yet important

Table 1: Enclave primitives leading to various attack vectors on the host application.

Attack \ Requirement	Arbitrary Read	Arbitrary Write	Arbitrary EEXIT
Information disclosure	✓	✗	✗
Control-flow attacks	(✓)	✓	(✓)
Data-only attacks	(✓)	✓	✗

line of research, as we discuss later. Also, this work does not focus on microarchitectural side channels, although SGXJail prevents certain classes of side-channel attacks. Finally, the CPU hardware is considered trusted.

4 Analyzing the Enclave Malware Threat

In this section, we analyze enclave primitives leading to different attack vectors violating memory safety of the application. This helps us design a proper defense mechanism and solve the enclave malware threat at the level of memory safety, leaving only high-level API attacks as a resort for the attacker, as discussed at the end of this section.

4.1 Enclave Primitives

Intel SGX entrusts enclaves with powerful primitives leading to different attacks violating memory safety, as depicted in Table 1. We outline these primitives in the following.

Arbitrary read. An enclave can read arbitrary memory of the host application. This is intended for exchanging data between enclave and host. Furthermore, an enclave can use hardware transactions to suppress exceptions stemming from reading inaccessible memory [53], giving a powerful fault-resistant arbitrary read primitive.

Arbitrary write. An enclave can write arbitrary writable host memory, which is intended for data exchange. Furthermore, it can use hardware transactions to suppress exceptions while writing inaccessible or non-writable memory [53], yielding a fault-resistant arbitrary write primitive.

Arbitrary EEXIT. An enclave can choose the precise code location in the application where execution shall continue after leaving enclave execution via the EEXIT instruction. Moreover, the enclave has control over many CPU registers immediately after an EEXIT, in particular the stack pointer, which gives enclaves the possibility to configure the application’s CPU state before resuming application execution.

4.2 Attack Vectors

Given the above primitives, a malicious enclave can mount a broad range of attacks violating memory safety of the host application. In the following, we cluster them into information disclosure, control-flow attacks as well as data-only attacks and give representative examples of these attacks. A detailed

overview of attacks violating memory safety was presented by Szekeres et al. [56].

4.2.1 Information disclosure

A malicious enclave can use the arbitrary read primitive to exfiltrate sensitive user data like cryptographic keys or passwords from the host application. Even if the application contains no such user secrets, an enclave can disclose other sensitive information, e.g., as used in various runtime protection mechanisms. For example, an enclave can derandomize application protection schemes like ASLR [45], stack canaries [14], code randomization [44] or randomization-based control-flow integrity schemes [31, 39]. The enclave can furthermore disclose the host application’s codebase and, subsequently, generate targeted exploitation payload like ROP chains on the fly. Thus, information disclosure is a powerful tool often used for subsequent exploitation.

4.2.2 Control-flow attacks

A malicious enclave can deliberately tamper with the application’s control flow in several ways. For example, it can directly corrupt code pointers, use rogue EEXITs and bypass various mitigation mechanisms.

Code pointer corruption. An enclave can manipulate an arbitrary code pointer of the host using the write primitive. This can be, e.g., return addresses on the stack or virtual function pointers on the heap. As soon as the application fetches a corrupted code pointer, execution is diverted to an attacker-chosen address. By carefully crafting a so-called ROP chain (cf. Section 2) and diverting execution to it, the attacker can gain arbitrary code execution with the privileges of the application, allowing to execute arbitrary syscalls in lieu of the application. To prepare a ROP chain, the enclave scans the host application for ROP gadgets using the arbitrary read primitive and writes the corresponding addresses on a fake stack using the arbitrary write primitive [53].

An enclave is by no means restricted to ROP attacks only. Similar to ROP, it can craft jump-oriented programming (JOP) attacks, loop-oriented programming (LOP) attacks, or call-oriented programming (COP) by overwriting indirect function pointers [5, 9, 10, 18, 32]. COOP attacks are also possible by overwriting virtual function pointers in C++ applications [51] or SROP attacks [6], faking a signal handler.

Rogue EEXIT. A malicious enclave can also mount control-flow attacks without corrupting a single code pointer. By using the arbitrary EEXIT primitive, the enclave can directly corrupt the CPU state. For example, it can manipulate the stack-pointer register to point to an attacker-crafted ROP chain. By doing an EEXIT instruction towards an arbitrary ret instruction of the host, the enclave can immediately trigger the ROP chain, leading to the same implications as for ROP.

Bypassing Defenses. Several defense mechanisms seek to protect the application’s control flow. Stack canaries [14] protect against linear buffer overflows overwriting return addresses on the stack. ASLR [45] hides code addresses via randomization, while others randomize code itself [44], both making the generation of ROP gadgets hard. More elaborate mechanisms enforce control-flow integrity (CFI), arguably at different granularity. CPI [31] hides code pointers in a shadow stack² while CCFI [39] encrypts code pointers. As these mechanisms rely on randomization, they can be easily broken by the enclave via information disclosure. If CFI metadata is involved, it can be easily corrupted using the write primitive. Stronger hardware-enforced CFI schemes like CET [27] are still unavailable on modern x86 CPUs, and it is unclear to what extent they consider rogue EEXIT attacks.

4.2.3 Data-only attacks

Apart from control-flow attacks, enclaves can corrupt application data other than code pointers or CFI metadata. For example, they can corrupt loop counters, function arguments or syscall arguments [8, 28] using the arbitrary read/write primitives. Typically, data-only attacks are much more restricted than control-flow attacks. For example, they can only reuse code reachable in the normal control flow. Yet, data-only attacks are agnostic to CFI protection schemes and can even achieve Turing-complete computation in many cases by chaining together valid execution paths [28].

4.3 API attacks

The previous attack vectors all violate memory safety of the application by reading, writing and executing application memory in an illegitimate way. It is clear that defeating these attacks is paramount to protecting an application from misbehaving enclaves. Only with such protections in place, it makes sense to reason about the application’s security on the API level. Obviously, SGXJail does not defend against too permissive OCALLs, e.g., giving an enclave the ability to access arbitrary files. Yet, we need to ask to what extent an enclave can attack its host application purely via the ECALL/OCALL interface, that is, without relying on the above SGX attack primitives. For example, an enclave can seek to attack the application by crafting invalid API calls or returning malformed data. For a successful attack, either the API itself needs to be flawed, or the underlying implementation misses important validation steps (e.g., confused deputy attacks [22] and Iago attacks [11]). Since such API-based attacks are highly application specific, they cannot be addressed by a generic defense mechanism anticipated in this work. We discuss proper mitigation strategies in Section 7. Also, we do not address misuse of computational power (e.g., for cryptocurrency mining).

²This corresponds to the weaker randomization scheme since the stronger segment-based isolation is unavailable for 64-bit execution mode.

5 SGXJail

In this section, we present SGXJail, a novel mechanism to protect host applications from untrusted (third-party) enclaves. SGXJail defeats entire classes of attacks by prohibiting enclave primitives outlined in Section 4 at the discretion of the host application. SGXJail can be implemented purely in user space and relies on process isolation and syscall filters, similar to other sandboxing techniques like Docker [40]. We evaluate SGXJail under different workloads to demonstrate its efficiency. Finally, we show how SGXJail can also be implemented via minimal changes to the SGX specifications and corresponding hardware, which we call HSGXJail.

5.1 SGXJail via Software Confinement

SGXJail defeats enclave malware by breaking all three enclave primitives described in Section 4.1. SGXJail does so by confining enclave operation to a strict set of memory pages.

Figure 2 illustrates the basic idea of SGXJail. To break the arbitrary read and write primitives, we rely on the operating system’s ability to isolate processes.³ Namely, we run potentially malicious or misbehaving enclaves in a separate *sandbox process* which does not have access to the host application’s memory. To still allow benign ECALL/OCALL interaction, we establish shared memory between the sandbox process and the host application to implement a form of inter-process communication.

Even with the above process isolation in place, a malicious enclave can perform an attack on the control flow of the sandbox process to issue arbitrary syscalls on behalf of the sandbox process. Such an attack can either be a rogue EEXIT attack, or a code-reuse attack (e.g., ROP) through manipulating the stack [53]. Breaking the primitives that allow an attacker to change the control flow is not trivial. EEXIT can jump to any executable page, and the target address cannot be restricted. Similarly, if the enclave rewrites the saved return address on the stack, the sandbox process cannot detect this modification. A possible—but rather expensive solution—is to mark all executable pages of the sandbox process (except for trampoline code) as non-executable before entering the enclave. When leaving the enclave, the sandbox immediately traps to the kernel, which can then assess the legitimacy of the address at which the sandbox process should resume and remap the pages as executable. However, this requires frequent and expensive page remapping by updating a majority of the page tables of a process. Instead of trying to prevent an attack from hijacking the control flow in the sandbox process, we confine the damage of such a hijacked control flow. In particular, we restrict the syscall interface of the sandbox process by using seccomp syscall filters [36] to whitelist only abso-

³Conforming with Intel’s and our extended SGX threat model, software-based side-channel attacks circumventing such isolation, e.g., Meltdown [37] or Rowhammer [29], are out of scope.

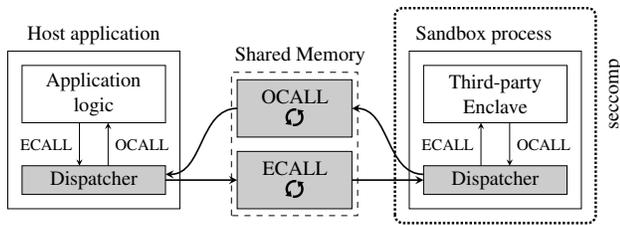


Figure 2: With SGXJail, the enclave is isolated within a separate sandbox process and can communicate with the host application only via shared memory. Also, the enclave is confined using seccomp filters.

lutely necessary syscalls. Even if a malicious enclave gains arbitrary code execution inside the sandbox process, it can no longer perform malicious actions. In contrast to sandboxing techniques like Docker isolating the entire system (e.g., via cgroups), we only need to restrict a single user process for which syscall filters are the appropriate choice.

Life Cycle. A complete SGXJail life cycle works as follows. First, SGXJail creates a new process, the *sandbox process*. The third-party enclave is then loaded within this sandbox process. Moreover, SGXJail creates a shared memory between host application and sandbox process and installs dispatchers for routing all ECALLs and OCALLs through this shared memory. Afterwards, SGXJail activates seccomp filters to restrict the syscalls of the sandbox process to an absolute minimum. Only syscalls required for the communication between application and sandbox process, as well as syscalls required to terminate the sandbox process, are whitelisted. After the initialization, the application can issue ECALLs and receive OCALLs, as follows. The application dispatcher automatically encapsulates ECALLs into messages and transfers them via the shared memory to the sandbox process. ECALL function arguments are copied from the host application to the shared memory. The sandbox process dispatcher listens for incoming messages, decapsulates arriving messages and performs the actual ECALL towards the enclave. Results are returned back to the application, again via message passing over shared memory. The application dispatcher finally copies ECALL results from the shared memory to application memory and hands over to the application. In the same way, OCALLs are routed from the sandbox process through the shared memory to the application host and vice versa. Upon termination of the application, the sandbox process is simply destroyed. Multiple enclaves are isolated via separate sandbox processes with individual shared memory segments.

Compatibility. SGXJail is a transparent enclave confinement mechanism. It does not require any changes to third-party enclaves themselves, *i.e.*, it is binary-compatible with existing enclaves and their existing cryptographic signatures. Also, no enclave source code needs to be available. Instead, SGXJail is tightly integrated within the SGX SDK [25]. All glue code for dispatching and redirecting ECALLs and OCALLs via

shared memory is automatically generated from an enclave’s EDL file [25] which needs to be shipped together alongside each pre-compiled third-party enclave. Also, code for instantiating the sandbox process, the shared memory and activating seccomp filters is provided by SGXJail. For SGXJail, only the untrusted application code has to be recompiled under the SGXJail toolchain.

The installation of seccomp filters is independent of the enclave itself. Since enclaves are not entitled to issue syscalls, the selection of proper syscall filters solely depends on SGXJail and does not affect compatibility with enclaves.

SGXJail enforces benign enclave communication to follow the ECALL/OCALL interface specified in the enclave’s EDL file. An enclave implementing other communication methods (e.g., by directly accessing host memory) breaks as soon as SGXJail is active. This is intentional, as enclave developers are strongly encouraged to clearly define the enclave’s API via ECALLs and OCALLs. In particular, SGXJail breaks unsafe usage of ECALLs and OCALLs where enclave and host application exchange and dereference raw, unchecked pointers rather than buffered data. For example, if one marks an ECALL function parameter with the so-called `user_check` attribute within the EDL file [25], the SDK passes this function parameter without further checking and copying into the enclave. A quick code inspection revealed usage of `user_check` in some Intel architectural enclaves and remote attestation code, all for performance reasons. They could be updated to avoid `user_check` at the cost of slight performance loss. To yet support `user_check`, one would need to manually share (*i.e.*, map) host memory with the sandbox process to which an enclave shall have unrestricted access. Also, host application pointers passed to the enclave need to be translated to the sandbox process due to ASLR. SGXJail could provide simple helper functions for sharing host memory and translating pointers.

5.2 Implementation Details

For generating dispatcher code, we extend the `edger8r` tool [25] accordingly. The sandbox dispatchers are generated in the files `Enclave_us.c|h`, while the application dispatchers are located in `Enclave_u.c|h`. An enclave always copies arguments to enclave memory before processing it. Similarly, our dispatcher code copies arguments to application memory before invoking an OCALL. This prevents TOCTOU vulnerabilities such as double-fetch bugs [58] by design.

ECALLs and OCALLs are routed between application and sandbox process via two distinct shared memory regions, one for each direction. The dispatchers synchronize ECALL/OCALL interaction via shared semaphores. This has the advantage that processes (application and sandbox) are consuming no CPU time while waiting for the other communication partner. For receiving OCALLs, the application installs a separate listener thread that only gets active upon incoming OCALLs.

Selection of appropriate syscall filters is crucial for the security of SGXJail, as a malicious enclave can directly exploit a lax configuration (e.g., via rogue EEXIT attacks). It is favorable to restrict both the number of syscalls as well as their complexity to reduce the attack surface given by the whitelisted syscalls. This also has an impact on the type of inter-process communication between sandbox and application process. By choosing shared memory as communication channel, we do not require any syscall for the actual communication, and only one syscall (`futex`) for synchronization. In summary, we configure `seccomp` [36] to only allow the syscalls `futex` necessary for semaphores as well as `exit_group` for terminating the sandbox process. Thus, the shared memory approach results in only one whitelisted syscall in addition to the required `exit_group` syscall. Unless the implementation of these two syscalls is buggy, they cannot cause a security violation when issued by a malicious enclave.

The SGX SDK passes OCALL function arguments from the enclave to the application via the application’s stack. The enclave knows the application’s stack location via the stack pointer (`RSP` register), which is preserved by the `EENTER` instruction. Hence, it can allocate a stack frame on the host stack via a function called `sgx_ocalloc` and store any outgoing OCALL arguments there. One can leverage this mechanism for reducing SGXJail overhead, as follows. Currently, when doing an OCALL, our sandbox dispatchers copy OCALL arguments from the sandbox to the shared memory. By modifying `RSP` immediately before an `EENTER` to point to the shared memory, one can instruct the enclave to write OCALL arguments directly to the shared memory instead of the sandbox application’s stack. When the enclave `EEXITS`, one can simply restore the original sandbox stack (namely, `RSP`).

In our current implementation, the size of the shared memory is hard-coded to three pages for each direction. For ECALL/OCALL arguments exceeding the shared memory, one can dynamically resize the shared memory on demand. Although multithreaded enclaves are currently not supported by our prototype implementation, support can be easily added. This is done by installing separate semaphores and shared buffers for all enclave threads, which are enumerated in a public enclave XML configuration file. Also, support for nested calls (OCALLs issuing ECALLs) can be added by adapting the synchronization mechanism appropriately.

An interesting question arises whether SGXJail should be integrated with the SGX SDK in a way that does not demand recompilation of the application. Thus, system administrators can globally enforce SGXJail by just installing corresponding shared libraries. Since the enclave’s EDL file is public anyway and will be distributed alongside third-party enclaves, the generation of dispatcher code is straight forward. Moreover, one would need to hook the enclave API of the unmodified application binary and inject dispatcher code, which can be done by preloading SGX SDK libraries (in particular, `sgx_urts.so`).

Table 2: ECALL and OCALL latency in CPU cycles of SGXJail compared to the unprotected Vanilla version. The standard deviation is shown in braces.

Latency	ECALL	OCALL
Vanilla	15 624 (± 301)	13 438 (± 1046)
SGXJail	22 094 (± 814)	19 515 (± 1360)

5.3 Evaluation

SGXJail does not affect runtime performance of host applications or enclaves in isolation. That is, as long as no interaction between enclave and application takes place, they can run without performance loss. The only performance overhead occurs when doing ECALLs and OCALLs due to the message passing via shared memory and the necessary synchronization between application and sandbox process. To evaluate this effect, we first present microbenchmarks for bare metal ECALL and OCALL latency, which are followed by macrobenchmarks on more representative workloads.

Test Setup. All evaluations are done on a commodity notebook featuring an Intel i5-6200U CPU, a Samsung SM951 SSD and running Ubuntu 16.04 Desktop and SGX SDK version 2.4. For the benchmarks, we disabled the screen as well as network interfaces to reduce noise from screen redrawing or external interrupts. Also, we fixed the CPU frequency to its maximum (2.3 GHz) and pinned the benchmark to a single core. The benchmarks include a warm-up phase.

Microbenchmarks. To measure the ECALL latency, we implemented a simple ECALL and measured its execution time from within the host application. That is, the ECALL latency includes `EENTER`, `EEXIT`, all glue code for the enclave and the host, as well as context switching and synchronization between application and sandbox for SGXJail. To measure the OCALL latency, we, in addition, perform one simple OCALL from within the ECALL and subtract the ECALL latency. We repeated the measurement 500 times. The resulting latencies are shown in Table 2. The raw ECALL latency increases from $15.6 \cdot 10^3$ cycles to $22.1 \cdot 10^3$ cycles while the OCALL latency increases from $13.4 \cdot 10^3$ cycles to $19.5 \cdot 10^3$ cycles. Hence, the absolute latency remains small. Since many practical usage scenarios of SGX involve somewhat complex computations inside the enclave, the actual runtime overhead is much lower than the pure ECALL/OCALL overhead.

Macrobenchmarks. Quantifying performance of enclaves is highly application specific. Unfortunately, enclaves are not widely deployed yet and standardized benchmarking suites are unavailable to the best of our knowledge. A common approach is to port existing programs to an enclave [61]. While this sounds appealing, it tends to introduce many unnecessary OCALLs to the standard library which well-designed enclaves would not perform, e.g., the `getpid` syscall in `openVPN` [61].

Instead, we quantify the performance of SGXJail as follows. First, we benchmark a synthetic workload under dif-

ferent OCALL frequencies. The results of this benchmark are generic and can be applied to any enclave for which the OCALL frequency can be determined. Second, we benchmark storage of sensitive enclave data to disk via the Intel protected filesystem (PFS). The PFS is integrated within the SGX SDK and is likely to be used by a vast number of enclaves.

For our first benchmark, we observe that an enclave typically issues OCALLs to perform syscalls, e.g., writing to files. Our benchmarked OCALL performs a `close` syscall on an invalid file descriptor. Such a fast syscall gives an upper bound on the performance overhead since longer syscalls decrease the influence of the OCALL overhead. We repeated each measurement 100 times. The OCALL-to-enclave ratio (w.r.t. their runtime) as well as the overhead of SGXJail compared to unprotected Vanilla applications is given in Figure 3, whereas the simple standard deviation is shown as the area under the curves. We execute a fixed baseline workload inside the enclave, which corresponds to $2201.44 (\pm 25.67) \cdot 10^6$ cycles, or $0.96 (\pm 0.011)$ s on our 2.3 GHz CPU. As this workload runs within the enclave, we quantify it as enclave seconds, or Esec. While we keep the enclave workload constant, we issue OCALLs at different frequencies and measure the additional OCALL work. This is shown as ratio on the left axis of Figure 3 and allows us to decouple the OCALL overhead from the OCALL frequency, which we quantify as OCALLs/Esec.

One can see that the overhead of SGXJail is virtually non-existent for low-frequency OCALLs, meaning that pure enclave execution is not impeded by SGXJail at all. Even for 10 000 OCALLs/Esec the overhead is below 3% and for a large number of 50 000 OCALLs/Esec the overhead is only around 11%. To put these numbers into perspective, Netflix observed a maximum of 50 000 OCALLs/s across their systems [20]. For even higher OCALL frequencies the OCALL workload starts to exceed the enclave workload in the vanilla version already. With SGXJail, enclaves can issue up to 113 000 OCALLs/Esec before OCALL processing exceeds actual enclave computations (ratio=1). For unprotected apps this point is reached for 164 000 OCALLs/Esec. Such situations should be dealt with in practice by redesigning the enclave API and reducing or removing unnecessary OCALLs. Yet, SGXJail only introduces around 20% overhead even in this extreme case.

Our first benchmark measures the raw OCALL performance. However, this does not reflect the performance of copying OCALL arguments between enclave and application. To evaluate the maximum overhead of a real-world scenario, we benchmark an enclave which only accesses files via the Intel protected file system (PFS) library. PFS is shipped with the SGX SDK and is intended for sealing sensitive enclave data on the host file system for persisting state across reboots. To resemble a worst-case scenario of PFS, we implement and benchmark a single ECALL which opens a new file (`sgx_fopen_auto_o_key`), writes a fixed-size buffer (`sgx_fwrite`), and immediately closes the file again

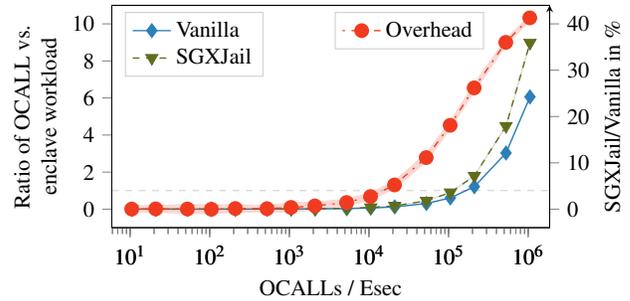


Figure 3: Benchmark on unprotected (Vanilla) and hardened (SGXJail) applications, plotted over different numbers of OCALLs per enclave second (Esec).

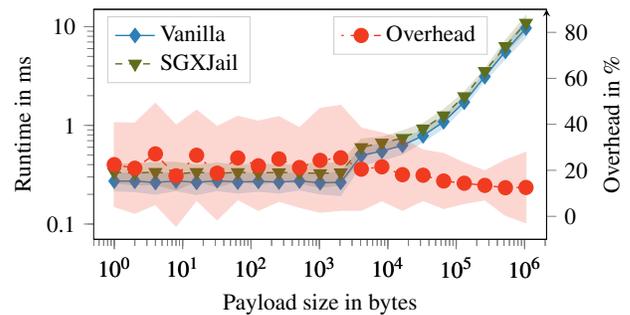


Figure 4: PFS runtime of SGXJail compared to unprotected Vanilla enclaves for different payload sizes.

(`sgx_fclose`). We repeat the measurements 200 times. After each run, we delete the file and synchronize the file system to reliably capture the overhead of PFS. Figure 4 shows the PFS performance for different payload sizes up to 1MB. The runtime includes enclave as well as OCALL computation. The simple standard deviation is shown as area around the curves.

The maximum overhead for protecting PFS with SGXJail is roughly around 20%. There is almost constant runtime up to 2 kB payloads for SGXJail and the unprotected vanilla enclave with a sudden increase at 4 kB payloads. The reason is that the PFS library caches smaller chunks of data and defers actual file writing to closing the file with `sgx_fclose` with 8 OCALLs in total. When exceeding the internal buffer of 3072 bytes, the PFS library flushes data to the file system using 7 more OCALLs, resulting in the sudden increase of the absolute runtimes for SGXJail and the vanilla enclave.

For larger payloads (4 kB and more), the overall overhead does not increase but falls below 20%. This suggests that argument copying itself is not the bottleneck of PFS. We verified this by manually removing argument copying in the sandbox for the actual file write OCALL. Using 1 MB payloads, the overhead dropped by roughly 3%. Rather than argument copying, the runtime overhead of SGXJail is dominated by the OCALL overhead since the PFS implementation chops

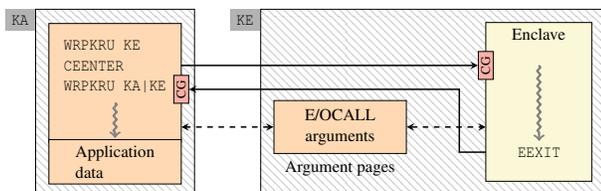


Figure 5: HSGXJail confines the enclave to pages marked with memory protection key `KE`. Thus, the application can protect its pages via a disjoint memory protection key `KA`. ECALL/OCALL interaction is constrained to non-enclave `KE` pages (dashed lines). Moreover, `EEXIT` can only target a single exit point, namely the instruction following a `CEENTER` (a new confined `EENTER` instruction).

larger payloads into a sequence of smaller OCALLs. In fact, for 1 MB payloads we observed 313 OCALLs in total.

We have shown that SGXJail does not impede pure enclave computation (0% overhead). For real-world workloads up to 10 000 OCALLs/Esec, the overhead is below 3% (cf. Figure 3). Even for uncommonly high OCALL frequencies (100 000 OCALLs/Esec), the overhead of SGXJail is still below 20%, whereas plain writing of protected files with high OCALL interaction comes at only 20% overhead. To further improve performance, SGXJail could use HotCalls for faster enclave communication [61]. Alternatively, we propose a lightweight hardware extension (HSGXJail) which provides SGXJail isolation at virtually no overhead.

Memory overhead. SGXJail requires one additional process for the sandbox. As for site isolation in browsers [47], this incurs only a slight (constant) increase in used memory for the sandbox and the shared memory used for communication.

5.4 HSGXJail via Hardware Confinement

In this section, we propose a more efficient defense mechanism via a minimal change to the SGX specification with respect to Intel memory protection keys (MPK), *i.e.*, disallowing one MPK instruction in SGX.

To prevent an enclave from accessing host application memory, we propose a stricter page access policy. To that end, HSGXJail introduces two extensions: first, data confinement and second, control confinement. First, memory regions that are not supposed to be used by the enclave shall be inaccessible to the enclave. Data confinement limits memory pages an enclave can read or write, thus breaking the arbitrary read and write primitives. Second, `EEXIT` shall be only allowed on well-defined exit points. Control confinement prevents the enclave from misusing `EEXIT` to jump to arbitrary host application code, thus breaking the arbitrary `EEXIT` primitive.

Data Confinement with Intel Memory Protection Keys. The central issue of enclave malware is an asymmetry in the memory access policy, granting enclaves unrestricted

access to host-application memory. Data confinement uses a recent protection mechanism called memory protection keys (MPK) [26] to partition virtual memory into enclave-accessible memory and protected application memory. If the enclave attempts to access protected application memory, the CPU raises a page fault. To prevent the enclave from reconfiguring MPK, HSGXJail disallows certain MPK instructions in enclave execution mode. Similar to SGXJail (cf. Section 5.1), we use this mechanism to confine enclave execution to a narrow ECALL/OCALL interface, as shown in Figure 5.

Memory protection keys work as follows: they augment page-based read, write and execute permissions with additional access policies. Each application page can be assigned one particular memory protection key. This protection key is stored directly in the corresponding page table entry (PTE). By assigning different protection keys to different pages, MPK allows to partition virtual memory pages into 16 disjoint protection domains. The PKRU CPU register controls which access policy is applied to those protection domains. For each of the 16 protection keys, PKRU allows to selectively disable write and read access for the current execution thread. The PKRU register can be updated via the unprivileged `WRPKRU` instruction, enabling frequent switching of protection domains within the application. Since each CPU thread maintains its own local PKRU register, MPK supports multithreading.

For HSGXJail, we partition the application into protection key `KA` comprising all application pages and `KE`, covering enclave memory as well as argument pages, as shown in Figure 5. Immediately before entering an enclave, the application configures PKRU to confine memory accesses to the enclave only (`WRPKRU KE`). During enclave operation, the enclave can only access argument pages for ECALL/OCALL arguments. After leaving the enclave, the application re-enables full access to the application itself (`KA`) as well as the argument pages (`KE`) via `WRPKRU KA|KE`.

To prevent the enclave from manipulating MPK by reconfiguring the PKRU register, HSGXJail demands a slight modification to the SGX specification. Whenever HSGXJail is active, the `WRPKRU` instruction is disallowed for the enclave and raises an invalid opcode exception instead. This change should be easily adaptable via a microcode update to the CPU.

HSGXJail poses no limit on the number of applications using third-party enclaves, however, the number of enclaves within a single application is restricted. Since MPK supports up to 16 different protection domains, HSGXJail can natively secure applications utilizing up to 15 distinct enclaves. Note that one protection domain is needed for the application itself. To support more enclaves per application, one can follow various approaches: First, in many cases enclaves provide simple functionality, e.g., ECALLs without OCALLs, or OCALLs for issuing syscalls but not towards other enclaves. In these cases, enclaves are never called in an interleaved way and thus, are never concurrently active. Hence, the application can safely share the same argument pages and also the same pro-

tection key among those enclaves. This increases the number of supported enclaves by the degree of enclaves which are not interleaved with other enclaves. Second, memory protection keys can be dynamically updated and scheduled among different enclaves. While this supports an arbitrary large number of enclaves per application, it incurs additional performance penalty in updating protection keys in the PTEs.

Control Confinement. Whenever leaving enclave execution (via ECALLs and OCALLs), the enclave jumps into the host application via an EEXIT instruction. However, since the enclave can freely choose the jump target of EEXIT, a variety of code-reuse attacks become possible (cf. Section 4).

Data confinement already limits an enclave’s read and write access by means of MPK. While MPK protects data accesses, it does not prevent fetching code from other protection domains. This design choice is intentional to enable application code to update protection domains without accidentally removing access to its own code. Hence, data confinement does nothing to protect an application from rogue EEXITs.

To break the arbitrary EEXIT primitive, HSGXJail restricts EEXIT to a single valid *exit point*. In particular, EEXIT can only target the instruction immediately following a so-called CEENTER instruction. This *exit point* is similar to the enclave *entry points* used to protect an enclave from malicious applications, both of which are shown as call gates (CG) in Figure 5.

Control confinement can be easily implemented via small changes to SGX. We propose to extend the semantics of EENTER via a novel confined CEENTER instruction. From the enclave’s perspective, CEENTER behaves exactly as EENTER. EENTER already stores the exit point (*i.e.*, the address of the instruction immediately following EENTER) in register RCX. However, SGX leaves it up to the enclave to store this exit point and later on pass it to EEXIT. In contrast, our CEENTER instruction additionally stores the exit point in a protected, thread-local CPU register called OEXIT which is inaccessible to the enclave. To make use of this exit point, we propose to adapt the semantics of the EEXIT instruction, as follows: Instead of jumping to a target provided by the enclave via register RBX, our EEXIT ignores RBX and instead directly jumps to the address stored in the protected OEXIT register. Both, CEENTER and EEXIT can be implemented in CPU microcode.

Compatibility. To be fully compatible with existing enclave software, we activate HSGXJail only on demand. If the application issues a normal EENTER instruction, HSGXJail is inactive and SGX behaves as usual. When entering the enclave via our new confined CEENTER instruction, HSGXJail is active until EEXIT. Moreover, HSGXJail’s slim design is fully compatible with advanced SGX features such as multithreading, dynamic memory management and virtualization [26]. Availability of HSGXJail can be indicated via a model-specific register.

Software Considerations. HSGXJail protects applications from existing, unmodified third-party enclaves. HSGXJail can be integrated entirely within the SGX SDK [25], thus being

fully transparent to existing application code. This allows to use HSGXJail by recompiling applications, without the need to rewrite any application code.

To use HSGXJail, the SDK needs the following slight adaptations. First, the SDK replaces EENTER with CEENTER in the untrusted `urts` library. The `urts` library already uses a single exit point, which is the address immediately following EENTER. The corresponding trusted `trts` library belonging to the enclave performs EEXIT only towards this single exit point. Since our modified EEXIT instruction enforces the same exit point, it does not change the behavior of benign enclaves. No changes to the `trts` library are required. Benign enclaves compiled under the original `trts` library work out of the box.

For data confinement, the SGX SDK needs to establish enclave-accessible argument pages reflecting the ECALL/OCALL interface and configure memory protection keys accordingly. By default, all application code runs with protection key *zero*. Thus, the SDK assigns protection keys starting with *one* to all enclave pages as well as the corresponding argument pages. Similar to the software-only variant, SGXJail, the SDK can do this once when loading a new enclave.

When doing an ECALL, the SDK additionally copies all input arguments from application memory to an enclave-accessible argument page. In the same way, the SGX copies back any output arguments from the argument page to application memory at the end of an ECALL. The same applies to OCALLs. While argument copying causes some overhead, it is deemed necessary to generically prevent TOCTOU attacks and guarantee the security of the application. For the same reason, the enclave copies untrusted application arguments to enclave memory before operating on it.

Before entering the enclave, the SDK saves all necessary CPU registers in application memory, clears sensitive content from the registers and configures the application’s stack pointer RSP to point to one of the argument pages. Configuring RSP in that way causes the enclave to read and write any OCALL arguments directly from/to the argument page, which is enclave-accessible, without additional copying overhead. After leaving the enclave, the SDK restores the application’s CPU registers, including the stack pointer.

Performance Estimates. The only functionally necessary change for HSGXJail is disallowing the `WRPKRU` instruction on CEENTER, which can be easily implemented in the CPU. The microcode changes we propose to CEENTER and EEXIT for control confinement are minimal and only comprise register operations rather than memory accesses, resulting in negligible performance overhead. Second, data confinement via MPK requires no change and shows the same performance as for MPK without HSGXJail. Hence, it is reasonable to expect a negligible overhead of HSGXJail in every aspect, far lower than the overhead of the software-based SGXJail variant.

6 Related Work

Defense by Detection. Researchers proposed to detect enclave malware by monitoring their I/O behavior [13, 16]. However, this is believed to be infeasible in practice [38]. Others proposed analyzing enclave code before actually running it [13], which is not feasible for generic loaders. Generic loaders can remotely fetch arbitrary malicious code at runtime. Refusing such generic loaders would annihilate all use cases for protecting intellectual property. Instead, Costan et al. [13] proposed to force generic loader enclaves to embed malware analysis code within the enclave. However, it is unclear how effective this technique is in detecting malicious code. It also raises the question who decides which analysis code to embed and to ensure the analysis code does not leak enclave secrets. Also, analysis code cannot be easily updated, and enclaves without analysis code cannot be executed without risk.

Defense by Prevention. While applying control-flow integrity (CFI) to the host application sounds appealing, it does not close all attack vectors outlined in Section 4. Although hardware-assisted CFI can prevent some control-flow attacks [27], they are not yet available and might miss rogue EEXIT attacks. Software CFI schemes like [31, 39] can simply be bypassed by leaking secrets and corrupting CFI metadata via the arbitrary read and write primitives. Moreover, no CFI scheme can prevent data-only attacks.

Readactor [15], Heisenbyte [57], and NEAR [62] severely limit the arbitrary read primitive necessary for many attacks by forcing page faults when trying to access sensitive code. However, they have significantly larger overhead than SGXJail, and blind ROP attacks might still be possible [4]. Ryoan [23] executes malicious enclaves inside a software sandbox using software fault isolation (SFI). However, Ryoan demands recompilation of the enclave with SFI, which cannot be applied in our setting. Also, Ryoan severely restricts the enclave life cycle to a single stateless invocation, which is incompatible to generic third-party enclaves.

7 Discussion

Since the very first blog post in 2013 [48], the enclave malware threat has been discussed at a high level but was mostly disregarded by the research community. With recent attacks showing powerful and practical enclave malware, research on proper defense mechanisms becomes pressing.

In this work, we identified three enclave primitives, namely arbitrary memory reads, writes and EEXITs, which lie at the heart of the enclave malware threat by exposing an application to a variety of runtime attacks originating from misbehaving enclaves. Although these primitives help support different SGX programming models, they not only give rise to enclave malware but they are unnecessary in practice, as enclaves ought to strictly comply with the defined ECALL/OCALL interface. In particular, the enclave runtime services offered by

the SGX SDK demand precise EDL specification of the data exchanged, and bypassing this specification is considered bad practice. Moreover, the SDK uses only a single enclave exit point, from which all ECALLs and OCALLs are dispatched.

Based on these observations, we proposed (H)SGXJail to confine enclave primitives to the narrow interface specified by the EDL. This applies the principle of least privileges [49] also to enclaves and closes a entire class of runtime attacks, including information disclosure, control-flow attacks, as well as data-only attacks. Even more, by automatically copying ECALL/OCALL arguments from and to application memory, (H)SGXJail prevents double-fetch bugs [58] by design.

Furthermore, SGXJail paves the way for reasoning about application security based on application code only (*i.e.*, without trusting any enclave code), and the ECALL/OCALL interface in particular. While SGXJail defeats a entire class of runtime attacks, it cannot solve the problem of too permissive host interfaces, e.g., a syscall proxy [38] which allows executing arbitrary syscalls. Further research on designing and validating ECALL/OCALL interfaces is needed to avoid API-level attacks via too permissive OCALLs or confused deputy [22] and Iago attacks [11]. In general, one has to consider enclave-to-host communication not as asymmetric (*cf.* the kernel's syscall interface) but as part of a mutually distrusted API where both communication parties distrust each other. Mutual distrust is an integral part of designing secure web APIs. Since enclave malware raises similar threats as web applications, we also see some overlap in defense strategies [43]. In special, input validation or sanitization [43, Section V5] can help prevent Iago-style attacks while verification of the logical execution flow [43, Section V11] can prevent confused deputy attacks.

Closing Side Channels. Several side-channel attacks mounted against benign SGX enclaves have been shown [7, 19, 34, 41, 59, 63]. Moreover, malicious enclaves themselves can mount side-channel attacks [21, 53, 54]. Although not the primary focus of this work, SGXJail prevents a variety of side-channel attacks that rely on accessing host application memory, e.g., Flush+Reload on shared host libraries used by the host application from within enclaves, Prime+Probe using host application arrays [54], Rowhammer attacks from within enclaves [21] as well as TSX-based address probing [53].

8 Conclusion

While designed to increase the security of a computing system, secure enclave technology such as Intel SGX might also be misused for shielding malware inside enclaves. However, research on potential enclave malware is still in its beginnings, and practical defense mechanisms are virtually non-existent.

In this work, we identified the root cause of enclave malware as an insufficient enclave-to-host isolation and proposed (H)SGXJail as a generic defense against a wide range of enclave malware threats. (H)SGXJail enforces mutual isolation

between host applications and enclaves, thus protecting applications from potentially misbehaving or malicious third-party enclaves. SGXJail is an efficient and transparent software defense, running third-party enclaves in an isolated sandbox. Our proof-of-concept implementation shows zero overhead for pure enclave computation and less than 3% for realistic workloads. SGXJail is tightly integrated within the SGX SDK and can be used out of the box. Furthermore, we propose SGXJail directly in hardware. Our HSGXJail mechanism provides enclave confinement by means of Intel MPK with slim extensions to the SGX specification at virtually no cost. We believe HSGXJail should be immediately rolled out via a microcode update to SGX-enabled CPUs to proactively enable our SGX malware defense. However, support for MPK is still rare. Although some server CPUs support MPK [64], it is unclear when x86-based desktop CPUs catch up.

Apart from defending against enclave malware, (H)SGXJail opens up new use cases for Intel SGX and similar isolation technologies. For example, we envision that (H)SGXJail can be used as lightweight and secure sandboxing mechanism for browser site isolation or plugin management, where third-party code has proven to be both, potentially malicious and potentially security critical.

Acknowledgements

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 681402). This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSS-net, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Additional funding was provided by a generous gift from Intel. 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 parties.

References

- [1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS*, 2005.
- [2] Erick Bauman and Zhiqiang Lin. A case for protecting computer games with SGX. In *Workshop on System Software for Trusted Execution*, 2016.
- [3] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems*, 2015.
- [4] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *S&P*, 2014.
- [5] Tyler K. Blensch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS*, 2011.
- [6] Erik Bosman and Herbert Bos. Framing signals - A return to portable shellcode. In *S&P*, 2014.
- [7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostianen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.
- [8] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.
- [9] Nicholas Carlini and David A. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security*, 2014.
- [10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *CCS*, 2010.
- [11] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *ASPLOS*, 2013.
- [12] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, 2001.
- [13] Victor Costan and Srinivas Devadas. Intel SGX explained. 2016.
- [14] 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.
- [15] 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 *S&P*, 2015.
- [16] Shaun Davenport and Richard Ford. SGX: the good, the bad and the downright ugly, January 2014. URL: <https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly>.
- [17] Anders T Gjerdrum, Robert Pettersen, Håvard D Johansen, and Dag Johansen. Performance of trusted computing in cloud infrastructures with intel sgx. In *International Conference on Cloud Computing and Services Science*, 2017.

- [18] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *S&P*, 2014.
- [19] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *EuroSec*, 2017.
- [20] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel isolation: From an academic idea to an efficient patch for every computer. *USENIX ;login*, 2018.
- [21] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O’Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.
- [22] Norman Hardy. The confused deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–38, 1988.
- [23] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Usenix OSDI*, 2016.
- [24] Intel. Intel Software Guard Extensions SSL. URL: <https://github.com/intel/intel-sgx-ssl>.
- [25] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, May 2016. Rev 1.5.
- [26] Intel. Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. (325384), 2016.
- [27] Intel. Control-flow Enforcement Technology Preview, June 2017. Revision 2.0.
- [28] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *CCS*, 2018.
- [29] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.
- [30] Klaudia Krawiecka, Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N Asokan. Protecting web passwords from rogue servers using trusted execution environments. *arXiv:1709.01261*, 2017.
- [31] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *OSDI*, 2014.
- [32] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. Loop-oriented programming: a new code reuse attack to bypass modern defenses. In *IEEE Trustcom/BigDataSE/ISPA*, 2015.
- [33] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, 2017.
- [34] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security Symposium*, 2017.
- [35] Xueping Liang, Sachin Shetty, Deepak Tosh, Charles Kamhoua, Kevin Kwiat, and Laurent Njilla. Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In *International Symposium on Cluster, Cloud and Grid Computing*, 2017.
- [36] Linux kernel. SECure COMPUTing with filters, 2017. URL: https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.
- [37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.
- [38] Marion Marschalek. The Wolf In SGX Clothing. *Bluehat IL*, January 2018.
- [39] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: cryptographically enforced control flow integrity. In *CCS*, 2015.
- [40] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014.
- [41] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *CHES*, 2017.
- [42] Nicolas Bacca. Soft launching ledger SGX enclave, 2017. URL: <https://www.ledger.fr/2017/05/22/soft-launching-ledger-sgx-enclave/>.
- [43] OWASP. OWASP application security verification standard 4.0, 2019.
- [44] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *S&P*, 2012.

- [45] PaX Team. Address space layout randomization (ASLR), 2003. URL: <http://pax.grsecurity.net/docs/aslr.txt>.
- [46] PaX Team. Rap: Rip rop. *Hackers to Hackers Conference*, 2015.
- [47] Charlie Reis. Mitigating spectre with site isolation in chrome, 2018. URL: <https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html>.
- [48] Joanna Rutkowska. Thoughts on Intel’s upcoming Software Guard Extensions (Part 2), 2013. URL: <http://theinvisiblethings.blogspot.com/2013/09/>.
- [49] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
- [50] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *S&P*, 2015.
- [51] 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 *S&P*, 2015.
- [52] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. *AsiaCCS*, 2018.
- [53] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with Intel SGX. In *DIMVA*, 2019.
- [54] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.
- [55] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.
- [56] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *S&P*, 2013.
- [57] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *CCS*, 2015.
- [58] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. A survey of the double-fetch vulnerabilities. *Concurrency and Computation: Practice and Experience*, 30(6), 2018.
- [59] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, 2017.
- [60] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in Intel SGX enclaves. In *ESORICS*, 2016.
- [61] Ofir Weisse, Valeria Bertacco, and Todd M. Austin. Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves. In *ISCA*, 2017.
- [62] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *AsiaCCS*, 2016.
- [63] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, May 2015.
- [64] Mingwei Zhang. XOM-Switch, 2019. URL: <https://github.com/intel/xom-switch>.

Fluorescence: Detecting Kernel-Resident Malware in Clouds

Richard Li* Min Du† David Johnson* Robert Ricci* Jacobus Van der Merwe* Eric Eide*

*University of Utah

†University of California, Berkeley

Abstract

Kernel-resident malware remains a significant threat. An effective way to detect such malware is to examine the kernel memory of many similar (virtual) machines, as one might find in an enterprise network or cloud, in search of anomalies: i.e., the relatively rare infected hosts within a large population of healthy hosts. It is challenging, however, to compare the kernel memories of different hosts against each other. Previous work has relied on knowledge of specific kernels—e.g., the locations of important variables and the layouts of key data structures—to cross the “semantic gap” and allow kernels to be compared. As a result, those previous systems work only with the kernels they were built for, and they make assumptions about the malware being searched for.

We present a new approach to detecting kernel-resident malware within a “herd” of similar virtual machines. Our approach uses limited knowledge of the kernels under examination—e.g., the location of the page global directory and the processor’s instruction set—to concisely *fingerprint* each kernel. It uses no kernel-specific semantics to *compare* the fingerprints and find those that represent anomalous hosts. We implement our method in a tool called Fluorescence and demonstrate its ability to identify Linux and Windows hosts infected with real-world, kernel-resident malware. Fluorescence can examine a herd of 200 virtual machines with Linux guests in about an hour.

1 Introduction

Kernel-resident malware is stealthy. Once a kernel-resident rootkit infects a machine, it will generally try to hide traces of its intrusion, disable security software, and install persistent backdoors for future unauthorized access [12, 22, 39, 40]. Despite recent advances in protecting kernel integrity, kernel rootkits remain a significant threat: for example, at Black Hat 2017, Bulygin et al. [7] demonstrated a successful kernel rootkit attack against Windows 10, which has multiple kernel-protection mechanisms enabled.

To persist and perform malicious activities, a kernel rootkit must inject code into the kernel’s address space [21]. Thus, in principle, an analyst can detect the presence of a kernel rootkit by comparing a memory snapshot of a suspect host against a memory snapshot of a clean, uninfected host running the same kernel. Such a comparison is difficult in practice for three reasons. First, the analyst must locate the baseline: a

host that is running the relevant kernel and that is guaranteed to be uninfected. A clever way to solve this problem is to leverage the fact that clouds commonly run many instances of a single virtual-machine (VM) image [6]. Given a large number of VMs running the same image and the assumption that infections are rare, an analyst can assume that the overwhelming majority of the VMs will be clean [3]. Second, kernel memory snapshots are large. While it is possible to transfer “raw” memory snapshots to a single point for analysis [3], the network cost of this collection can be high when many VMs are to be examined. Third and most significantly, the kernel-memory snapshots of two VMs that run “the same kernel” can differ widely, for a large number of reasons that do *not* indicate the presence of a rootkit infection. Across two snapshots, the routine differences due to divergent virtual-to-physical memory mappings, address-space layout randomization (ASLR), paravirtualization-related patching, and other factors can make it very challenging to identify the differences that are indicators of kernel malware.

To distinguish between benign differences and potentially important ones, previous work relies on knowledge of the kernels being inspected. For example, the Blacksheep system [3] uses knowledge of the Windows XP and 7 kernels—e.g., the identities and layouts of important data, the locations of kernel entry points, and the structure of Portable Executable (PE) files—so that it can give special attention to differences in Windows’ key components. Bridging the “semantic gap” in this way is effective but has three practical weaknesses. First, the analyzer becomes specialized: it works only on the kernels for which it has special (and accurate) implementation knowledge. Second, the analyzer becomes more complex: it must include code to walk individual data structures, extract specific kernel features, assign weights to the extracted features, and so on. Third, by choosing to give special attention to certain parts of the kernel, the analyzer inherently makes assumptions about how malware will integrate with the kernel. These assumptions may or may not be accurate, and as malware evolves, the analyzer’s assumptions may become less true over time.

In this paper, we present a new and alternative approach to detecting kernel-resident malware within a large group of similarly configured VMs (a “herd”). Our approach uses no malware-specific knowledge, e.g., no signatures or assumptions about how malware attaches to the kernel, except for

the assumption that the malware has code that resides in the kernel. Our approach uses *limited kernel-specific knowledge* to obtain, for each VM in the herd, a meaningful but concise *fingerprint* of the code in that VM’s kernel. The knowledge used in this process is low-level: e.g., the location of the page global directory, allowing the rest of the kernel’s memory to be located, and knowledge of the processor’s instruction set, allowing the kernel’s code to be disassembled. Each fingerprint is a collection of hashes that summarize the (normalized) contents of a kernel’s code pages: we use fuzzy hashing [20] so that similar page contents map to similar hash values. The fingerprints are generated on the physical machines that host the VMs, and then they are sent to a central analysis server. The server uses *no kernel-specific knowledge* to carry out its task. It compares the fingerprints by first performing feature alignment (identifying the elements that “line up” over all the fingerprints); then putting the fingerprints into a space over which distances can be computed; and finally, computing clusters over the fingerprints. The fingerprints of most VMs form a single cluster, representing the healthy members of the herd. Fingerprints that fall outside the main cluster correspond to VMs with anomalous, possibly malware-infected, kernels.

We have implemented our approach in a tool called *Fluorescence* and evaluated its ability to detect VMs that are infected with real-world kernel rootkits. Fluorescence can examine both Linux (3.13–4.15, x64) and Windows 7 (x64) kernels. Because the kernel-specific knowledge needed for memory acquisition and normalization is minimal and low-level, it tends to be stable across many versions of a single kernel: in particular, we report that Fluorescence’s single Linux-specific agent works correctly across the range of Linux kernels we have tested, 3.13.0–4.15.0. We also report on our experiments using Fluorescence to detect the presence of kernel rootkits within herds. Fluorescence was able to find all of the infected hosts in the Linux- and Windows-based herds that we created; in addition, when multiple types of malware were present, Fluorescence was able to correctly cluster the infected hosts by type. Finally, we report that the time taken by Fluorescence is reasonable, even for herds containing a few hundred VMs. In our experience, Fluorescence can analyze a 50-host herd in less than ten minutes, and 200 hosts in ~60–80 minutes.

Our contributions are threefold. First, we present a new method for detecting kernel-resident malware within a group of VMs that run the same kernel. Unlike previous methods that require detailed knowledge of the kernel under examination, our method uses limited kernel-specific knowledge to construct fingerprints and no kernel-specific knowledge to analyze the fingerprints. Second, we describe the implementation of our method in Fluorescence. Our implementation shows that our approach is general: Fluorescence works with both Windows and Linux kernels, and a single Linux agent suffices for a wide range of Linux kernel versions. Third, we evaluate Fluorescence in terms of its detection abilities and speed. In our experiments, Fluorescence was able to detect

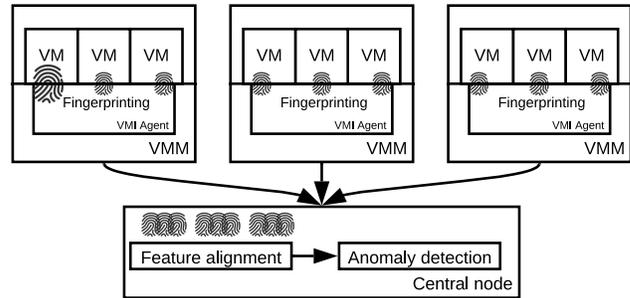


Figure 1: Fluorescence architecture.

all the real-world kernel rootkits in our VM herds. The time required by Fluorescence is reasonable for regular (e.g., daily) scanning of large herds, containing up to a few hundred VMs.

2 Design

Figure 1 illustrates the overall design of Fluorescence, our system for detecting kernel-resident malware within a herd of similarly configured VMs. Fluorescence implements a three-step algorithm. First, it collects the current *fingerprint* of every VM in the herd (§2.1). A fingerprint summarizes the code pages within the kernel of a VM’s guest; fingerprints are computed by agents that run on the physical machines that are being monitored by Fluorescence, and these agents send the fingerprints to a central node for analysis. Second, Fluorescence’s central node performs *feature alignment* (§2.2). Each fingerprint is an unordered multiset that represents the content of one VM’s kernel, and the feature-alignment step finds the elements that best correspond to each other across the multisets. The output of this step is a matrix. Each row encodes the fingerprint of one VM, and each column represents a *feature*; the elements of each fingerprint (row) are permuted so that the best-corresponding elements across all fingerprints are aligned (columns). Third, the central node performs *anomaly detection* over the data in the matrix (§2.3). Fluorescence does this in two steps: the first transforms the data in the matrix so that one can compute “distances” between the fingerprints, and the second uses machine learning—deep learning (§2.3.1) and clustering (§2.3.2)—to find anomalies. The fingerprints of most VMs form a single cluster. Given the assumption that malware infections are rare [3], that cluster represents VMs that are healthy. Outliers correspond to VMs with anomalous kernel-memory code, e.g., malware infections. The clustering pattern among the outliers can help an analyst determine if the outlier VMs are infected by a single kind of malware (one cluster) or different kinds (multiple clusters).

Fluorescence is designed to be general in two ways. First, it relies on no malware-specific knowledge: no signatures or assumptions about how malware works, except for the assumption that it must have code in order to stay resident. Second, it requires only very limited information about the kernels that are being monitored. As Table 1 shows, the

Analysis Step	Kernel/Arch.-specific Knowledge
Fingerprinting	
<i>kernel page pinning (opt.)</i>	debug info used by pinning tool
<i>VM pause/resume</i>	<i>none</i>
<i>kernel page acquisition</i>	page global dir. (KPGD) location x86_64 page table
<i>normalization</i>	Intel Extended Page Tables general ELF/PE loading layout OS-dependent addr. space layout x86 instruction set
<i>hashing</i>	<i>none</i>
Feature alignment	<i>none</i>
Anomaly detection	<i>none</i>

Table 1: Kernel- and architecture-specific knowledge needed by Fluorescence.

fingerprinting agents need some basic, low-level information in order to locate a kernel’s pages (§2.1.1) and normalize their contents (§2.1.2). Fluorescence’s central server, which does feature alignment and anomaly detection, needs no kernel- or architecture-specific knowledge at all.

2.1 Fingerprinting

Fluorescence’s agents, which are co-located with the VMs being monitored, produce a fingerprint for every VM in the herd. Fluorescence’s feature-alignment and clustering steps operate on these fingerprints, rather than kernel memory snapshots, which greatly reduces the amount of data that is transferred to the central server. The key goal of fingerprinting, therefore, is to preserve the most important characteristics of a VM’s guest kernel code memory while also being concise.

Creating a fingerprint involves four steps. First, the agent pauses the target VM. Second, the agent uses virtual machine introspection (VMI) to locate the VM guest’s kernel code memory pages. It copies the pages and their metadata into its own memory—a quick operation—and then resumes the target VM.¹ Third, the agent *normalizes* the contents of the copied pages to reduce expected sources of “noise,” e.g., the effects of address-space layout randomization (ASLR). The agent has multiple ways to normalize the raw data, resulting in multiple *feature views* of each page. Fourth, the agent uses fuzzy hashing [20] to compute a hash for each feature view. A fuzzy hash function produces similar hashes for similar inputs, and is therefore a summarizer: the “distance” between the hashes of pages *A* and *B* can be used to estimate the similarity of the full contents of pages *A* and *B*.

The complete fingerprint of a VM guest’s kernel is a multiset, and each element of the multiset is a tuple that describes one 4 KB-page of the kernel’s code memory. The first element of the tuple is the hash for the first feature view of the page, the second is the hash of the second feature view, and

¹A future version of Fluorescence could use page sharing between the target and agent VMs, in conjunction with copy-on-write, to reduce the pause time for the target VM. We have not implemented this because the pause time is already short, and reducing pause time is not the focus of our research.

so on. When it is complete, the agent sends the fingerprint to Fluorescence’s central server for analysis.

2.1.1 Finding Kernel Code Pages

There are three main challenges that Fluorescence addresses in obtaining the kernel code pages of a monitored VM. The first is to ensure that all the code pages are in memory. Some kernels, including the Windows 7 kernel, can swap their own code pages to disk; on-disk pages cannot easily be read through VMI, and unreadable pages would result in incomplete fingerprints. For such “swappable” kernels, our Fluorescence implementation simply invokes a tool inside the VM guest to pin all of the kernel’s code pages, prior to Fluorescence starting the fingerprinting process (§3.1). The second challenge lies in accessing the target VM’s memory. For this, our implementation uses libVMI [31], a popular and open-source library that implements virtual machine introspection. The third challenge is to find all of the kernel code pages. To do this, the Fluorescence agent starts from the kernel page global directory (KPGD) and makes a breadth-first traversal of the page table to collect and copy all of the kernel’s executable pages. The location of the KPGD is kernel-specific, but easily obtainable via libVMI (§3.1).

The x64 architecture supports multiple page sizes—4 KB, 2 MB and 1 GB—and kernels use pages of different sizes to improve memory management. So that fingerprint generation is not influenced by the use of huge pages (which changes over time), Fluorescence uses a uniform 4 KB page size. When the agent finds a huge kernel page, it divides that page into multiple 4 KB pages within its own representation of the kernel’s memory. As described next, each 4 KB page becomes the basis of a feature in the kernel’s fingerprint.

2.1.2 Normalization

Consider a single page of code that is loaded into the guest kernels of two VMs. One might assume that in the running kernels, the contents of the two pages would be identical, but this is often not the case. In particular, the kernel-loading process may patch the loaded code to replace symbolic references (e.g., to functions in other code) with actual (virtual) addresses. The two kernels may patch *different* addresses into the code for various reasons, including the use of ASLR, thus causing the two copies of the code to be slightly different across the two VMs. This difference is benign—expected, and not indicative of an anomaly. Unless differences like this are accounted for, however, they can make it difficult for Fluorescence to identify differences that *are* anomalous.

To reduce the effects of benign differences, the Fluorescence agent performs *normalization*: it applies a set of functions to reduce the “noise” introduced by factors such as ASLR. A perfect normalization function would effectively undo the benign changes, mapping every copy of “the same code page” across all the monitored VMs onto a single value that is similar to the originally loaded, unpatched code. With

enough kernel information (e.g., debug symbols) such perfect normalization is feasible, but our aim is for Fluorescence to work with minimal knowledge of the kernels it monitors. Our implemented normalization functions (§3.2) therefore rely only on basic knowledge about ELF/PE loading and the x86 ISA. As a consequence, they are approximate, but still effective at reducing “noise.”

The Fluorescence agent applies normalization to each of the 4 KB pages that it obtains from a monitored VM (§2.1.1). The agent supports multiple normalization functions and applies each one individually, resulting in multiple translations of each page. We refer to each of these translations as a *feature view*. By convention, the identity function is always one of the normalization functions: i.e., the “raw content” is always one of the feature views.

2.1.3 Hashing

Finally, the agent hashes every feature view of every page that the agent obtained from the monitored VM. Our implementation uses *ssdeep* [20], which is a fast fuzzy hash function. For each page, the agent collects the hashes of the page’s feature views into a tuple. It then collects the tuples into a multiset, which is the completed fingerprint of the VM.

2.2 Feature Alignment

The fingerprints of all the monitored VMs are sent to Fluorescence’s central server for analysis. The first step of the analysis is feature alignment, which aims to find the best correspondences—i.e., the best matches—of all of the pages across all of the VMs. Imagine putting all of the fingerprints into a 2D matrix, where each row contains the tuples from a single fingerprint. The goal of feature alignment is to permute each row so that the tuples in each matrix column represent versions of “the same page” across all of the VMs.

Feature alignment has three phases. The first simplifies the fingerprints by removing tuples that represent content present in all of the VMs. The second computes a *basis*, which is a vector that contains the most representative elements (tuples) across all of the fingerprints. The third phase translates all of the fingerprints into vectors, ordering the elements by matching them against the basis.

2.2.1 Remove Tuples for Ubiquitous Content

Recall that our ultimate goal is to find anomalous VMs within the herd. Pages that are identical across all of the VMs are not useful for anomaly detection, so their tuples can simply be removed from the fingerprints.

The algorithm for removing “ubiquitous tuples” considers each normalization separately, starting from the identity function. Let i be the number of the current normalization, and let F be the collection of n fingerprints. Let F_i be the “ i -th projection of F ”: the fingerprints F , but replacing every tuple with just its i -th element. Now compute H as the (multiset) intersection of the members of F_i . The hashes in H represent

tuples that represent the same content (under normalization i) across all of the fingerprints. Now we can remove those tuples from the fingerprints. For each f in F , for each h in H , remove the tuple whose i -th element is equal to h .²

The above process is repeated for each normalization function. By starting with the identity normalization, the algorithm considers exact page-content matches first: their tuples are matched and removed, allowing subsequent matching to be more accurate. In practice, the above algorithm removes a large fraction of the tuples from all of the fingerprints, greatly speeding up all of the subsequent analysis steps.

2.2.2 Compute a Basis

The next step is to compute a basis: a vector that Fluorescence can use to impose an order on the elements of all of the (simplified) fingerprints. The order is arbitrary; its only purpose is to allow similar pages across the VMs to be associated with each other, so that the fingerprints can sensibly be compared, and outliers identified.

Fluorescence creates the basis by selecting the most *representative* elements from the actual fingerprints. We say that an element of a fingerprint is representative if it is *similar* to an element in every other fingerprint, where the similarity of elements (i.e., tuples) is defined in terms of the hashes they contain. Given two hashes, *ssdeep* can compute a similarity score between 0 and 100: a high score means that the hashes represent highly similar strings (i.e., normalized page content) and a low score means that the hashes represent very dissimilar strings. We define the similarity σ of two fingerprint elements as the maximum similarity scores of their hashes for every feature view:

$$\begin{aligned} \sigma(e_1, e_2) &= \max(sim_1, sim_2, \dots) \\ \text{where } e_1 &= \langle \alpha_1, \alpha_2, \dots \rangle, e_2 = \langle \beta_1, \beta_2, \dots \rangle, \\ sim_i &= \text{ssdeep}(\alpha_i, \beta_i) \end{aligned}$$

Fluorescence collects the most representative elements into a basis, called Λ , by using the following algorithm. Initialize Λ to be a zero-element vector. For every element e in every fingerprint, find the element in every other fingerprint that is most similar to e . Call that set $neighbors_e$, and let sim_e be the sum of $\sigma(e, n)$ for all $n \in neighbors_e$. (If no element in a given fingerprint has a positive similarity score when compared to e , then that fingerprint contributes nothing to $neighbors_e$.) Now, from all the elements in all of the fingerprints, choose the element e that has the greatest sim_e : this is the most representative element. Extend Λ by adding e , and remove every element of $neighbors_e$ from further consideration. Now repeat: from the remaining eligible elements, choose the element e with the greatest sim_e ; extend Λ , and remove e ’s neighbors from further consideration. Repeat until there are no more elements to consider.

²By construction, such a tuple is guaranteed to exist. If there is more than one tuple whose i -th element is h , arbitrarily choose one to remove.

In general, Λ may contain more elements than any of the actual fingerprints. (It must be at least as long as the longest fingerprint.) This is because Λ includes elements that represent all of the “rare” and unique elements that are present in any fingerprint.

2.2.3 Compute the Fingerprint Vectors

After computing the basis Λ , it is straightforward to transform the fingerprints into vectors that can be sensibly compared. Fluorescence does this by computing a 2D matrix, called T (for “tuples”), in which every row represents a fingerprint and every column corresponds to an element of Λ . Fluorescence fills each row from left to right, i.e., from the most representative to the least representative elements of the basis. Consider the translation of a fingerprint f . To fill the leftmost cell of f ’s row in the matrix, Fluorescence finds the element in f that has the greatest similarity with the leftmost value of the basis. (I.e., choose the element e from f that maximizes $\sigma(\Lambda_0, e)$.) Store that element in the leftmost cell of f ’s row, and remove the element from f . Repeat the selection process for the remaining cells, moving from left to right across the row. When filling the cell at index j , if no remaining element of f has a positive similarity score with Λ_j , leave cell j empty.

2.3 Anomaly Detection

At its central server, Fluorescence performs two analyses to detect anomalous VMs. The first (§2.3.1), based on deep learning, detects anomalies by measuring a neural network’s ability to reconstruct (encoded) fingerprints. The network is able to reconstruct “typical” fingerprints more accurately than it can reconstruct anomalous ones. The second (§2.3.2) applies a clustering algorithm to distinguish between typical fingerprints (a large cluster, representing healthy VMs) and atypical ones (small clusters, representing anomalous VMs). The two algorithms have different strengths and weaknesses (§2.3.3) but generally agree in practice, so each serves to validate the other’s results.

The versions of these algorithms that we use in Fluorescence operate on data points that are represented as vectors of numbers. The matrix T that Fluorescence computed in §2.2.3, however, has cells filled with tuples. To prepare for anomaly detection, therefore, Fluorescence computes a new matrix, called S (for “similarities”), in which the tuples of T are replaced by similarity scores as follows:

$$S_{ij} = \begin{cases} \sigma(\Lambda_j, T_{ij}) & \text{if } T_{ij} \text{ is not empty} \\ -1 & \text{otherwise} \end{cases}$$

That is, each tuple is represented by its similarity to the corresponding element of the basis, and empty cells are represented by -1 . This transformation is not distance-preserving in principle,³ but it preserves the essential qualities of the

³If two fingerprints α and β have the same similarity to the basis in dimension j , then $\sigma(\Lambda_j, T_{\alpha j}) = \sigma(\Lambda_j, T_{\beta j})$. Thus $S_{\alpha j} = S_{\beta j}$ and in S , the

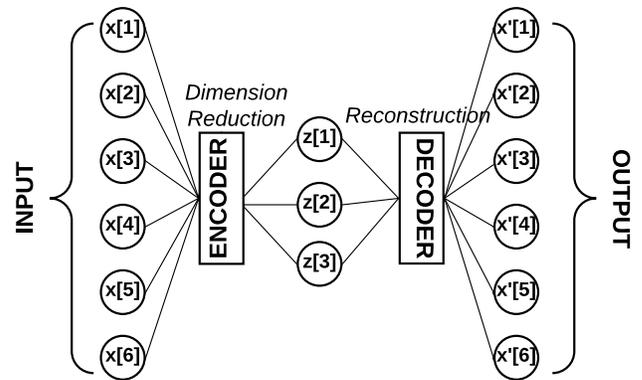


Figure 2: Autoencoder architecture.

fingerprints in practice. If two fingerprints are “close to” the basis in a particular dimension (column) in T , they remain close to each other in that dimension in S . Similarly, fingerprints with unique and/or rare contents (i.e., that have tuples in columns of T where most fingerprints have none) continue to be distinguished in S .

2.3.1 Deep Learning Approach

Fluorescence’s deep-learning approach to detecting anomalous VMs is based on an *autoencoder*. The purpose of an autoencoder is to learn, in an unsupervised manner, an efficient method for representing a dataset. As illustrated in Figure 2, an autoencoder contains an encoder, which reduces the number of dimensions in an input, and a decoder, which attempts to reconstruct the original input from its reduced representation. To minimize the error between the input and output, an autoencoder must learn to preserve the maximum information while encoding. The major patterns within the input dataset are learned and preserved, and as a consequence, the minor patterns—found in “outliers”—are lost.

Fluorescence leverages these properties to identify the rows in S that represent anomalous VMs. It trains an autoencoder over the rows of S . The encoder reduces the dimensionality of each row vector to 1/30th of its original (with a minimum of two encoded dimensions), and the decoder attempts to reconstruct the original vector. The learning goal is to minimize the sum of mean square errors between all the corresponding inputs and outputs, i.e., to minimize $\sum_{i=1}^m \sum_{j=1}^n (S_{ij} - S'_{ij})^2$, where m is the number of VMs (rows of S), n is the number of features per VM (columns of S), and S'_{ij} is the “reconstructed value” that corresponds to S_{ij} .

After training the autoencoder, Fluorescence calculates the *error score* of each VM as the squared error between its original vector representation in S and the reconstructed vector in S' . The VMs that have significantly larger error scores are identified as anomalies. To find such scores, Fluorescence models the error score as a Gaussian distribution. The VMs

distance between the fingerprints in dimension j is zero. In general, however, $T_{\alpha j}$ and $T_{\beta j}$ may not be identical.

that have error scores within an acceptable confidence interval of this distribution are considered to be normal, and all others are flagged as anomalous.

The encoder and decoder networks are 1-layer fully connected neural networks having dimensions $D \times 2$ and $2 \times D$, respectively, where D is the input encoding dimension. For optimization, we use ADADELTA [45] as the optimizer and cross-entropy loss as the loss function. The autoencoder model is trained on the entire input data for 50 epochs, with a mini-batch size of 2.

2.3.2 Clustering Approach

Fluorescence’s clustering approach to identifying anomalous VMs uses DBSCAN [11], a density-based algorithm. (Unlike some other methods, such as k -means, density-based clustering does not require prior knowledge of the number of clusters to be formed.) The Euclidean distance d between two vector rows of S , representing two VMs, is computed in the usual way: $d(\alpha, \beta) = \sqrt{(S_{\alpha 1} - S_{\beta 1})^2 + (S_{\alpha 2} - S_{\beta 2})^2 + \dots}$

DBSCAN requires two parameters: ϵ , which is a distance threshold, and m , which is the minimum number of points needed to form a cluster. DBSCAN builds a cluster by choosing an unvisited data point, “expanding” toward all of the neighboring points that are within ϵ of that point, and then recursively expanding from each of those neighbors. A point is marked as an outlier if it is not a member of a cluster that contains at least m points.

Fluorescence’s goal is for all normal VMs to be contained in one or more large (many-member) clusters and for anomalous VMs to be either (1) contained in small clusters or (2) classified as outliers. To do this, we need to choose appropriate values for ϵ and m .

To illustrate how we choose ϵ , we performed experiments in which we deployed herds of similar VMs with some instances infected with malware, measured the distance between every pair of VMs, and plotted the distances as a CDF. Figure 3 summarizes six of these experiments: in each, we deploy 95 normal VMs and five that are compromised with one type of malware. Note the “plateau” in the CDF of each experiment. The slope to the left of the plateau corresponds to VM-pairs that are near each other and that we would like to cluster together. The slope to the right corresponds to VM-pairs where the VMs are far apart, and where we would like the VMs not to be clustered together. Thus, the plateau represents the difference between intra-cluster distances and inter-cluster distances for a given dataset; a good choice of ϵ is one that lies near the left edge of the plateau. Based on our tuning experiments, we set ϵ to 100. Tuned in this way, Fluorescence can collect normal VMs into a small number of clusters—ideally, a single cluster—each with many members.

We set m to three so that DBSCAN will cluster even small numbers of VMs. This allows Fluorescence to identify clusters of anomalous VMs, e.g., VMs infected by the same

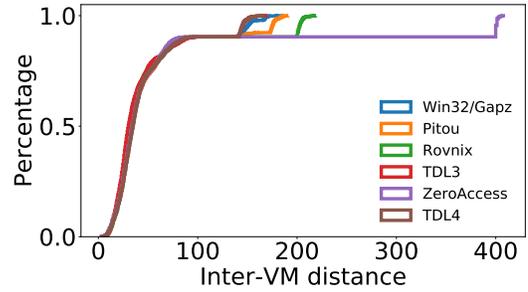


Figure 3: CDFs of all inter-VM distances in six tuning experiments. Each experiment involves 100 VMs, where five are infected with a single kind of malware.

malware (§4.2.2). After clustering, Fluorescence reports all outliers and all members of small clusters as anomalous.

2.3.3 Discussion

In cases where a small number of VMs are infected with malware, Fluorescence’s deep-learning and clustering techniques are likely to identify the same VMs as anomalous, and thus serve to validate each other. In other circumstances, however, the two approaches have different strengths and weaknesses.

One advantage of the autoencoder-based approach is that it does not require one to tune the algorithm by choosing clustering parameters: the autoencoder simply learns the dominant patterns in the dataset and can identify the data points that are most atypical. However, this comes at a cost in three ways. First, in the common case where no VMs are infected, the autoencoder may be overly sensitive. If the error scores of all VMs are small, then the threshold for flagging anomalies is also small. In these cases, the clusters and intra-VM distances computed by the clustering method can be used as a sanity check. Second, in the rare case that many VMs are infected, the autoencoder may start to become insensitive: it may learn the patterns of the infected VMs, even if those VMs are in the minority (say, 10%). In comparison, the clustering method is less sensitive, and can be accurate even when the number of infected VMs is high, as long as the uninfected VMs constitute the largest cluster. Third, while the autoencoder method can detect anomalies, it cannot report which anomalies are similar to each other. DBSCAN, of course, can compute clusters among the anomalous VMs.

The main shortcoming of the clustering approach is that the threshold for building clusters, ϵ , may be too “generous.” If the data points for normal VMs are actually much closer to each other than ϵ , then a poor choice of ϵ means that anomalous VMs that are nevertheless “within ϵ ” will be classified as normal by DBSCAN. In contrast, in this situation, the autoencoder would essentially learn the threshold automatically. Similarly, consider a situation in which there are two types of VMs (e.g., two different kernels) that are similar, and the data points of these two types form “overlapping” regions when

measured in *S*. DBSCAN may group instances of these VMs into a single cluster and thereby obscure small anomalies in each type. Conversely, the autoencoder could potentially differentiate between the types and identify anomalous VMs at a finer grain.

3 Implementation

Fluorescence depends on its fingerprinting agents (§2.1) to (1) “snapshot” the kernel code pages within all of the monitored VMs and (2) perform normalization, so that benign differences do not mask actual anomalies. This section provides implementation detail about how the agent performs these tasks. The agent requires some kernel- and architecture-specific knowledge to carry out these tasks; our implementation works with VM guests that run Linux (3.13–4.15) or Windows 7 kernels for x64 within VMs managed by Xen 4.9. The agent is implemented in C and uses libVMI [31] to manage and access the monitored VMs.

3.1 Walking the Page Table

To obtain all of the executable pages from the kernel of a VM guest, the Fluorescence agent performs a breadth-first traversal of a page table, starting from the KPGD. Since hypervisors typically maintain a KPGD for each VM, the KPGD can be easily accessed through VMI libraries. (The kernel-specific knowledge about the location of the KPGD is encapsulated within libVMI.) Using knowledge of x86_64 page tables, it is straightforward for Fluorescence to find all of the kernel code pages.

In practice, ensuring that the kernel code pages are in memory is a concern. As previously described (§2.1.1), some kernels can swap their own code pages to external storage, where they are not easily accessible through VMI. In particular, we found that Windows 7 swaps its own code pages aggressively: if a page is not used for long time, Windows may move the page to external storage or mark it as “in transition” (another kind of invalid page table entry). If the Fluorescence agent encounters such a page, the resulting fingerprint will be incomplete, possibly leading to spurious anomaly reports.

To avoid problems caused by swapped-out pages, Fluorescence invokes a tool on VMs running Windows, prior to taking a fingerprint, that pins all of the guest kernel’s code pages into memory. Fluorescence uses DRAKVUF [23] for this purpose. DRAKVUF is a system, built atop libVMI, that enables one to hijack a process to run an injected executable. Fluorescence injects code into the `winLogon` process, which is a privileged process present on every Windows machine, that loads and runs a kernel module that pins the kernel’s executable pages. The Fluorescence agent does this before each fingerprint of a Windows guest, in order to catch any pages that might have been injected since the VM was last fingerprinted. We did not develop a similar page-pinning tool for Linux guests, because none of the Linux kernels we used ever swapped any of their code pages out.

3.2 Normalizing Code-Page Content

When a kernel is loaded into memory, the loader patches code pages to replace symbolic references (to code or data) with the actual addresses of the things being referenced. Due to ASLR and other factors, when the same kernel is loaded into two different VMs, the loaded copies of the kernel will be patched with different addresses (§2.1.2).

Normalization aims to reduce these differences by replacing patched addresses with constants, in a way that is *consistent* across all of the VMs. One can think of this as undoing the patching, replacing all resolved references with symbolic ones. Consider an original (unloaded) code page *P*. In every loaded copy of *P*, every patched reference to a function *f* should be replaced with a constant that represents *f*—the same constant in all copies of *P*. To preserve as much information as possible in the normalized pages, a “mostly unique” constant should be used for each referent: e.g., if a page refers to two functions *f* and *g*, the constants chosen to represent references to *f* and *g* should be different.

To perform this replacement, the Fluorescence agent must solve two problems. It must determine the constants it will use to replace patched references, and it must find the references.

Determine the constants. To find constants in a consistent way across all of the protected VMs, Fluorescence leverages the fact that a kernel image consists of many loaded “objects” (i.e., object files). A single object generally defines many functions and variables, and these things appear at different offsets within the loaded object. ASLR may randomly arrange the objects in the kernel’s memory, but it does not rearrange the contents *inside* the objects. The offset of a function or variable within its containing object is thus constant across all of the kernels that have loaded that object. Fluorescence uses these constants to normalize loaded pages: given a resolved reference to a function or variable *f*, it replaces that reference with the byte offset of *f* within the object that contains *f*.

More specifically, Fluorescence divides the address space of a kernel into a number of 4 KB-page-aligned *regions* of virtually contiguous memory. (Certain address ranges are excluded, based on knowledge of how kernels use their address spaces: e.g., the “direct mapping” region in Linux, and the system cache region in Windows, are excluded.) Fluorescence makes regions that correspond to the kernel’s loaded objects by searching for objects in the appropriate area of the kernel’s address space. For Windows, the search is easy: the Portable Executable (PE) header value “MZ” appears at the start of each loaded object. For Linux, the search is also straightforward. Although ELF headers are not present in memory, loaded ELF objects follow a common pattern: a series of executable (code) pages, followed by some read-only (data) pages, and then by some writable (data) pages. Using these patterns, it is simple to create regions that correspond to the kernel’s loaded objects.

After making regions for objects, Fluorescence creates re-

gions that cover the remainder of the kernel’s address space (with some exceptions as previously noted). Whenever a normalization function wants to replace a resolved reference with a constant, Fluorescence determines the region that the reference points into. It replaces the reference with the difference between the referenced address and the region’s start.

Find the references. Our implemented Fluorescence agent can provide up to three normalized versions, a.k.a. feature views (§2.1.2), of every kernel code page. The first is the page content just as it appears in the VM, or the *original* view. If a page is not modified at all by the kernel loader, then all of the copies of that page will be identical in this view. The second feature view, called *sub-base*, and the third, called *disassembly*, both modify the original page content by heuristically searching for resolved references and then replacing those references with constants as described above. The sub-base and disassembly views differ in how they attempt to find resolved references within the code.

The sub-base view is simple, treating the input page as an array of eight-byte elements. The sub-base normalization function examines each element: if the value of an element can be interpreted as an address that falls within a defined region, that element is replaced by a constant (i.e., the difference between the element value and the referenced region’s start). This heuristic is very fast, correctly transforms constructs like jump tables that are eight-byte aligned, and “works” because the 64-bit address space is sparsely filled. In general, however, the sub-base method may overlook many resolved references (e.g., those that are not eight-byte aligned) and may modify values that are not actually addresses.

The disassembly view attempts to address the shortcomings of the sub-base view. Given a code page, the disassembly normalization function uses Capstone [33] to interpret the code found on the page. It searches the disassembled code for operands that appear to be (64-bit) absolute addresses and (32-bit) PC-relative addresses. Whenever one of these appears to point into a defined region, the normalization function replaces the address with the appropriate constant as described above. The disassembly view is often more precise than the sub-base view, but it is still heuristic; in our experience, it is still subject to false positives (incorrect modifications) and false negatives (overlooked references). This is why Fluorescence relies on multiple feature views of each code page: when one view “fails,” another may succeed.

4 Evaluation

We evaluate our Fluorescence implementation by testing its ability to identify VMs that are infected with kernel-resident malware, picking those VMs out of a mostly healthy herd of similarly configured VMs running Linux or Windows 7. We focus on answering four questions. *First, can Fluorescence correctly identify the VMs in the herd that are infected with malware (§4.2)?* In our experiments, the answer is yes: Fluorescence accurately identified the infected VMs. We

discuss the performance of the autoencoder (§4.2.1) and clustering (§4.2.2) detection methods. *Second, is normalization necessary and effective (§4.3)?* We find that the answer is yes: our implemented normalization process can drastically reduce the number of features that need to be considered during anomaly detection, and it allows different classes of VMs to be clustered for identification. *Third, what is the run-time performance of fingerprint generation (§4.4)?* In our experiments, fingerprint generation takes 11–13 s, and the monitored VM is paused for only 0.6 s. *Fourth, how does the run time of Fluorescence scale as the number of monitored VMs increases (§4.5)?* We find that the run time of Fluorescence is acceptable even for moderately sized herds: Fluorescence can process a herd of 200 VMs in about an hour. Larger herds can be handled by treating them as multiple sub-herds, each analyzed by a separate instance of Fluorescence.

4.1 Experiment Setup

We deploy herds containing various numbers of VMs running Linux or Windows. We use Xen 4.9 as the hypervisor and run Ubuntu 16.04 within dom0. Each Linux VM runs (64-bit) Ubuntu 16.04 as the guest, with Linux kernel version 4.4.0. Each Windows VM runs (64-bit) Windows 7. Each VM is configured with one virtual CPU and 1 GB RAM. The VMs are distributed over twenty “d430” physical machines within the Utah Emulab network testbed [43].⁴ Each d430 has two 2.4 GHz Intel Xeon E5-2630v3 8-core CPUs and 64 GB RAM. Each VM-hosting machine runs an instance of the Fluorescence agent in its dom0. Fluorescence’s central server, which performs feature alignment and anomaly detection, is located on a separate d430 (hosting no VMs) that runs Ubuntu 16.04.

We collected samples of kernel-resident malware to use in our experiments. We focus on malware that persists in the kernel by injecting or modifying kernel code, because that is the type of malware that Fluorescence is designed to detect. For Windows, we collected samples of Pitou, Rovnix (a.k.a. Cidox), ZeroAccess (a.k.a. Sirefef), Win32/Gapz, and two variants of TDSS known as TDL3 and TDL4. Some of these have recently been active in the wild [41]. For Linux, we collected three rootkit samples: Diamorphine [26], Nurupo [29], and Reptile [1].

For experiments with Linux VMs, we configured Fluorescence to compute all three feature views of every page (§3.2). For experiments with Windows VMs, we configured Fluorescence to compute only the *original* and *sub-base* views. We did this because the Windows 7 kernel uses Position Independent Executables (PIE); as a result, most code pages are unpatched by the kernel-loading process and are already identical across VMs. We leave *sub-base* enabled in our Windows 7 experiments to handle the small number of code pages that do vary across loaded Windows 7 kernels.

⁴We used Emulab’s “experiment firewall” feature to block traffic between the VMs and the Internet.

Fluorescence transfers VMs’ fingerprints to its central server for analysis. For the VMs described above, we found that that size of a Windows VM fingerprint was approximately 2.7 MB. A Linux VM fingerprint was approximately 1.8 MB. These numbers compare favorably to prior work [3] that collects and analyzes physical memory dumps.

4.2 Malware Detection

We performed a set of experiments to test Fluorescence’s ability to identify the VMs in a herd that are infected with kernel-resident malware. For each test, we set up the herd as follows. We created 50 VMs, running either Windows or Linux. To perturb the collection, we randomly selected ten VMs, logged into them, and performed some activities manually. On Windows, we opened some files, played some small games, and/or used a web browser to download some files. On Linux, we ran Apache or nginx and then downloaded random files from those servers. We then randomly installed malware on some of the VMs. For Windows, we chose a subset of our six samples to inject (1–6 samples); for each sample, randomly chose the number of instances to inject (1–4); randomly chose the set of VMs to be infected (one VM per instance); and then injected the samples. For Linux, we followed the same process, injecting 1–3 rootkit types, with 1–4 instances each.

After setting up the herd, we ran Fluorescence to test its ability to discover the infected VMs. For each herd configuration that we selected, we repeated the herd setup and Fluorescence test five times to check for repeatability.

In every test we performed, Fluorescence identified the infected VMs. The DBSCAN method always found the infected VMs, without false positives or negatives, and properly clustered the VMs that were infected with a common type of malware. The autoencoder method was also accurate but produced false positives at negatives when a large fraction of the VMs were infected (as discussed below, §4.2.1).

Figure 4 explains this result. It visualizes S , the similarity matrix (§2.3), for one of our tests involving all six of our Windows malware samples. Each row represents a VM, and each column represents a feature, i.e., a “matching page” across the VMs. We sort the rows and columns for clarity in the visualization. Each cell is colored according to its value: light cells contain high values (i.e., are similar to the corresponding basis value) and dark cells contain low values (are dissimilar to the corresponding basis value). In the visualization, it is clear that the infected VMs present distinct patterns in S , and each type of malware has a different signature. These are the patterns that Fluorescence detects. The figure is annotated to show the different malware families.

Figure 8 presents a similar visualization for one of our Linux tests, one involving all three of our Linux rootkits.

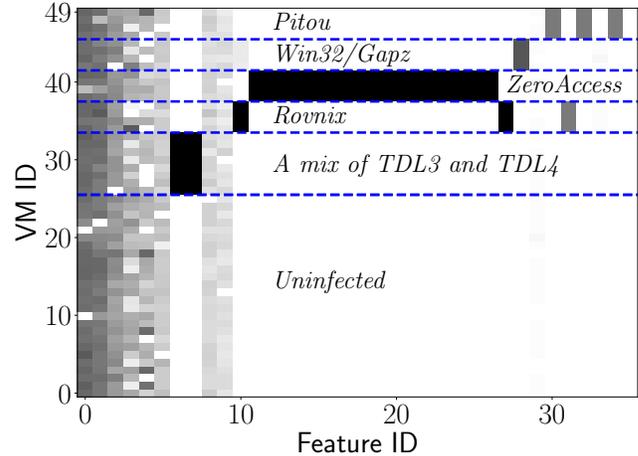


Figure 4: Similarity matrix for an experiment involving 50 VMs running Windows, many infected with malware. In this visualization, higher similarity scores have lighter colors.

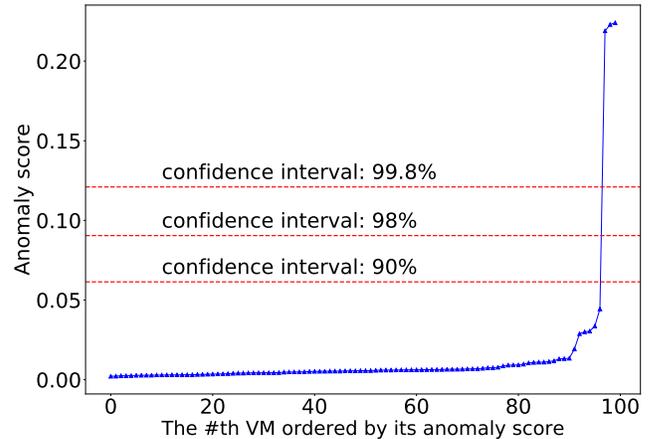


Figure 5: Error scores from the autoencoder-based analysis of an example herd of 100 VMs, three of which are infected with malware. The scores of the infected VMs (upper right) are flagged because they exceed a threshold, as determined by a confidence interval.

4.2.1 Anomaly Detection with Autoencoder

We conducted another set of experiments to better characterize the performance of Fluorescence’s autoencoder-based method for detecting anomalies. Recall that the autoencoder method calculates an error score for each monitored VM (§2.3.1). It models the error score as a Gaussian distribution; if the error score of a VM lies outside a chosen confidence interval, Fluorescence flags the VM as anomalous. Figure 5 illustrates this idea. To generate this example, we created and processed a fingerprint set for 100 VMs, three of which were infected with malware. We sorted the resulting scores and plotted the results. The three points at the right side of the figure, with scores above 0.20, correspond to the infected VMs. For confidence interval choices between 90%

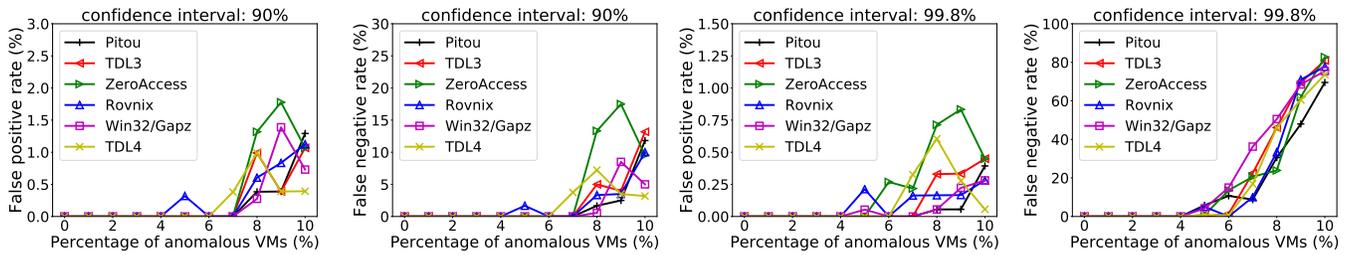


Figure 6: Autoencoder performance, with 90% and 99.8% confidence intervals, for different types of Windows kernel malware and rates of infection in a herd of 100 VMs. Points represent averages over twenty trials.

and 99.8%, all of these VMs are reported to be anomalous, and there are no false positives or negatives.

To perform a large set of experiments similar to the one described above, we reused the fingerprints of Windows VMs that we collected during the experiments described in §4.2. In each of these new experiments, we simulate a herd of 100 VMs by randomly selecting 100 fingerprints from our set of previously generated fingerprints. Each experiment involves one kind of Windows malware, and we vary the number of infected VMs from zero to ten. (I.e., in each configuration, we choose 0–10 fingerprints of VMs infected with the chosen malware, and 90–100 fingerprints of clean VMs.) We then perform anomaly detection over the assembled herd, using the autoencoder method, and record the number of false positives and false negatives reported at the 90% and 99.8% confidence intervals. To account for randomness introduced by the autoencoder—e.g., randomly initialized parameters—we repeat the analysis of each herd twenty times. For each herd, we compute and report the average false positive and false negative rates over the twenty trials.

Figure 6 presents the results of these experiments. (Note the varying ranges of the y-axes in the figure.) We make three observations about this data. First, if the rate of anomalous VMs was under 4%—a realistic threshold for large herds in practice—the autoencoder detected all anomalies without any false positives or false negatives. Second, when the rate of anomalous VMs increases beyond 4%, the autoencoder becomes less effective. Third, the higher confidence interval leads to more false negatives, but also to possibly fewer false positives. This suggests an incremental strategy to maintaining a herd: fix reported VMs, thus lowering the rate of infection, and then repeat analysis with Fluorescence. From our experiments, we conclude that the autoencoder method works well when few anomalies are present, which is likely to be the case in practice. Despite its limitations, it is widely applicable for detecting unknown anomalies.

4.2.2 Anomaly Detection with DBSCAN

We reuse the simulated herds described in §4.2.1 to test the performance of Fluorescence’s clustering approach, utilizing DBSCAN, for detecting anomalies. Recall that each herd is

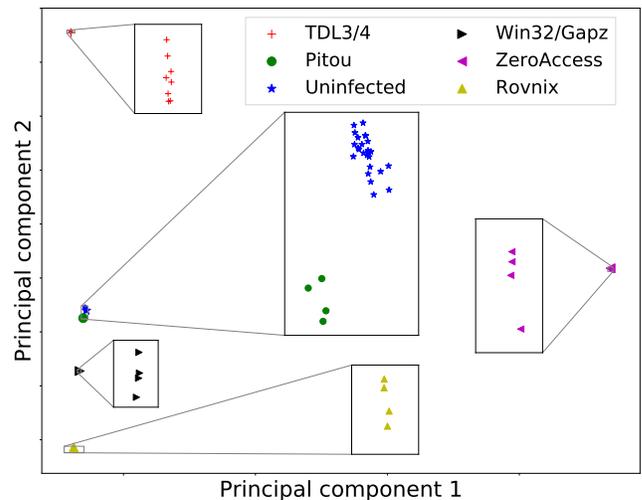


Figure 7: Visualization of clusters found by DBSCAN within a herd of 100 Windows VMs. Using principal component analysis (PCA), the data for each VM was reduced to a 2D coordinate.

represented by the fingerprints of 100 VMs with Windows 7 guests, where 0–10 of those guests have been infected by one type of malware. In each of these tests, DBSCAN correctly partitioned the normal and infected VMs into separate clusters, with no false positive or negatives.

To visualize the reason for DBSCAN’s success in our tests, we simulated another herd of 100 Windows VMs. In this herd, we included fingerprints of VMs infected with all of our Windows malware samples: four instances of each malware sample, for a total of 24 infected VMs. We analyzed this herd with Fluorescence—again, all the VMs were properly classified and clustered by family—and obtained the similarity matrix S for the herd. We used principal component analysis (PCA) to reduce the number of features for each VM to just two. Finally, we used the two values for each VM to plot the VMs on the X-Y plane.

Figure 7 shows the result. Each color/shape represents one cluster identified by DBSCAN. The green-dot and blue-star clusters are relatively close to each other, but they are identified as separate clusters, as shown in a zoom-in view. The blue-star cluster has the greatest number of VMs among

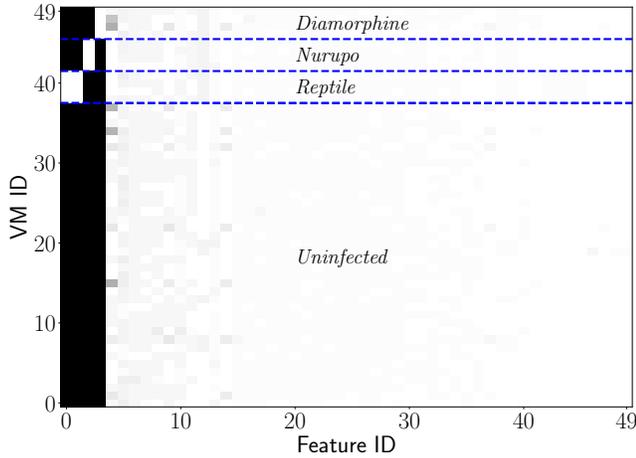


Figure 8: Similarity matrix for an experiment involving 50 VMs running Linux, many infected with malware. In this visualization, *higher* similarity scores have lighter colors.

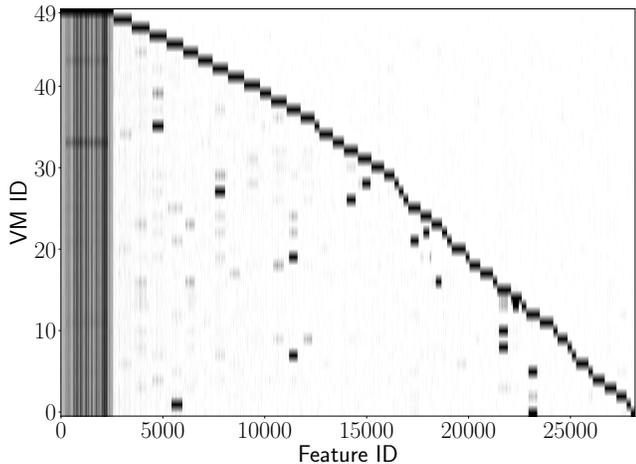


Figure 9: Similarity matrix for the 50 VMs portrayed in Figure 8, but with *sub-base* and *disassembly* normalization disabled. In this visualization, *lower* similarity scores have lighter colors.

all clusters, so the VMs in this cluster are identified as normal while the other, smaller clusters represent different kinds of anomalies (i.e., VMs infected by different families of rootkits). This analysis by DBSCAN matches the ground truth.

4.3 Impact of Normalization

Fluorescence performs normalization to reduce benign differences in the contents of acquired kernel pages. We performed an experiment to assess the effectiveness of our implemented normalization procedure.

Figure 8 visualizes the similarity matrix S for a herd of 50 Linux VMs, many of which are infected with malware. As explained previously for Figure 4, each row represents a VM, each column represents a feature, and cells are shaded according to their values. We have sorted the rows and columns for

Step	Windows 7	Linux
Pin kernel code pages	6.4	N/A
Copy kernel code pages	0.6	0.6
Compute kernel memory regions	0.6	0.6
Normalization	0.9	7.7
Hashing	2.7	4.2
Total	11.2	13.1

Table 2: Time (secs) to generate a fingerprint. The VM being fingerprinted is paused only while pages are being copied (step 2).

clarity. The visualization makes the different groups of VMs apparent; each has a pattern that Fluorescence detects.

We removed the *sub-base* and *disassembly* feature views from the fingerprints of this herd—leaving only hashes for the *original* page contents—and analyzed the resulting fingerprints to produce another similarity matrix. Figure 9 visualizes this result. (In Figure 9, low values are light and high values are dark; this is the opposite of the convention used in Figure 8.) Two things are immediately apparent. First, compared to the original matrix, the number of features per VM has increased by more than two orders of magnitude. This happens because Fluorescence is no longer able to remove content that is ubiquitous across the VMs; normalization is necessary for finding such content and is very effective. Second, the patterns that are so clear in Figure 8 are not apparent in Figure 9. This suggests that Fluorescence would have a difficult time finding the malware-infected VMs in this data.

4.4 Fingerprint Generation Time

We measured the time needed for the Fluorescence agent to produce the fingerprint of a VM. We ran the fingerprinting procedure for a Windows VM and a Linux VM and recorded the elapsed time for each step of the process. We repeated the procedure ten times for each VM and computed the average elapsed times over the trials. Table 2 presents our results.

The VM being fingerprinted is paused only while Fluorescence copies its kernel code pages (§2.1). For both Windows 7 and Linux, the pause time is short, less than a second.

For Windows 7, the longest step is the one that uses DRAKVUF to pin the kernel’s code pages into memory (§3.1). (The pinning step is unnecessary for our Linux VMs because the Linux kernel does not swap-out its code.) Although pinning for Windows takes several seconds, the VM continues to run during that time. The pinning procedure forces about 2,000 pages to be present in memory, or about 8 MB. Of those, it is common for half to be marked as “in transition,” meaning that the page contents are in memory but marked as inaccessible; the kernel can quickly make those pages present again. In this way, the pinning procedure imposes about a 4 MB overhead on the Windows kernel.

For Linux, the most time-consuming step is normalization. For Windows 7, we configured Fluorescence not to compute the *disassembly* feature view (§4.1). Fluorescence spends sev-

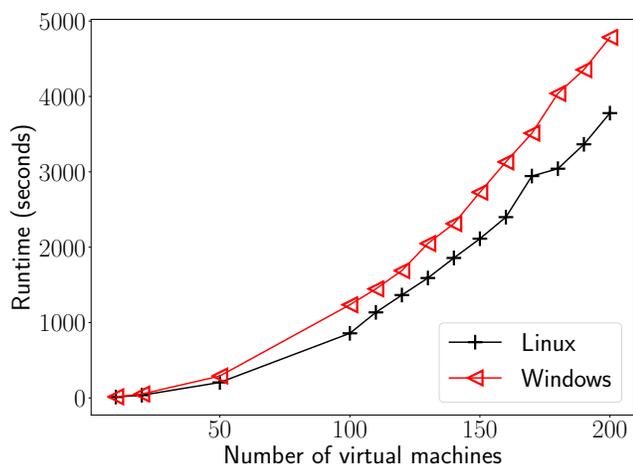


Figure 10: Time required for Fluorescence’s central server to analyze VM herds of varying sizes.

eral seconds performing this normalization on Linux kernel pages, and in our experience, it is necessary in order to get good anomaly detection results (§4.3).

4.5 Scalability

Because Fluorescence operates on herds of VMs, it is important to understand the performance of Fluorescence on herds of VM of various sizes. Again, we reused the fingerprints of VMs that we collected in earlier experiments to simulate herds of Windows and Linux VMs of varying sizes, from 10 to 200 VMs. For each herd, we measured the time required for Fluorescence’s central node to analyze the collected fingerprints, i.e., to perform both feature alignment and anomaly detection. Anomaly detection is fast—less than a minute in all of our tested configurations—so the majority of the time is spent on feature alignment.

Figure 10 presents the results of these experiments. In brief, Fluorescence required less than ten minutes to analyze each 50-VM herd. It analyzed our herd of 200 Linux VMs in approximately 63 minutes, and it analyzed our herd of 200 Windows VMs in approximately 80 minutes. We believe that this performance is reasonable for periodically measuring the health of a VM herd. Moreover, Fluorescence scales horizontally. For monitoring a herd containing more than a few hundred VMs, one can divide the herd into subherds, each monitored by a separate instance of Fluorescence.

To put these results in context, we note that Bianchi et al. reported [3] that their Blacksheep system, which looks for kernel-level anomalies in herds of Windows machines, needs ten minutes to compare two 1 GB memory dumps. In that time, Fluorescence can search for kernel-level anomalies in at least 50 VMs.

5 Security Analysis

Fluorescence aims to detect kernel-resident malware within a large group of similarly configured VMs, and its design is based on three main assumptions.

The first is that the malware to be detected within the VMs cannot compromise the virtual machine monitors (VMMs) on which those VMs run. In other words, Fluorescence assumes that the VMMs are trustworthy. This assumption allows Fluorescence to fingerprint the monitored VMs efficiently by running its agents on the same physical hosts as the monitored VMs (§2, Figure 1); the agents use virtual machine introspection to access the memory of the monitored VMs and read their kernel code pages (§2.1). If malware is able to compromise the VMMs, then the Fluorescence agents may be disabled or otherwise compromised as well, and the fingerprinting process may not be trustworthy. VMM integrity continues to be an important area of concern [30], but it is not the concern that Fluorescence is intended to address.

The second assumption is that the VMs being monitored are similar to each other in terms of configuration. They boot from a single “golden image,” which is a common practice in cloud-based application deployments [6], and therefore they have the same kernel patches applied and the same kernel modules installed, at least at boot time. Like the VMM integrity assumption, the VM similarity assumption helps Fluorescence to be efficient: normalization accounts for anticipated but benign differences (§2.1.2), and normalized pages that are identical across all of the VMs can be removed from fingerprints (§2.2.1), greatly speeding up subsequent analysis. More significantly, the similarity assumption allows Fluorescence to automatically identify anomalous VMs (§2.3), because their fingerprints are most different from the basis that Fluorescence computes (§2.2.2).

The third assumption is also related to automatic anomaly detection. Fluorescence assumes that, in a large group of initially healthy VMs, malware-infected VMs will be the exception, not the rule (§2.3). This assumption, which is also made by prior work [3], allows Fluorescence to distinguish “healthy” VMs (the majority) from “abnormal” ones (the minority) without kernel-specific knowledge.

The second and third assumptions introduce the risk of misclassification. Our experiments showed, for example, that at infection rates above 4%, the autoencoder-based detector produced both false positives and false negatives (§4.2.1). While our experiments with DBSCAN produced no false positives or false negatives (§4.2.2), it is conceivable that an attacker could design malware in a way that would cause Fluorescence to overlook it. For example, an attacker could learn the distribution of pages known to be very different across benign VMs (e.g., ten pages in the Windows kernel) and figure out how to inject code only in those pages. The code injected into each VM would need to be unique to prevent Fluorescence from clustering the infected VMs; just a few similar features are

enough for Fluorescence to differentiate healthy VMs from infected ones (Figure 4, Figure 8). We believe that hiding kernel-resident malware from Fluorescence in this way would require a great deal of sophistication and effort.

If healthy VMs are dissimilar from one another, then Fluorescence would need reconfiguration—or additional information—in order for it to automatically identify anomalous VMs. Consider a herd of healthy VMs in which half have a particular kernel module installed and half do not. This might happen, for example, if the herd is in the middle of an upgrade. Such a split is likely to (1) decrease the effectiveness of the autoencoder-based detector and (2) cause the DBSCAN-based detector to divide the healthy VMs into two clusters. A cloud administrator could deal with such a split in two ways. The first way is to run two instances of Fluorescence, one for each class of VM; the administrator would migrate VMs from one instance to the other as the VMs are upgraded. The second way is for the administrator to *manually* label healthy clusters in some fashion, so that Fluorescence would not need to assume that only the largest cluster is healthy. (This would require a change to Fluorescence, and it would be a kind of kernel-specific knowledge being added to the classifier.) In practice, the first approach is likely to be preferable, because manual labeling would not improve the performance of the autoencoder-based classifier.

The current implementation of Fluorescence does not consider dynamically generated code (“JIT-compiled code”), which can cause the kernel code pages of healthy VMs to diverge. If a VM’s kernel contains JIT-compiled code, it will likely be identified as an anomaly. Legitimately JIT-compiled kernel code, such as that produced by eBPF, is typically verified before it is executed, and as such can generally be considered to be benign. Code pages created by eBPF can be recognized by the magic code `0xb9f` [18], but it would be an obvious security problem for Fluorescence to simply ignore pages marked with this code. We leave the development of appropriate normalization (§2.1.2) methods for JIT-compiled kernel code as future work.

6 Related Work

We classify existing rootkit detectors into three general categories: baseline-based approaches, integrity-protection approaches, and comparative anomaly-detection approaches. By contrasting Fluorescence’s herd anomaly-detection approach to systems in these categories, and we show that Fluorescence advances the state of the art.

Baseline approaches. Many rootkit detectors [2, 5, 9, 28, 34, 42] require the computation or collection of a *baseline* prior to deployment, such as a sample of a known rootkit (a negative baseline) or a sample of an uninfected operating system (a positive baseline). Hancock [16] and Hamsa [24] generate signatures of rootkit samples; others [8, 19, 35, 36, 44] learn from known rootkits’ behavior and generate patterns to match future rootkit execution. These approaches are lim-

ited because they can only detect rootkits similar to known signature or behavior patterns. Invariant-based detection [10, 14, 25, 32] establishes a correct view of key kernel structures or state, and detects anomalies when that state invariant is violated. These techniques require comprehensive knowledge of specific kernels and assume kernel source code availability. In contrast, Fluorescence requires no baseline and minimal kernel-specific knowledge, which enables it to detect anomalies in both Windows and Linux VMs.

Integrity protection. Some systems periodically check the integrity of kernel critical components, including code and key data structures. Although not yet fully adopted by the Linux kernel, Kernel Patch Protection [13] and Driver Signature Enforcement [17] have been applied to 64-bit Windows and have raised the bar for kernel rootkit development [37]. However, if a rootkit evades these defenses, it can simply disable the protection [7]. The `pitou` rootkit [41], which affects Windows XP through Windows 10, infects the MBR, and bypasses kernel-mode code signing to load a malicious kernel driver. In contrast to integrity protection within a VM, Fluorescence works outside the VM, and thus cannot be disabled by a rootkit unless it escapes the VM and executes code in the hypervisor; this is considerably more difficult.

Comparative approaches. Baseline-based tools may be difficult to deploy, due to the difficulty of constructing an appropriate baseline sample. Cross-view detection, applied in tools like GMER [15] and RootkitRevealer [38], compares multiple different views of the same system state to find inconsistencies [4]. These approaches require significant manual effort to identify the system state to be observed and compared. Diffy [27] is a live cloud triage tool that provides mechanisms for both baseline- and clustering-based anomaly detection. It runs a special agent in each monitored VM to collect OS-level information. In contrast, Fluorescence detects anomalies without relying on in-VM software and semantics.

Blacksheep [3] makes the assumption that infection begins in a minority of a set of homogeneous machines. It detects anomalies by collecting memory dumps from the machines, extracting selected system information, and comparing pairwise to measure distance via clustering. Fluorescence starts from the same assumption as Blacksheep, but is designed to be less reliant on OS semantics and to detect code injection and modification attacks.

Because Blacksheep makes heavy use of OS semantics to exclude “benign differences” between VMs, its approach is difficult to port to non-Windows systems. For example, Blacksheep requires the PE header to handle load-time relocations, but object headers in Linux are removed during object load. Furthermore, the Linux kernel performs additional load-time code patching beyond object relocation and kASLR, e.g., in the `.paravirtualization` ELF section, and this noise should also be considered benign. Instead, Fluorescence uses very limited OS semantics, and its approach is viable on mul-

multiple versions of both Windows and Linux. For each memory dump, Blacksheep must transport gigabytes of files across a network for analysis. In contrast, a Fluorescence fingerprint is less than 3 MB. Blacksheep weaves its data summary and comparison together while making use of a large amount of OS semantics. Fluorescence decouples its data-summary and VM-comparison processes: by comparing VM fingerprints, rather than “raw” memory snapshots, Fluorescence enables scalable monitoring and clustering.

7 Conclusion

Fluorescence is a novel tool that detects kernel-resident malware infections within a “herd” of similar virtual machines. It uses virtual machine introspection-based observations to identify anomalies, without the need for training over specific anomalies. Previous work relied on knowledge of specific kernels to cross the “semantic gap” and compare kernel state, or made assumptions about the malware being searched for. Fluorescence’s more general approach to anomaly detection does not rely on information that is specific to a single target kernel: the fingerprinting procedure needs some low-level information to acquire and normalize kernel memory samples, but the anomaly-detection process needs no kernel- or architecture-specific information at all. Fluorescence is scalable because the examination and summarization of VMs happens in parallel across the VM hosts; the central server receives and operates on fingerprints, not full memory snapshots; and the feature-alignment and analysis algorithms are chosen and engineered to be fast. We report that Fluorescence can analyze a herd of 200 VMs in ~60–80 minutes. Large herds can be divided into subherds for faster analysis.

Acknowledgments

We thank VirusTotal and VirusShare for making malware samples available to us, T. Roy from CodeMachine Inc. for sharing his expertise about Windows rootkits, and Mingbo Zhang at Rutgers University for helping us to debug our Windows kernel-memory-pinning driver. We thank the anonymous RAID reviewers and our shepherd, Andrea Lanzi, for their valuable comments and help in improving this paper. This material is based upon work supported by the National Science Foundation under Grant Numbers 1314945 and 1642158.

References

- [1] Ighor Augusto. Reptile rootkit. Commit b0a2d0f, April 2018. URL <https://github.com/f0rb1dd3n/Reptile>.
- [2] Rishi Bhargava and David P. Reese, Jr. System and method for passive threat detection using virtual memory inspection, March 14, 2017. U.S. Patent 9,594,881 B2.
- [3] Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Blacksheep: Detecting

- compromised hosts in homogeneous crowds. In *Proc. CCS*, pages 341–352, October 2012. doi: 10.1145/2382196.2382234.
- [4] Bill Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones & Bartlett Learning, 2009. ISBN 978–1598220612.
- [5] Michael Boelen, John Horne, et al. The Rootkit Hunter project. Release 1.4.6, February 2018. URL <http://rkhunter.sourceforge.net/>.
- [6] Ed Bukoski, Brian Moyles, and Mike McGarr. How we build code at Netflix. Netflix Technology Blog, March 9, 2016. URL <https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15>.
- [7] Yuriy Bulygin, Mikhail Gorobets, Andrew Furtak, and Alex Bazhaniuk. Fractured Backbone: Breaking modern OS defenses with firmware attacks. Presentation at Black Hat USA, July 2017. URL <https://youtu.be/ryKy9LvmSIs>.
- [8] Chen Chen, Darius Suci, and Radu Sion. POSTER: KXRy: Introspecting the kernel for rootkit timing footprints. In *Proc. CCS*, pages 1781–1783, October 2016. doi: 10.1145/2976749.2989053.
- [9] Amit Dang, Preet Mohinder, and Vivek Srivastava. System and method for kernel rootkit protection in a hypervisor environment, June 30, 2015. U.S. Patent 9,069,586 B2.
- [10] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007. doi: 10.1016/j.scico.2007.01.015.
- [11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. KDD*, pages 226–231, August 1996. URL <http://www.aaai.org/Library/KDD/1996/kdd96-037.php>.
- [12] Nicolas Falliere, Liam O. Murchu, and Eric Chien. W32.Stuxnet dossier. White paper, version 1.4, Symantec Corporation, February 2011. URL https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.
- [13] Scott Field. An introduction to kernel patch protection. MSDN Blog, August 12, 2006. URL <https://blogs.microsoft.com/windowsvistasecurity/2006/08/12/an-introduction-to-kernel-patch-protection/>.

- [14] Francesco Gadaleta, Nick Nikiforakis, Jan Tobias Mühlberg, and Wouter Joosen. HyperForce: Hypervisor-enforced execution of security-critical code. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research: SEC 2012*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 126–137. Springer, June 2012. doi: 10.1007/978-3-642-30436-1_11.
- [15] GMER. GMER - rootkit detector and remover, 2016. URL <http://www.gmer.net/>.
- [16] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-Cker Chiueh. Automatic generation of string signatures for malware detection. In Engin Kirda, Somesh Jha, and Davide Balzarotti, editors, *Recent Advances in Intrusion Detection: RAID 2009*, volume 5758 of *Lecture Notes in Computer Science*, pages 101–120. Springer, September 2009. doi: 10.1007/978-3-642-04342-0_6.
- [17] Ted Hudek and Cymoki. Driver signing. Microsoft Windows documentation, April 2017. URL <https://docs.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing>.
- [18] The Kernel Development Community. BPF type format (BTF), 2019. URL <https://www.kernel.org/doc/html/latest/bpf/btf.html>.
- [19] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proc. USENIX Security*, pages 351–366, August 2009. URL https://www.usenix.org/legacy/events/sec09/tech/full_papers/kolbitsch.pdf.
- [20] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3(Supplement):91–97, September 2006. doi: 10.1016/j.diin.2006.06.015.
- [21] Jesse D. Kornblum. Exploiting the rootkit paradox with Windows memory analysis. *International Journal of Digital Evidence*, 5(1):1–5, Fall 2006. URL <http://www.utica.edu/academic/institutes/ecii/publications/articles/EFE2FC4D-0B11-BC08-AD2958256F5E68F1.pdf>.
- [22] Andrea Lanzi, Monirul Sharif, and Wenke Lee. K-Tracer: A system for extracting kernel malware behavior. In *Proc. NDSS*, February 2009. URL <https://www.ndss-symposium.org/ndss2009/k-tracer-system-extracting-kernel-malware-behavior/>.
- [23] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proc. ACSAC*, pages 386–395, December 2014. doi: 10.1145/2664243.2664252.
- [24] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proc. IEEE S&P*, pages 32–46, 2006. doi: 10.1109/SP.2006.18.
- [25] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proc. NDSS*, February 2011. URL <https://www.ndss-symposium.org/ndss2011/siggraph-brute-force-scanning-of-kernel-data-structure-instances-using-graph-based-signatures/>.
- [26] Victor Ramos Mello. Diamorphine rootkit. Commit ba97922, March 2018. URL <https://github.com/m0nad/Diamorphine>.
- [27] Forest Monsen and Kevin Glisson. Netflix Cloud Security SIRT releases Diffy: A differencing engine for digital forensics in the cloud. Netflix Technology Blog, July 17, 2018. URL <https://medium.com/netflix-techblog/netflix-sirt-releases-diffy-a-differencing-engine-for-digital-forensics-in-the-cloud-37b71abd2698>.
- [28] Nelson Murilo and Klaus Steding-Jessen. Chkrootkit. Version 0.52, March 2017. URL <http://www.chkrootkit.org/>.
- [29] nurupo. Nurupo rootkit. Commit 78faabd, December 2017. URL <https://github.com/nurupo/rootkit>.
- [30] Rajendra Patil and Chirag Modi. An exhaustive survey on security concerns and solutions at different components of virtualization. *ACM Comput. Surv.*, 52(1): 12:1–12:38, February 2019. doi: 10.1145/3287306.
- [31] Bryan D. Payne, Tamas K. Lengyel, Steven Maresca, Antony Saba, et al. LibVMI: Simplified virtual machine introspection. Version 0.12.0, April 2018. URL <https://github.com/libvmi/libvmi>.
- [32] Nick L. Petroni, Jr., Timothy Fraser, Aaron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. USENIX Security*, pages 289–304, July/August 2006. URL <https://www.usenix.org/legacy/events/sec06/tech/petroni.html>.
- [33] Nguyen Anh Quynh. Capstone disassembly framework. Version 3.0.5-rc2, March 2017. URL <https://github.com/aquynh/capstone>.

- [34] Jayakrishnan Ramalingam. Rootkit monitoring agent built into an operating system kernel, September 17, 2013. U.S. Patent 8,539,584.
- [35] Junghwan Rhee, Ryan Riley, Zhiqiang Lin, Xuxian Jiang, and Dongyan Xu. Data-centric OS kernel malware characterization. *IEEE Trans. on Information Forensics and Security*, 9(1):72–87, January 2014. doi: 10.1109/TIFS.2013.2291964.
- [36] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-aspect profiling of kernel rootkit behavior. In *Proc. EuroSys*, pages 47–60, April 2009. doi: 10.1145/1519065.1519072.
- [37] T. Roy. Personal communication, October 2016.
- [38] Mark Russinovich. RootkitRevealer v1.71. Microsoft Windows documentation, November 2006. URL <https://docs.microsoft.com/en-us/sysinternals/downloads/rootkit-revealer>.
- [39] Dan Sullivan. Beyond the hype: Advanced persistent threats. White paper, 2011. URL <https://www.realtimedpublishers.com/book.php?id=197>.
- [40] Symantec Corporation. Advanced persistent threats: A Symantec perspective. White paper, 2011. URL https://www.symantec.com/content/en/us/enterprise/white_papers/b-advanced_persistent_threats_WP_21215957.en-us.pdf.
- [41] Gianfranco Tonello. Bootkits are not dead. Pitou is back!, January 2018. URL https://www.tgsoft.it/english/news_archivio_eng.asp?id=884.
- [42] Trend Micro Inc. OSSEC, open source host-based intrusion detection system, 2019. URL <https://github.com/ossec/ossec-hids>.
- [43] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, December 2002. URL <https://www.usenix.org/legacy/event/osdi02/tech/white.html>.
- [44] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proc. NDSS*, February 2008. URL <https://www.ndss-symposium.org/ndss2008/hookfinder-identifying-and-understanding-malware-hooking-behaviors/>.
- [45] Matthew D. Zeiler. ADADELTA: An adaptive learning rate method. arXiv preprint arXiv:1212.5701, December 2012. URL <https://arxiv.org/abs/1212.5701>.

Now You See It, Now You Don't: A Large-scale Analysis of Early Domain Deletions

Timothy Barron
Stony Brook University

Najmeh Miramirkhani
Stony Brook University

Nick Nikiforakis
Stony Brook University

Abstract

Domain names are a valuable resource on the web. Most domains are available to the public on a first-come, first-serve basis and once domains are purchased, the owners keep them for a period of at least one year before they may choose to renew them. Common wisdom suggests that even if a domain name stops being useful to its owner, the owner will merely wait until the domain organically expires and choose not to renew.

In this paper, contrary to common wisdom, we report on the discovery that domain names are often deleted before their expiration date. This is concerning because this practice offers no advantage for legitimate users, while malicious actors deleting domains may hamper forensic analysis of malicious campaigns, and registrars deleting domains instead of suspending them enable re-registration and continued abuse. Specifically, we present the first systematic analysis of early domain name disappearances from the largest top-level domains (TLDs). We find more than 386,000 cases where domain names were deleted before expiring and we discover individuals with more than 1,000 domains deleted in a single day. Moreover, we identify the specific registrars that choose to delete domain names instead of suspending them. We compare lexical features of these domains, finding significant differences between domains that are deleted early, suspended, and organically expiring. Furthermore, we explore potential reasons for deletion finding over 7,000 domain names squatting more popular domains and more than 14,000 associated with malicious registrants.

1 Introduction

The Domain Name System (DNS) makes the modern web possible by allowing users to navigate by human readable names rather than machine route-able IP addresses. Domain names are often core to the identity of

a Web site so it is no surprise that they can sometimes hold significant monetary value depending on popularity, brand recognition, or specific keywords [8].

Domain names are not registered permanently and eventually they expire and become available for anyone to re-register. Prior work has studied security issues revolving around expiring domains. Specifically, in 2016 Lever et al. showed how the new owner can abuse the *residual trust* inherited by a re-registered domain name [34]. Other work has studied the ecosystem around domain name re-registering [32, 37]. In most cases this occurs at the end of the domain life-cycle when the domain is expiring, but little attention has been given to domains which are deleted before their expiration date.

Deleting a domain name before expiration is not recommended and not supported by many registrars. Domain names are paid for in full upon registration and registrars generally will not offer refunds prorated or otherwise. Therefore, deleting a domain name wastes the investment that was made in it. A registrant who no longer wishes to use a domain name might as well keep it and choose not to re-register when it normally expires. In spite of this, we discover that domain names deleted prior to their expiration date are surprisingly common. Such deletions by malicious registrants may hamper forensic analysis of their malicious campaigns, while registrars deleting domains instead of suspending them enable re-registration and continued abuse.

For years, TLD zone files have been a popular tool in the security community to find active domain names [12, 30, 34, 36, 37, 43]. Zone files are publicly available to researchers who request access and they represent a snapshot of *resolvable* domains, but not all registered domains will appear in zone files. Halvorson et al. observed that 5.5% of registered domains do not appear in zone files, but they specifically refer to domains which are purchased and not assigned name servers by the registrant so they are not yet added to zone files [20]. Alowaisheq et al. discuss suspensions via `EPP client hold` status

which remove domains from zone files [11]. In this paper we present evidence of other cases of registered domains disappearing from zone files, making it clear that zone files alone should not be relied upon as a record of domain registrations and de-registrations.

In this paper, we present the first systematic study of domain names deleted prior to expiration. Over three weeks studying live data, we find the surprising result that 6.4% of all dropping domains are actively deleted prior to expiration. We confirm this phenomenon over the long-term using available historic data sets, finding a combined total of over 386,000 prematurely deleted domains. Among other trends, we find that domains names deleted early are longer on average and much more likely to be pronounceable compared to normally expiring domains. We explore potential motivations for deletions finding more than 7,000 domains abusing trademarks and squatting more popular domains, and over 14,000 associated with malicious activity such as phishing and malware. Furthermore, we investigate the participating parties finding that over 100 registrants deleted domains in bulk, and several registrars, including GoDaddy and Domain.com, delete large numbers of domains instead of suspending them. Our results lead us to a discussion of registrar policies regarding domain name deletion, as well as the advantages of publicly available sources for registration information.

2 Background

To enable the translation from domain names to IP addresses for over 300 million domains, DNS utilizes a hierarchical look-up process beginning at the root zone which leads to top level domain (TLDs) zones such as .com, or .net. These TLD zones are maintained by *registries*, such as Verisign, which then delegate the selling of domain names to *registrars*, such as GoDaddy and eNom. Registrars communicate with the registry through the Extensible Provisioning Protocol (EPP) [23] in order to perform a series of domain-name-related operations, such as checking domain availability, registering new domains, and deleting domains. Finally, *registrants* are the users who buy domain names from the registrars.

Domain names are registered for a period of at least one year, and optionally longer for additional cost. Registrants pay the full cost of the domain at the time of registration and control it until the registration period expires, after which the registrant has the option to renew the domain and pay for another period before anyone else has the opportunity to buy it. This process is often automatic, but if the registrant chooses not to renew, then they lose control of the domain name and it becomes publicly available for anyone to register again.

The complete life-cycle of a domain name contains several phases, details of which are not obvious to typical registrants and easy to confuse. Figure 1 illustrates these phases showing the duration of each and indicating when the domain appears in the TLD zone file. These phases revolve around the registration and expiration of domains in an attempt to make the process fair and reduce the risk of accidental expirations.

The first phase after registration is the 5 day *add-grace period* which is the only time when it is possible to receive a refund for a domain name. This can lead to a form of abuse called domain tasting which was studied in detail by Coull et al. [15] so registrars are limited to issuing a maximum number of add-grace deletions [25]. After the first 5 days, refunds are no longer available and the longest phase is the *registered period* during which the domain should be in the zone files or else it cannot be resolved.

Once the domain expires it enters the *auto-renew grace period*. The registrar is in control at this stage and based on their policies they will notify the registrant and often remove the DNS records from the zone file after a certain point. If the registrar does nothing, the domain will automatically renew between the registrar and registry, so if the registrant does not choose to renew then the registrar will typically delete the domain (to avoid having to pay for a domain whose owner does not want) right before the end of the 45 day period (though it is possible for them to delete sooner). The domain then moves to the *redemption grace period* in which the domain is not in the zone file, but the registrar still has an opportunity for 35 days to reclaim the domain name at an increased cost. Finally, if the registrar takes no further action, the domain enters the *pending delete phase*. At this point the domain is still not in the zone file and after 5 days it becomes publicly available for re-registration.

This domain life-cycle is typical for most domain names, but there is also the possibility for a registrant to delete their domain name before the end of the registration period. This is unusual behavior, because the registrant will not be refunded the registration cost. Even if the registrants are no longer using the domain and do not want it, there is no obvious reason to actively request its deletion, as opposed to allowing it to expire. Yet, as we show later, more than 8,000 domains are deleted early every day.

3 Methodology

In this section we describe the data sources, the method for finding early domain deletions, and a discussion of obstacles and limitations of these data sets.



Figure 1: Stages in the life-cycle of a domain name registration. Green or red indicate the domain is present in or absent from the zone files respectively. Orange means the domain may or may not be in the zone files depending on registrar policies.

3.1 Data Collection

Zone files. Top level domain (TLD) zone files contain at the very least name server records for all resolvable domain names. These zone files are publicly available for all generic TLDs (gTLDs) once requested from the corresponding registries. We use zone files from ten of the largest TLDs: .com, .net, .org, .info, .biz, .top, .xyz, .loan, .club, and .online. These zone files were collected every day and the deltas between each day were computed to find when domains appeared and disappeared. In most cases, appearances and disappearances correspond to new registrations and expirations respectively. These deltas were indexed to create a searchable database of registration periods for each domain. In practice, we find that zone files are not a perfect representation of registration status, but they are a useful starting point using public data.

Drop Lists. Domain drop lists provide a reliable view of domains which have been de-registered. These domains will become available for re-registration on a specific date and registrars want to advertise as many domains as they can to increase sales on the drop date. These drop lists were collected on a daily basis starting from 1/10/2017 and aggregated from SnapNames, DropCatch, Pool, Namejet and Dynadot, all of which are companies that allow users to re-register valuable domains which were left to expire. These lists cover .com, .net, .org, .info, and .biz.

Historic WHOIS. Whenever a registrant purchases a new domain, their details are added to a WHOIS record which is publicly accessible. Before the recent GDPR legislation, WHOIS records contained personally identifying details (PII) about the owners of each domain, such as their name, address, and phone number. Since the GDPR legislation went into effect, WHOIS records are mostly anonymized although they still provide EPP status, dates of registration/expiration, and the registrar.

WHOIS queries are restrictive (i.e. individual WHOIS servers set limits as to how many queries a client can perform per day) and therefore difficult to obtain in large numbers, especially for past records. For our experiments, we used commercial services to obtain historical WHOIS data taken at the time of registration for all

new domains in 2017 and all dropping domains between February and October of 2017 [6,9,47].

RDAP. The recent pilots for the new Registration Data Access Protocol (RDAP) have made it practical to collect live information about domains as they disappear. Unlike WHOIS which is just a text protocol, RDAP provides structured data which allow us to straightforwardly extract important domain information, such as a domain’s registration and expiration dates. Between 12/19/2018 and 1/27/2019 we collected registration information for all of the domains that were removed from the zone files each day. In cases where RDAP failed we fell back to WHOIS. Between these two methods, only 7.0% of queries failed and 74% of failures were from .xyz while .com had only 3.8% failures. To account for delay in server updates and temporary status changes, we also re-queried the same domains one week after disappearance starting with domains that disappeared on 12/31/2018.

Blacklists. We collected hphosts, malc0de, zeustracker, conficker, and malware domains blacklists [1,3–5,7] on a daily basis and used the Internet Archive to retrieve snapshots of older versions of the blacklists. We supplemented blacklists by querying for domains using the Google Safe Browsing API (GSB) [2].

Typosquatting, bitsquatting, and combosquatting. Starting from the Alexa top 1 million on 7/1/2017 and 12/31/2018, we generated a set of typo domain names using the typo models described by Wang et al. [46] and a set of bit-flipped domain names as described by Dinaburg [17]. We also compiled a list of 279 popular trademarks modified from those used by Kintis et al. [29] to find combosquatting domains. Each of these lists are used to find squatting domains taking advantage of the trademarks and brand names of others.

3.2 Finding Early Deletions

WHOIS/RDAP 2019 data set. Our primary experiment during the first three weeks of 2019 makes use of zone file deltas from the 10 TLDs mentioned above and public registration information to study domain names as they disappear and become unresolvable on a daily basis. The combination of both RDAP and WHOIS provide EPP status codes [26] and expiration dates, allowing

us to investigate these domains and determine the cause of their disappearance.

The two situations that would typically cause a domain name to be removed from the zone files are deletion (including expiration) and suspension. There are five relevant statuses that we track on disappearing domains. A status of `redemption period`, `auto renew period`, or `pending delete` indicates that the domain is in the process of being deleted as shown in Figure 1. The `client hold` and `server hold` statuses indicate suspension by the domains' registrar and registry respectively. In some cases a registrar may set the `server hold` status on a domain that is expiring, but this would not impact our results for domains prior to their expiration date and the domain may also have the `auto renew period` status indicating that this is a deletion and not a suspension. Once we know the reason for the domain's disappearance, we then check its expiration date to determine if it was deleted prematurely. We query the same domains again a week later allowing us to remove inaccuracies due to delays in updates on the registration servers and determine whether the status changes are transient or permanent.

Historical 2017 data set. In order to longitudinally study the phenomenon of early deletions, we also conduct a measurement using historical data from 2017. Even though we initially attempted to rely as much as possible on the indexed zone file deltas, we quickly discovered that zone files alone are not a reliable indicator of a domain's registration. We discovered a large number of cases where domains disappeared and reappeared days, weeks, or months later, and in some cases multiple times without dropping or changing name servers or WHOIS information. Below, we describe our process using these zone file deltas and the additional data sources we used to improve the accuracy of our experiments.

Based on the domain life-cycle shown in Figure 1, we are able to narrow down the list of domains that may have been deleted early. First, if a domain name is newly registered and then de-registered within 365 days then it is a candidate early deletion since the shortest possible registration period is one year.

We ignore any domains that were deleted within the 5 day add-grace period because, as described in Section 2, this falls under domain tasting which is a well known practice and may result in a refund. Domain names may be suspended or deleted by the registrar after 15 days if their owners do not respond to inquiries regarding the accuracy of their WHOIS contact information [24]. While this is 15 days from an inquiry and not necessarily since creation, we conservatively choose to ignore disappearances 15 days after creation when contact information was given to the registrar.

Given our observation of domain names disappearing and reappearing without their registration status changing, we filter these domains further, identifying cases where they could not have been registered or de-registered. As shown in Figure 1, a domain name that is deleted should be absent for *at least* 35 days during the redemption period and pending delete phase following its disappearance from the zone files.

We still observed domains which remained registered according to WHOIS, but were absent for more than 35 days so we also used publicly available drop lists. A domain appearing in a drop list must have been de-registered so we use this to filter out temporary domain disappearances from the zone files. This requirement limits the results of this data set to `.com`, `.net`, `.org`, `.info`, and `.biz`, which are covered by our collected drop lists. On the other end of the life-cycle, when we see a domain name appear we do not know if it is newly registered or reappearing from a temporary disappearance. We could similarly require the domain name to be absent for 35 days before appearing, but again the domain may be reappearing after a longer period and the redemption period/pending delete phase do not apply to domains deleted during the add-grace period. Therefore, we require additional information analogous to the drop lists to verify new registrations. We use historic WHOIS records to confirm the zone file appearance date with the listed creation date. This also has the advantage of obtaining registrar information and registrant contact details which we use to characterize early deletions.

4 Results

In the following we present the results with a data-driven approach to explain the phenomenon of early deletions.

4.1 Categorizing Disappearances

Collecting registration information with RDAP and WHOIS allows us to broadly categorize unexpected domain disappearances from the zone files. Based on the EPP statuses of each domain, we assign one of four labels: i) *active* domains which are still registered and have no adverse status, ii) *suspended* domains which were removed by the registrar or registry, iii) *deleted* domains which are on their way to being de-registered (typically due to expiration), and iv) domains for which the queries *failed* to obtain registration information.

On average, we observed 127,318 domains disappearing from the zone files every day. Figure 2 shows that on a typical day about 70% of these disappearing domains were deleted with most of the remainder caused by suspensions by either the registry or registrar.



Figure 2: Percentage of disappearing domains each day with status indicating deleted, suspended, or active. (Top) Queried on the day of disappearance. (Bottom) Queried one week after disappearance.

There is a notable weekly cycle of days where the majority of the disappearing domains are still listed as active. This occurs on Saturdays and Sundays and on one Monday (1/21) which was a US national holiday. By querying the same domains one week later (bottom of Figure 2) we obtain more consistent results and we no longer see this pattern. This data depends heavily on Verisign’s RDAP pilot deployment and our observations suggest that they have some weekend delay in updating registration information on their servers for newly expired and suspended domains.

Ignoring these weekends and looking specifically at domains which change status one week later, we find that by far the most common change on a typical day is domains having their suspension lifted and becoming active again. Table 1 shows how many domains changed status out of the total number of disappearances. Overall we find the results are very stable which lends confidence to the results from RDAP and WHOIS. For the rest of this section we use the second round of results to avoid temporary status changes and server inconsistencies.

Transition	Percentage of domains
Active → Deleted	0.0002%
Active → Suspended	0.0004%
Deleted → Active	0.075%
Deleted → Suspended	0.005%
Suspended → Deleted	0.36%
Suspended → Active	5.28%
Failed → Not Failed	1.69%
Not failed → Failed	1.44%

Table 1: Changes in EPP status one week after disappearance.

4.2 Early Disappearances

To find domains that disappeared early we compare the date they were removed from the zone files to the expiration date returned by RDAP/WHOIS. Figure 3 shows the breakdown of status for domains that disappeared before their expiration date. This ignores domains with failed queries since we do not have their expiration dates or status. Since domains disappearing after expiration is the normal expected behavior, it is unsurprising that when we look at these cases, we find that most are caused by suspensions. However, a surprising 22.1% (173,292 total) of the early disappearances are related to domain names that were deleted before their expiration date. This corresponds to 6.4% of all disappearing domains and an average of 8,252 early deletions per day.

We also observe 5,061 domains which still have an active status after both queries despite their removal from the zone files. 99.96% of these occur before expiration so we know these are not expired domains with inaccurate status. Without an entry in the zone files these domains cannot be resolved, but they still appear to be registered and they do not have a client or server hold. Registrants do not typically have the ability to directly remove a domain from the zone files while it is still registered, but this may be a process controlled by the registrar or registry. Because of this behavior, we conclude that zone files alone cannot be used to indicate whether a domain is registered or not. Therefore systems that rely on zone files [12, 30, 34, 36, 37, 43] are bound to be missing domains that are temporarily suspended and ones that are registered but are missing.

To find early deletions over a longer period of time we use zone files, drop lists and historical WHOIS as described in Section 3. We classify 212,864 domains as early deletions over 13 months which is about 0.5% of all disappearing domains during the same time period. Figure 4 shows the number of early deletions over time which is fairly steady on typical days, but there are multiple outliers, the largest being 3/16/2017 with more than 20 times the average number of deletions. These cases are discussed in more detail later in this section. We also observe a slight increase over time, as the number of



Figure 3: Percentage of domains which disappeared before their expiration date each day with status indicating deletion, suspension, or that the domain is still active. (Top) Queried on the day of disappearance. (Bottom) Queried one week after disappearance.

monitored, newly-registered domains grows. These data sets cover fewer TLDs than our RDAP/WHOIS measurement, but based on the percentage of disappearing domains, it is still significantly fewer early deletions. The main reason is that this method can only capture early deletions that occur within one year of registration, missing domains which were registered for a longer period.

This data set provides another vantage point from which we observe domains disappearing and reappearing while still registered. Without using drop lists or WHOIS to filter domain changes in the zone files, we found 5,907,109 cases of domains disappearing before expiration. We do not have access to domain status when these domains disappeared, but based on our above findings where we observed about 20,000 suspended domains per day we expect this to be the primary cause of early disappearances. However, most of these suspended domains names will remain suspended and will not reappear in the zone files for the lifespan of the domain.

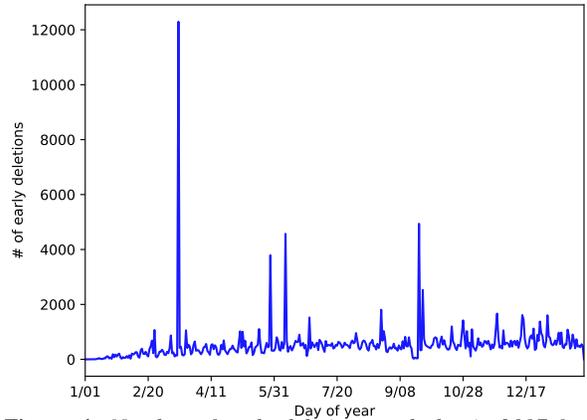


Figure 4: Number of early deletions each day in 2017 from .com, .net, .org, .info, and .biz.

4.3 Duration of Registration

To understand potential patterns in early deletions, we start by examining the lifespan of deleted domains. Figure 5 shows the distributions of registered duration and days remaining. We measure the duration of registrations as the number of days from a domain’s creation to the observed date of deletion. The remaining time is the number of days prior to expiration that we observe deletion. The median remaining time was 322 days so domains are often deleted long before expiration, but only 4% of domains had more than 1 year remaining. Interestingly, deleted domains are not all short-lived. In fact, 81% of deleted domains were registered for over one year with a median of 458 days. This makes it clear that the primary cause for the difference in number of observed early deletions in our two data sets is the limitation of domains in zone files for under one year in our historical data set.

We also found that domains have a bias towards being deleted early in their overall lifespan. This can be observed in Figure 6 which shows the distribution of remaining time as a percentage of the total number of days from creation to expiration. We also observe that there are multiple sharp jumps in each of these graphs which likely correspond to bulk activity relating to a few registrants that had many domains which were registered on the same date and deleted on a specific following date.

4.4 Trademark Abuse

One explanation for domain deletions could be an association with a brand that requests for the domain to be taken down. For serious disputes, trademark holders can file UDRP (Uniform Domain-Name Dispute-Resolution Policy) requests, but this can cost thousands of dollars [48]. In 2017 we found records of only 1,557 total filed disputes covering 3,487 domains using a third-party

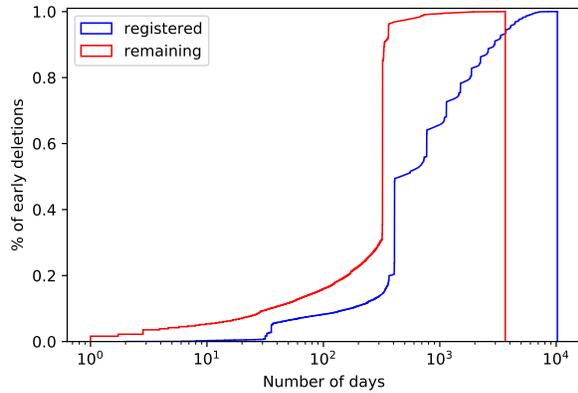


Figure 5: Cumulative distribution of duration of registration and days remaining before expiration for domains deleted early. Number of days on log scale.

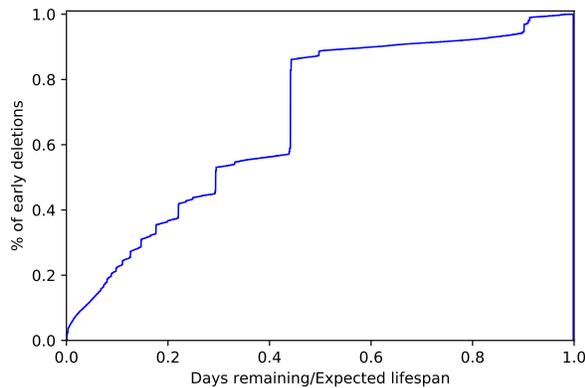


Figure 6: Cumulative distribution of percentage of paid lifespan remaining before deletion.

tool that crawls and indexes UDRP cases [28]. In most cases where the ruling is in the complainant’s favor, the domain is transferred. This gives the trademark holder the most control as they can now sinkhole or redirect it to their primary site and they can choose to continue renewing it, so it stays under their control indefinitely. In only 3% of these cases, the decision was to delete the domain name instead of transferring it. As such, we argue that formal disputes have a minimal impact on early deletions. More often, the first step in dealing with a squatting domain is sending a cease and desist to the offending domain registrant demanding that the domain name be deleted, transferred, or abandoned. In this scenario, a registrant who is concerned about legal trouble may choose to delete the domain name right away to satisfy the trademark holder. To estimate the number of early deletions that may have been caused by this situation before an official complaint, we looked for typosquatting, bitsquatting, and combosquatting domains.

Typosquatting domains target popular Web sites and capitalize on users making typos of those sites. Bitsquatting is very similar, but it involves registering domains with an ASCII encoding one bit different from the target to capitalize on random bitflips during the resolution process. We generated a list of typos and bitflips from the Alexa top 1 million on 12/31/2018 for the RDAP/WHOIS data set and from 7/1/2017 for the historic data set as described in Section 3.1. From these lists we found 4,751 typosquatting/bitsquatting early deletions. We also explored combosquatting which is a broader category of domains which contain popular trademarks along with additional keywords meant to confuse and deceive users reading the URL. As described in Section 3, we obtained the list of combosquatting trademarks compiled by Kintis et al. [29] which we extended with trademarks that we observed during our manual analysis of early domain deletions. From the resulting list of 279 trademarks, we found 2,612 combosquatting early deletions in our combined data sets.

By combining the discovered typosquatting, bitsquatting, and combosquatting results, we found a combined total of 7,322 (1.9%) early deletions abusing trademarks. It is clear that this represents a level of malicious activity although we cannot differentiate whether these early deletions were due to cease-and-desist orders or due to malicious registrants attempting to cover their tracks by deleting domains after they abuse them. Only 134 of these squatting domains were present in our collected blacklists which amounts to less than 2% coverage by popular blacklisting approaches.

4.5 Domain Features

We analyze features of domains which are deleted early and compare them to popular domains, domains which expire normally, and suspended domains. Since the domain names we are studying are offline by the time we observe them, we are limited to lexical features of the names themselves. The simplest metrics are length, and whether the domain contains at least one number or hyphen. We also look at Shannon entropy which can be an indicator of names created by Domain Generation Algorithms (DGA) which create a large number of domains where botnets know to contact command and control servers. Plohmman et. al. [41] discuss various types of DGAs and show that most have high entropy. The domain names are segmented into the most likely words based on known frequencies of words and pairs of words in the English language [40]. From this we get number of words, and check those words against a blacklist of adult keywords [16]. We filter out extracted words which do not appear in a dictionary, but since many domain names contain made-up brand names or newer slang, we also

Metric	Alexa Top 1M	Normal Expirations	Suspensions	Early Deletions
<i>Length</i>	10.55	12.14	11.59	12.85
<i>Entropy</i>	2.825	2.937	2.893	3.005
<i>Number of words</i>	1.377	1.534	1.368	1.719
<i>Percent containing numbers</i>	6.46%	14.04%	15.99%	9.87%
<i>Percent containing hyphens</i>	10.05%	7.44%	11.20%	8.43%
<i>Percent containing adult keywords</i>	3.129%	2.633%	2.562%	2.649%
<i>Percent unpronounceable</i>	14.89%	21.64%	24.58%	15.02%

Table 2: Comparison of lexical domain features between popular, expired, suspended, and deleted domain names.

measure pronounceability based on frequency of letter bi-grams in the English language. Average pronounceability scores were very similar across all domains, but we set a threshold to label domains as pronounceable or not which we applied equally to each category. Using only English words is a potential limitation, but in many cases pronounceability may work well across languages and we exclude from this analysis Internationalized Domain Names which begin with xn--. Similar metrics are commonly used in part for domain name appraisals which combine many features of a domain to estimate its monetary value [10].

From the extracted English words we used WordNet Domains [13] to label domain names with more general topics. Wordnet Domains has 181 possible topics, many of which are closely related, so we manually aggregated similar topics to more appropriately group domain names. For example, banking, commerce, and finance are distinct topics which we include under the *Economy* label along with eight other sub-topics. Since domain names can have multiple words and each word may have multiple meanings each mapping to different topics with WordNet Domains, we take the most common topic from the list of all related topics for that name. If a domain name has no words or the words do not have a mapped topic, then we label it as *Unknown*. A breakdown of the most represented topics among early deleted domain names can be seen in Table 3. A full table with all 28 topics and a statistical comparison between different categories of domains appears in Appendix B.

Table 2 compares each metric between domains in the Alexa top 1 million, domains which disappear normally after their expiration, domains which were suspended, and domains deleted early between 12/31/18 and 1/20/19. One difference is that early deletions are the longest in both number of characters and number of words. Early deletions also have fewer numbers and hyphens than suspended domain names and are almost as likely to be pronounceable as popular domains. Shorter domain names are generally considered more valuable, but the pronounceability and number of words suggests that early deleted domain names may be well formed.

Topic	% of Early Deletions
Unknown	26.71%
Economy	8.79%
Science/technology	8.49%
Play/sports	6.16%
Health/medicine	5.64%
Architecture	4.24%
Geography	4.09%
Politics/government	3.95%
Art	3.54%
Travel/transport	3.46%
Person	3.07%
Writing/language	2.32%
Other	19.56%

Table 3: Top domain topics in early deleted domain names.

If the domain is considered valuable we expect it is less likely a registrant chooses to delete it, but one explanation for the length could be DGA names that are built from a dictionary of words. Three examples of such wordlist-based DGAs are Matsnu, Suppobox, and Gozi which were studied by Plohmann et. al. [41] in 2016. These are less likely to be suspended or blacklisted because they do not look like random strings of numbers and letters. At the same time, the paper points out that these DGAs are more likely to collide with existing domain names than other methods. To counteract this issue, modern wordlist-based DGAs may require longer names with more words than would be common among popular domains in order to generate a consistent variety of unused domain names.

We wish to determine if the observed differences between early deletions and the three other categories in Table 2 are statistically significant or if they could have occurred randomly as a sample of all domain names. We use the Welch's *t*-test to test the null hypothesis that early deletions are sampled from the same population as the other groups. Due to our large sample sizes, we obtained very small *p*-values for almost all tests leading us to reject the null hypothesis in these cases and conclude that early deletions are drawn from distinct populations. Ap-

pendix A includes more details of the statistical analysis including Cohen's d effect size, t -statistic, and p -value for each test. The fact that early deletions have statistically significant differences means that it is very unlikely that domain names are being deleted indiscriminantly. Rather, some reasoning went into which domains were deleted, or the entities that deleted domain names had some pattern as to the domains they held.

4.6 Registrant Patterns

While domain owners can choose to use a WHOIS privacy service to hide personal information, such as email addresses, we found that in our 2017 data set, after filtering out anonymous addresses, 73% of early deletions had real registrant email addresses. Out of 58,773 registrant email addresses, 97% of them deleted less than 10 domains before expiration, but 100 registrants deleted more than 100 domains early. We clustered the email addresses using DBSCAN and Levenshtein distance to find registrants who are likely the same person registering under different accounts, but this had only a small impact on our analysis. After clustering we had 56,279 registrants, 105 of whom deleted more than 100 domains early. These bulk deletions may contribute to the spikes seen in Figure 4. We count the number of deletions each day by individual registrants and find that the spike on 9/23/19 was caused by a few bulk registrants, one of whom deleted 952 domains on that day.

We argue that these types of mass deletions suggest malicious use. Bulk registrants are often domain speculators which are not necessarily malicious, but hope to sell the domain for a profit and collect small amounts of advertisement money from parked pages in the meantime [44]. Therefore, they have no incentive to delete their domains. Contrastingly, spam domains and DGA C&C domains are also commonly registered in bulk and tend to be short lived [21, 45].

4.7 Registrar Patterns

We take advantage of the registrar listed by RDAP/WHOIS to find the registrars which either enable or otherwise actively delete domains. Table 4 shows the top 10 registrars in terms of number of early deletions. Godaddy has by far the most early deleted domain names, but according to ICANN transaction reports [27] it is also the largest registrar in the world by number of registered domains. To address this, we also ranked registrars normalized by their total number of .com domains reported by ICANN, excluding registrars with less than 100 domains. This reveals some additional smaller registrars which had notable portions of their domains deleted. Dropcatch.com is at the top

of this list which is interesting because they sell expired and re-registered domains which Miramirkhani, et al. showed are rarely used for legitimate web pages [37]. The ranking of registrars by total deletions is similar for 2017 with six of the top ten appearing on both lists. Notably, the registrar Web Drive deleted 27% of their total domains early over the course of our measurement.

While we discovered that many domains are suspended by the registrar with a `client hold` status, some registrars may delete abusive domain names instead, which could explain high numbers of deletions from certain registrars. In particular, we found that Godaddy only suspended 412 during the same time period which is only 0.0009% of their total domains. Godaddy and Wild West Domains are both among the lowest of all registrars in terms of percentage of domains suspended. While these registrars deleting domains could explain a large number of early deletions, it is surprising that registrars choose deletion over `client hold`, thereby allowing the domain to be re-registered.

We investigated the largest 15 registrars as well as all of those shown in Table 4 to determine how easy it is for registrants to delete their own domain names. We found that only GoDaddy, Google, Hetzner Online, and RegistryGate refer to or provide an option for users to delete domains [18, 19, 22, 42]. This is further evidence that the other registrars are deleting domain names early, but we cannot rule out the possibility that these registrars would allow deletions if requested through customer support.

Figure 4 exhibited multiple significant outliers which we investigate further using WHOIS listings for the registrar and registrant email addresses. As with our analysis of registrants (Section 4.6), for each outlier, we group together early deletions of domains belonging to the same registrar. Three of these outliers are dominated by single registrars:

- **3/16:** The largest outlier by far, Domain.com had 12,014 early deletions while the second most was 1&1 Internet with only 119. Several Hotmail email addresses were tied to hundreds of these domains.
- **5/28:** Cronon had 3,576 early deletions, again significantly more than Godaddy having the second most at 182, while no registrants deleted more than 16.
- **6/9:** 1&1 Internet had 4,267 early deletions with GoDaddy trailing at 207 with no dominant registrant emails.

The evidence strongly suggests that certain registrars are responsible for these deletions. While registrars are welcome to take action against abusive domain names before they expire, it is surprising that they choose to delete the domain names instead of placing them on `client hold`. Once a domain is deleted, a malicious actor can re-register the same domain and regain control of the infrastructure associated with it. In fact, prior stud-

Registrar	# Early Deletions	Registrar	% of Total Domains
GoDaddy.com, LLC	125,059	DropCatch.com, LLC	7.89
Tucows Domains Inc.	6,084	Ednit Software Private Limited	2.37
Cronon AG	4,896	NetTuner Corp. dba Webmasters.com	1.72
Wild West Domains, LLC	4,713	Vautron Rechenzentrum AG	0.97
Google, Inc.	3,599	Deutsche Telekom AG	0.90
Key-Systems GmbH	2,712	Metaregistrar BV	0.79
Name.com, Inc.	2,545	Cronon AG	0.78
CSC Corporate Domains, Inc.	2,502	RegistryGate GmbH	0.77
RegistryGate GmbH	2,160	Hetzner Online GmbH	0.74
PSI-USA, Inc. dba Domain Robot	1,821	HTTP.NET Internet GmbH	0.68

Table 4: Registrars with the most early deletions.

ies have shown that malicious domains are *more* likely to be re-registered [34, 37]. Therefore, by deleting suspicious domains (instead of placing them on hold) registrars are merely inconveniencing malicious actors who can use a new registrar to re-register their domains and resume their malicious campaigns.

4.8 Blacklisted Domains

A possible motive for registrants deleting domains is for them to cover their tracks after malicious use. For example, associated IP addresses and WHOIS information cannot be obtained for deleted domains which may stump later forensic investigations of abusive domains.

To gauge the level of malicious activity we rely on blacklists. Blacklists cannot possibly cover all malicious domains, but with the domains deleted and the web sites down, they are the best available method to estimate the number of malicious early deletions. We referenced multiple sources to find blacklisted deleted domains as described in Section 3.1. We found 402 (0.23%) blacklisted domain names that were deleted early in the 2019 data set and another 1,107 (0.52%) in the 2017 data set. We can expand the number of potentially malicious domains by grouping early deletions by owner. We know from Section 4.6 that some registrants were associated with large numbers of deleted domains. If one of those domains was blacklisted then we may suspect that the registrants' other deleted domains may also be malicious. Using the clusters of registrant emails from the previous section, we mark a registrant as malicious if they deleted at least one blacklisted domain. The number of 2017 domains deleted by these malicious registrants is 9,782 (4.60%), significantly more than was found with blacklists alone. Despite the unavailability of registrant email addresses, we are able to apply a similar analysis to the 402 blacklisted domains in the 2019 data set. We grouped deleted domain names by registrar and date, then clustered very

similar domain names using Levenshtein distance and DBSCAN to find groups of domains that were likely created by the same registrant. This resulted in 8,577 clusters containing two or more domains and an average cluster size of 4.5. Then, with the same approach of labeling a group as malicious if at least one of its domains was blacklisted, we extend the number of potentially malicious domains to 5,028 (2.90%). While associating domains in this way does not guarantee that they are malicious, we apply this method because shared ownership is a concrete connection between domain names. Malicious domains are often registered in bulk, and this is within the capabilities of registrars who ultimately control early deletions.

Even though the percentage of domains found in blacklists may appear small, to get a rough comparison of coverage we check domains that were suspended between 12/31/18 and 1/20/19 and find that only 2,163 (0.47%) appear in the same set of blacklists. We expect that most suspended domains were malicious, and yet only a small percentage of them appear in blacklists. The percentage of blacklisted early deletions is similar and our estimates for potentially malicious domains after association are 6-10 times more than the coverage for suspended domains. Therefore, we are confident that many of these domains were malicious but did not make their way onto blacklists before their deletion.

We suggest that deleting a domain name before it would normally expire is suspicious enough that it may be a signal worth considering when blacklisting/suspending domains. This signal can be used to tie this information to the registrant and scrutinize other domains that they register. Moreover, since malicious domains are often re-registered after their deletion [34, 37] or after sink-holing systems let them expire [11], knowing that a domain was deleted early can predict future abusive behavior. In fact, this is one reason why the owner of a blacklisted domain may

choose to delete it, i.e., to make the association with their other live domains less conspicuous.

5 Discussion

Registrars deleting domains. Through two distinct data sets we quantify disappearing domains and identify hundreds of thousands of domain names which were deleted before expiration. By analyzing registrar patterns over time we conclude that at least 10% of deletions may be initiated by registrars. For registrars that are deleting domains to deal with abuse, we recommend that they instead suspend them with `client hold` and/or disable the registrant's account, but maintain control of the domain so that it cannot be re-registered for malicious use. As we mentioned in Section 4.7, most registrars already make it difficult for registrants to delete domain names or at least warn against it which is appropriate for the majority of users. For these registrars we recommend that they go a step further and screen requests to delete domains to determine if the domain names were abused which may be used to find other abusive domains from the same owner.

Registrants deleting domains. For cases where the registrant initiated the deletion, explanations are not obvious, but we present the following possibilities: i) a negative association with another name, ii) attempts to hide malicious activity, or iii) due to ignorance or indifference regarding the domain life-cycle. Since most registrars do not make deleting domains easy, and even the ones that do warn against it, we believe that category three is an unlikely cause. We found many domains fitting the first category and due to our thorough approach to checking multiple types of domain squatting we expect that we found the majority of cases in this category. For the second category, we identify as many malicious domains as possible using blacklists and association by registrant. Due to the limited coverage of blacklists and the difficulty of finding malicious activity after the domain has been deleted, we argue that without another likely explanation, unexplained cases are likely to fall into the second category.

Registration information. Another lesson from this study is that registration data including status, registrar, and dates should be maintained as a publicly available resource. Public access to zone files has been very successful in aiding security research and applications [12,30,34,36,37,43], but it is not enough to identify all registered domain names, nor does it cover all stages of the domain life-cycle making cases like early deletion and dropcatching more difficult to monitor. In the past, query limits on WHOIS were a reasonable precaution to prevent mass collection of registrants' personal

information, but with recent changes to WHOIS privacy, largely driven by the EU's GDPR, this is no longer necessary. Now there is an opportunity to make generic, non-personal domain registration information publicly available. RDAP is a step in the right direction particularly with unified data structure and the addition of authorization, but as it is still in pilot and specific zones are managed by different parties, it remains to be seen whether this will continue to be a public resource.

Limitations. A limitation of our study is the inability to determine for each domain name whether its deletion was initiated by the domain owner or the registrar. In Section 4.7, we are only able to use indirect evidence to estimate that at least 9.33% were deleted by registrars, and cannot guarantee that the remainder were all initiated by registrants. Nevertheless, we hope that this work motivates security-conscious registrars who do have this visibility for their customers' domains to further explore premature deletions.

For an analysis of domain names after they have already been deleted, we are also limited in our ability to determine what the domain name was used for while it was still active. Our primary tool is analysis of lexical features, but in many cases this is not enough to identify whether a domain name was registered and used for malicious purposes. We were able to find that 1.9% of early deletions were domain squatting, and based only on blacklists we estimated another 3.84% were likely malicious. We acknowledge that this leaves a majority of cases unexplained, and future work should aim to fill this gap. However, even these limited findings draw attention to the phenomenon of early deletions and warrant a closer look into a practice that has been ignored up to this point.

6 Related Work

A domain name may enter the pending delete state of the domain life-cycle as a result of early deletion or expiration. While re-registration of expired domains, and the security implication of this practice have received extensive attention [31–34, 37], to the best of our knowledge, no one has exclusively looked into the unusual behavior of early deletion. Since our work is the first systematic study on this phenomenon, we review prior work on deleted domains which are closest to our work.

Lauinger et al. [32] showed that there is an intense competition between dropcatch registrars in registering desirable deleted domains. Some of these registrars maintain large numbers of registrar accreditations to be able to submit more requests and in this way increase their chance of catching available domains. Only three large dropcatch registrars control 75% of registrars

which translates to millions of dollars in accreditation fees. In more recent work [31], they took a closer look at when expired domain names are re-registered. Using a model to infer the deletion time of domains, they showed that 9.5% of deleted domains are re-registered less than one second after they became available. Miramirkhani et al. found that the domains which are shorter, older, have more residual traffic, and malicious history are more likely to be re-registered and the majority of registrations are for speculative or malicious purposes [37]. They also reported that premature deletions are relatively uncommon compared to the usual yearly life-cycle. In this paper we investigated the reasons and motivations behind early deletions because, although it is a smaller percentage of cases, it creates an opportunity for malicious registrants to avoid detection from DNS monitoring systems.

In related work, potential and actual abuse of the reputation of deleted domains are discussed. Nikiforakis et al. showed that popular websites contain remote JavaScript inclusions that are pointing to expired domains which can be re-registered to perform code injection attacks [39]. Later, Moore et al. investigated the domains of US banks and reported that 33% of these domains are re-registered to host advertisement, distribute malware, or to carry out search engine optimization (SEO) activities [38]. A recent study by Vissers et al. showed that re-registration of expired domain names can result in hijacking thousands of domain names through their name servers [43]. Lever et al. explored the domains that were deleted in a span of six years and for those that were abused for malicious intentions, they examined whether the malicious activity occurred before or after domain re-registration. They found hundreds of thousands of re-registrations occurred with the intention of abusing negative or positive residual trust of the original domains [34]. Recent studies [14, 35] have shown a similar use-after-free problem exists for IP addresses as well. Specifically, Borgolte et al. [14] demonstrated that stale DNS records pointing to cloud IP addresses could be abused to hijack domains' traffic and create new SSL certificates under an attacker's control.

7 Conclusion

In this paper we presented the first systematic analysis of early domain name disappearances. Using historical zone files, drop lists, and WHOIS for all of 2017 and collecting live RDAP/WHOIS between 12/19/18 and 1/27/19, we uncovered the surprising phenomenon of domain names deleted before their expected expiration date, with thousands of cases every day. We showed that domains deleted early are longer, contain fewer numbers, and are much more likely pronounceable than domains expiring normally. We found thousands of cases

of squatting domain names which may have been deleted due to complaints from trademark holders, and domains deleted by malicious registrants potentially to cover digital tracks of abusive activity. We showed that many registrants deleted domains in bulk and a few registrars such as GoDaddy seem to delete domain names rather than suspending them.

We demonstrate issues that arise when relying on zone files to study the global state of domain names and advocate for public access to anonymous registration information to aid automated security tools and forensics. Finally, we recommend that registrars scrutinize registrants who delete domain names and utilize suspensions rather than deletions to prevent malicious domains from being re-registered.

Acknowledgements: We thank our shepherd Katharina Krombholz and the anonymous reviewers for their helpful feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-16-1-2264 and by the National Science Foundation (NSF) under grants CMMI-1842020, CNS-1617593, and CNS-1617902.

8 Availability

To motivate further research into the unexpected disappearances of domain names and their effect on security, we are releasing our compiled list of 386K domains that disappeared from zone files before their expected expiration date (together with their associated metadata). The compiled dataset can be downloaded from: <https://github.com/timothy-barron/now-you-see-it>.

References

- [1] Cert.at Conficker. https://www.cert.at/static/conficker/all_domains.txt.
- [2] Google Safe Browsing API. <https://developers.google.com/safe-browsing/>.
- [3] hpHosts. <https://hosts-file.net/download/hosts.zip>.
- [4] Malc0de. <https://malc0de.com/bl/B00T>.
- [5] Malware Domains. <https://mirror1.malwaredomains.com/files/domains.zip>.
- [6] WHOISDataCenter. <https://whoisdatacenter.com/>.
- [7] ZeuS Tracker. <https://zeustracker.abuse.ch/blocklist.php?download=domainblocklist>.
- [8] Sex.com now officially the most expensive domain name in the world. <https://techcrunch.com/2011/02/22/sex-com-now-officially-the-most-expensive-domain-name-in-the-world/>, 2011.
- [9] Domainiq (a domain intelligence service). <https://www.domainiq.com/>, 2017.
- [10] Estibot appraisal tool. <http://www.estibot.com>, 2017.

- [11] ALOWAISHEQ, E., WANG, P., ALRWAI, S., LIAO, X., WANG, X., ALOWAISHEQ, T., MI, X., TANG, S., AND LIU, B. Cracking Wall of Confinement: Understanding and Analyzing Malicious Domain Takedowns. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)* (2019).
- [12] ANTONAKAKIS, M., PERDISCI, R., DAGON, D., LEE, W., AND FEAMSTER, N. Building a dynamic reputation system for dns. In *USENIX security symposium* (2010), pp. 273–290.
- [13] BENTIVOGLI, L., FORNER, P., MAGNINI, B., AND PIANTA, E. Revising the wordnet domains hierarchy: semantics, coverage and balancing. In *Proceedings of the Workshop on Multilingual Linguistic Resources* (2004), Association for Computational Linguistics, pp. 101–108.
- [14] BORGOLTE, K., FIEBIG, T., HAO, S., KRUEGEL, C., AND VIGNA, G. Cloud strife: mitigating the security risks of domain-validated certificates. In *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)* (2018).
- [15] COULL, S. E., WHITE, A. M., YEN, T.-F., MONROSE, F., AND REITER, M. K. Understanding domain registration abuses. In *IFIP International Information Security Conference* (2010), Springer, pp. 68–79.
- [16] DAVID COHN. Parental Controls Bad Word List:NSFW. <https://www.webseoanddesign.com/parental-controls-bad-word-listnsfw/>.
- [17] DINABURG, A. Bitsquatting: Dns hijacking without exploitation. *Proceedings of BlackHat Security* (2011).
- [18] GODADDY. Cancel my domain. <https://www.godaddy.com/help/cancel-my-domain-412>.
- [19] GOOGLE DOMAINS HELP. Delete a domain. <https://support.google.com/domains/answer/6202231?hl=en>.
- [20] HALVORSON, T., DER, M. F., FOSTER, I., SAVAGE, S., SAUL, L. K., AND VOELKER, G. M. From academy to zone: An analysis of the new tld land rush. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference* (2015), ACM, pp. 381–394.
- [21] HAO, S., THOMAS, M., PAXSON, V., FEAMSTER, N., KREIBICH, C., GRIER, C., AND HOLLENBECK, S. Understanding the domain registration behavior of spammers. In *Proceedings of the 2013 conference on Internet measurement conference* (2013), ACM, pp. 63–76.
- [22] HETZNER ONLINE. How do I delete a domain from my Managed Server? <https://hetzner.co.za/help-centre/products-and-services/how-do-i-delete-a-hosting-package-from-my-server/>.
- [23] HOLLENBECK, S. Extensible provisioning protocol (epp). <https://tools.ietf.org/html/rfc5730>, 2009.
- [24] ICANN. Registrar advisory concerning the “15-day period” in whois accuracy requirements. <https://www.icann.org/news/advisory-2003-04-03-en>, 2003.
- [25] ICANN. AGP (Add Grace Period) Limits Policy. <https://www.icann.org/resources/pages/agp-policy-2008-12-17-en>, 2008.
- [26] ICANN. EPP Status Codes. <https://www.icann.org/resources/pages/epp-status-codes-2014-06-16-en>, 2014.
- [27] ICANN. Monthly registry reports. <https://www.icann.org/resources/pages/registry-reports>, 2019.
- [28] INTELUM. Trademark247. <https://www.trademark247.com/>, 2019.
- [29] KINTIS, P., MIRAMIRKHANI, N., LEVER, C., CHEN, Y., ROMERO-GOMEZ, R., PITROPAKIS, N., NIKIFORAKIS, N., AND ANTONAKAKIS, M. Hiding in plain sight: A longitudinal study of combosquatting abuse. In *Proceedings of 24th ACM Conference on Computer and Communications Security (CCS)* (2017).
- [30] KOUNTOURAS, A., KINTIS, P., LEVER, C., CHEN, Y., NADJI, Y., DAGON, D., ANTONAKAKIS, M., AND JOFFE, R. Enabling network security through active dns datasets. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2016), Springer, pp. 188–208.
- [31] LAUINGER, T., BUYUKKAYHAN, A. S., CHAABANE, A., ROBERTSON, W., AND KIRDA, E. From deletion to re-registration in zero seconds: Domain registrar behaviour during the drop. In *Proceedings of the 2018 Internet Measurement Conference* (2018), IMC’18.
- [32] LAUINGER, T., CHAABANE, A., BUYUKKAYHAN, A., ONARLIOGLU, K., AND ROBERTSON, W. Game of Registrars: An empirical analysis of post-expiration domain name takeovers. In *Proceedings of the USENIX Security Symposium* (August 2017).
- [33] LAUINGER, T., ONARLIOGLU, K., CHAABANE, A., ROBERTSON, W., AND KIRDA, E. Whois lost in translation:(mis) understanding domain name expiration and re-registration. In *Proceedings of the 2016 ACM on Internet Measurement Conference* (2016), ACM, pp. 247–253.
- [34] LEVER, C., WALLS, R., NADJI, Y., DAGON, D., MCDANIEL, P., AND ANTONAKAKIS, M. Domain-z: 28 registrations later measuring the exploitation of residual trust in domains. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 691–706.
- [35] LIU, D., HAO, S., AND WANG, H. All your dns records point to us: Understanding the security threats of dangling dns records. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1414–1425.
- [36] MCGRATH, D. K., AND GUPTA, M. Behind phishing: An examination of phisher modi operandi. *LEET 8* (2008), 4.
- [37] MIRAMIRKHANI, N., BARRON, T., FERDMAN, M., AND NIKIFORAKIS, N. Panning for gold.com: Understanding the dynamics of domain dropcatching. In *Proceedings of the Web Conference* (April 2018).
- [38] MOORE, T., AND CLAYTON, R. The ghosts of banking past: Empirical analysis of closed bank websites. In *International Conference on Financial Cryptography and Data Security* (2014), Springer, pp. 33–48.
- [39] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 736–747.
- [40] PETER NORVIG. Natural Language Corpus Data: Beautiful Data. <http://norvig.com/ngrams/>.
- [41] PLOHMANN, D., YAKDAN, K., KLATT, M., BADER, J., AND GERHARDS-PADILLA, E. A comprehensive measurement study of domain generating malware. In *Proceedings of the USENIX Security Symposium* (2016), pp. 263–278.
- [42] REGISTRYGATE. Registration and management conditions for domain names. http://www.registrygate.com/fileadmin/user_upload/RyG_KS_ICANN_Registration-Agreement_Registrant_ENGLISH.pdf.

- [43] VISSERS, T., BARRON, T., VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. The wolf of name street: Hijacking domains through their nameservers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 957–970.
- [44] VISSERS, T., JOOSEN, W., AND NIKIFORAKIS, N. Parking sensors: Analyzing and detecting parked domains. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)* (2015).
- [45] VISSERS, T., SPOOREN, J., AGTEN, P., JUMPERTZ, D., JANSSEN, P., VAN WESEMAEL, M., PIESSENS, F., JOOSEN, W., AND DESMET, L. Exploring the ecosystem of malicious domain registrations in the .eu tld. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2017), Springer, pp. 472–493.
- [46] WANG, Y.-M., BECK, D., WANG, J., VERBOWSKI, C., AND DANIELS, B. Strider typo-patrol: Discovery and analysis of systematic typo-squatting. *SRUTI 6* (2006), 31–36.
- [47] SecurityTrails: The World’s Largest Repository of historical DNS data. <https://securitytrails.com/dns-trails>.
- [48] WIPO. Schedule of Fees under the UDRP. <https://www.wipo.int/amc/en/domains/fees/>, 2002.

A Statistical Analysis of Domain Features

In order to test our domain features presented in section 4.5, the Welch’s t -test was chosen for its robustness on large sample sizes and to handle unequal variances. For each feature in Table 2, we test the difference between sample means under the null hypothesis that each sample is drawn from the same population. Since our samples are very large ($>172K$), the distribution of sample means is closely approximated by the normal distribution. Table 5 shows the Cohen’s d effect size, t -statistic, and p -value for each test, comparing the feature means of early deletions against the three other categories of domains. In three cases we have large enough p -values that these differences are not considered significant. Two of these are tests for the feature *percent containing adult keywords* which is very similar between early deletions, normal expirations and suspensions. The third is *percent unpronounceable* where early

deletions and Alexa domains are similar despite the difference with the other two categories. In all other tests we obtain very small p -values leading us to reject the null hypothesis. The effect sizes indicate that the most significant difference between early deletions and normal expirations is in the mean *number of words* and *percent unpronounceable*. Overall, the effect sizes coincide with our observations in section 4.5 of the most notable differences between domain categories.

B Domain Name Topics

In section 4.5 we presented the top domain topics among domains deleted early. We extend this to the Alexa top 1M popular domains, and domains that were suspended or expired between 12/31/18 and 1/20/19. Early deleted domain names are much more likely to have a mapped topic with only 27% unknown while the other categories have between 33% and 40% unknown. This intuitively follows our observation that domain names deleted early tend to be longer and contain more words, making it more likely that we find a match in WordNet Domains. Table 6 compares these categories with the full list of 28 topics. For a more fair comparison, this table only represents domain names that are not unknown. There are a few topics that are ranked in different orders between columns, but overall the distribution of topics appears similar. To support this observation we applied a Chi-square test comparing early deletions to each of the other three categories. The effect size and p -value are shown at the bottom of Table 2 below the column being compared to early deletions. The effect size used is $\varphi = \sqrt{\frac{\chi^2}{N}}$. Because the samples are so large, each test produced a p -value very near zero, but the effect sizes are small enough that it is not clear that there is a meaningful difference.

Feature	Alexa Top 1M			Normal Expirations			Suspensions		
	Cohen's <i>d</i>	<i>t</i> -statistic	<i>p</i> -value	Cohen's <i>d</i>	<i>t</i> -statistic	<i>p</i> -value	Cohen's <i>d</i>	<i>t</i> -statistic	<i>p</i> -value
<i>Length</i>	0.53	174.79	0.00	0.13	51.05	0.00	0.23	82.11	0.00
<i>Entropy</i>	0.38	155.43	0.00	0.14	58.15	0.00	0.23	84.06	0.00
<i>Number of words</i>	0.33	112.54	0.00	0.15	60.40	0.00	0.29	102.85	0.00
<i>Percent containing numbers</i>	0.16	51.76	0.00	-0.12	-54.45	0.00	-0.18	-68.07	0.00
<i>Percent containing hyphens</i>	-0.04	-15.85	0.00	0.04	14.28	0.00	-0.09	-33.87	0.00
<i>Percent containing adult keywords</i>	-0.03	-11.21	0.00	0.00	0.3984	0.69	0.01	1.918	0.06
<i>Percent unpronounceable</i>	0.00	0.6791	0.50	-0.16	-72.26	0.00	-0.23	-89.43	0.00

Table 5: Statistical comparison of domain features of early deletions against popular, expired, and suspended domains.

Topics	Early Deletions	Alexa Top 1M	Normal Expirations	Suspensions
Economy/commerce/banking	11.99%	9.90%	12.19%	10.90%
Science/technology	11.58%	13.53%	12.32%	13.14%
Play/sports	8.40%	9.25%	7.91%	7.70%
Health/medicine	7.69%	6.89%	7.61%	7.53%
Architecture	5.79%	6.16%	6.37%	5.57%
Geography	5.58%	5.41%	5.48%	5.79%
Politics/government	5.39%	3.92%	4.48%	4.51%
Art	4.83%	4.52%	5.50%	5.48%
Travel/transport	4.72%	5.43%	5.04%	4.88%
Person	4.19%	3.95%	4.07%	4.13%
Writing/language	3.17%	4.27%	3.22%	3.30%
History/humanity	2.64%	2.41%	2.35%	2.47%
Psychology	2.56%	1.76%	2.14%	2.26%
Media/telecommunication	2.39%	3.64%	2.30%	2.42%
Religion	2.31%	1.55%	1.72%	1.99%
Earth/environment	2.21%	2.11%	2.25%	2.31%
Time period	2.12%	1.93%	1.96%	2.10%
Food	2.05%	1.90%	2.27%	2.12%
Education	2.01%	2.16%	1.55%	1.62%
Quality	1.77%	1.89%	1.96%	1.80%
Animals	1.61%	1.69%	1.70%	2.00%
Administration	1.27%	1.45%	1.17%	1.30%
Metrology	1.27%	1.50%	1.48%	1.87%
Fashion	0.92%	0.95%	1.34%	1.05%
Sexuality	0.54%	0.77%	0.55%	0.62%
Number	0.51%	0.58%	0.59%	0.66%
Color	0.48%	0.46%	0.47%	0.48%
Paranormal	0.01%	0.02%	0.01%	0.01%
Chi-square test		ϕ , <i>p</i> -value 0.07, 0.00	ϕ , <i>p</i> -value 0.03, 0.00	ϕ , <i>p</i> -value 0.05, 0.00

Table 6: Comparison of domain name topics between popular, expired, suspended, and deleted domain names.

HinDom: A Robust Malicious Domain Detection System based on Heterogeneous Information Network with Transductive Classification

Xiaoqing Sun¹, Mingkai Tong², Jiahai Yang³
*Institute for Network Sciences and Cyberspace,
Tsinghua University
National Research Center for Information Science
and Technology, Beijing, China*

Xinran Liu⁴, Heng Liu⁵
⁴ *National Computer Network Emergency
Response Center, Beijing, China*
⁵ *Institute for Network Sciences
and Cyberspace Beijing, China*

Abstract

Domain name system (DNS) is a crucial part of the Internet, yet has been widely exploited by cyber attackers. Apart from making static methods like blacklists or sinkholes infeasible, some weasel attackers can even bypass detection systems with machine learning based classifiers. As a solution to this problem, we propose a robust domain detection system named *HinDom*. Instead of relying on manually selected features, *HinDom* models the DNS scene as a Heterogeneous Information Network (HIN) consist of clients, domains, IP addresses and their diverse relationships. Besides, the metapath-based transductive classification method enables *HinDom* to detect malicious domains with only a small fraction of labeled samples. So far as we know, this is the first work to apply HIN in DNS analysis. We build a prototype of *HinDom* and evaluate it in CERNET2 and TUNET. The results reveal that *HinDom* is accurate, robust and can identify previously unknown malicious domains.

1 Introduction

Though improved increasingly, the Internet is still widely used by adversaries who misuse benign services or protocols to run malicious activities. As a foundation of the Internet, Domain Name System (DNS) provides mappers among IP addresses and domain names, identifying services, devices or other resources in the network. As a consequence, domains are one of the major attack vectors used in various cybercrimes, such as spams, phishing, malware and botnets, etc. Therefore, it is essential to effectively detect and block malicious domains when combating cyber attackers.

After some flexibility-increasing techniques (e.g. Fast-Flux, Domain-Flux, Double-Flux, etc) make static block methods like blacklists infeasible, extensive researches are proposed for malicious domain detection. Traditional systems [3–6, 8, 21] mostly follow a feature based approach. Though these researches get relatively good performance, potential problems are commonly ignored. First, in the training phase,

these detection systems require labeled datasets large enough to guarantee accuracy and coverage. However, the fickle nature of DNS makes accurate labeling an arduous process. Second, it seems they treat each domain individually and rely on some manually selected statistical features (e.g. number of distinct IP addresses, the standard deviation of TTL, etc), making the detection system easy to be evaded by sophisticated attackers [2, 7, 14]. Some researchers [18, 25, 28, 38] intend to utilize structural information for a more robust detection system. However, under the limitations of homogeneous network methods, almost all these researchers model the DNS-related data into a client-domain bipartite graph [28] or a domain-IP bipartite graph [19]. In this case, they can represent at most two types of entities and utilize only one kind of relationship, leaving plenty of information untapped.

Facing the problems mentioned above, we propose an intelligent domain detection system named *HinDom*. First, to fuse more information and introduce higher-level semantics, we model the DNS scene into a Heterogeneous Information Network (HIN), as a HIN model can represent diverse components and relations. Second, a transductive classification method is applied to make use of the structural information, and therefore reduces the dependence on labeled datasets. Besides, considering real-world practicality, we design a series of filtering rules to improve efficiency and reduce noises.

In *HinDom*, we hold the intuitions that, 1) a domain which has strong associations with the known malicious domains is likely to be malicious and 2) attackers can falsify domains individually but cannot easily distort their associations. Thus, to be more robust against attackers' evasion tactics, we first naturally model the DNS scene into a HIN with client nodes, domain nodes, IP address nodes and the following six types of relations among them: (i) *Client-query-Domain*, client *a* queries domain *b*. (ii) *Client-segment-Client*, client *a* and client *b* belong to the same network segment. (iii) *Domain-resolve-IP*, domain *a* is resolved to IP address *b*. (iv) *Domain-similar-Domain*, domain *a* and domain *b* have similar character-level distribution. (v) *Domain-cname-Domain*, domain *a* and domain *b* are in a CNAME record. (vi) *IP-domain-IP*, IP address

a and IP address b are once mapped to the same domain. Then multiple meta-paths are built to represent connections among domains and the PathSim algorithm [32] is applied to compute the similarity among domain nodes. The similarities derived from different meta-paths are combined according to Laplacian Scores [15], excavating associations among domains over multiple views. Finally, illuminated by LLGC [37], GNetMine [17] and HetPathMine [22], a meta-path based transductive classification method is introduced to HinDom to make full use of the information provided by unlabeled samples.

To sum up, we make the following contributions in this research:

1) A comprehensive represent model. We naturally represent the DNS scene by modeling clients, domains, IP addresses and their diverse relations into a HIN. To the best of our knowledge, this is the first work to introduce HIN in malicious domain detection. The combined domain similarity formulated over multiple meta-paths fully represents the rich semantics contained in DNS-related data.

2) Transductive classification in HIN. To reduce the cost of obtaining label information, we apply a meta-path based transductive classification method in HinDom. The experiment results show that HinDom yields ACC: 0.9960, F1-score: 0.9902 with 90% labeled samples and can still detect malicious domains with ACC: 0.9626, F1-score: 0.9116 when the initial labeled sample rate decreases to 10%.

3) Practicality evaluation. We implement a prototype of HinDom and evaluate its performance in two realistic networks, CERNET2 and TUNET. During the deployment, we are able to detect long-buried mining botnets in these two educational networks. The evaluation results show that HinDom is practical in real-world and can identify malicious domains unrevealed by public blacklists.

The rest of this paper is organized as follows. After presenting related work in Section 2 and introducing necessary preliminaries in Section 3, we describe HinDom's framework and the technical details of each component in Section 4. Section 5 reports the experiment results and real-world evaluations. We discuss limitations and future work in Section 6 and summarize our work in Section 7.

2 Related Work

Malicious domain detection. As static block methods like blacklists become infeasible, plenty of researches have been proposed to detect malicious domains. We group them into two categories: *object-based approaches* and *association-based approaches*. It is hard to practice a fair comparison between HinDom and these prior researches, as both their datasets and system implementations are unavailable. In this section, we provide detailed introductions of researches in each group and discuss why HinDom is more advanced.

object-based approaches. Their general method is to first build a classifier based on features extracted from various DNS-related data. Then after being trained with a ground truth dataset, the classifier can be used to inspect unlabeled domains. Notos [3] assigns reputation scores to domains by analyzing network and zone features. It trains classifiers to measure a domain's closeness with five pre-labeled groups (Popular, Common, Akamai, CDN and Dynamic DNS) and uses the calculated scores as features for final detection. Exposure [6] extends the scope of detection to malicious domains involved in spams, phishing, etc and obtains higher efficiency with lower requirements for training data. Kopis [4] gets larger visibility by leveraging traffic among top-level-domain servers. Some works aim at detecting a specific kind of malicious domains. Pleiades [5] detects algorithmically generated domains (AGDs) by analyzing NXDomain responses in DNS traffic while others [20, 29, 36] focus on AGDs' distinguish character distributions. In these researches, various resources are accessed for data enrichment (e.g. ASN, WHOIS, geo-location, network traffic, etc), yet they are analyzed in a coarse-grained way. The classifier treats each domain individually and relies on many statistic results as features, which makes the detection system easy to be evaded by sophisticated attackers. For instance, character patterns of malicious domains can be designed to imitate those of the benign ones [2, 14]. It is also easy for attackers to change temporal patterns like request intervals or TTL values, which commonly service as major features of the classifiers. HinDom is more robust by further utilizing the rich structural information among domains.

association-based approaches. Systems in this group get more macro perspectives by utilizing the relationships among domains. Manadhata et al. [25] build a bipartite client-domain graph and apply belief propagation to discover malicious domains. Segugio [28] focuses on the *who is querying what* information and constructs a machine-domain bipartite graph based on DNS traffic between clients and the resolver. Khalil et al. [18] build a domain-IP graph based on a passive DNS dataset and then simplify it to a domain graph for detection. Futai Zou et al. [38] try to utilize both the client-query-domain relation and the domain-resolve-IP relation by constructing a DNS query response graph and a passive DNS graph. However, due to the limitations of homogeneous network analysis methods, all the above researches can represent at most two types of nodes and utilize only one type of relationship, leaving plenty of information in DNS-related data untapped. HinDom solves this problem with a HIN model which can represent multiple types of nodes and relations for a more comprehensive analysis.

Heterogeneous information network. In recent years, an increasing number of researches start to focus on the importance of heterogeneous information network and apply it to various fields, such as link prediction, recommender system, information fusion, etc [31]. Hindroid [16] is the first work to apply HIN in information security field. By analyz-

ing different relations among API calls in Andriod program, HinDriod extracts higher-level semantics to discover Android malware precisely. Scorpion [11] use HIN to model relations among archives, files, APIs and DLLs for malware detection. As for transductive classification in HIN, GNetMine [17] is the first work to expand a transductive classification method named LLGC [37] from homogeneous network to HIN. Het-PathMine [22] utilizes metapaths to set different classification criterions for different types of objects. Grempt [35] generates local estimated labels for unlabeled samples and expands the transductive method from classification to regression. Illuminated by these researches, our work shows HIN's usefulness in malicious domain detection.

3 PRELIMINARIES

3.1 Heterogeneous Information Network

In the real world, most systems contain diverse interactions among different types of components. However, for ease of analysis, they are usually modeled as homogeneous networks with unique type of nodes and links. In this case, information loss is caused by ignoring differences among objects and relationships. Recently, researchers start to model these systems into Heterogeneous Information Networks (HINs) [33], which can fuse richer semantics and support more comprehensive represents. The basic concepts of HIN are as follows.

Definition 1. Heterogeneous Information Network (HIN) [33]. Given a graph $G = \langle V, E \rangle$, where V is the set of nodes, E is the set of links. m types of objects are denoted as $V_1 = \{v_{11}, v_{12}, \dots, v_{1n_1}\}, \dots, V_m = \{v_{m1}, v_{m2}, \dots, v_{mn_m}\}$, where n_i is the number of the i -th type nodes and p types of relationships are denoted as $E_1 = \{E_{11}, E_{12}, \dots, E_{1q_1}\}, \dots, E_p = \{e_{p1}, e_{p2}, \dots, e_{pq_p}\}$, where q_i is the number of the i -th type of relations. We regard G as a HIN if $m \geq 2$ or $p \geq 2$. When $m=p=1$, G reduces to a homogeneous network.

Definition 2. Network Schema [33]. $T_G = \langle A, R \rangle$ is the network schema of a HIN $G = \langle V, E \rangle$, with type mapping function $\varphi: V \rightarrow A$ and $\psi: E \rightarrow R$, where A is the set of object types and R is the set of relationship types.

Definition 3. metapath [32]. Given a network schema $T_G = \langle A, R \rangle$, a metapath P defines a composite relation $R = R_1 \circ R_2 \circ \dots \circ R_L$ between A_1 and A_{L+1} , where \circ is the relation composition operator. P is denoted as $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \dots \xrightarrow{R_L} A_{L+1}$, where L is the length of the metapath.

Figure 1 shows a HIN model of the bibliographic dataset DBLP [12]. It represents four types of nodes: paper (P), author (A), conference (C) and keyword (K), as well as four kinds of links: authors write papers, papers are published in conferences, papers contain keywords and a paper cites other

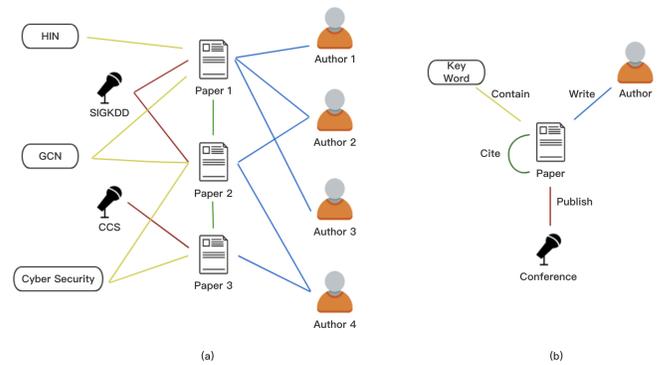


Figure 1: An example of HIN instance (a) and its network schema (b)

papers. The relation between author and conference can be represented as metapath $A \xrightarrow{write} P \xrightarrow{published} C$, or APC for short. Metapath $APCPA$ indicates the relation that authors have published papers on the same conference. To sum up, a HIN instance contains detailed information while its network schema describes the structural constraints and metapaths are used to represent complex relations among entities.

3.2 Transductive Classification

Unlike inductive classification, instead of learning general decision functions from training data, transductive classification infers from specific training cases to specific test cases. The situation is more like to propagate label information over the whole network. Therefore, when there are many test samples but few labeled training samples, transductive methods can classify more effectively with the utilization of information from the unlabeled data. Based on definitions in Section 3.1, transductive classification in HIN can be defined as follow.

Definition 4. Transductive classification in HIN [17]. Given a HIN $G = \langle V, E \rangle$ and a subset of its labeled nodes $\tilde{V} \subseteq V$ with their label information denoted by vector Y , transductive classification is to predict labels for nodes in $V - \tilde{V}$.

4 HinDom System Description

The intuition of HinDom is that domains with strong relationships tend to belong to the same class (benign or malicious). Besides, attackers can only falsify domain's features individually but cannot easily control the natural associations generated in DNS. We model clients, domains, IP addresses as well as their relations into a HIN and analyze six types of associations among domains based on the following two observations: (i) Attackers are subjected to the cost of network resources. That is, though trying to stay dynamic, attackers tend to reuse network resources due to economic constraints.

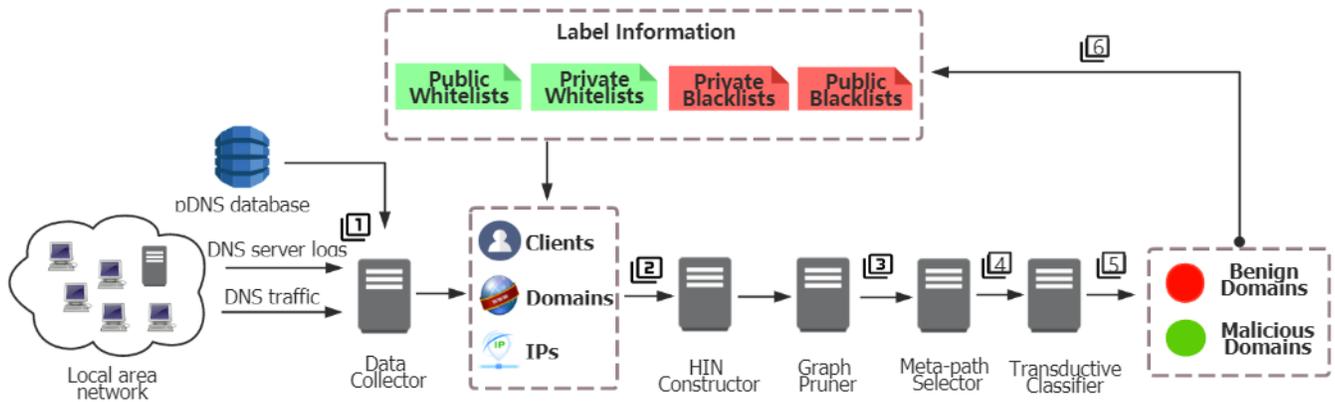


Figure 2: The architecture of HinDom

(ii) The set of malicious domains queried by victims of the same attacker tend to overlap.

As shown in Figure 2, HinDom has five main components: Data Collector, HIN Constructor, Graph Pruner, Meta-path Combiner and Transductive Classifier. After DNS-related data are collected (step 1), a HIN consist of clients, domains, IP addresses and their various relations is constructed to represent the DNS scene (step 2). Then some nodes in the graph are pruned to filter noises and reduce computing complexity (step 3). We analyze six different meta-paths and combine them according to their influences on domain detection (step 4). Finally, based on some initial label information, the transductive classifier categorizes unlabeled domains (step 5). We analyze the classification result and add it to private whitelist or blacklist for further detection (step 6). In the following, we will introduce each component in detail.

4.1 Data Collector

To obtain richer information that reveals the behavior of actual users, instead of sending specific DNS queries on purpose, we execute DNS data collection passively. Three major data sources that we collect are: (i) *DNS server log*. When dealing with queries, DNS servers generate logs to collect information like source IP, queried domain, time, etc. Among all the logs, those of the recursive servers are widely used to extract information about "who is querying what" in local area networks. (ii) *DNS traffic*. It contains the most comprehensive information with various fields such as NS, MX, TXT, PTR, etc. Yet, considering privacy issues, this kind of data is hard to share publicly. (iii) *Passive DNS dataset*. Some organizations (e.g. Internet Systems Consortium, Farsight Security [10], 360 NetLab [26], etc.) have constructed passive DNS (pDNS) systems with sensors voluntarily deployed by contributors in their infrastructures. They aggregate the captured DNS messages before making them publicly available. Records in pDNS do not contain client information. They only offer the

first and last timestamps of a domain's appearance, as well as the total number of domain-IP resolutions in between.

The data collector collects resolver's logs or DNS traffic between clients and the resolver in a local area network (LAN) during a time window T , which can be set to an hour, a day or a week, considering computing resources and the network size. Noting that HinDom can construct the HIN model just based on DNS response traffic. But pDNS dataset can provide richer information on domain-IP relations in both spatial and temporal dimensions. Besides, when DNS traffic data is unavailable due to permission or technique restrictions, HinDom can utilize DNS logs construct the client-domain part and use pDNS dataset for the domain-IP part.

4.2 HIN Constructor

As shown in Figure 3, based on the collected data, HinDom naturally models the DNS scene as a HIN consist of clients, domains, IP addresses and six types of relations among them. The details of these relations are as follows and Table 1 lists their corresponding adjacent matrices.

- **Client-query-Domain**, we use matrix Q to denote that domain i is queried by client j .
- **Client-segment-Client**, we use matrix N to denote that client i and client j belong to the same network segment.
- **Domain-resolve-IP**, we use matrix R to denote that domain i is resolved to IP address j .
- **Domain-similar-Domain**, we use matrix S to denote the character-level similarity between domain i and j .
- **Domain-cname-Domain**, we use matrix C to denote that domain i and domain j are in a CNAME record.
- **IP-domain-IP**, we use matrix D to denote that IP address i and IP address j are once mapped to the same domain.

All these adjacent matrices can be naturally extracted from the DNS-related data except matrix S which indicates the

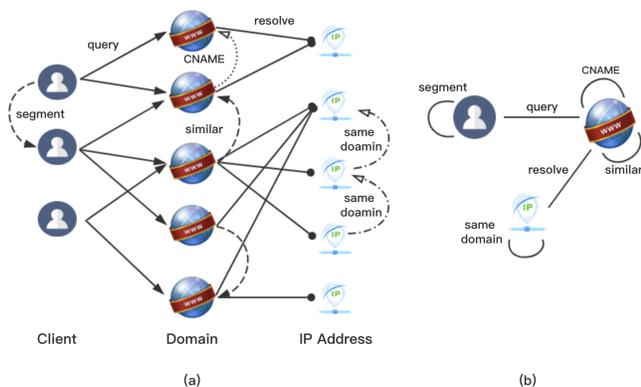


Figure 3: HIN instance (a) and its network schema (b) in HinDom

character-level similarity among domains. We use n-gram to process the domain name strings, regard the results of the entire dataset as a vocabulary and embed each domain into a characteristic vector. Then we use the K-Means algorithm to cluster these vectors into K categories and transform the clustering result into matrix S . In our experiments, we test uni-gram, bi-gram and tri-grams for both types of features. Tri-grams brings a marginal improvement with much more cost on memory requirements. Considering performance and complexity, we concatenate uni-grams and bi-grams as features and empirically set $K = 20$.

Table 1: Elements and descriptions of the relation matrices

Matrix	Element	Description
Q	$q_{i,j}$	if domain i is queried by client j , then $q_{i,j} = 1$, otherwise, $q_{i,j} = 0$.
N	$n_{i,j}$	symmetric, if client i and j belong to the same network segment, then $n_{i,j} = n_{j,i} = 1$, otherwise, $n_{i,j} = n_{j,i} = 0$.
R	$r_{i,j}$	if domain i is resolved to ip j , then $r_{i,j} = 1$, otherwise, $r_{i,j} = 0$.
S	$s_{i,j}$	symmetric, if domain i and j are similar on character level, then $s_{i,j} = s_{j,i} = 1$, otherwise, $s_{i,j} = s_{j,i} = 0$.
C	$c_{i,j}$	symmetric, if domain i is the cname of domain j , then $c_{i,j} = c_{j,i} = 1$, otherwise, $c_{i,j} = c_{j,i} = 0$.
D	$d_{i,j}$	symmetric, if IP address i and j are once resolved to the same domain, then $d_{i,j} = d_{j,i} = 1$, otherwise, $d_{i,j} = d_{j,i} = 0$.

4.3 Graph Pruner

Because we aim at detecting malicious domains in a campus or enterprise network, the HIN may contain millions of nodes and billions of edges, it is a waste of computing resources to model all these entities and perform the corresponding matrix operations. Besides, The data we collect directly from DNS traffic or logs is quite dirty with noises like irregular domains, "large" clients, etc. Therefore, we add a graph pruning module in HinDom to improve its performance and practicality. The graph pruner filters nodes according to the following conservative rules.

- **Unusual domains.** We remove domains that fail to meet the naming rules, for example, *icmsb2018(at)163.com*, which may be caused by mistyping, misconfiguration or benign services' misuse. Besides we discard domains that are queried by only one client to focus on those that have greater impacts over the LAN.
- **Popular domains.** In most cases, popular domains queried by a large fraction of clients in a LAN are inclined to be benign; otherwise, there will be a significant attack event and will be easily detected by the security management department. Besides, these popular domains cause much computational complexity as they are all nodes with high degrees in HIN. Therefore, we filter out domains that queried by $K_d\%$ clients in a network. To be conservative, we set $K_d\%$ to be 25% in our experiments.
- **Large clients.** There are some "large" clients outstanding by querying a large fraction of the whole domain set. We find these devices are often DNS forwarders or large proxies and thus can not represent the behavior of regular clients. HinDom removes them to eliminate the ambiguousness and complicity they bring into the system. In our evaluations, the top $K_a\%$ (empirically set to 0.1%) most active clients are discarded.
- **Inactive clients.** We regard clients that query less than K_c domains as the inactive ones. They are discarded for the lack of effects on mining associations among domains. In our experiments, K_c is set to be 3.
- **Rare IPs.** For the same reason as above, IP addresses that only map to one domain are also filtered out to boost performance and save computing resources.

To be more conservative for information loss caused by graph pruning, we set some exceptions against the mentioned rules based on label information. Considering that some attackers try to hide by reducing activities, we keep domains with clear malicious labels even if they are regarded as unusual ones. Same to their related clients or IP addresses.

4.4 Meta-path Combiner

As mentioned in Section 3, meta-paths are used in HIN to denote complex associations among nodes. Because we are in-

Table 2: Commuting Matrix of each metapath

PID	metapath	Commuting Matrix M	Description
1	$d \xrightarrow{S} d$	S	domains similar to each other on character level
2	$d \xrightarrow{C} d$	C	the cname relationship among domains
3	$d \xrightarrow{Q} c \xrightarrow{Q^T} d$	QQ^T	domains queried by same clients
4	$d \xrightarrow{R} ip \xrightarrow{R^T} d$	RR^T	domains resolved to same IP address
5	$d \xrightarrow{Q} c \xrightarrow{N} c \xrightarrow{Q^T} d$	QNQ^T	domains queried by clients belong to the same subnet
6	$d \xrightarrow{R} ip \xrightarrow{D} ip \xrightarrow{R^T} d$	RDR^T	domains resolved to IPs that belong to the same attacker

terested in malicious domain detection, HinDom only chooses symmetric meta-path where $A_1 = A_{L+1} = domains$ and derives six types of meta-paths from the six relations mentioned above. Table 2 displays the description and corresponding commuting matrix M_k of each meta-path while the reasons for choosing them are listed as follows.

- **P1:** $d \xrightarrow{S} d$. We have noticed that benign and malicious domains differ in character distributions. Besides, malicious domain names from the same family tend to follow a similar textual pattern.
- **P2:** $d \xrightarrow{C} d$. The cname domain of a benign domain is unlikely to be malicious, vice versa.
- **P3:** $d \xrightarrow{Q} c \xrightarrow{Q^T} d$. Infected clients of the same attackers tend to query partially overlapping sets of malicious domains while normal clients have no reasons to reach out for them.
- **P4:** $d \xrightarrow{R} ip \xrightarrow{R^T} d$. IP resources are relatively stable in Internet, domains resolved to the same IP address in a period tend to belong to the same class.
- **P5:** $d \xrightarrow{Q} c \xrightarrow{N} c \xrightarrow{Q^T} d$. Adjacent clients are vulnerable to the same attacks. For example, malware propagating in a subnet or spams aiming to clients on the same segment.
- **P6:** $d \xrightarrow{R} ip \xrightarrow{D} ip \xrightarrow{R^T} d$. Even trying to keep flexible, with funding limits, attackers are likely to reuse their domain or IP resources.

Based on meta-paths, an algorithm named PathSim [32] can be used to measure the similarity among nodes. Yet different meta-paths represent associations from different points of view which are not equally important in malicious domain detection. HinDom obtains a combined meta-path with the corresponding similarity matrix denoted as follow, where w_k is the weight assigned to each meta-path.

$$M' = \sum_{k=1}^6 \omega_k \cdot PathSim(M_k) = \sum_{k=1}^6 \omega_k \cdot \frac{2M_{k(i,j)}}{M_{k(i,i)} + M_{k(j,j)}}$$

Many methods can be used to compute the weight vector, for example, linear regression with gradient descent. HinDom

chooses to use the Laplacian Score (LS) [15] for two reasons: First, LS can be applied to unsupervised situations. Second, as a "filter" method, LS is independent of further learning algorithms and can evaluate features directly from the local geometric structure of data. The basic idea of LS is to evaluate features according to their locality preserving power. LS constructs a nearest neighbor graph and seeks features respecting this graph structure. Specifically, We code all meta-paths into a tensor $T \in R^{6 \times n \times n}$, where $T_{k,i,j} = M_k(i,j)$, n is the number of domains, M_k is the commuting matrix of meta-path P_k . Then a domain meta-path representation matrix $W \in R^{n \times m}$, where $W_{k,i} = \sum_j T_{k,i,j}$ is generated as the input of LS.

4.5 Transductive Classifier

Though some public domain lists are commonly used as label information in malicious domain detections, some subtle issues are ignored. For whitelists, the widely used Alexa top K list only contains second-level domains (2LD) sorted by popularity, which leads to many false positives. For example, a prevalent 2LD may hold proxies to malicious activities and some malicious domains may rank high with a burst of queries from the infected clients. As for blacklists, though usually generated with robust evidences, some discrepancies are still caused by the fickle nature of DNS. For instance, domains like *alipay.com* are in DGArchive [27], a database of DGAs and the corresponding domains. Besides, when new malicious domains come, blacklists cannot update in time. In a word, none of these lists is completely reliable. It is a time-consuming and cost-expensive process to obtain an accurately labeled dataset as the ground truth.

To reduce the cost of labeling, HinDom applies a meta-path based transductive classification method which can perform well even with a small fraction of labeled samples. The basic two assumptions in transductive classification are (i) *smoothness assumption*, objects with tight relationships tend to belong to the same class; (ii) *fitting assumption*, the classification results of the known nodes should consist with the pre-labeled information. Therefore, the cost function of the transductive classifier is as follow,

$$Q(F) = \frac{1}{2} \left(\sum_{i,j=0}^{n-1} M'_{i,j} \left\| \frac{F_i}{\sqrt{D_{ii}}} - \frac{F_j}{\sqrt{D_{jj}}} \right\|^2 + \mu \sum_{i=0}^{n-1} \|F_i - Y_i\|^2 \right)$$

where n is the number of domain nodes in HIN, $M' \in R^{n \times n}$ is the similarity matrix we get from the combined metapath, $D \in R^{n \times n}$ is a diagonal matrix whose (i, i) -element equals to the sum of the i -th row of M' . $F \in R^{n \times 2}$ contains each domain's probability of being benign or malicious while $Y \in R^{n \times 2}$ denotes their pre-labeled information. We can see the first term of this cost function represents the smoothness assumption while the second term follows the fitness assumption. Trade-off between the two assumptions is adjusted by parameter μ . In order to find the F^* that minimize $Q(F)$, we get

$$\frac{dQ}{dF} = F^* - F^*S + \mu(F^* - Y) = 0$$

$$F^* = \beta(I - \alpha S)^{-1}Y$$

where $\alpha = \frac{1}{1 + \mu}$, $\beta = \frac{\mu}{1 + \mu}$ and $S = D^{-1/2}M'D^{-1/2}$.

We get the theoretical optimal solution, yet in the real world, inverting a large matrix will consume too much computing resources. Thus, illuminated by LLGC [37], in HinDom we perform iterations $F(t+1) = \alpha SF(t) + \beta Y$ to approach the optimal solution. We refer the readers to LLGC [37] for the theoretical proof that this iteration can coverage to the optimal solution. The algorithm of Transductive Classifier is summarized as follow.

Step 1, Given a HIN $G = \langle V, E \rangle$ with incomplete domain label information Y , get similarity matrix M' from Metapath Combiner.

Step 2, Regularize the similarity matrix with $S = D^{-1/2}M'D^{-1/2}$, where D is a diagonal matrix whose (i, i) -element equals to the sum of the i -th row of M' .

Step 3, Set $F(0) = Y$, iterate $F(t+1) = \alpha SF(t) + \beta Y$ until it converges.

Step 4, Label domain i 'benign' if $F_{i,0} \geq F_{i,1}$, vice versa.

We further analyze the classification results and add domains with solid labels, namely the difference between $F_t[i, 0]$ and $F_t[i, 1]$ is higher than the threshold θ , into local whitelist or blacklist as a supplement for further detection. Considering the dynamic nature of DNS, we only keep local label information within 7 days.

5 Experiments

In this section, we present comprehensive experiments to evaluate HinDom from three aspects: performance, robustness and practicality. For performance, we first analyze detection results and the corresponding weight of each meta-path to prove the effectiveness of Meta-path Combiner. Then, we test HinDom in insufficient labeling scenario and multi-classification

scenario. For robustness, we test HinDom's ability to deal with label noises in the training dataset. For practicality, we test HinDom when only public labels are available and deploy it in two real-world networks: CERNET2 and TUNET.

5.1 Setup

We evaluate HinDom in two real-world networks: CERNET2 and TUNET. Our research has obtained permissions from the relevant security management teams. The DNS-related data we get has been processed to minimize privacy disclosure, for example, the IP addresses of clients are desensitized by numerical identifiers.

CERNET2, the second generation of China Education and Research Computer Network. Jointly built by 26 universities, CERNET2 is the first IPv6 national backbone network in China and is the world's largest next-generation Internet backbone network using pure IPv6 technology. At present, CERNET2 has 25 core nodes distributed in 20 cities with $2.5G \sim 10Gbps$ bandwidth and provides IPv6 access services for more than 5 million users in about 500 research institutes. We capture DNS traffic in CERNET2 at Tsinghua node.

TUNET, the campus network of Tsinghua University. By statistics, we find that over 0.24 million clients request about 1.5 million unique domains per day. With close supervision and control, TUNET is much purer than CERNET2 and hides less malicious domains.

In this research, we use DNS traffic of CERNET2 to construct its HIN and just use 360 pDNS dataset for data enrichment. As for TUNET, due to permission restrictions, we only get the logs of its central DNS resolver. The logs and 360 pDNS dataset are used respectively to construct HIN's domain-client part and domain-IP part. Besides, as for DNS traffic, we only use A, AAAA and CNAME records currently and may expand to PDG, MX, SRV, NS, PTR for richer information in the future.

Table 3: Description of the testing HIN instance

Nodes					
Clients	Benign Domains	Malicious Domains		IPs	
~0.49M	~0.7M	~0.25M		~0.26M	
Edges (C-clients, D-domains, IP-IPs, c-cname, s-similar)					
C-C	C-D	D-c-D	D-s-D	D-IP	IP-IP
~93M	~112M	~1.3M	~15M	~3.1M	~4.3M
Nodes Total			Edges Total		
~1.7M			~228.7M		

To build the test dataset, we labeled about 1 million domains queried in CERNET2 on 13 April 2018 by referring to various whitelists/blacklists and expertise. For benign information, we regard domains whose 2LD appear in Alexa Top 1K list [1] or our local whitelist as benign ones. For malicious

domains, we use multiple sources like *Malwaredomains.com* [24], *Malwaredomainlist.com* [23], DGArchive [27], etc. We also refer to integrated services like Google Safe Browsing [9] and VirusTotal [34]. Besides, we manually investigate dubious domains which have both benign and malicious information. According to the graph pruning rules mentioned in Section 4.3, we discard some unhelpful nodes by setting $K_d = 25$, $K_a = 0.1$, $K_c = 3$. Table 3 shows details about the HIN instance we construct and Table 4 lists the evaluation metrics used in the experiments.

Table 4: Metrics for evaluation

Metric	Description
TP	malicious domains labeled as malicious
FP	benign domains labeled as malicious
TN	benign domains labeled as benign
FN	malicious domains labeled as benign
accuracy	$(TP + TN) / (TP + FP + TN + FN)$
precision	$TP / (TP + FP)$
recall	$TP / (TP + FN)$
F1	$2 \times (precision \cdot recall) / (precision + recall)$
ROC	a curve plotting TPR against FPR with various thresholds
AUC	area under the ROC curve

5.2 Metapath Combiner

HinDom relies on six meta-paths to formulate the final similarity matrix. These meta-paths represent associations among domains from different perspectives and thus have different influences on domain detection. We test the detection performance of each meta-path by randomly keep the label information for 70% domains, leaving the remained 30% as test samples and repeat the procedure for 10 times. The accuracy, precision and recall of labels generated by each meta-path are shown in Table 6 while the ROCs are shown in Figure 4.

Table 5: Labeled Metrics of each meta-path

PID	metapath	accuracy	precision	recall
1	S	0.9376	0.8703	0.8712
2	C	0.9999	0.9999	0.9998
3	QQ^T	0.9567	0.9198	0.8860
4	RR^T	0.9888	0.9879	0.9989
5	QNQ^T	0.9571	0.9251	0.8710
6	RDR^T	0.9754	0.9580	0.9281

Noting that some meta-paths have low connectivity, which means the classifier cannot reach plenty of domains if it only relies on one of the meta-paths. Thus, when assigning weights

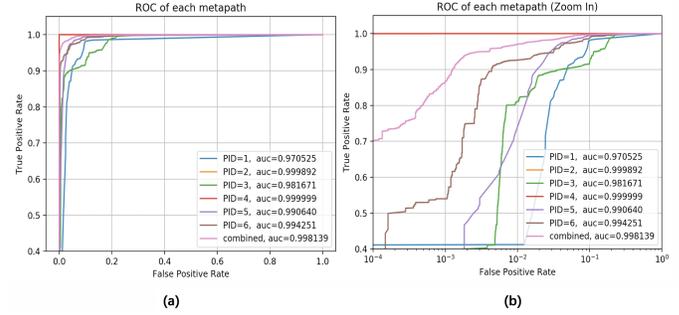


Figure 4: ROC for the labeled result of each metapath

to each meta-path, we need to consider two aspects: *coverage* and *accuracy*. For coverage, it means with this meta-path, we can fully exploit connections so that few domains will be left unlabeled. For accuracy, it means among the labeled domains, few are misclassified. The Meta-path Combiner assigns a weight to each meta-path according to Laplacian Scores. To test its effectiveness, we list the average detection results and the corresponding weight of each meta-path in Table 6, where the *Unlabeled rate* and *F1 score* reveal coverage and accuracy respectively.

Table 6: Detection results of each meta-path

PID	metapath	F1 Score	Unlabeled Rate	Weight
1	S	0.8708	0.0027	0.1698
2	C	0.9996	0.9133	0.0003
3	QQ^T	0.9026	0.0917	0.1386
4	RR^T	0.9934	0.5317	0.0125
5	QNQ^T	0.8973	0.0049	0.3826
6	RDR^T	0.9428	0.2057	0.2962
combined path		0.9743	0	-

From Table 5, we can see that HinDom combines these meta-paths in order: PID 5, PID 6, PID 1, PID 3, PID 4 and PID 2. The combined path gets the ability to cover the whole set of domains with high detection accuracy. It is worth noting that some meta-paths with strong relations in domain detection (e.g. PID2: $d \xrightarrow{C} d$) get extremely high accuracy but very low coverage. This is consistent with the fact that domains in a CNAME record tend to belong to the same class, yet not many domains are in this kind of relationship. With Laplacian Score, the Meta-path Combiner assigns these meta-paths with relatively low weights to ensure that HinDom can detect as many domain names as possible. Take PID 4: $d \xrightarrow{R} ip \xrightarrow{R^T} d$ and PID 6: $d \xrightarrow{R} ip \xrightarrow{D} ip \xrightarrow{R^T} d$ for instance, the latter extends its coverage by utilizing relations among IP addresses, though introducing some noises, it can reach more domains and thus plays a more important role in this scenario.

Table 7: Detection results with different fraction of labels

Initial Label Fraction	Metrics of each method							
	NB		SVM		RF		HinDom	
	accuracy	F1 Score	accuracy	F1 Score	accuracy	F1 Score	accuracy	F1 Score
90%	0.9632	0.9499	0.9864	0.9827	0.9813	0.9803	0.9960	0.9905
70%	0.9429	0.9276	0.9682	0.9550	0.9700	0.9611	0.9880	0.9743
50%	0.9020	0.8912	0.9286	0.9090	0.9257	0.9180	0.9840	0.9776
30%	0.8235	0.8260	0.8527	0.8516	0.8613	0.8544	0.9698	0.9453
10%	0.7929	0.7834	0.8120	0.8031	0.8141	0.7902	0.9626	0.9116

5.3 Transductive Classification

As mentioned in Section 3.5, the public whitelists or blacklists of domains are not completely reliable and have a number of subtle issues. In order to reduce the cost of labeling domains manually, HinDom utilizes a meta-path based transductive classification method to make better use of structural information of the unlabeled samples. To test the effectiveness of the Transductive Classifier, in this section, we compare HinDom with three inductive classification methods: Navie Bayes (NB), Support Vector Machine (SVM) and Random Forest (RF), on the situation where 90%, 70%, 50%, 30%, 10% labels are kept randomly. For the inductive methods, we extract all details about entities and relations of the HIN instance as features to learn the classification functions. The *accuracy* and *F1 score* of each method are shown in Table 7. As we can see, HinDom maintains relatively stable performance when the fraction of initial label information decreases. It yields *accuracy*: 0.9960, *F1*:0.9905 when 90% domains are pre-labeled and still obtains *accuracy*: 0.9626, *F1*: 0.9116 when only 10% labels are left at the beginning. As for the inductive methods, they can obtain relatively good performance with sufficient labels, yet the accuracy drops to around 0.8 as the size of training dataset decreases. The reason behind is that by using HIN assisted with a transductive classifier, HinDom can not only learn from the labeled data but also fully exploit associations among domains and generally propagates the initial label information over the whole network.

To find the minimum threshold of label fraction for relatively high performance, we gradually reduce the initial label information and draw curves of accuracy and F1 score in Figure 5. The breakpoint is around 10%, which means we need to provide at least 10 percent labeled samples for domain detection, otherwise, the performance of HinDom will suffer a dramatic decrease.

5.4 Robutness

Though consuming lots of efforts, the manually labeled training dataset often contains noises which might be caused by attackers' tricks or human mistakes. In this section, to test

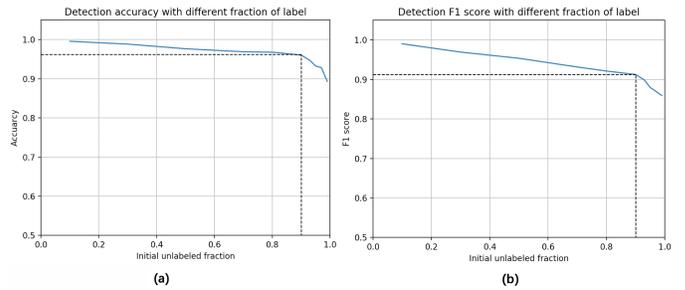


Figure 5: Accuracy and F1 score with different initial label fraction

HinDom's robustness to label noises, we keep 70% initial label information, randomly change labels of $k_d\%$ training samples and compare the detection results of HinDom with those traditional methods: NB, SVM and RF. We increase $k_d\%$ gradually and stop at 50% where none of these methods can generate a tolerable detection result. We repeat each scenario for 10 times. Figure 6 shows the average accuracy trend and F1-score trend of each method. We can see that the traditional machine learning methods are susceptible to mislabeling. Meanwhile, with a better understanding of structural information, HinDom can hold relatively stable performance when dealing with some label noises.

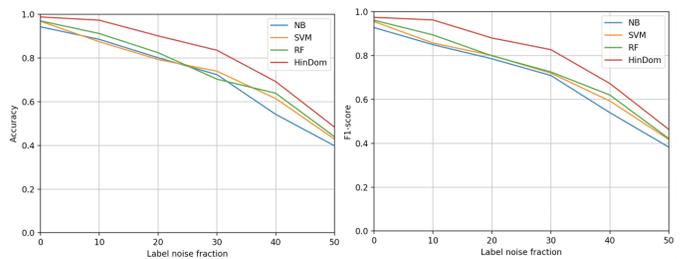


Figure 6: Accuracy and F1 score with $k_d\%$ label noise

5.5 Multi-classification

It is observed that malicious domains from the same attacker tend to follow the same patterns, regardless of character distributions, the victim groups or the sets of IP addresses they map to. From the definition of Transductive Classifier, we can see that apart from detecting malicious domains, HinDom can support multi-classification and identify which categories the malicious domains belong to. The domain family information provided by multi-classification is useful for follow-up work like reverse engineering and security reports. To test HinDom’s multi-classification ability, we further label the 0.25 million malicious domains mentioned in Section 4.1 into 13 categories based on the malware or cybercrimines they related to. Note that for those categories with less than $F=150$ domains, we group their domains together as Class *Rare*.

Table 8: Multi-classification with different fraction of labels

Initial Label Fraction	Metrics			
	accuracy	precision	recall	F1 Score
90%	0.9814	0.9786	0.9815	0.9801
70%	0.9783	0.9759	0.9765	0.9762
50%	0.9720	0.9673	0.9706	0.9689
30%	0.9644	0.96178	0.9654	0.9636
10%	0.9598	0.9543	0.9585	0.95647

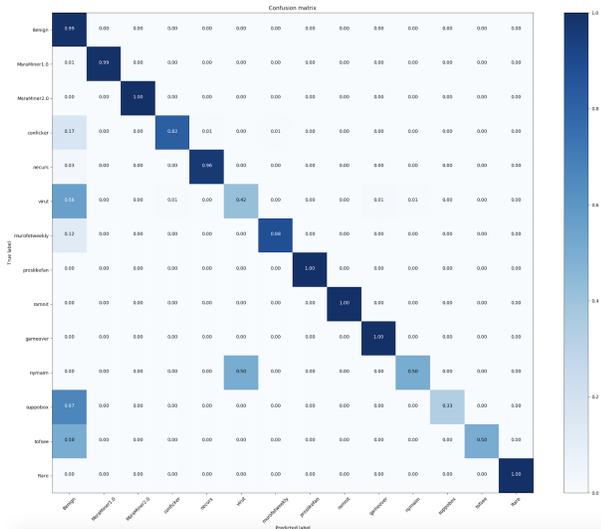


Figure 7: Confusion matrix of multi-classification with 50% initial labels

Table 8 lists the detection results with different fractions of initial labels. Considering the sample imbalance of each category, we use the weighted-average metrics to evaluate the multi-classification. With this method, the metrics of each label will be weighted by support to find their average. Due

to space limitations, Figure 7 only displays the confusion matrix of multi-classification when there are 50% initial labels. The X axis denotes the true category of each domain while the Y axis denotes their predicted labels. The confuse matrix shows that most misclassifications are between Class *Benign* and some malicious classes with relatively small sample size, which we suppose is caused by data skew. The additional information about domain family brings higher TPR yet lead to worse FPR. To solve this problem, we will separate HinDom into two stages, first distinguish malicious domains from the benign ones and then multi classify these malicious domains to identify their families.

5.6 Public Information Only

To test the real-world practicality, we eliminate the influence of human decisions by running HinDom with initial labels only from public whitelists or blacklists. When using lists with only 2LDs (e.g. Alexa Top List, DGArchive), we set domains with the same 2LD to the same class. For instance, *agoodm.m.taobao.com* and *chat.im.taobao.com* are in benign class because their 2LD *taobao.com* is in Alexa Top 1K. For those dubious domains appear in both whitelists and blacklists, we randomly set them to benign or malicious class. In the end, we label about 24% of the 0.95 million domains with about 0.04 million confused ones. HinDom yields *accuracy*: 0.9634, *F1*: 0.9253 with this kind of initial label information.

Table 9 shows the performance of HinDom with public labels and with 20%, 30% manual labels while Figure 8 displays their ROC. We can see that HinDom gets similar performance no matter with only public information or with manual labels of the same proportion. Besides, by analyzing the detection results, we find that with a relatively small $\mu = 0.3$, which means we do not firmly insist on the pre-labeled information, HinDom can correct the label of some domains that are misclassified at the very beginning. For example, *memberprod.alipay.com* and *hosting.rediff.com* were randomly assigned to the malicious class because they have both benign and malicious information. HinDom adjusts their labels from 1 to 0 after 11 times iteration because of the strong associations they have with the benign domains.

Table 9: Detection results with public and manual labels

Metrics	Labels		
	30% Manual	Public only	20% Manual
accuracy	0.9698	0.9634	0.9633
precision	0.9510	0.9367	0.9380
recall	0.9396	0.9142	0.9087
F1 score	0.9453	0.9253	0.9232

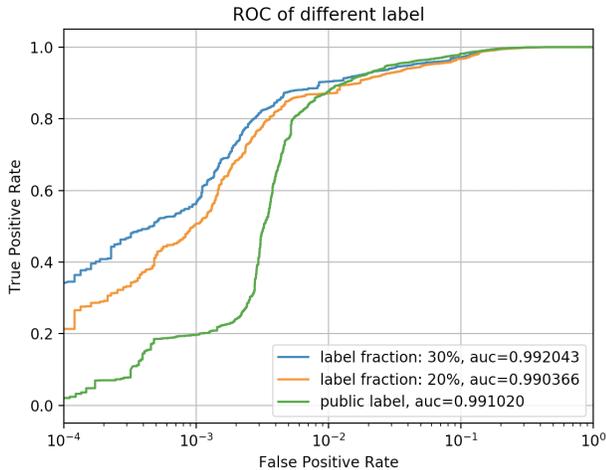


Figure 8: ROC of detection with public or manual labels

5.7 Compare with detection engines

To test HinDom’s practicality, we also compare its performance with existing Detection Engines (DE) on VirusTotal [34]. We randomly choose 30% domains from the experimental dataset as testing samples and collect the detection results of all 66 engines in VirusTotal. Due to space limitations, Table 10 only displays nine engines that have relatively good performance or that are widely used by security vendors like McAfee, Kaspersky, etc. We can see that HinDom outperforms six out of nine engines with *accuracy*: 0.9697 and *F1-score*: 0.9460. This is a remarkable result as these engines are supported by more expertise and the label information of our experimental dataset is partly depended on VirusTotal.

Table 10: Detection results of engines from VirusTotal

Engines	HinDom	DE1	DE2	DE3	DE4
ACC	0.9697	0.9038	0.9785	0.8952	0.9603
F1-score	0.9460	0.8898	0.9701	0.8824	0.9511
Engines	DE5	DE6	DE7	DE8	DE9
ACC	0.9812	0.9846	0.9023	0.9116	0.8974
F1-score	0.9747	0.9800	0.8975	0.9048	0.8423

5.8 Real-world Deployment

We implement a prototype system of HinDom and deploy it in CERNET2 and TUNET. Though HinDom is not a real-time detection system, it can be run every hour or every day to detect ongoing malicious activities. Taking computing resources and the network scale into consideration, we choose to set the time window T to one hour in our deployment. In other words, we construct a HIN instance based on the DNS-related

data within an hour, get the detection results, accumulate label information, and the cycle repeats.

In CERNET2, on average, about 40 thousand clients initiate 3.8 million DNS requests within an hour and about 0.25 million unique domains exist after graph pruning. With HinDom we find 3.34% domains are malicious. The result is proved to be reliable by experts certification and some of these malicious domains are detected several months before they are reported by public services. In particular, HinDom detects a bunch of domains that are not listed in any public blacklists, neither are the IP addresses they are resolved to. After consulting Qihoo 360, a Chinese internet security company, we confirm these domains belong to a long-buried mining botnet named *MsraMiner*. In TUNET, about 50 thousand clients request for 0.4 million unique domains per hour. With closer network supervision, the proportion of malicious domains drops to 1.21%. However, we still detect the variation of *MsraMiner*, with domains like *ral.kziu0tpofwf.club* and *sim.jiovt.com*, in the campus network. The above detection results have been reported to the relevant network management department.

6 Limitation and Future Work

Currently, HinDom constructs a HIN of clients, domains and IP addresses based on the DNS-related data and can perform well even with a small fraction of labeled samples. Yet there are some potential problems in views of scalability and practicability. We discuss HinDom’s limits and our future work in this section.

First, efficiency. With a graph-based mechanism, HinDom cannot be deployed in a real-time mode. We need to choose a proper time window T to collect data and then conduct the detection procedure off-line. If T is too small, the collected data will not be sufficient for accurate detection. Yet with a very big time window, HinDom will need more computing resources and longer detection time. Thus, there is a trade-off between accuracy and efficiency. We have mentioned that the multiplication between adjacency matrices in HinDom is a resource-consuming operation. We recommend matrix block calculation and parallel computation framework like Hadoop in the real-world deployment. Besides, we find that embedding is an up-and-coming approach to represent graphs in a low dimensional way. Some embedding methods such as HIN2Vec [13] and ESim [30] have been proposed to represent nodes in HIN, which can be used to improve HinDom’s efficiency.

Second, the detection range. As an association-based detection system, HinDom can only detect malicious domains that have direct or indirect relations with others. When a new type of malicious domain is just registered and has few relations with other entities, HinDom cannot detect it immediately. Besides, HinDom may not hold a good performance when detecting malicious domains hosted by network services like

CDN, as they are related to a large number of benign domains. To eliminate these problems, we plan to utilize more types of DNS-related data and dig out richer associations among domains. For example, WHOIS dataset is an important clue with information about registered users or assignees. Besides, for now, we only use A, AAAA and CNAME records in DNS traffic and may expand to PDG, MX, SRV, NS, PTR for richer information in the future.

Third, further analyzation. After getting domain detection results, we can design functions and go further to find the infected clients and malicious IPs in the network based on the rich semantic information represented in HinDom. With this information, the security management team can narrow down their investigation range and focus on the most dangerous hosts. We will add the client and IP detection module in future work.

7 Conclusion

In this paper, we present an intelligent malicious domain detection system named HinDom. HinDom constructs a HIN of clients, domains and IP addresses to model the DNS scene and generates a combined meta-path to analyze the associations among domains. With a meta-path based transductive classification method, HinDom performs well even when the initial label fraction drops to 10%, which reduces the cost of acquiring labeled samples. In our extensive evaluation, we verified HinDom's performance, robustness and practicality. In the real-world deployment, we are able to discover a long-buried mining botnet named MsraMiner and some other malicious domains ahead of the public services. As for further developments, we plan to extend to other types of DNS-related data for more comprehensive semantic information and integrate graph embedding in HinDom to improve efficiency.

Acknowledgment

We thank Hui Zhang, Chenxi Li, Shize Zhang for constructive recommendations on experiments and data processing. Additionally, we appreciate 360netLab, VirusTotal for permissions of their advanced APIs and we thank Information Technology Center of Tsinghua University for authorizing the use of their data in our experiments. This work is supported by the National Key Research and Development Program of China under Grant No.2017YFB0803004.

References

- [1] Amazon Web Services, Inc. AWS | Alexa Top Sites - Up-to-date lists of the top sites on the web. <https://aws.amazon.com/alexa-top-sites/>, 2019. [Online].
- [2] Hyrum S Anderson, Jonathan Woodbridge, and Bobby Filar. Deepdga: Adversarially-tuned domain generation and detection. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, pages 13–21. ACM, 2016.
- [3] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for dns. In *USENIX security symposium*, pages 273–290, 2010.
- [4] Manos Antonakakis, Roberto Perdisci, Wenke Lee, Nikolaos Vasiloglou, and David Dagon. Detecting malware domains at the upper dns hierarchy. In *USENIX security symposium*, volume 11, pages 1–16, 2011.
- [5] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou, Saeed Abu-Nimeh, Wenke Lee, and David Dagon. From throw-away traffic to bots: Detecting the rise of dga-based malware. In *USENIX security symposium*, volume 12, 2012.
- [6] Leyla Bilge, Sevil Sen, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. Exposure: A passive dns analysis service to detect and report malicious domains. *ACM Transactions on Information and System Security (TISSEC)*, 16(4):14, 2014.
- [7] Yizheng Chen, Yacin Nadji, Athanasios Kountouras, Fabian Monrose, Roberto Perdisci, Manos Antonakakis, and Nikolaos Vasiloglou. Practical attacks against graph-based clustering. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1125–1142. ACM, 2017.
- [8] Daiki Chiba, Takeshi Yagi, Mitsuaki Akiyama, Toshiki Shibahara, Takeshi Yada, Tatsuya Mori, and Shigeki Goto. Domainprofiler: Discovering domain names abused in future. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 491–502. IEEE, 2016.
- [9] Google Developers. Google Safe Browsing. <https://developers.google.com/safe-browsing/>, 2019. [Online].
- [10] Dnsdb.info. DNSDB. <https://www.dnsdb.info>, 2019. [Online].
- [11] Yujie Fan, Shifu Hou, Yiming Zhang, Yanfang Ye, and Melih Abdulhayoglu. Gotcha-sly malware!: Scorpion a metagraph2vec based malware detection system. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 253–262. ACM, 2018.
- [12] Schloss Dagstuhl Leibniz Center for Informatics. DNS-BH – Malware Domain Blocklist by RiskAnalytics. <url=https://dblp.uni-trier.de/>, 2019. [Online].

- [13] Tao-yang Fu, Wang-Chien Lee, and Zhen Lei. Hin2vec: Explore meta-paths in heterogeneous information networks for representation learning. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1797–1806. ACM, 2017.
- [14] Jason Geffner. End-to-end analysis of a domain generating algorithm malware family. *Black Hat USA*, 2013, 2013.
- [15] Xiaofei He, Deng Cai, and Partha Niyogi. Laplacian score for feature selection. In *Advances in neural information processing systems*, pages 507–514, 2006.
- [16] 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. ACM, 2017.
- [17] Ming Ji, Yizhou Sun, Marina Danilevsky, Jiawei Han, and Jing Gao. Graph regularized transductive classification on heterogeneous information networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 570–586. Springer, 2010.
- [18] Issa Khalil, Ting Yu, and Bei Guan. Discovering malicious domains through passive dns data graph analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 663–674. ACM, 2016.
- [19] Issa M Khalil, Bei Guan, Mohamed Nabeel, and Ting Yu. A domain is only as good as its buddies: Detecting stealthy malicious domains via graph inference. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 330–341. ACM, 2018.
- [20] Pierre Lison and Vasileios Mavroeidis. Automatic detection of malware-generated domains with recurrent neural models. *arXiv preprint arXiv:1709.07102*, 2017.
- [21] Daiping Liu, Zhou Li, Kun Du, Haining Wang, Baojun Liu, and Haixin Duan. Don’t let one rotten apple spoil the whole barrel: Towards automated detection of shadowed domains. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 537–552. ACM, 2017.
- [22] Chen Luo, Renchu Guan, Zhe Wang, and Chenghua Lin. Hetpathmine: A novel transductive classification algorithm on heterogeneous information networks. In *European Conference on Information Retrieval*, pages 210–221. Springer, 2014.
- [23] Malwaredomainlist.com. MDL. <https://www.malwaredomainlist.com>, 2019. [Online].
- [24] Malwaredomains.com. DNS-BH – Malware Domain Blocklist by RiskAnalytics. <http://www.malwaredomains.com>, 2019. [Online].
- [25] Pratyusa K Manadhata, Sandeep Yadav, Prasad Rao, and William Horne. Detecting malicious domains via graph inference. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2014.
- [26] Passivedns.cn. Sign In-passiveDNS. <https://passivedns.cn>, 2019. [Online].
- [27] Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla. A comprehensive measurement study of domain generating malware. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 263–278, 2016.
- [28] Babak Rahbarinia, Roberto Perdisci, and Manos Antonakakis. Segugio: Efficient behavior-based tracking of malware-control domains in large isp networks. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 403–414. IEEE, 2015.
- [29] Samuel Schüppen, Dominik Teubert, Patrick Herrmann, and Ulrike Meyer. {FANCI}: Feature-based automated nxdomain classification and intelligence. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1165–1181, 2018.
- [30] Jingbo Shang, Meng Qu, Jialu Liu, Lance M Kaplan, Jiawei Han, and Jian Peng. Meta-path guided embedding for similarity search in large-scale heterogeneous information networks. *arXiv preprint arXiv:1610.09769*, 2016.
- [31] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and S Yu Philip. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):17–37, 2017.
- [32] 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.
- [33] Yizhou Sun, Yintao Yu, and Jiawei Han. Ranking-based clustering of heterogeneous information networks with star network schema. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 797–806. ACM, 2009.
- [34] Virustotal.com. Virustotal. <https://www.virustotal.com>, 2019. [Online].

- [35] Mengting Wan, Yunbo Ouyang, Lance Kaplan, and Jiawei Han. Graph regularized meta-path based transductive regression in heterogeneous information network. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 918–926. SIAM, 2015.
- [36] Jonathan Woodbridge, Hyrum S Anderson, Anjum Ahuja, and Daniel Grant. Predicting domain generation algorithms with long short-term memory networks. *arXiv preprint arXiv:1611.00791*, 2016.
- [37] Dengyong Zhou, Olivier Bousquet, Thomas N Lal, Jason Weston, and Bernhard Schölkopf. Learning with local and global consistency. In *Advances in neural information processing systems*, pages 321–328, 2004.
- [38] Futai Zou, Siyu Zhang, Weixiong Rao, and Ping Yi. Detecting malware based on dns graph mining. *International Journal of Distributed Sensor Networks*, 11(10):102687, 2015.

DomainScouter: Understanding the Risks of Deceptive IDNs

Daiki Chiba¹, Ayako Akiyama Hasegawa¹, Takashi Koide¹,
Yuta Sawabe², Shigeki Goto², and Mitsuaki Akiyama¹

¹NTT Secure Platform Laboratories, Tokyo, Japan

²Waseda University, Tokyo, Japan

Abstract

Cyber attackers create domain names that are visually similar to those of legitimate/popular brands by abusing valid internationalized domain names (IDNs). In this work, we systematize such domain names, which we call deceptive IDNs, and understand the risks associated with them. In particular, we propose a new system called DomainScouter to detect various deceptive IDNs and calculate a deceptive IDN score, a new metric indicating the number of users that are likely to be misled by a deceptive IDN. We perform a comprehensive measurement study on the identified deceptive IDNs using over 4.4 million registered IDNs under 570 top level domains (TLDs). The measurement results demonstrate that there are many previously unexplored deceptive IDNs targeting non-English brands or combining other domain squatting methods. Furthermore, we conduct online surveys to examine and highlight vulnerabilities in user perceptions when encountering such IDNs. Finally, we discuss the practical countermeasures that stakeholders can take against deceptive IDNs.

1 Introduction

Domain names are indispensable resources or assets of online service providers on the Internet. Although the Internet was not designed to distinguish borders and languages, domain names were originally written in English only (i.e., using ASCII codes, digits, and hyphens). After some time, internationalized domain names (IDNs) were proposed to enable Internet users to create domain names in their local languages and scripts [25]. Since IDNs were successfully standardized and implemented in 2003, characters in the Unicode Standard can now be used in domain names while maintaining backward compatibility with previously implemented English-based domain names and the domain name system (DNS). The backward compatibility was implemented using the Punycode representation of the Unicode characters with a special prefix (xn--). For example, 例え[.]test in the IDN format is transformed into xn-r8jz45g[.]test in the ASCII-compatible format. IDNs are essential for enabling

the multilingual Internet to serve culturally and linguistically diverse populations.

At the same time, cyber attackers abuse the IDN mechanism to register their domain names for cyber attacks. In fact, cyber attackers create domain names that are visually similar to those of legitimate and popular brands by abusing IDNs [36, 48, 58]. The attackers aim to trick innocent users into falsely recognizing a purposely created misleading domain name as a legitimate brand's domain name by its visual appearance. This type of attack, called an IDN homograph attack, poses a real threat to Internet users. For example, a security researcher used an IDN similar to apple[.]com with a valid SSL certificate to demonstrate a proof-of-concept of an almost complete phishing attack; many users could not distinguish the fake IDN from the genuine one by its appearance in April 2017 [68]. Similarly, another security researcher discovered an IDN homograph attack that used an IDN visually similar to adobe[.]com to distribute a fake flash player with malware [40]. Recently, a researcher reported a new vulnerability in Apple's Safari browser that renders a specific Unicode letter as a normal Latin small "d" in the browser's address bar, which can lead to IDN homograph attacks [56].

In this paper, first, we systematize such visually distorted IDNs, which we call *deceptive IDNs*, to understand the risks associated with them. Unlike the previously reported similar studies [36, 48], the deceptive IDNs in this paper include not only homograph IDNs, wherein some of the characters in English brand domain names are replaced with visually similar characters, but also other types of lookalike IDNs targeting both English and non-English brands comprehensively. On the basis of the systematization, we propose a new system called DOMAINSCOUTER for detecting deceptive IDNs and calculating a *deceptive IDN score* for each IDN. This score is a new metric indicating the number of users that are likely to be misled by a deceptive IDN. The purpose of DOMAINSCOUTER is to score the suspiciousness of an attempt to deceive users on the basis of IDN characteristics. In particular, it is designed to capture distinctive visual characteristics of deceptive IDNs, consider characteristics of targeted legitimate

brand domain names, and use the domain knowledge of both IDNs and targeted domain names.

The contributions of this paper are summarized as follows.

- Propose a new system called **DOMAINSCOUTER** to detect more various types of deceptive IDNs than previously proposed systems and calculate a deceptive IDN score, a new metric indicating the number of users likely to be misled by a deceptive IDN (Sections 3 and 4).
- Perform by far the most comprehensive measurement study on the deceptive IDNs detected by the proposed **DOMAINSCOUTER** using over 4.4 million registered real-world IDNs under 570 top level domains (TLDs) (Section 5).
- Conduct online surveys ($N=838$) to examine vulnerabilities in user perceptions when encountering deceptive IDNs and evaluate that the deceptive IDN score we proposed reflects the tendency of users to be deceived by the attacks. To the best of our knowledge, this is the first user study on deceptive IDNs (Section 6).
- Discuss the practical countermeasures that stakeholders can take against deceptive IDNs (Section 7).

2 Systematization of Deceptive IDNs

We systematize all possible deceptive IDNs targeting users' visual perception. We focus on IDNs that look similar to those of legitimate brands to deceive users to take actions such as clicking links in spam emails and inputting personal information on phishing sites. To the best of our knowledge, this study is the first attempt in security research to systematize deceptive IDNs.

First, we divide deceptive IDNs into those targeting English brands and those targeting non-English brands since these two categories have quite different characteristics. Since English is the world's standard language and the Internet was originally available only in ASCII and English character sets, most globally popular brands have their websites and domain names in English. At the same time, many local brands in non-English-speaking communities have started to use their native languages and characters to create domain names. Thus, English and non-English brand names should be treated differently, especially when researching the Internet-related topics such as domain names. Whereas previous studies have focused only on deceptive IDNs targeting English brands [36, 48], IDNs targeting non-English brands have not been studied well so far.

Second, we reveal that there are three types of deceptive IDNs in theory: combosquatting (combining brand name with keywords) (*combo*), homograph (*homo*), and homograph+combosquatting (*homocombo*) IDNs. We define a combo IDN as an IDN that combines a brand domain name with some additional English or non-English phrases. Kintis

et al. [30] conducted the first study to reveal English-based combosquatting domains; our paper extends this concept to IDNs. The homo IDN is an IDN wherein some of the characters of a brand domain name are replaced with characters that are visually similar. Some previous studies analyzed the characteristics of homo IDNs in 2018 [36, 48]. The homocombo IDN is defined as an IDN that does not match the above combo or homo definitions exactly but has characteristics of both the combo and homo IDNs; e.g., an IDN containing words similar to a legitimate brand name and some additional phrases. Our paper is the first to define, measure, and analyze the homocombo IDNs. Note that we do not include any non-IDN squatting domains such as typosquatting (typographical errors) [1, 29, 55, 62] or bitsquatting (accidental bit flips) [41] since our paper focuses on user misbehavior caused by deceptive IDNs.

On the basis of the above conditions, we consider six types of IDN-based attacks in this paper. In particular, when considering English brands (e.g., `example[.]test`) as targets, the brand could be targeted by combo IDNs (*eng-combo*; e.g., `exampleログイン[.]test`), homo IDNs (*eng-homo*; e.g., `êxämplê[.]test`), and homocombo IDNs (*eng-homocombo*; e.g., `êxämplêログイン[.]test`). When considering non-English brands (e.g., `例え[.]test`), the brand could be targeted by combo IDNs (*noneng-combo*; e.g., `例えログイン[.]test`), homo IDNs (*noneng-homo*; e.g., `イ列え[.]test`), and homocombo IDNs (*noneng-homocombo*; e.g., `イ列えログイン[.]test`).

In terms of creating/registering deceptive IDNs (especially combo and homocombo), attackers are free to use one or more arbitrary words as prefixes or postfixes of brands. That is, similar to non-IDN combosquatting [30], a deceptive IDN lacks a generative model. Therefore, we cannot rely on the generative model but need to design a system to grasp the nature of deceptive IDNs.

3 DomainScouter System

We propose a new system called **DOMAINSCOUTER** to detect the six types of deceptive IDNs (*eng-combo*, *eng-homo*, *eng-homocombo*, *noneng-combo*, *noneng-homo*, and *noneng-homocombo*) defined in Section 2. Figure 1 shows an overview of **DOMAINSCOUTER**. The inputs to **DOMAINSCOUTER** are registered IDNs and selected brand domains. **DOMAINSCOUTER** automatically detects deceptive IDNs on the basis of various features focusing on visual similarities, brand information, and TLD characteristics. The outputs of **DOMAINSCOUTER** are detected deceptive IDNs, targeted brands, and deceptive IDN scores for each IDN. The deceptive IDN score is a new metric indicating the number of users likely to be deceived when encountering a deceptive IDN. **DOMAINSCOUTER** consists of five steps: IDN extraction, brand selection, image generation, feature extraction, and score calculation. The following sections explain these steps in turn.

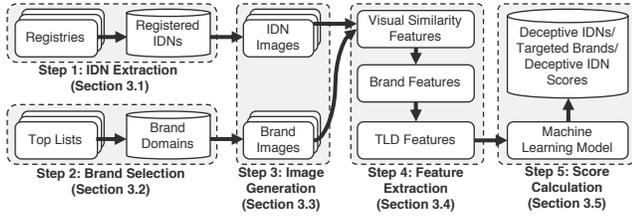


Figure 1: System Overview

3.1 Step 1: IDN Extraction

The first step involves extracting already existing IDNs from the domain registry databases. Unfortunately, since each domain registry corresponding to a TLD has been operated separately, there is no single (unified) database with all registered domains freely available for researchers. Thus, we need to collect registered domain names from more than 1,400 TLD registries to study all IDNs that exist in the world.

In general, TLDs can be divided into two categories: generic TLDs (gTLDs) and country-code TLDs (ccTLDs) [21]. In this paper, we further separate gTLDs and ccTLDs to understand the relationship between deceptive IDNs and TLDs’ characteristics. We separate gTLDs into three types: *legacy gTLD*, *new gTLD*, and *new IDN gTLD*. The legacy gTLD consists of 22 TLDs (.aero, .asia, .biz, .cat, .com, .coop, .edu, .gov, .info, .int, .jobs, .mil, .mobi, .museum, .name, .net, .org, .post, .pro, .tel, .travel, and .xxx) introduced before the new gTLD program started by ICANN in 2013 [24, 31]. The new gTLD is composed of 1,042 non-IDN TLDs (e.g., .top, .xyz, and .loan) introduced by the ICANN’s program. The new IDN gTLD is made up of 84 IDN TLDs (e.g., .网址 (.xn-ses554g), .在线 (.xn-3ds443g), and .pyc (.xn-placf)) also used by the program, especially for allowing the entire domain names to be represented in a local language and characters. Furthermore, we separate ccTLDs into two types: *legacy ccTLD* and *new IDN ccTLD*. The legacy ccTLD is composed of 245 TLDs (e.g., .cn, .jp, and .uk) that were two-letter codes representing countries listed by the ISO 3166-1 standard [23]. The new IDN ccTLD consists of 42 IDN TLDs (e.g., .新加坡 (.xn-yfro4i67o), .한국 (.xn-3e0b707e), and .ph (.xn-plai)) registered after 2009 [22].

To collect and extract all registered IDNs under the above-mentioned TLD types, we leveraged the commercial WHOIS database [64] containing information about nearly all domains as of May 2018. Table 1 shows the breakdown of our collected dataset. In total, we processed over 294 million domains (including IDNs and non-IDNs) under 1,435 TLDs. From all domains, we extracted over 4.4 million IDNs under 570 TLDs. Note that the remaining 865 TLDs have no registered IDNs.

3.2 Step 2: Brand Selection

The second step of DOMAINSCOUTER is selecting brand domains targeted by deceptive IDNs. We need to select both

Table 1: Domain Dataset

TLD type	# TLDs (IDNs)	# TLDs (Total)	# Domains (IDNs)	# Domains (Total)
Legacy gTLD	13	22	1,482,709	171,016,371
New gTLD	328	1,042	424,024	21,523,232
New IDN gTLD	84	84	599,559	599,559
Legacy ccTLD	103	245	988,963	100,398,597
New IDN ccTLD	42	42	931,062	931,062
Total	570	1,435	4,426,317	294,468,821

English and non-English brands since our paper focuses on deceptive IDNs targeting both types of brands as stated in Section 2.

For English brands, we leveraged three major top domain lists (Alexa [2], Umbrella [11], and Majestic [38] top 1 million lists) that record representative Internet domains. As discussed in recent studies [35, 50], each list has its own ranking mechanism; thus, we used the three major lists in the Internet measurement community to collect English brands in an unbiased way. We extracted the top 1,000 domains from each list, removed redundant domains, and finally collected 2,310 domains in total.

For non-English brands, we used the same three top domain lists as for English brands. Since there are far fewer non-English brand domains than English ones, we extracted non-English IDNs from the top 1 million domains in each list, removed redundant domains, and finally collected 4,774 domains in total. Note that we excluded some low-ranked malicious domains accidentally listed in the top lists by referring to multiple domain blacklists such as VirusTotal [61], hpHosts [20], Google Safe Browsing [16], and Symantec DeepSight [54].

3.3 Step 3: Image Generation

The third step of DOMAINSCOUTER is generating images from both registered IDNs (step 1) and brand domains (step 2) for the following calculation of visual similarities in step 4. In particular, we generate three types of images for each domain in both registered IDNs and brand domains. We select the default font used in the address bar of Google Chrome in Windows 10 since the browser/OS has the biggest market share [52].

RAW images. The first type is a *raw* image, simply generated from each domain’s string without any modifications. RAW is used for specifying a very similar combination of a deceptive IDN (e.g., eng-homo and noneng-homo) and a brand domain as a whole.

PSR images. The second type is a public suffix-removed (PSR) image generated from substrings excluding a public suffix [39] from a domain name string. A public suffix consists of strings in domain names that cannot be controlled by individual Internet users [9]. For example, in the case of PSR images, example is extracted from both example[.]com and example[.]co[.]jp since .com and .co.jp are in public suffixes. PSR images can help distinguish deceptive IDNs that have different public suffixes from targeted brand domains

Table 2: List of Features

Type	No.	Feature	Importance
Visual Similarity	1	Max of SSIM indexes between RAW images	0.123
	2	Max of SSIM indexes between PSR images	0.158
	3	Max of SSIM indexes between WS images	0.391
Brand (RAW)	4	Alexa rank of identified RAW brand domain	0.019
	5	Umbrella rank of identified RAW brand domain	0.017
	6	Majestic rank of identified RAW brand domain	0.012
Brand (PSR)	7	Alexa rank of identified PSR brand domain	0.012
	8	Umbrella rank of identified PSR brand domain	0.004
	9	Majestic rank of identified PSR brand domain	0.009
Brand (WS)	10	Alexa rank of identified WS brand domain	0.041
	11	Umbrella rank of identified WS brand domain	0.046
	12	Majestic rank of identified WS brand domain	0.040
TLD	13	TLD type of Input IDN	0.085
	14	TLD type of RAW brand domain	0.024
	15	TLD type of PSR brand domain	0.006
	16	TLD type of WS brand domain	0.015

since attackers do not necessarily use the same public suffixes of the brand domains [48].

WS images. The third type is a word segmented (WS) image. A WS image is generated by applying word segmentation algorithms to a domain name string. For example, `example` and `テスト` are segmented from `exampleテスト[.]com`. We use the polyglot [44] implementation for multilingual word segmentation. The intuition behind generating WS images is to help detect combosquatting-based deceptive IDNs such as `eng-combo`, `eng-homocombo`, `noneng-combo`, and `noneng-homocombo`.

3.4 Step 4: Feature Extraction

The fourth step of DOMAINSOUTER is extracting features from registered IDNs (step 1), brand domains (step 2), and their corresponding images (step 3). This step is intended to design features that can detect the six types of deceptive IDNs defined in Section 2. In particular, we use three types of features: visual similarity, brand, and TLD features.

Visual similarity features. The visual similarity features, Nos. 1–3 listed in Table 2, are designed to grasp the most distinguishing characteristics of a deceptive IDN, the IDN’s appearance. In other words, these three features are used to measure the extent to which an IDN can deceive users. We utilize image similarity between registered IDNs and brand domains as the visual similarity features. To measure similarity between two images, we use the Structural SIMilarity (SSIM) index [63] since it is reported to achieve the best performance when detecting one type of the deceptive IDNs (`eng-homo`) [36]. For our prototype implementation, we used `pyssim` [46], a python module for computing the SSIM index. The SSIM index ranges between 0.0 (non-identical) and 1.0 (perfectly identical). As explained in Section 3.3, we prepare images of three different types (RAW, PSR, and WS) to detect various deceptive IDNs; accordingly, we calculate the SSIM index for pairs of images of the same type. We use the maximums of the SSIM indexes between RAW, PSR, and WS images as features No. 1, 2, and 3, respectively. We identify the brand domain with the highest SSIM indexes as the targeted brand domain corresponding to the input IDN.

Brand features. The brand features, Nos. 4–12 listed in Ta-

ble 2, are designed to consider characteristics of targeted brand domains. We hypothesize that more popular domains are targeted to create deceptive IDNs. Thus, we use the rank information in the three top lists (Alexa [2], Umbrella [11], and Majestic [38]) as our brand features. The reason for using multiple top lists is to measure popularity from several ranking mechanisms in an unbiased way. We refer to the Alexa, Umbrella, and Majestic ranks of the targeted brand domain identified on the basis of the visual similarity features as mentioned above in RAW, PSR, and WS images as features Nos. 4–6, 7–9, and 10–12, respectively.

TLD features. The TLD features, Nos. 13–16 listed in Table 2, are designed to use domain names’ own characteristics of both input IDNs and targeted brand domains. We introduce these features since our analysis reveals that the usage of TLDs has changed dramatically in recent years, and deceptive IDNs do not always use the same TLD as the targeted brand domains. We use the TLD types defined in Section 3.1 (e.g., legacy gTLD, new gTLD, new IDN gTLD, legacy ccTLD, and new IDN ccTLD) as the TLD features for the input IDN (No. 13) and the targeted brand domain based on RAW (No. 14), PSR (No. 15), and WS (No. 16) images.

3.5 Step 5: Score Calculation

The fifth step of DOMAINSOUTER is calculating the deceptive IDN score, which is the estimated probability of the user being deceived by the corresponding input IDN. We use a supervised machine learning approach to calculate the score. The input of this step consists of the input IDN with the features listed in Table 2. We use one-hot encoding for categorical features (Nos. 13–16). Supervised machine learning is generally composed of two phases: training and testing. The training phase generates a machine learning model from training data that includes extracted features and labels. For labeling, we hypothesize that some deceptive IDNs have already been used for phishing or social engineering attacks. Thus, we rely on multiple blacklists that have phishing or social engineering categories and carefully label the input IDN *deceptive* or *non-deceptive*. Note that our aim is not labeling many known deceptive IDNs but labeling reliable deceptive IDNs for estimating the scores for unlabeled IDNs. In the testing phase, the model generated in the training phase is used to calculate the probabilities of input IDNs being deceptive IDNs. We define these probabilities as the deceptive IDN scores. The higher the score, the more likely the user is to be deceived by the IDN. Consequently, this step outputs detected deceptive IDNs, their targeting brand domains, and the deceptive IDN scores.

Among many traditional and deep learning algorithms, we select Random Forest [8] for three reasons. First, Random Forest has good interpretability, i.e., it makes clear how features contribute to the result and how they are treated. Second, the parameters of Random Forest include the number of decision

trees to employ and the features considered in each decision tree, which makes the model easy to tune. Finally, in our preliminary experiments, Random Forest outperformed other popular algorithms such as Logistic Regression, Naïve Bayes, Decision Tree, and Support Vector Machine. In Random Forest, the probability or deceptive IDN score is calculated by averaging results of each decision tree. The higher the number of decision trees predicted to be deceptive, the higher the deceptive IDN score.

3.6 Limitation

DOMAINSCOUTER has two limitations. First, it does not aim to detect various kinds of malicious domain names but only deceptive IDNs that may lead to user misbehavior. Thus, a deceptive IDN is not always used for specific malicious attacks (e.g., phishing, social engineering, and malware). However, identifying deceptive IDNs itself provides incentives for various stakeholders as discussed later in Section 7. There are many previous systems aiming at detecting malicious domain names in terms of the lexical characteristics [37, 55, 67], the relationship between domains and IP addresses [3, 10, 32], and the behavior of DNS queries [4, 5, 7]. Our system complements these systems. In particular, we can combine the systems to achieve better detection coverage.

The second limitation is in the coverage of non-English brands in step 2. In particular, we selected non-English brands on the basis of the top lists; however, there could be more non-English brands for each country, region, and language. We will explore other sources such as registered trademarks or search engine results for each country in our future work.

4 Evaluation

In this section, we show the results of comparing our system DOMAINSCOUTER with those proposed in previous works in terms of system properties and detection performance.

4.1 Comparison of Properties

We compared the properties of DOMAINSCOUTER and those of two previous systems [36, 48] from four perspectives. Table 3 summarizes the results.

Dataset. When comparing datasets used in each study, there are clear gaps between our system and the other two. In particular, our system contains 570 studied TLDs, whereas that of Liu et al. [36] contains only 56. No description regarding TLDs is provided by Sawabe et al. [48]. Furthermore, our system contains many more IDNs than the two previous systems: 3 times more than that of Liu et al. [36] and 2.3 times more than that of Sawabe et al. [48]. To the best of our knowledge, our domain dataset that includes both gTLDs and ccTLDs is the most comprehensive dataset ever used in security research.

Targeted Brand. In terms of the targeted brands used in each study, DOMAINSCOUTER uses both English and non-English brand domains, and the number of these domains is much

Table 3: Results of Comparing Properties

		Our System	Liu et al. [36]	Sawabe et al. [48]
Dataset	# TLDs (IDNs)	570	56	-
	# Domains (IDNs)	4,426,317	1,472,836	1,928,711
Targeted Brand	# Domains (English)	2,310	1,000	1,000
	# Domains (Non-English)	4,774	0	0
Deceptive IDN	Combo	●	●	○
	Homo	●	○	○
	Homocombo	●	○	○
Method	Visual Similarities	●	●	●
	Brand Features	●	○	○
	TLD Features	●	○	○

●: Fully Covered, ●: Partially Covered, ○: Not Covered

bigger than that of the Liu et al. [36] and Sawabe et al. [48] systems.

Deceptive IDN. DOMAINSCOUTER focuses on various deceptive IDNs (eng-combo, eng-homo, eng-homocombo, noneng-combo, noneng-homo, and noneng-homocombo), whereas Liu et al. [36] studied only eng-homo and a part of eng-combo IDNs, and Sawabe et al. [48] detected only eng-homo IDNs.

Method. Liu et al. [36] used visual similarities (the SSIM index) between IDNs and brand domains to detect eng-homo IDNs targeting the Alexa Top 1,000 brands. Sawabe et al. [48] calculated visual similarities between non-ASCII and ASCII characters using optical character recognition (OCR) to detect eng-homo IDNs. However, both methods need to tune the thresholds of either the SSIM index or OCR manually, which tends to cause false positives and false negatives, and do not consider how popular the targeted brand domain is. In addition, Liu et al. did not focus on eng-homo IDNs between different TLDs (e.g., `example[.]com` and `êxâmplê[.]test`).

To solve the above problems, as stated in Section 3.4, DOMAINSCOUTER utilizes not only multiple visual similarity features but also targeted brand ranking and TLD features and applies a machine learning approach to eliminate tuning thresholds for visual similarity features.

4.2 Comparison of Detection Performance

We compared the deceptive IDN detection performance of DOMAINSCOUTER with that of the previously proposed systems [36, 48]. First, we describe the experimental setups in the other two systems and our system. Then, we illustrate the comparison results using real registered IDNs.

Setups of the Previous Systems. We replicated the previously proposed systems on the basis of their descriptions provided in the corresponding papers [36, 48] since the systems are not open-source. For the Liu et al. [36] system, we needed to set a threshold for the SSIM index to detect eng-homo IDNs. The original paper set the threshold to 0.95. However, in our re-implemented system, the 0.95 threshold caused non-negligible false positives, which may be due to the differences in the font and image settings between the original system and our re-implemented one. We manually verified the SSIM index results to determine the threshold of 0.99, which caused only few false positives. For the Sawabe et al. [48] system, we used the mappings between non-ASCII and correspond-

ing similar ASCII characters kindly provided by Sawabe et al. themselves [48] to re-implement the detection method of eng-homo IDNs. To match the brand domains employed in the previous works, we used all English brand domains shown in Section 3.2 for fair evaluation, even though the original papers used only the top 1,000 brand domains.

Setup of DOMAINSCOUTER. Section 3 describes the implementation of our system, DOMAINSCOUTER. As stated in Section 3.5, we need to set up a labeled training dataset. In our evaluation, we used 10,000 labeled IDNs consisted of 242 deceptive (positive) and 9,758 non-deceptive (negative) IDNs for building our machine learning model. The positive IDNs were labeled by referring to the latest three blacklists (hpHosts [20], Google Safe Browsing [16], and Symantec DeepSight [54]) as of November 2018 and manually verified by the authors. The 242 positive IDNs are composed of only eng-homo deceptive IDNs since even the latest blacklists do not cover other types of deceptive IDNs. However, we design our proposed features to grasp the nature of various deceptive IDNs, thus DOMAINSCOUTER can identify other types of deceptive IDNs other than eng-homo. The negative IDNs were randomly sampled from the IDNs shown in Table 1 and manually verified by the authors.

We performed 10-fold cross-validation (CV) on the training dataset and achieved a true positive rate of 0.981, a true negative rate of 0.998, a false positive rate of 0.002, a false negative rate of 0.019, and an F-measure of 0.972 on average. Regarding the two parameters in Random Forest, we set the number of decision trees as 100 and the number of sampled features in each individual decision tree as 6 on the basis of the best results in our preliminary experiments. Note that, as explained in Section 3.6, DOMAINSCOUTER does not aim to detect malicious IDNs but only deceptive IDNs. Thus, the *positive* does not mean *malicious* but *deceptive*. Similarly, *negative* does not mean *legitimate/benign* but *non-deceptive*. This has been a typical evaluation setting regarding detecting deceptive IDNs (e.g., eng-homo). Similar to ours, previous studies [36, 48] did not provide true positive/negative rates in terms of detecting *malicious* IDNs since they focused on detecting eng-homo IDNs.

The last column of Table 2 shows relative feature importance of all features. The higher the importance score, the more the feature contributed to the correct detection. The results demonstrated that the three visual similarity features (Nos. 1–3) are more effective than the other features. In particular, the visual similarity based on word segmented images (feature No.3) appeared to contribute to the correct detection the most. The remaining proposed features (Nos. 4–16) were confirmed to contribute to detecting deceptive IDNs as well.

Detection Performance. Here, we compare the detection performance of the three systems. The input IDNs for each system were the same 4,426,317 IDNs described in Table 1.

Unfortunately, there is no ground truth to label all IDNs. Thus, we used the re-implemented previous systems and the

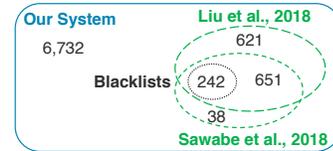


Figure 2: Venn Diagram of Detected Deceptive IDNs

trained DOMAINSCOUTER, which proved to be accurate in the CV evaluation, to explore unknown deceptive IDNs in the dataset. Of course, there could be unavoidable false negatives or missed deceptive IDNs. We manually excluded false positives or falsely detected non-deceptive IDNs from the results of the three systems.

We did not exclude the 242 positive IDNs used for the training dataset of DOMAINSCOUTER from the input IDN in this evaluation for two reasons. One is that the goal of our paper is not just to compare the detection performance but also to conduct a comprehensive measurement study of deceptive IDNs (shown later in Section 5). The other is all the 242 positive IDNs were confirmed to be easily detected by the three systems since they were easily identifiable eng-homo deceptive IDNs.

Figure 2 is a Venn diagram showing intersections of deceptive IDNs detected by the three systems and the 242 positive IDNs labeled using blacklists. The Liu et al. system detected 1,514 deceptive IDNs (=621+651+242) and the Sawabe et al. system detected 931 (=38+651+242). Our analysis revealed that the difference between the coverage achieved by the Liu et al. and Sawabe et al. systems originated from the difference in handling the input IDN by each system: the Liu et al. system handled an IDN string as one image, whereas the Sawabe et al. system handled each non-ASCII character contained in an IDN.

Surprisingly, DOMAINSCOUTER fully covered the 1,552 (=621+38+651+242) deceptive IDNs detected by the two previous systems. Moreover, DOMAINSCOUTER detected 6,732 further deceptive IDNs that were not detected by the two systems. The extra detected deceptive IDNs mainly consisted of our new targets such as eng-combo, eng-homocombo, noneng-combo, and noneng-homocombo. The results of the 8,284 IDNs detected in total are explained in the next section.

5 Measurement Study

So far, we have evaluated the detection performance of DOMAINSCOUTER compared with those of the two previously proposed systems. This section focuses on the 8,284 deceptive IDNs detected by DOMAINSCOUTER. To the best of our knowledge, this is the most comprehensive study in terms of the numbers of both the input IDNs (more than 4.4 million registered IDNs under 570 TLDs as shown in Table 1) and the detected deceptive IDNs. In the following sections, we describe our measurement results in terms of the characteristics of deceptive IDNs, the impacts caused by deceptive IDNs, and the brand protection of deceptive IDNs.

Table 4: Breakdown of Detected Deceptive IDNs

Type	# IDNs
eng-combo	368
eng-homo	1,547
eng-homocombo	3,697
noneng-combo	144
noneng-homocombo	2,528
Total	8,284

5.1 Characteristics of Deceptive IDNs

Deceptive Types. We begin by investigating the types of deceptive IDNs found in the registered IDNs as of May 2018. The identified deceptive IDNs were grouped into the defined types on the basis of the information we obtained when extracting our proposed features, i.e., identified targeted brands (eng/noneng) and which SSIM index of images (RAW/WS/PSR) is the highest. Table 4 provides a breakdown of the detected deceptive IDNs. Our system found 368 eng-combo, 1,547 eng-homo, 3,697 eng-homocombo, 144 noneng-combo, and 2,528 noneng-homocombo IDNs. As explained in Section 2, some eng-homo IDNs were already analyzed in the previous studies [36, 48]. We successfully revealed that there were many deceptive IDNs other than eng-homo IDNs, which were found in the research literature for the first time. We defined a noneng-homo IDNs; however, our system did not detect any noneng-homo IDNs that targeted our selected non-English brand domains from the input IDNs.

Targeted Brands. Next, we focused on the targeted brands among the detected deceptive IDNs. Table 5 lists the 10 most targeted English brands, along with their Alexa ranks, among the detected deceptive IDNs. The results highlight three major outcomes. First, more popular brand domains (i.e., those with higher Alexa ranks) are targeted for creating deceptive IDNs as hypothesized in Section 3.4. Second, all websites of the top 10 targeted brands offer user accounts and user login functions. A possible explanation for this is attackers targeted these websites to obtain sensitive information such as user IDs and passwords via phishing or social engineering attacks. Finally, DOMAINSOUTER successfully detected many eng-combo and eng-homocombo IDNs that were defined in this paper for the first time. For example, we found that Amazon was targeted the most (56 eng-combo IDNs, 64 eng-homo IDNs, and 843 eng-homocombo IDNs).

Table 6 lists the 10 most targeted non-English brands, along with their Alexa rankings and English meanings. The result proves the existence of many noneng-combo and noneng-homocombo IDNs that are defined and studied in this paper for the first time. Noneng-combo IDNs were found for only one target brand in the top 10 brands. We found many noneng-homocombo IDNs that targeted place names (e.g., Austria, Pattaya, and Antalya) and common words (e.g., sport, flights, and weather) in non-English languages.

Creation Dates. We examined when the detected deceptive IDNs were registered and started to be used. To this end, we leveraged the WHOIS database [64] explained in Section 3.1. From the WHOIS database, we extracted the dates

Table 5: Top 10 Targeted English Brands

Target	Alexa	eng-combo	eng-homo	eng-homocombo	Total
amazon[.]com	10	56	64	843	963
hotel[.]com	622	2	13	457	472
google[.]com	1	14	122	100	236
apple[.]com	71	20	59	129	208
facebook[.]com	3	18	78	58	154
target[.]com	410	0	6	135	141
youtube[.]com	2	23	37	61	121
bet365[.]com	274	79	0	22	101
office[.]com	38	5	6	84	95
yahoo[.]com	7	7	18	64	89

Table 6: Top 10 Targeted Non-English Brands

Target	Alexa	Meaning	noneng-combo	noneng-homocombo	Total
xn-sterreich-z7a[.]at	487,222	Austria	0	1,032	1,032
xn-nlabehi[.]kz	479,087	sport	0	307	307
xn-o3cnn2dh[.]ws	977,559	Pattaya	0	159	159
xn-flge-1ra[.]de	199,379	flights	0	155	155
xn-80ahnhrfk[.]shop	419,929	presents	0	42	42
xn-mto-bmab[.]fr	58,899	weather	42	0	42
xn-72c0ao2e4bzd[.]com	475,666	cash	0	28	28
xn-80abmi5aecftcl4j[.]su	459,704	security	0	26	26
xn-90acjmnclhybf[.]su	900,952	ad	0	23	23
xn-hgbkak5kj5a[.]net	234,297	Antalya	0	23	23

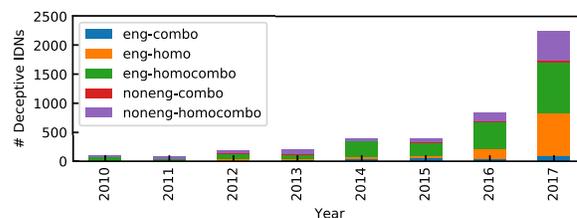


Figure 3: Number of Detected Deceptive IDNs

of registration corresponding to deceptive IDNs. Due to some limitations of the WHOIS dataset (e.g., dates of registration were not provided in some registries), we were able to extract the dates for only 62% (=5,176 / 8,284) of the detected deceptive IDNs.

Figure 3 illustrates the number of deceptive IDNs in each year by the deceptive type. The results revealed two major facts about deceptive IDNs. First, the number of deceptive IDNs increases year by year. Second, many deceptive IDNs that are newly defined in this paper (e.g., eng-combo, eng-homocombo, noneng-combo, and noneng-homocombo) were registered after 2014.

5.2 Impacts of Deceptive IDNs

Accesses. To understand the impacts of the detected deceptive IDNs, we investigate how many accesses or queries to the detected deceptive IDNs were observed over time. To this end, we leveraged the passive DNS database (DNSDB) [13] covering the period from 6-24-2010 to 9-19-2018. The DNSDB enabled us to investigate statistics of DNS queries to the deceptive IDNs such as the dates of first- and last-seen queries and the number of queries. Note that, because the provider could not identify specific users from aggregated DNS queries, the DNSDB data inevitably counted queries from victims, attackers, and security researchers. Table 7 lists the total number of DNS queries to each type of deceptive IDNs. For the deceptive IDNs targeting English brands, 1,547 eng-homo IDNs were queried over 1 million times in total. For those targeting non-English brands, 2,528 noneng-homocombo IDNs were

Table 7: Accesses to Deceptive IDNs

	# Deceptive IDNs	Sum of Queries
eng-combo	368	226,546
eng-homo	1,547	1,019,613
eng-homocombo	3,697	737,696
noneng-combo	144	317,043
noneng-homocombo	2,528	1,440,388
Total	8,284	3,741,286

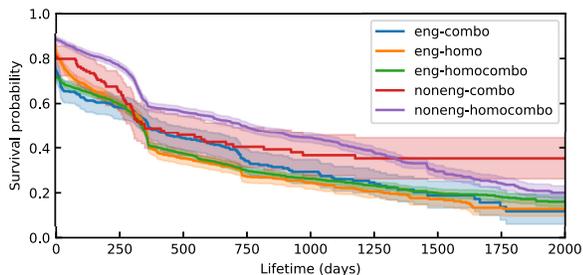


Figure 4: Lifetime of Deceptive IDNs

queried over 1.4 million times in total. These results show that all types of deceptive IDNs accumulated a non-negligible number of accesses over time.

Lifetime. Next, we focus on the lifetime of the detected deceptive IDNs. In this paper, we define the lifetime of deceptive IDNs as the period from the first-seen to the last-seen DNS queries on the basis of the DNSDB data. To analyze the lifetime, we used a survival analysis based on the Kaplan-Meier estimator [28], commonly used to estimate lifespan in cyber security [15, 33, 42]. Figure 4 shows the results of the survival analysis. The x-axis is the lifetime (in days), while the y-axis shows the survival probability, meaning the probability that the deceptive IDN remains after the elapsed number of days. From the figure, we can see that the characteristics of the deceptive IDNs targeting English brands and those targeting non-English brands are different. In particular, eng-combo, eng-homo, and eng-homocombo IDNs have shorter survival probability than noneng-combo and noneng-homocombo IDNs.

5.3 Brand Protection

So far, we have shed light on the characteristics of the detected deceptive IDNs overall. This section focuses on the deceptive IDNs that are now protected by their legitimate domain owners or rightsholders. In particular, we investigated each detected deceptive IDN to identify whether it is protected by its targeted brand’s owner. To this end, we used the WHOIS dataset to extract registrant emails of both the deceptive IDN and the targeted brand domain. In this work, a deceptive IDN is considered to be protected if both emails are the same and the domain part of the email (e.g., @example[.]com) is identical to the targeted brand domain (e.g., example[.]com). This identification process has two limitations. One is the process does not work when an email address is not properly extracted from the WHOIS dataset. The other is the process cannot properly identify a protected deceptive IDN if the legitimate

Table 8: Top 10 Protected Brands

Brand Domain	Alexa	# Protected	# Detected	Protective Ratio
amazon[.]com	10	42	963	4.4%
google[.]com	1	35	236	14.8%
gmail[.]com	536	18	81	22.2%
skype[.]com	456	17	56	30.4%
android[.]com	990	16	45	35.6%
blogger[.]com	299	16	26	61.5%
bet365[.]com	274	15	101	14.9%
cloudflare[.]com	256	14	16	87.5%
youtube[.]com	2	14	121	11.6%
symantec[.]com	310	14	16	87.5%

domain owner uses different email addresses for the brand domain and its deceptive IDN, or if the owner uses a WHOIS privacy protection service to hide their email addresses.

Using the above identification processes, we revealed that only 3.8% (=316 / 8,284) of the detected deceptive IDNs were protected by their targeted brand owners. Table 8 lists the top 10 protected brands; it contains the Alexa rank, the number of protected deceptive IDNs, the number of all detected deceptive IDNs, and the protective ratio. From the table, one can derive two noteworthy facts regarding brand protection. One is no brand domain in the top 10 or the world’s most popular Internet companies protected themselves from all of its corresponding detected deceptive IDNs. This strongly indicated that the deceptive IDN problem is difficult for one company to solve by itself. The other fact is only the few companies offering Internet security services (e.g., Cloudflare and Symantec) protected themselves from the deceptive IDNs more than other companies did.

6 User Study

The attacks that use deceptive IDNs target the perceptions of the users accessing websites. In this section, we examine whether the deceptive IDN score we proposed reflects the tendency of users to be deceived by the attacks. Understanding the impact of the attacks on users helps stakeholders to discuss more practical countermeasures. We conducted two separate online surveys on Amazon Mechanical Turk (MTurk): *User Study 1* to investigate users’ knowledge of IDNs, and *User Study 2* to examine the extent to which users are deceived by deceptive IDNs. Our Institutional Review Board (IRB) approved both surveys. Participants were limited to U.S. residents with an approval rating over 97% and more than 50 tasks approved. We conducted these surveys in November 2018.

6.1 User Study 1

The first survey was designed to ask participants about their demographics and knowledge of IDNs.

Method. The survey consisted of 12 closed-ended questions. We asked the participants about the characters used in domain names. This was a multiple-choice question with the following options: English (Upper case), English (Lower case), digit, hyphen, punctuation, Cyrillic, Greek, Chinese, Japanese, and Korean. IDNs can contain all these characters except for punctuation.

Table 9: Participants’ Misunderstanding of the Characters That Can be Used in Domain Names

	# Participants (all)	# Participants (Computer Eng. / IT Pro.)
Correct Answer	20 (5.5%)	7 (13.5%)
Incorrect Answer	344 (94.5%)	45 (86.5%)
Total	364 (100.0%)	52 (100.0%)

The median time to complete the survey was 3.2 minutes, and we compensated the participants \$0.50 each. After removing 15 participants who gave incomplete or careless answers, we analyzed the remaining 364 participants. 61.0% of the participants were male, and their ages ranged from 19 to 71, with a median of 33 (mean 36.3). Our sample had a wide range of education levels (from high school to graduate degree) and various occupations.

Results. Each language other than English was selected by only one-fourth of the participants at most, whereas English, hyphen, and digit were selected by over a half of the participants. As shown in Table 9, a small number, only 5.5% (=20 / 364), of the participants knew enough about IDNs (i.e., they selected all choices except for punctuation). Only 11.3% (=41 / 364) of the participants seemed to have *some* knowledge about IDNs, i.e., they selected some languages other than English and did not select punctuation. Surprisingly, only 13.5% of the computer engineers or IT professionals answered the question correctly.

In summary, the majority of the participants did not know enough about IDNs, even those engaged in IT-related occupations.

6.2 User Study 2

In the second survey, we aimed to examine the extent to which users are deceived by attacks employing deceptive IDNs.

Method. The survey consisted of 18 closed-ended questions regarding users’ demographics and visual perception of deceptive IDNs.

To measure how many users are not aware of the deceptive IDNs that disguise domains of popular online services, we prepared 70 actual deceptive IDNs for seven popular brands (online services): Google, YouTube, Facebook, Amazon, Twitter, Instagram, and PayPal. We prepared five high-scoring deceptive IDNs (with the score of 1.0) and five low-scoring deceptive IDNs (with the scores ranging from 0.06 to 0.56) for each target brand.

After demographic questions, the participants were first asked which services they used more than once a month. The list of the seven popular brands mentioned above was used to formulate this question. After a few dummy questions, we then gave the participant a deceptive question, asking “Have you ever visited [SERVICE].com?” as a closed-ended question, which could be answered with “yes” or “no.” Note that [SERVICE].com was actually replaced by a deceptive IDN in this question. For example, `example[.]test` would be used instead of `example[.]test`. The displayed deceptive IDNs

Table 10: The Ratio of the Participants Who Were Aware of Deceptive IDNs. We Prepared Five High-scoring Deceptive IDNs and Five Low-scoring Deceptive IDNs for Each Brand

Brand (Score)	Score			# Potential Victims			# Participants*			Insensible Rate		
	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
Google (H)	1.0	1.0	1.0	40	45	42.8	41	47	43.8	0.96	1.0	0.98
Google (L)	0.07	0.40	0.26	23	41	35.0	46	50	47.2	0.50	0.85	0.74
YouTube (H)	1.0	1.0	1.0	34	42	38.6	40	48	44.8	0.79	0.93	0.86
YouTube (L)	0.28	0.40	0.35	29	42	35.8	40	46	43.6	0.71	0.91	0.82
Facebook (H)	1.0	1.0	1.0	30	37	33.2	33	39	36.4	0.82	0.95	0.91
Facebook (L)	0.11	0.40	0.26	29	36	31.4	36	41	38.6	0.76	0.90	0.81
Amazon (H)	1.0	1.0	1.0	40	46	44.0	41	47	44.8	0.96	1.0	0.98
Amazon (L)	0.25	0.40	0.36	28	43	35.6	39	47	43.2	0.72	0.91	0.82
Twitter (H)	1.0	1.0	1.0	22	28	24.2	23	31	27.2	0.77	0.96	0.89
Twitter (L)	0.38	0.53	0.41	15	25	20.6	25	31	27.4	0.60	0.84	0.75
Instagram (H)	1.0	1.0	1.0	20	26	22.2	22	28	25.0	0.75	1.0	0.89
Instagram (L)	0.39	0.56	0.42	15	23	19.0	16	26	23.6	0.71	0.94	0.81
PayPal (H)	1.0	1.0	1.0	23	28	25.6	25	28	26.4	0.92	1.0	0.97
PayPal (L)	0.06	0.38	0.26	18	26	21.4	24	30	27.0	0.68	0.88	0.79

	# Potential Victims	# Participants*	Insensible Rate
All (H)	1,153	1,242	0.92
All (L)	994	1,253	0.79
Total	2,147	2,495	0.86

*Participants who answered that they use the brand’s service more than once a month
H: High Score, L: Low Score

and their order were randomized for each participant. We defined *potential* victims of the attack as those who answered “yes” in the deceptive questions about a certain brand’s service among those who used the service more than once a month in the previous question. We assumed that the participants who answered “no” recognized the deceptive IDNs.

The median time to complete the survey was 4.3 minutes, and we compensated the participants \$0.75 each. After removing 17 participants who gave incomplete or careless answers, we analyzed the remaining 474 participants. The participants’ ages ranged from 18 to 72, with a median of 34 (mean 35.7). 59.7% of the participants were male. Similar to the first survey, the sample of the second survey had a wide range of education levels and occupations.

A limitation of this user study is that we did not measure the actual *success rate* of the attacks. As an ethical consideration, we did not provide the hyperlinks of the actual deceptive IDNs in the questionnaires to avoid harming the participants. Another limitation is that the study was limited to 70 deceptive IDNs. However, we believe this study can provide unique and adequate results to show the risks of deceptive IDNs.

Results. We defined the *insensible rate* = v/p , where p is the number of participants who answered that they used a certain brand’s service once a month, and v is the number of potential victims who answered that they visited the deceptive IDN disguising the brand’s service. The results are shown in Table 10. Most participants did not really notice the deceptive IDNs with high scores; the insensible rate for the IDNs with high scores was 0.92 (=1,153 / 1,242). Surprisingly, many participants did not notice deceptive IDNs even if their scores were not high; the insensible rate for the IDNs with low scores was 0.79 (=994 / 1,253) in total. The insensible rate of all IDNs was 0.86 (= (1,153+994) / (1,242+1,253)). Some participants who noticed deceptive IDNs commented: “[...] I marked these as no because they contained these special characters” and “[...] questions are supposed to be phishing or intentionally fake sites but I marked no on the ones that aren’t plainly the

Table 11: Correlation between Deceptive IDN Score and Insensible Rate

Brand	Correlation Coefficient (γ)	p -value
Google	0.83	0.0027*
YouTube	0.35	0.31
Facebook	0.74	0.014*
Amazon	0.84	0.0021*
Twitter	0.73	0.016*
Instagram	0.46	0.18
PayPal	0.87	0.0011*
All	0.68	<0.0001*

We note statistically significant differences with asterisks.

real domain.” Unfortunately, the participants who were IT professionals and computer engineers were also likely not to notice deceptive IDNs, similar to other participants.

Overall, as shown in Table 11, we found a positive correlation between the deceptive IDN score and the insensible rate of the attacks ($\gamma=0.68$, p -value<0.0001), although the correlation was not significant for YouTube and Instagram. This result indicates that the proposed system can successfully measure the reasonable scores that reflect the tendency of users to be deceived by deceptive IDN attacks.

In summary, our user study newly revealed deceptive IDNs are difficult for end users to recognize even if they are IT-professionals or computer engineers. Through correlation analysis, we confirmed that the deceptive IDN score successfully reflects the tendency of users to be deceived by the considered type of cyber attacks.

7 Discussion

In the previous section, the user studies revealed that most end users do not notice deceptive IDNs. To mitigate the risks of deceptive IDNs and enhance cultural and linguistic diversity on the Internet with IDNs, various stakeholders should take countermeasures against deceptive IDNs. We believe that our findings based on the measurements and user studies can help improve countermeasures for stakeholders. Now, we briefly provide discussions and suggestions for client applications, domain registrars/registries, domain owners, and certificate authorities (CA) on how to reduce the spread of deceptive IDNs.

7.1 Client Application

Client applications such as web browsers and other applications displaying URLs or domain names can prevent users accessing deceptive IDNs by detecting them. For example, to mitigate eng-homo deceptive IDNs, many web browsers have original policies/rules about whether to display IDNs in Unicode or Punycode format in their address bars [18, 45]. Moreover, very recently, the Google Chrome browser has implemented a new experimental feature for warning against look-alike URLs including eng-homo deceptive IDNs [51]. DOMAINSCOUTER found many newly defined deceptive IDNs other than simple eng-homo, thus, DOMAINSCOUTER can help improve the rules/functions for providing better detection coverage of deceptive IDNs.

Unfortunately, the mitigation in client applications can only prevent users from accessing deceptive IDNs and does not address the root cause that such deceptive IDNs exist. The existence of a deceptive IDN similar to a legitimate brand is a risk of brand defamation, especially for companies. Therefore, not only client applications but also the other stakeholders should take other countermeasures against them.

7.2 Registrar and Registry

The guidelines for implementing IDNs [17] for mainly TLD registries describe that visually confusing characters from different scripts must not be allowed to co-exist in a single IDN label unless a corresponding IDN policy and IDN Table [12, 47] are defined to minimize confusion between domain names. The majority of eng-combo and eng-homocombo exhibit the prohibited pattern, mixing cross-script code points in a single label. According to Table 4, eng-combo and eng-homocombo account for 49% ($= (368+3,697) / 8,284$) of all 8,284 detected deceptive IDNs. If registries strictly followed the guidelines prohibiting the mixture of cross-script code points, approximately half of the discovered deceptive IDNs could have been avoided.

Registrars and registries make an effort to enable rightsholders to protect their rights when registering domain names; however, they do not investigate IDNs comprehensively. Although the trademark clearinghouse (TMCH) [57] contributes to protecting domains, deceptive IDNs are beyond its technical scope. The TMCH serves as a database for verified trademark rights information. Trademarks are submitted to the TMCH by rightsholders. Verified marks are provided with a priority-registration period and the Trademark Claims service for all new gTLDs. The Trademark Claims service identifies potentially abusive registrations by comparing TMCH-recorded trademark strings to domain names and sends a notice to rightsholders. The technical problem is a domain name is considered as an exact match to a TMCH-recorded string. This method results in false negatives when detecting deceptive IDNs. Our system discovered various deceptive IDNs unexplored by other methodologies. This means that registrars and the TMCH should broaden the scope of the detection to include IDNs and adopt the method proposed in this paper to prioritize defending high-scoring deceptive IDNs. Furthermore, the TMCH should serve not only new gTLDs but also legacy ccTLDs and new IDN ccTLDs.

7.3 Domain Owner

Brand protection is an essential way for rightsholders to fight against the violation of their rights. The mindset of those owning famous domain names (or trademarks) should be to make an effort to protect their brands and not to allow visually confusing domain names to be operated by other parties. The owners of famous domains (or trademarks) can take preventive actions to protect their brands. They can proactively

register additional domain names that are similar to their own brands to prevent abusive registrations by other parties. They can also use brand protection services (e.g., the TMCH) or take measures by themselves. According to our measurement results, only 3.8% of the visually confusing domain names that we discovered as deceptive IDNs were legitimately registered for brand protection. We assume that most domain owners (and also brand protection services) are not aware of such IDNs because they were unexplored by other existing methodologies; thus, domain owners should broaden the scope of brand protection to include IDNs.

When domain owners find squatted domain names (e.g., deceptive IDNs) targeting their brands, they can use the Uniform Domain-Name Dispute-Resolution Policy (UDRP) [59] to confiscate or cancel such domain names. The UDRP, a policy for resolving disputes regarding the registration of domain names, has been adopted by all ICANN-accredited registrars of gTLDs [26]. Many registrars of ccTLDs also adopt the UDRP or regionally localized policies based on it (e.g., JP-DRP [27]). Dispute resolution services based on the UDRP are widely used by rightsholders. The World Intellectual Property Organization (WIPO), one such service provider, handled over 73,000 cases from 1999 to 2017 and successfully transferred the rights to rightsholders [65, 66]. A case filed with the WIPO is normally concluded within two months. The Uniform Rapid Suspension System (URS) [60], which complements the UDRP, provides rightsholders with a quick and a low-cost process to take down squatted domain names. The fees of the URS start from almost \$1,000 less than those of the UDRP (\$1,500 [49]). The identified invalid domain names are suspended by the registry within two or three weeks; however, they are not deleted or transferred to the rightsholders. To counter deceptive IDNs, domain owners can select one of the two services (the UDRS or the URS) by taking both the urgency and the monetary costs into consideration.

7.4 Certificate Authority

Outreach efforts to spread HTTPS by security engineers, researchers, and browser vendors made many large websites serve HTTPS by default. The major browsers also require HTTPS; e.g., Google Chrome started to mark all HTTP sites as “not secure” in July 2018.

Certificate authorities (CAs) should not issue certificates to suspicious domain names (websites) to protect end users from deceptive IDNs. However, in reality, many CAs have issued certificates to squatted domain names, including deceptive IDNs [58]. The baseline requirement for the issuance and management of publicly trusted certificates published by the CA Browser Forum [6] mentions that CAs should do additional verification activities for high-risk certificate requests. We recommend that CAs accommodate the brand-protection policies and procedures that are followed by domain registrars. If all responsible CAs proactively shared trademark information similar to the TMCH, they would NOT issue certificates to

squatted domain names. In addition, CAs would be able to revoke certificates for the domain names that violate trademarks if they received such claims from rightsholders. Domain owners are able to explore certificates of squatted domain names in the log server of certificate transparency [34] because all CAs are now encouraged to submit new certificates to it. Many responsible CAs receive claims from rightsholders.

8 Related Work

We summarize related research literature in terms of deceptive IDNs and non-IDN squattings.

Deceptive IDNs. Gabrilovich and Gontmakher first mentioned an IDN homograph attack using non-ASCII characters in 2002 [14]. In 2006, Holgers et al. investigated a campus network traffic to find eng-homo IDNs targeting the Alexa top 500 [19]. As mentioned in Section 4, in 2018, Liu et al. proposed an eng-homo IDN detection method using the SSIM index between IDNs and brand domains [36]. Sawabe et al. proposed using OCR-based similarities between non-ASCII and ASCII characters [48]. In 2019, Le Pochat et al. explored candidate IDNs that brand owners may want to register [43]. Suzuki et al. developed a framework to identify IDN homographs in an automated manner [53]. Whereas the above studies focused mainly on eng-homo IDNs using a smaller number of IDNs under a limited number of TLDs, our work has advanced these studies by focusing on various deceptive IDNs (e.g., eng-combo, eng-homocombo, noneng-combo, and noneng-homocombo), analyzing more IDNs under almost all TLDs, and studying the extent to which users are deceived by deceptive IDNs.

Non-IDN Squattings. In addition to deceptive IDNs, many previous studies analyzed a wide range of domain squatting methods in non-IDN (ASCII) domains such as combosquatting (combining brand name with keywords) [30], bit squatting (accidental bit flips) [41], and typosquatting (typographical errors) [1, 29, 55, 62].

9 Conclusion

This paper proposed a system called DOMAINSCOUTER to detect deceptive internationalized domain names (IDNs) and calculate the deceptive IDN score. We performed the most comprehensive measurement study to show that (1) there are many previously unexplored deceptive IDNs, (2) their number has kept increasing since 2014, and (3) only 3.8% of them are protected by their targeted brand owners. Moreover, we conducted online surveys to reveal that the majority of users cannot recognize deceptive IDNs and confirm that the deceptive IDN score successfully reflects the tendency of users to be deceived. To reduce the risk of deceptive IDNs, we provided suggestions for client applications, domain registrars/registries, domain owners, and certificate authorities. We hope that our results can be used to enable a secure and multilingual Internet for all users.

References

- [1] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. Seven months' worth of mistakes: A longitudinal study of typosquatting abuse. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.
- [2] Alexa Top Sites. <http://www.alexa.com/topsites/>.
- [3] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for DNS. In *Proc. 19th USENIX Security Symposium*, pages 273–290, 2010.
- [4] Manos Antonakakis, Roberto Perdisci, Wenke Lee, Nikolaos Vasiloglou II, and David Dagon. Detecting malware domains at the upper DNS hierarchy. In *Proc. 20th USENIX Security Symposium*, 2011.
- [5] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou II, Saeed Abu-Nimeh, Wenke Lee, and David Dagon. From throw-away traffic to bots: Detecting the rise of DGA-based malware. In *Proc. 21st USENIX Security Symposium*, pages 491–506, 2012.
- [6] Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates, v.1.1.7. <https://cabforum.org/wp-content/uploads/BRv1.1.7.pdf>.
- [7] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. EXPOSURE: finding malicious domains using passive DNS analysis. In *Proc. 18th Network and Distributed System Security Symposium (NDSS)*, 2011.
- [8] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [9] Daiki Chiba, Mitsuaki Akiyama, Takeshi Yagi, Kunio Hato, Tatsuya Mori, and Shigeki Goto. DomainChroma: Building actionable threat intelligence from malicious domain names. *Computers & Security*, 77:138–161, 2018.
- [10] Daiki Chiba, Takeshi Yagi, Mitsuaki Akiyama, Toshiaki Shibahara, Takeshi Yada, Tatsuya Mori, and Shigeki Goto. DomainProfiler: Discovering domain names abused in future. In *Proc. 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 491–502, 2016.
- [11] Cisco Umbrella 1 Million. <https://umbrella.cisco.com/blog/2016/12/14/cisco-umbrella-1-million/>.
- [12] K. Davies and A. Freytag. Representing Label Generation Rulesets Using XML. RFC 7940 (Proposed Standard), August 2016.
- [13] Farsight Security, Inc. DNSDB. <https://www.dnsdb.info/>.
- [14] Evgeniy Gabrilovich and Alex Gontmakher. The homograph attack. *Commun. ACM*, 45(2):128, 2002.
- [15] Carlos Gañán, Orcun Cetin, and Michel van Eeten. An empirical analysis of ZeuS C&C lifetime. In *Proc. 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 97–108, 2015.
- [16] Google Safe Browsing. <https://developers.google.com/safe-browsing/>.
- [17] Guidelines for the Implementation of Internationalized Domain Names Version 4.0. <https://www.icann.org/en/system/files/files/idn-guidelines-10may18-en.pdf>.
- [18] Guidelines for URL Display. https://chromium.googlesource.com/chromium/src/+/master/docs/security/url_display_guidelines/url_display_guidelines.md.
- [19] Tobias Holgers, David E. Watson, and Steven D. Gribble. Cutting through the confusion: A measurement study of homograph attacks. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 261–266, 2006.
- [20] hpHosts. <http://www.hosts-file.net/>.
- [21] IANA. Root zone database. <https://www.iana.org/domains/root/db>.
- [22] ICANN. ICANN IDN ccTLD Fast Track Process. <https://www.icann.org/resources/pages/fast-track-2012-02-25-en>.
- [23] ICANN. ICANN IDN Glossary. <https://www.icann.org/resources/pages/glossary-2014-02-04-en>.
- [24] ICANN. ICANN new gTLDs delegated strings. <https://newgtlds.icann.org/en/program-status/delegated-strings>.
- [25] ICANN. Internationalized domain names. <https://www.icann.org/resources/pages/idn-2012-02-25-en>.
- [26] ICANN-Accredited Registrars. <https://www.icann.org/registrar-reports/accredited-list.html>.
- [27] JP Domain Name Dispute Resolution Policy (JP-DRP). <https://www.nic.ad.jp/en/drpf/>.

- [28] E. L. Kaplan and Paul Meier. Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association*, 53(282):457–481, 1958.
- [29] Mohammad Taha Khan, Xiang Huo, Zhou Li, and Chris Kanich. Every second counts: Quantifying the negative externalities of cybercrime via typosquatting. In *Proc. 36th IEEE Symposium on Security and Privacy (SP)*, pages 135–150, 2015.
- [30] Panagiotis Kintis, Najmeh Miramirkhani, Charles Lever, Yizheng Chen, Rosa Romero Gómez, Nikolaos Pitropakis, Nick Nikiforakis, and Manos Antonakakis. Hiding in plain sight: A longitudinal study of com-bosquatting abuse. In *Proc. 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 569–586, 2017.
- [31] Maciej Korczynski, Maarten Wullink, Samaneh Tajalizadehkhoob, Giovane C. M. Moura, Arman Noroozian, Drew Bagley, and Cristian Hesselman. Cybercrime after the sunrise: A statistical analysis of DNS abuse in new gTLDs. In *Proc. 13th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 609–623, 2018.
- [32] Marc Kühner, Christian Rossow, and Thorsten Holz. Paint it black: Evaluating the effectiveness of malware blacklists. In *Proc. 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, volume 8688, pages 1–21, 2014.
- [33] Tobias Lauinger, Kaan Onarlioglu, Abdelberi Chaabane, William Robertson, and Engin Kirda. WHOIS lost in translation: (mis)understanding domain name expiration and re-registration. In *Proc. 16th ACM on Internet Measurement Conference (IMC)*, pages 247–253, 2016.
- [34] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013.
- [35] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proc. 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.
- [36] Baojun Liu, Chaoyi Lu, Zhou Li, Ying Liu, Hai-Xin Duan, Shuang Hao, and Zaifeng Zhang. A reexamination of internationalized domain names: The good, the bad and the ugly. In *Proc. 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 654–665, 2018.
- [37] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond blacklists: learning to detect malicious web sites from suspicious URLs. In *Proc. 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1245–1254, 2009.
- [38] Majestic Million. <https://majestic.com/reports/majestic-million>.
- [39] Mozilla foundation. Public suffix list. <https://publicsuffix.org/list/>.
- [40] NewSky Security. Fake Adobe website delivers BetaBot. <https://blog.newskysecurity.com/fake-adobe-website-delivers-betabot-4114d1775a18>.
- [41] Nick Nikiforakis, Steven Van Acker, Wannes Meert, Lieven Desmet, Frank Piessens, and Wouter Joosen. Bit-squatting: exploiting bit-flips for fun, or profit? In *Proc. 22nd International World Wide Web Conference (WWW)*, pages 989–998, 2013.
- [42] Arman Noroozian, Maciej Korczynski, Carlos Hernandez Gañán, Daisuke Makita, Katsunari Yoshioka, and Michel van Eeten. Who gets the boot? analyzing victimization by DDoS-as-a-Service. In *Proc. 19th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, volume 9854, pages 368–389, 2016.
- [43] Victor Le Pochat, Tom van Goethem, and Wouter Joosen. Funny accents: Exploring genuine interest in internationalized domain names. In *Proc. 20th International Conference on Passive and Active Measurement (PAM)*, volume 11419, pages 178–194, 2019.
- [44] Polyglot. <http://polyglot.readthedocs.org>.
- [45] The Chromium Projects. IDN in Google Chrome. <https://www.chromium.org/developers/design-documents/idn-in-google-chrome>.
- [46] Pyssim. <https://github.com/jterrace/pyssim>.
- [47] Repository of IDN Practices. <https://www.iana.org/domains/idn-tables>.
- [48] Yuta Sawabe, Daiki Chiba, Mitsuaki Akiyama, and Shigeki Goto. Detecting homograph IDNs using OCR. In *Proc. Asia-Pacific Advanced Network (APAN) Research Workshop*, volume 46, pages 56–64, 2018.
- [49] Schedule of Fees under the UDRP (valid as of December 1, 2002). <https://www.wipo.int/amc/en/domains/fees/>.
- [50] Quirin Scheitle, Oliver Hohlfeld, Julien Gamba, Jonas Jelten, Torsten Zimmermann, Stephen D. Strowes, and Narseo Vallina-Rodriguez. A long way to the top: Significance, structure, and stability of Internet top lists. In *Proc. 18th ACM Internet Measurement Conference (IMC)*, pages 478–493, 2018.

- [51] Emily Stark. The URLephant in the room. In *USENIX Enigma 2019*. <https://www.usenix.org/conference/enigma2019/presentation/stark>.
- [52] StatCounter. Browser Market Share Worldwide. <http://gs.statcounter.com/browser-market-share>.
- [53] Hiroaki Suzuki, Daiki Chiba, Yoshiro Yoneya, Tatsuya Mori, and Shigeki Goto. ShamFinder: An automated framework for detecting IDN homographs. In *Proc. 19th ACM Internet Measurement Conference (IMC)*, 2019.
- [54] Symantec. Deepsight intelligence. <https://www.symantec.com/services/cyber-security-services/deepsight-intelligence>.
- [55] Janos Szurdi, Balazs Kocso, Gabor Cseh, Jonathan Spring, Márk Félegyházi, and Chris Kanich. The long "taile" of typosquatting domain names. In *Proc. 23rd USENIX Security Symposium*, pages 191–206, 2014.
- [56] Tencent Security Xuanwu Lab. Spoof All Domains Containing ‘d’ in Apple Products [CVE-2018-4277]. <https://xlab.tencent.com/en/2018/11/13/cve-2018-4277/>.
- [57] The Trademark Clearinghouse. <http://trademark-clearinghouse.com>.
- [58] Touched by an IDN: Farsight Security shines a light on the Internet’s oft-ignored and undetected security problem. https://www.farsightsecurity.com/2018/01/17/mschiffm-touched_by_an_idn/.
- [59] Uniform Domain-Name Dispute-Resolution Policy. <https://www.icann.org/resources/pages/help/dndr/udrp-en>.
- [60] Uniform Rapid Suspension (URS). <https://www.icann.org/resources/pages/urs-2014-01-09-en>.
- [61] VirusTotal. <https://www.virustotal.com/>.
- [62] Yi-Min Wang, Doug Beck, Jeffrey Wang, Chad Verbowski, and Brad Daniels. Strider typo-patrol: Discovery and analysis of systematic typo-squatting. In *Proc. 2nd USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2006.
- [63] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Processing*, 13(4):600–612, 2004.
- [64] Whois XML API. <https://www.whoisxmlapi.com/>.
- [65] WIPO Cybersquatting Cases Reach New Record in 2017. https://www.wipo.int/pressroom/en/articles/2018/article_0001.html.
- [66] WIPO UDRP Domain Name Decisions (gTLD). <https://www.wipo.int/amc/en/domains/decisionsx/index-gtld.html>.
- [67] Sandeep Yadav, Ashwath Kumar Krishna Reddy, A. L. Narasimha Reddy, and Supranamaya Ranjan. Detecting algorithmically generated malicious domain names. In *Proc. 10th ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 48–61, 2010.
- [68] Xudong Zheng. Phishing with Unicode Domains. <https://www.xudongz.com/blog/2017/idn-phishing/>.

Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC

Wei Song¹, Peng Liu²

¹State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China

²The Pennsylvania State University, University Park, USA

Abstract

Minimal eviction sets are essential for conflict-based cache side-channel attacks targeting the last-level cache (LLC). In the most restricted case where attackers have no control over the mapping from virtual addresses to cache sets, finding rather than computing minimal eviction sets becomes the only solution. It was believed that finding minimal eviction sets is a long process until a recent discovery that it can be done in linear time.

This paper focuses on improving the existing algorithms and finding minimal eviction sets with the minimal latency. A systematic analysis of the existing algorithms has been done using an ideal cache. Our analysis shows: The latency upper bound of finding minimal eviction sets can be further reduced from $O(w^2n)$ to $O(wn)$; the average latency is seriously less than the upper bound; the latency assumption used by recent defenses is significantly overestimated. Overall, the latency is significantly shorter than we ever expected. Practical experiments are produced on three modern processors. Using a handful of new techniques proposed in this paper, including using concurrent multithread execution to circumvent the thrashing resistant cache replacement policies, we demonstrate that minimal eviction sets can be found within a fraction of a second on all processors, including a latest Coffee Lake one. It is also the first time to show that it is possible to find minimal eviction sets with totally random addresses without fixing the page offset bits, which provides a starting point towards a viable attack against fully randomized LLCs if they are ever adopted in the future.

1 Introduction

Cache-based side-channel attacks [31] has become serious security problems in recent years. They have been utilized to recover cryptographic keys [11], bypass the address space layout randomization (ASLR) [6], inject faults directly into the DRAM [7], and construct covert channels for attacks against speculative execution [15].

Many of the aforementioned attacks require the adversary to bring specific cache blocks into controlled states. This can be achieved by two types of attacks: *flush-based* attacks and *conflict-based* attacks. Flush-based attacks use explicit cache control instructions, such as `clflush` on x86 [31], to invalidate target cache blocks. These attacks are accurate but the explicit cache control instructions may require privilege to execute [32] and the target cache blocks must be shared between the adversary and the victim. If either of the conditions is not satisfied, an attacker could launch conflict-based cache attacks to achieve similar effect. By accessing a sequence of carefully chosen addresses, called an *eviction set*, enough cache replacements are triggered so that the target cache block is evicted out of the cache by one of them [22]. Conflict-based attacks can be launched in a sandbox without privilege [21] and, when attacking the last-level cache (LLC), the target cache block can belong to another process or virtual machine [33] located on another core [18]. Compared with flush-based attacks, conflict-based attacks are more versatile. Conflict-based attacks targeting the LLC are the type of attacks researched by this paper.

An eviction set is a collection of (virtual) addresses that contains enough number of addresses mapping to the same cache set containing the target cache block. Accessing this collection in a certain order triggers a cache replacement that evicts the target cache block from the cache set [26]. It is widely known that accessing a large number of addresses is sufficient to evict any cache block from any levels of caches [29]. However, accessing extra addresses beyond the necessary can introduce undesirable noise [7] and reduce the speed of attack below the required minimum [5]. According to a study on an ARM Cortex-A53, evicting a cache block using a set of 800 congruent addresses can be 33 times slower and less accurate than using a small set of 21 addresses [16]. Pruning a large eviction set to its minimal is crucial for the success of all targeted and stealth side-channel attacks.

Computing minimal eviction sets typically involves partially reversing the mapping from virtual addresses to physical addresses [16]. Reversing this mapping is relatively easy

if the mapping is already exposed by the operating system (`/proc/self/pagemap` in old Android systems [16]) or the adversary has already obtained the root privilege (a malicious kernel [8]). If neither is the case, the adversary may infer a part of the physical address by acquiring large chunks of virtually and physically contiguous memory, which is normally done by using huge pages [11, 18]. In the most restricted case where the adversary has no control over the mapping from virtual to physical addresses, such as in a sandbox [7, 21], computing minimal eviction sets becomes a challenging task.

The recent development in the cache hardware adds another layer of difficulties to the challenge. One of them is the undocumented complex addressing [10, 19] used in Intel processors. The LLC in these processors is sliced and the mapping from physical addresses to slices is determined by a set of undisclosed hash functions. For an adversary targeting the LLC, she needs to decipher the hash functions even when the mapping from virtual to physical addresses is fully reversed. Consequently, the complex addressing in several Intel processors has been deciphered [13, 19]. A set of undisclosed but static hash functions is hardly a strong defense. Nevertheless, a recently proposed LLC remapping technique, namely CEASER [23], may turn all the mentioned reversing effort in vain. CEASER dynamically and randomly remaps physical addresses to cache sets in the LLC using a low latency block cipher. It is a very compromising defense against all conflict-based side-channel attacks. To defeat CEASER in the same way as the complex addressing, an adversary needs to compute an eviction set and exploit it all in the short remapping period. This would be an extremely intimidating task.

Instead of trying to reverse the mapping from virtual addresses to cache sets, several approaches [18, 21, 26] in the literature discussed the possibility of finding minimal eviction sets with limited or even no control over the mapping. These algorithms normally comprise two steps: (1) the adversary first blindly collects a large collection of addresses enough to evict the target cache block, and then (2) prunes the large collection into a minimal eviction set. The size of the initial large collection is normally proportional to the size of the cache [26]. For a collection of n addresses, the original pruning algorithm proposed by Liu *et al.* [18] and Oren *et al.* [21] requires $O(n^2)$ memory accesses, which is terribly slow for large LLCs. Vila *et al.* [26] have recently proposed an optimized pruning algorithm which reduces the bound to $O(w^2n)$, where w is the number of ways in each cache set. In other words, *the necessary time for finding a minimal eviction set is linear with the size of the cache*. In addition, this linear bound invalidates the latency assumption used by CEASER which still assumed the naive bound of $O(n^2)$ memory accesses [23].

Although, in theory, the optimized algorithm [26] finds minimal eviction sets in linear time, in practice, the success rate on modern processors (after Haswell) is as low as just around 15% [26] and the latency in finding eviction sets is still too long for practical attacks.

Several reasons are known to contribute to this. *Translation lookaside buffer (TLB) noise*: Pruning from a large eviction set can cause false positive errors [5] as the TLB entry for the target cache block can be undesirably evicted from the TLB leading to a long latency similar to a miss in the LLC. *Adaptive cache replacement policies*: Introduced from Ivy-Bridge [12, 29], adaptive cache replacement policies are used to resist thrashing and scanning patterns in caches. This leads to both false positive errors where the target cache block is prematurely evicted and false negative errors where accessing an eviction set fails to evict the target cache block due to its scan-like pattern. *Hardware prefetching, inaccurate system timer, process scheduling* and potentially other undocumented hardware optimizations in the cache system may all contribute to the low success rate.

Vila's work [26] is great in discovering the linear time algorithm but it fails to push the algorithm to the limit, which we would try in this paper. We therefore focus on two questions: **(a) In theory, how fast can an adversary find a minimal eviction set?** Vila *et al.* [26] provided only the upper bound but not the average latency. Our analyses show: *The upper bound can be further reduced from $O(w^2n)$ to $O(wn)$; the average number of memory accesses is seriously less than the upper bound; the latency assumption used by CEASER [23] is significantly overestimated.* We then ask: **(b) In practice, how fast can a minimal eviction set be found on modern processors?** We have analyzed the source code provided by Vila *et al.* [26] and studied various eviction techniques [5, 7, 29]. A handful of new techniques are proposed to improve the speed and the accuracy of the optimized algorithm [26]. *Most importantly, we propose to use concurrent multithread execution to circumvent the thrashing resistant cache replacement policies [12].*

Utilizing minimal eviction sets in actual attacks is out of the scope of this paper. We assume they can be efficiently used in attacks without reversing the virtual to physical address mapping [18, 20] and attacks launched inside a sandbox [5, 7, 21] or an SGX enclave [25] without huge pages.

Summary of major contributions: Our major contributions are both theoretical and practical. On the theoretical side, we reduce the upper bound of the pruning algorithm [26] from $O(w^2n)$ to $O(wn)$ and provide an estimation of the average number of memory accesses using an ideal cache. On the practical side, we propose to use concurrent multithread execution to circumvent the thrashing resistant cache replacement policies and provide a handful of new techniques to significantly improve the speed and the success rate of finding minimal eviction sets. To our best knowledge, we are the first to successfully find minimal eviction sets on a modern Intel processor (Skylake) with literally no information of the address mapping, even without exploiting the fact that page offset bits control a part of the cache set index.

2 Preliminaries

2.1 Caches and Virtual Memory

In modern processors, caches are utilized to store recently or frequently used data to reduce access time. Data are stored in units of fixed-sized cache blocks. Commonly, caches are set-associative which allow a group of cache blocks to reside in one of the many cache sets. Cache sets are addressed by cache index, which is typically a subset of the address bits shared by all cache blocks in the same set. When the processor accesses an address, the cache checks whether the corresponding cache set has a block matching with the address (a hit). If no match is found (a miss), the cache block is fetched from memory and stored in the cache set for future use. If the cache set is fully occupied at this moment, a block is chosen by a replacement policy and evicted from the set. As a commonly used policy, least-recently used (LRU) would keep recently accessed cache blocks in the cache.

Caches are hierarchically organized in modern processors. Each core contains a pair of small level-one (L1) caches for data and instruction. The core may contain a medium-sized level-two (L2) cache. All cores share a large last-level cache (LLC). An inclusive cache hierarchy is normally adopted. When a cache block is evicted from the LLC, it is also purged from all cache levels. A cache coherence protocol is utilized to ensure that data are correctly updated in all caches.

User land applications run in the virtual memory space while physical memory is dynamically allocated to virtual memory in unit of pages, normally 4KB sized. The mapping is stored in page tables which are also cached in the cache hierarchy. L1 caches are typically addressed by virtual addresses, while the LLC is addressed by physical addresses. Figure 1 depicts a virtually indexed and physically tagged cache (normally used as L1). The virtual to physical address translation proceeds in parallel with cache set accesses. A translation lookaside buffer (TLB) is used to directly translate the virtual page number into the corresponding physical page number, if such translation has been recently used and cached in the TLB. If TLB fails to translate a virtual page number, the page table is accessed to refill the TLB, which leads to a long latency penalty. Also shown in Figure 1, virtual and physical addresses share the same page offset bits, which is also partially used in the cache index. The LLC has a similar structure but without the TLB as it is indexed by physical addresses.

2.2 Eviction Set

Definition 1. For a specific cache, two virtual addresses x and y are *congruent*, denoted as $x \simeq y$, if and only if x and y are mapped to the same set, $set(x) = set(y)$ [26], but they do not address the same cache block, $cb(x) \neq cb(y)$:

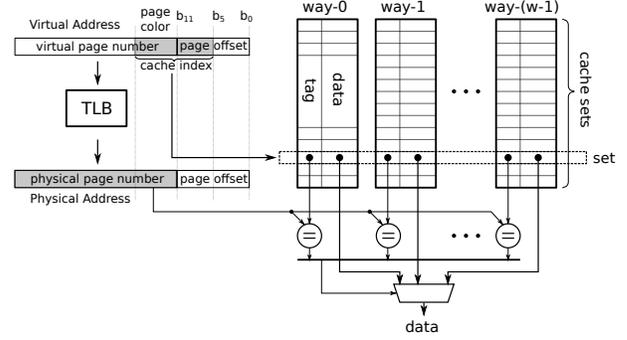


Figure 1: A virtually indexed physically tagged cache.

$$x \simeq y \iff set(x) = set(y) \wedge cb(x) \neq cb(y) \quad (1)$$

Congruence is an equivalence relation. The equivalence class $[x]$ of x with regarding to \simeq is the set of cache blocks mapping to the same cache set with x . We now give the definition of an eviction set.

Definition 2. A set of virtual addresses S is an eviction set for a target address x if $x \notin S$ and at least a addresses in S are congruent with x [26]:

$$x \notin S \wedge |[x] \cap S| \geq a \quad (2)$$

The intuition behind Definition 2 is that, if x is initially stored in a cache set, accessing congruent elements in a certain order can systematically evict x from the cache set, where a is the minimal number of congruent elements needed.

For caches adopting permutation-based replacement policies [2], such as FIFO, least-recently used (LRU) and pseudo-LRU (PLRU) [3], sequentially and repeatedly accessing w congruent cache blocks, where w is the number of ways in a cache set, guarantees the eviction of any block originally stored in the set. In this case, a is equal with w . However, for processors adopting thrashing resistant replacement policies [12], sequentially accessing is recognized as a scan and thus the accessed blocks are the first to be replaced rather than those originally stored. As a result, sequentially accessing an eviction set no longer guarantees the eviction of the target, even when $a \gg w$. The key here is to change the access pattern. As analyzed in Section 4.1, choosing a proper traverse function and utilizing concurrent multithread execution reduce the minimally required number of congruent elements to w as well. Finally, we can define the concept of a minimal eviction set.

Definition 3. A set of virtual addresses S is a minimal eviction set for a target address x if and only if $x \notin S$, S has w elements, and all the w elements are congruent with x [26]:

$$x \notin S \wedge |S| = w \wedge [x] \cap S = S \quad (3)$$

3 Find Eviction Sets in Ideal Caches

In this section, we answer the first question: **In theory, how fast can an adversary find a minimal eviction set?** A systematic analysis of the existing algorithms has been done using an ideal cache. We have further reduced the latency upper bound, revealed several parameters which can be tuned to reduce the average latency, and improved existing algorithms to boost their tolerance to noise.

3.1 An Ideal Cache

To conduct a systematic analysis of the complexity in finding eviction sets and reduce the noise introduced by the actual hardware implementation, an ideal cache model is adopted. It has the following characteristics:

Universal access latency: Accessing a cache block has the same latency disregarding to its location (level, set, way or slice) and status (hit or miss). This assumption allows measuring the search time by the number of memory accesses.

Ideal hit/miss status: Search algorithms can directly and accurately inquire the status of a cache block (hit or miss) with no time penalty. This removes the errors of measuring the accessing latency on actual processors.

No TLB noise: The model assumes a uniformly randomized mapping from virtual to physical addresses. Virtual addresses are translated into physical addresses before cache accesses. TLB is not needed and page table entries are not cached.

LRU replacement: Adaptive or random replacement policies can cause a large quantity of errors. To remove this effect, the model assumes a strict LRU replacement policy.

Randomized cache layout: The model adopts a fully randomized cache layout similar to the static version of CEASER [23]. Virtual addresses are randomly mapped to all cache sets; therefore, search algorithms cannot reduce their complexity by reversing the mapping.

3.2 Find a Candidate Set

The first step in finding a minimal eviction set is to create a large (non-minimal) eviction set, called a *candidate set*. Since the mapping from virtual addresses to cache sets is randomized, a candidate set is acquired by randomly collecting a large collection of virtual addresses.

According to Definition 2, a collection of virtual addresses is a candidate set for the target address x if there are more than a congruent addresses in the collection. The probability that a collection of n virtual addresses is a candidate set for x can be calculated using the binominal distribution:

$$Pr(X \geq a) = 1 - \sum_{i=0}^{a-1} \binom{n}{i} p^i (1-p)^{n-i} \quad (4)$$

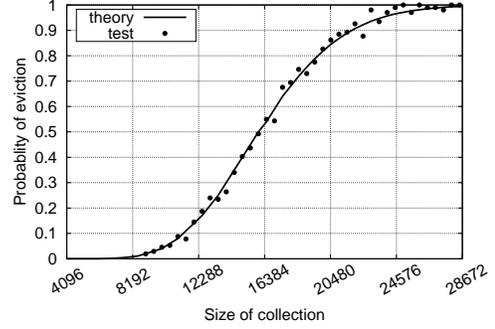


Figure 2: Probability of finding a candidate set as a function of its size. The cache has 1024 sets and 16 ways in each set. **Theory** shows the probability calculated from Equation 5. **Test** shows the test results using the ideal cache. Each result is averaged from 100 independent experiments.

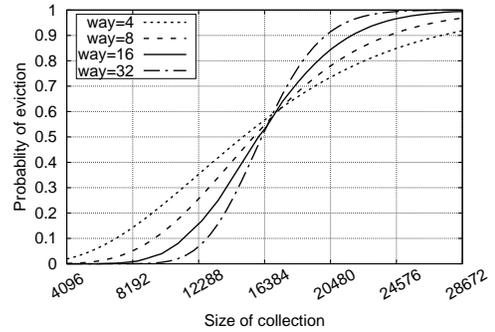


Figure 3: The probability of finding a candidate set in caches with the same size (16384 blocks) but different ways.

where p is the probability that a random virtual address is congruent with x . For a cache with s sets and w ways in each set, $a = w$ and $p = s^{-1}$:

$$Pr(X \geq w) = 1 - \sum_{i=0}^{w-1} \binom{n}{i} \frac{(s-1)^{n-i}}{s^n} \quad (5)$$

Figure 2 depicts the probability of finding a candidate set as a function of its size. It is shown that the test results using the ideal cache closely match the probability calculated from Equation 5. The probability of finding a candidate set increases with its size.

There is an interesting observation: When the size is around the number of blocks in a cache (16384 in this case), the probability of finding a candidate set is around 50%. Figure 3 depicts the probability of finding a candidate set in caches with the same number of blocks but different number of ways. It is shown that, independent of the set-way configuration, achieving a probability around 60% requires the same number of virtual addresses which is just above the total number of blocks in the cache. It is possible to detect the size of the LLC by searching the size achieving the 60% probability.

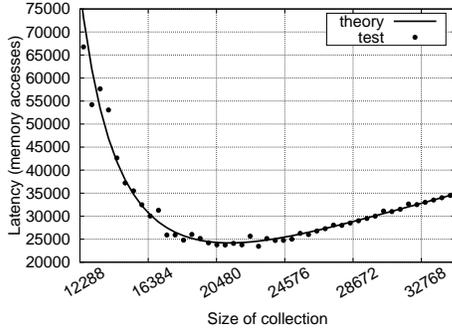


Figure 4: The average latency of finding a candidate set as a function of its size. The cache has 1024 sets and 16 ways in each set. **Theory** shows the probability calculated from Equation 6. **Test** shows the averaged latency of finding a candidate set in the ideal cache. Each result is averaged from 100 independent experiments.

To estimate the average latency of finding a candidate set, we assume the adversary to choose a fixed size for the candidate set. She repeatedly acquires a collection and verifies it until a candidate set is found. In the verification, the adversary accesses all elements in the collection after first visiting the target address, and then checks whether the target address is evicted. Using the number of memory accesses as the measurement of time, the average latency of finding a candidate set with n elements can be calculated as $T_c(n)$:

$$T_c(n) = \frac{n}{Pr(X \geq w)} \quad (6)$$

Figure 4 demonstrates the average latency of finding candidate sets of different sizes. For a 1024-set 16-way cache, finding a candidate set of 20480 elements in around 25000 memory accesses is the quickest. Finding a smaller candidate set needs more retrials while larger sets suffer from longer verification latency.

3.3 Prune an Eviction Set

With a candidate set available, the rest is to prune it into a minimal eviction set. Algorithm 1 is the original pruning algorithm proposed by Liu *et al.* [18] and Oren *et al.* [21]. The algorithm takes the target virtual address x and a candidate set C as inputs. For each element c in the candidate set, the algorithm tests whether the candidate set is still an eviction set without c . If yes, $test(C \setminus \{c\}, x)$ returns true and c is then removed from the set. When all elements are tested, the remaining candidate set becomes a minimal one. As analyzed in Section 2.2, the size of the minimal eviction set should be w , the number of ways.

Proposition 4. Algorithm 1 prunes a candidate set of n elements into a minimal eviction set in $O(n^2)$ memory accesses:

Algorithm 1: Baseline algorithm: $prune_base(C, x)$

Input: C , candidate set; x , target address.
Output: Minimal eviction set for x .

```

1 function  $prune\_base(C, x)$ 
2   foreach  $c$  in  $C$  do
3     if  $test(C \setminus \{c\}, x)$  then
4        $C \leftarrow C \setminus \{c\}$ 
5     end
6   end
7   return  $C$ 
8 end
```

$$T_{base}(n) \sim O(n^2) \quad (7)$$

Proposition 4 can be proved by finding that both the upper and the lower bounds approach $O(n^2)$. The baseline algorithm is quadratic, which is slow for large caches. Nevertheless, it has a benefit that the algorithm does not need to know the number of ways. It can be used to detect this information.

An optimized algorithm was recently proposed by Vila *et al.* [26], as illustrated in Algorithm 2. Instead of removing at most one element in each iteration, multiple elements are removed in Algorithm 2. For each **while** iteration, the remaining candidate set is split into l groups $\{G_1, \dots, G_l\}$. For each group G , it is tested whether the target x can be evicted without it. If yes, $test(C \setminus G, x)$ returns true and G is removed. The success of the algorithm depending on the split parameter l . In the worst scenario, elements of the minimal eviction set distribute evenly in all groups. If $l > w$, it is guaranteed that $l - w$ groups can be removed in each **while** iteration. Vila *et al.* set l to $w + 1$ to maximize the group size [26].

Algorithm 2: Prune with split: $prune_split(C, x, w, l)$

Input: C , candidate set; x , target address; w , number of ways; l , split parameter.
Output: Minimal eviction set for x .

```

1 function  $prune\_split(C, x, w, l)$ 
2   while  $|C| > w$  do
3      $\{G_1, \dots, G_l\} \leftarrow split(C, l)$ 
4     foreach  $G$  in  $\{G_1, \dots, G_l\}$  do
5       if  $test(C \setminus G, x)$  then
6          $C \leftarrow C \setminus G$ 
7       end
8     end
9   end
10  return  $C$ 
11 end
```

Proposition 5. Assuming $l > w$, the number of memory accesses of using Algorithm 2 to prune a candidate set of n elements has an upper bound:

$$T_{split}(n) < \frac{l(l-1)n}{l-w} \quad (8)$$

See Appendix A for a proof similar to the proof provided in [26]. When l is set to $w + 1$, the upper bound is $w^2n + wn$

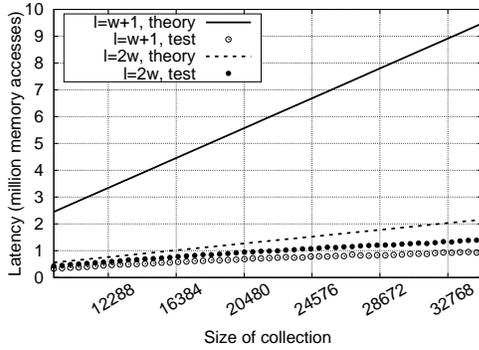


Figure 5: The average latency of pruning a candidate set as a function of its size. The target cache has 1024 sets and 16 ways in each set. **Theory** shows the latency upper bound from Proposition 5. **Test** shows the average latency of pruning a candidate set using different split parameters in the ideal cache. Each result is averaged from 100 independent experiments.

approaching $O(w^2n)$ as described in [26]. However, this is not the optimal bound for $w \geq 4$. **The optimal upper bound is $(4w-2)n$ approaching $O(wn)$ when the split parameter is set to $2w$ (for caches with $w \geq 4$).**

Figure 5 reveals the average pruning latency compared with different upper bounds in a 1024-set 16-way cache. The optimal upper bound using $l = 2w$ is significantly lower than the upper bound using $l = w + 1$. Both upper bounds are higher than the average latency from cache tests but **the optimal upper bound of $(4w-2)n$ is a more accurate estimator.**

Note that using an algorithm with a lower upper bound does not result in a lower average latency. The average latency of using $l = 2w$ is constantly higher than using $l = w + 1$. The reason is the early termination optimization of the inner **foreach** loop which is normally applied. When $l = w + 1$, only one group is guaranteed removable in the worst case. The **foreach** loop in Algorithm 2 can finish immediately when this group is found. However, nearly all groups are tested when $l = 2w$ because there are at least w removable groups.

On actual processors, errors are inevitable due to hardware noise. Algorithm 2 might fail when removable groups are mistakenly found irremovable. Algorithm 3 is a more robust algorithm. It combines the **foreach** inner loop into the **while** outer loop. In every iteration, $random_split(C, l)$ picks $|C|/l$ random elements from C , equivalent to a group in Algorithm 2, and tests whether they are removable. Algorithm 3 takes the benefit of the early termination optimization while keeps trying when a removable group is mistakenly tested irremovable. Although we cannot derive a mathematical upper bound for Algorithm 3, its average latency performance is similar to Algorithm 2.

Figure 6a shows the overall latency of finding a minimal eviction set, including the latency in finding a candidate set

Algorithm 3: Random split: $prune_random(C, x, w, l)$

Input: C , candidate set; x , target address; w , number of ways; l , split parameter.

Output: Minimal eviction set for x .

```

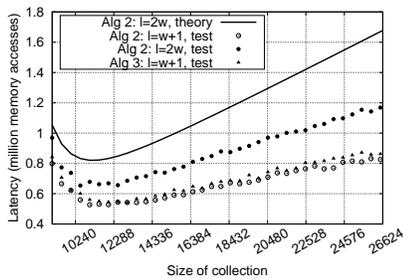
1 function  $prune\_random(C, x, w, l)$ 
2   while  $|C| > w$  do
3      $G \leftarrow random\_split(C, l)$ 
4     if test( $C \setminus G, x$ ) then
5        $C \leftarrow C \setminus G$ 
6     end
7   end
8   return  $C$ 
9 end
```

and pruning it. Algorithm 2 and 3 have similar latency when using the same split parameter. All algorithms achieve their lowest latency around size 11000 disregarding the algorithm and the split parameter, while $l = w + 1$ produces the lowest average latency of around 0.55 million memory accesses in all cases, which is around 50 times of the candidate size. However, neither $w + 1$ nor $2w$ is the optimal split parameter for Algorithm 3. As shown in Figure 6b, $l = 14$ produces an even lower latency for the 16-way cache. The long tail distribution of the overall latency is revealed in Figure 6c.

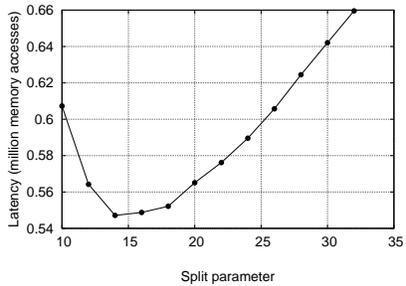
For defenses trying to prohibit an adversary from finding minimal eviction sets by dynamically remapping the cache layout, such as CEASER [23], **the remapping period should be shorter than the lowest overall latency of finding a minimal set.** As shown in Figure 6c, the lowest latency is located at leftmost point of the distribution, which is smaller than the median latency. Figure 7 shows the 1st (1%) and 5th percentile (5%) of the overall latency in different caches. To securely prohibit an adversary from finding a minimal eviction set with a probability of 99%, the remapping period is constrained by the 1st percentile of the latency, which is roughly 40% of the optimal upper bound. For the 1024-set 16-way cache, the 1st percentile is roughly $25n$, where n is around 11500. It is only 0.2% of the naive bound of $O(n^2)$ used in CEASER [23], **which has significantly overestimated the time complexity in finding minimal eviction sets.**

4 Find Eviction Sets on Actual Processors

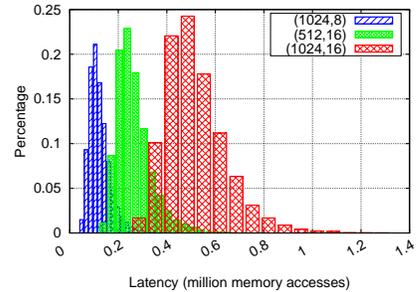
In this section, we answer the second question: **In practice, how fast can a minimal eviction set be found on modern processors?** As shown in Table 1, three different platforms are used in the evaluation. On all platforms, the search algorithm runs in user land without the root privilege. Algorithm 3 is used as the pruning algorithm due to its near optimal latency and tolerance to noise. Candidate sets are collected from a pre-allocated pool of 1GB in the virtual address space. By default, elements in the pool has the same page offset so that eviction sets are found at the granularity of pages. Later in Section 4.4, results for finding eviction sets comprised of elements with



(a) The overall latency of finding minimal eviction sets in a 1024-set and 16-way cache as a function of the size of the candidate set. **Theory** shows the optimal upper bound while **test** shows the latency using different algorithms and split parameters.



(b) The overall latency of finding minimal eviction sets in a 1024-set and 16-way cache using Algorithm 3 with different split parameters.



(c) The long tail distribution of the overall latency in different (set, way) caches using Algorithm 3 with $l = w$.

Figure 6: Analyses for the overall latency in finding minimal eviction sets.

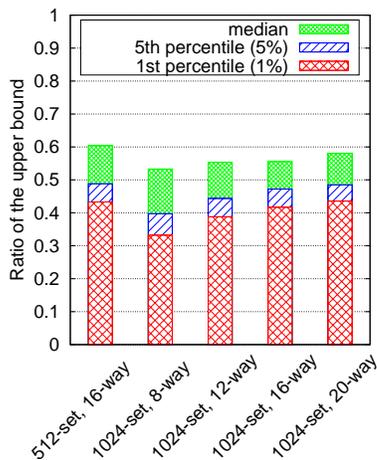


Figure 7: The overall latency of different cache configurations at the 1st (1%) and the 5th (5%) percentile, and the median. Results are normalized by their optimal upper bounds.

arbitrary page offset bits (at the granularity of cache blocks) are presented. The complex address scheme in the LLC is not reversed. Huge pages are not used unless explicitly noted. The initial size of candidate sets is chosen to achieve 50% of eviction using a sliding test similar to Figure 2, which is 2700 for i7-3770, and 3500 for both Xeon-4110 and i7-8700. The split parameter is set to the number of ways. The high resolution time stamp counter is used for time measurement [24] and the `clflush` instruction is used only in the automatic calibration of the LLC miss threshold.

4.1 Eviction Test

The key challenge in finding minimal eviction sets on actual processors is to quickly and accurately test whether a collection of virtual addresses is an eviction set. Algorithm 4

Table 1: Evaluation Platforms

	i7-3770	Xeon-4110	i7-8700
Architecture	IvyBridge	SkyLake	Coffee Lake
Cores	4	8	6
Threads	8	16	12
LLC Size	8MB	11MB	12MB
Cache Way	16	11	16
Memory	4GB	32GB	32GB
OS (Ubuntu)	16.04	16.04	18.04

illustrates a generic `test()` function used in the pruning algorithms. Instead of sequentially accessing a candidate set as on the ideal cache, the set is accessed using a dedicated `traverse()` function for multiple times. Normally the result is averaged from b repeated tests and the candidate set is traversed d times in each test. The latency of accessing the target address x is measured using a `time()` function.

Algorithm 4: Eviction test function

Input: C , candidate set; x , target address.
Output: Whether C evicts x .
Parameter: h , LLC eviction threshold; b , number of repeats; d , number of traverses; `traverse()`, `traverse` function.

```

1 function test( $C, x$ )
2    $i \leftarrow 0, j \leftarrow 0$ 
3   while  $i < b$  do
4     access( $x$ )
5     while  $j < d$  do
6       traverse( $C$ )
7        $j \leftarrow j + 1$ 
8     end
9     record  $t \leftarrow \text{time}(\text{access}(x))$ 
10     $i \leftarrow i + 1$ 
11     $j \leftarrow 0$ 
12  end
13  return  $\bar{t} > h$ 
14 end

```

Time measurement and LLC eviction threshold: The access latency of x is used to tell whether x is evicted from the LLC. The accuracy in this measure affects the speed and accuracy of the pruning algorithm. In our experiment, the time stamp counter provided by the Intel processors is directly used as an accurate time source [24]. On platforms where such time source is unavailable, it is possible to construct an accurate counter using a separate thread [6, 24]. If the latency of accessing x is larger than a threshold h , x is assumed evicted from the LLC. Such threshold is obtained from an automatic calibration [26] on each processor using the `clflush` instruction. For platforms where `clflush` is unavailable, measuring the data accessing latency by deliberately thrashing the LLC [29] produces the same result.

Traverse function: The choice of traverse functions depends on the cache replacement policy of the LLC. For processors using permutation-based replacement policies [2], sequentially accessing candidate sets would be sufficient. However, complex traverse functions become necessary when thrashing resistant replacement policies [12] are used, as on modern processors. Four types of traverse functions are analyzed in our experiments:

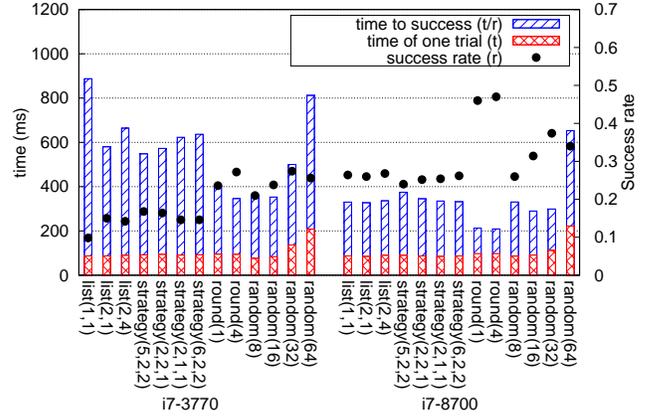
List traverse $list(m, n)$: Traversing an eviction set using a moving window has been found effective on some processors [29]. For each $c_i \in C$, $list(m, n)$ accesses $\{c_i \dots c_{i+m-1}\}$ for n times. It tries to hide its scan-like pattern by accessing the elements multiple times in short intervals. However, it might be ineffective when the eviction set C is not a minimal one, where $c_i \dots c_{i+m-1}$ might belong to different sets and most accesses to the LLC are filtered by the inclusive L1 cache.

Strategy traverse $strategy(m, n, \delta)$: Cache eviction strategy was first proposed in [7]. Strategy traverse takes a more generic form than list traverse as $list(m, n)$ is equivalent to $strategy(m, n, 1)$ in theory. The parameter δ controls the incremental step of the outer loop. To be specific, if c_i is traversed using $list(m, n)$ in the current iteration, $c_{i+\delta}$ would be traversed in the next iteration. When $\delta > 1$, $strategy(m, n, \delta)$ reduces the total number of memory accesses by a factor of δ compared to $list(m, n)$.

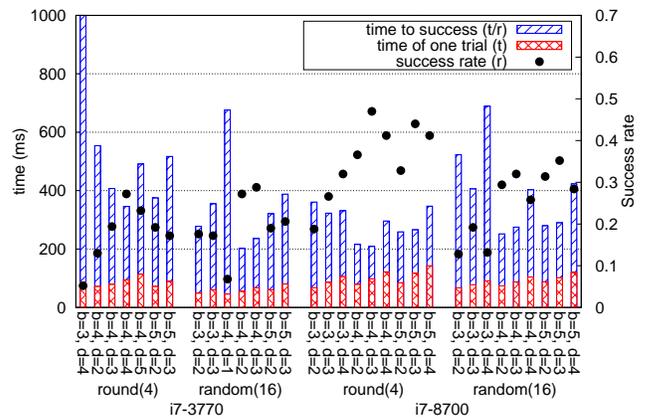
Round trip traverse $round(n)$: First proposed in [18], a round trip traverse sequentially accesses an eviction set forward and then backward. For each $c_i \in C$, it is accessed n times. This guarantees that the access pattern seen by each LLC set is a round trip rather than a simple scan.

Random traverse $random(n)$: This is a new traverse function proposed in this paper. For each $c_i \in C$, it is placed in a buffer along with $n - 1$ previously accessed and randomly selected elements. All the n elements in the buffer are accessed instead of c_i alone. This random pattern is likely to trigger repeated accesses to the LLC and therefore break its scan-like pattern.

Figure 8a demonstrates the results of finding minimal evic-



(a) The performance of different traverse functions on i7-3770 ($b = 4, d = 4$) and i7-8700 ($b = 4, d = 3$).



(b) The performance of using different (b, d) combinations.

Figure 8: Experiments using different configurations. Each result is averaged from 500 independent trials. *Time to success* refers to the estimated average latency of successfully finding a minimal eviction set, which is calculated as *time of one trial* (t) divided by the *success rate* (r).

tion sets using different traverse functions. We use *time to success* as the main criterion in comparing different algorithms. As an estimation of the average latency of successfully finding a minimal eviction set, it is calculated as the *time of one trial* divided by the *success rate* averaged from all trials. On i7-3770 (IvyBridge), $round(4)$ and $random(16)$ perform equally well while $round(4)$ performs the best on the latest i7-8700 (Coffee Lake). Interestingly, on Xeon-4110 (Skylake), no eviction set is found no matter which function is used.

Repeat parameters: The number of repeats b and the number of traverses d in each repeat also affect the results. Figure 8b reveals the performance of using the best traverse function with different repeat parameters (b, d). On i7-3770, the best time to success is produced by $b = 4, d = 2, random(16)$, while it is $b = 4, d = 3, round(4)$ on i7-8700.

Multithread traverse: To circumvent the thrashing resistant cache replacement policies used in modern processors [12], we propose to use concurrent multithread execution to do parallel rather than sequential traverse. The idea is simple. Although different traverse functions try to break the scan-like pattern by making multiple accesses to the same cache block, the inclusive L1 cache filters most of the extra accesses. Modern processors are usually multiprocessors containing four to eight cores (double for hardware threads). Meanwhile, attackers are capable of motivating multiple threads even in a JavaScript sandbox thanks to the latest support of web workers and JavaScript’s SharedArrayBuffer [5, 6, 9].¹ When a cache block is concurrently accessed by multiple cores, the LLC almost always receives multiple requests unless the block has been cached in all the L1 caches of all cores. This would effectively break the scan-like pattern and significantly increases the success rate of eviction.

Instead of doing the d traverses sequentially as described in Algorithm 4, the candidate set is traversed concurrently using d worker threads. It was found that creating and destroying threads cause significant amount of noise. Worker threads are created beforehand and kept in idle waiting status until traverse jobs are broadcast through global atomic variables.

Listing 1 shows the creation of worker threads (`worker()`). The traverse jobs are scheduled using the two global atomic variables: `jobs` (the number of traverse jobs to be claimed) and `done` (the number of traverse jobs being done). The `create_thread()` function is called only once at the beginning to initialize the global atomic variables, create several worker threads, and make them detached from the main thread. The number of workers depends on the chosen number of traverses (d).

Listing 1: Worker Initialization

```

1 // global variables
2 atomic<int> jobs;
3 atomic<int> done;
4 int d = number of traverse;
5
6 void create_threads() {
7     jobs = 0;
8     done = 0;
9     for(int i=0; i<d; i++) {
10         thread t(worker);
11         t.detach();
12     }
13 }

```

Listing 2 explains the internal procedure of worker threads. For each worker (`worker()`), it constantly checks whether there are unclaimed jobs (`jobs > 0`). If yes, the worker consequently claims a job (`jobs--`), does the traverse, and increases the job count (`done++`). To avoid race conditions among workers and the main thread, the reading of `jobs`

¹ As a mitigation of the Spectre attack, SharedArrayBuffer was disabled by default in Firefox from 52 ESR [1].

and the modification of both atomic variables are guarded by locks.

Listing 2: Worker Thread

```

1 void worker() {
2     bool work = false;
3     while(true) {
4         lock(); // critical section begin
5         if(jobs > 0) {
6             jobs--;
7             work = true;
8         } else {
9             work = false;
10        }
11        unlock(); // critical section end
12
13        if(work) {
14            traverse(); // the chosen traverse func
15            lock(); // critical section begin
16            done++;
17            unlock(); // critical section end
18        }
19    }
20 }

```

The sequential traverse in Algorithm 4 (line 5–8) is then replaced by the code segment as shown in Listing 3. `traverse.cfg()` broadcasts the chosen traverse function and the candidate set to all workers. Consequently, the main thread triggers all workers by clearing the job count (`done = 0`) and setting the number of unclaimed jobs to the number of traverses (`jobs = d`). To avoid race conditions, the modification of `jobs` is guarded by locks. The multithread traverse finishes when all jobs are claimed and done (`jobs == 0 && done == d`).

Listing 3: Multithread Traverse

```

1 traverse.cfg() // set the traverse function
2
3 done = 0; // clear job count
4 lock(); // critical section begin
5 jobs = d; // trigger works
6 unlock(); // critical section end
7
8 while(jobs != 0 || done != d); // detect end

```

The results of using multithread traverse are shown in Figure 9. The success rate of using multithread traverse is significantly higher than that of using a single thread. The time for each trial is also shortened as traverses are done in parallel. These two factors together result in a much lower time to success compared with the results in Figure 8a. When the best configuration of traverse function and repeat parameter is used, multithread traverse reduces the time to success by 50% and 52% on i7-3770 (IvyBridge) and i7-8700 (Coffee Lake) respectively. Finding minimal eviction sets also becomes possible on Xeon-4110 (Skylake) by using as many as 15 concurrent workers. The best configuration is found to be $b = 2, d = 14, round(1)$.

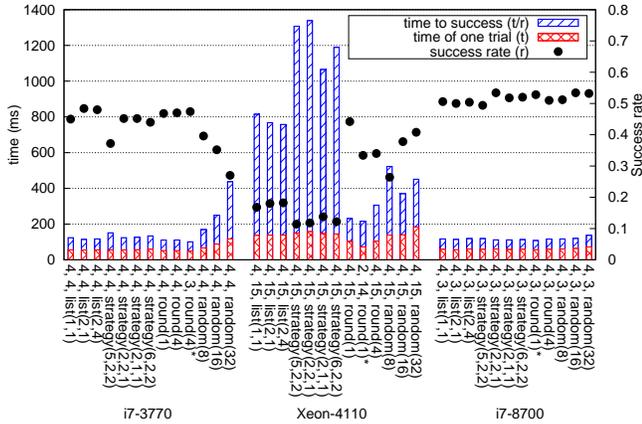


Figure 9: Experiments using concurrent multithread traverse. The parameter of each configuration is shown as $(b, d, \text{traverse})$. The configuration with the lowest time to success is labelled with ‘*’.

4.2 Algorithm Optimizations

It is also possible to improve the performance by further optimizing Algorithm 3. We analyze two methods in this paper: One is the *rollback support* first utilized in [26] and the other one is to *reuse the failed set*, a new optimization proposed by this paper.

Rollback support: When some elements are mistakenly removed from a candidate set, Algorithm 3 may fail to prune it into a minimal one. To tolerate this type of false positive errors, the algorithm can roll back by adding the removed elements back and try again. In practice, the algorithm keeps a limited number of recently removed groups of elements. When a predefined maximal number of retries is reached, the latest removed group is added back and the algorithm redoes the test. The algorithm ultimately fails when all kept groups are added back but the maximal number of retries is still reached.

Reuse the failed set: When a trial fails to find a minimal eviction set, the remaining collection is normally significantly smaller than a new candidate set and has a high density of the irremovable elements for a minimal eviction set. Instead of throwing the collection away, it is kept as a residue set. If the next trial fails as well, the new remaining collection and the stored residue set is combined as the new candidate set for the next trial. Using this combined candidate set has a significantly higher chance in producing a minimal one than a normal candidate set. If it still fails, the remaining collection becomes the new residue set and the whole process starts again.

Figure 10 depicts the improvement achieved by the two optimizations. To produce a fair comparison, the time of a single trial for cases reusing the failed set is the accumulated time of all retries targeting the same virtual address. On all

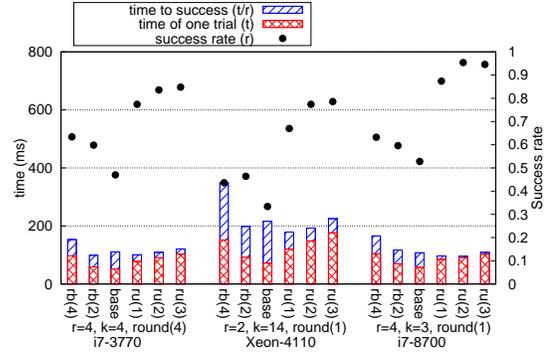


Figure 10: Experiments using algorithm optimizations. Multithread traverse is used. $rb(i)$ denotes the maximal level of allowed rollbacks is i . $ru(i)$ denotes the maximal number of reused trials is i . Both rollback and reuse are disabled in *base*.

processors, reusing the failed set boosts the success rate significantly with a moderate latency penalty. As for rollback, the improvement on the success rate is moderate but the latency overhead becomes significant when the maximal level of rollbacks is more than four. A combination of reusing the failed set and a shallow rollback should reduce the time to success while boosting the success rate.

4.3 Other Optimizations

Besides all the aforementioned techniques, there are extra optimizations that can be utilized to reduce the time to success.

TLB preload: Accessing a large candidate set may trigger false positive errors when TLB entries are mistakenly evicted from the TLB. To reduce this effect, Genkin *et al.* proposed to access another virtual address inside the same page with the target address before checking its cache status [5]. This effectively preloads the TLB entries and therefore reduces false positive errors.

Use huge pages: The underlying reason for the TLB noise is that the large number of pages visited by a candidate set over-stress the comparatively small TLB. Allocating candidate sets from huge pages [26] would significantly reduce the number of pages visited in traversing a candidate set, which thus reduces false positive errors.

Increase retry limit: Allowing extra retries for each candidate set increases the chance of finding removable elements. However, this also increases the time wasted on the candidate sets which would fail ultimately.

Tune the size of candidate sets and split parameters: As described in Section 3.3, finding the optimal candidate size and split parameter would reduce the time of a single trial, which then leads to reduced time to success.

Figure 11 demonstrates the results of using preloaded TLB and huge pages. When multithread traverse is not utilized, preloading TLB indeed improves the success rate, which is significant on i7-3770 (IvyBridge). However, its benefit is

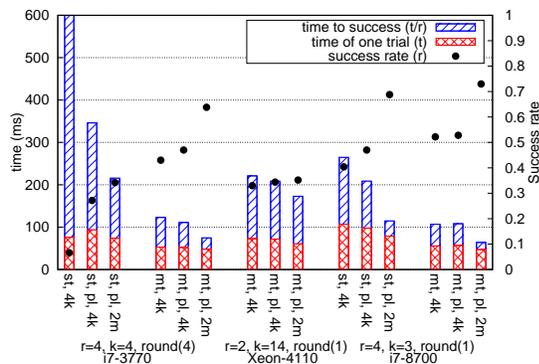


Figure 11: Experiments with preloaded TLB and huge pages. *st* (*mt*) denotes the tests using a single (multiple) thread(s) in traversing. *pl* denotes the TLB entries are preloaded before the eviction test. *4k* and *2m* denote the page size. So *2m* means huge pages are enabled.

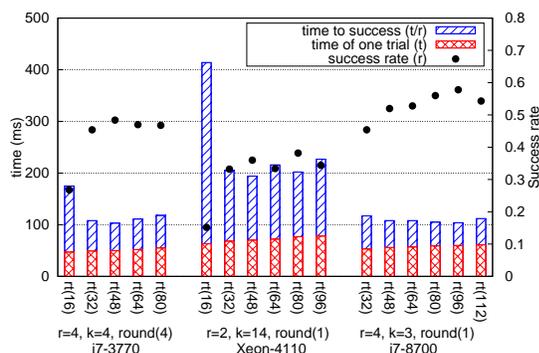


Figure 12: Experiments with different retry limits. Multi-thread traverse is used. *rt*(*i*) denotes the maximal number of retries for each level is *i*.

marginal when multithread traverse is used. In this case, the traverse of candidate sets is done by worker threads rather than the main thread. The TLB entries for the main thread are therefore less disturbed than worker threads and preloading TLB has little effect on the testing results. Using huge pages substantially boosts the success rate in all cases except for Xeon-4110 (Skylake). Another benefit of using huge pages is the reduced time for a single trial as the penalty introduced by page faults is reduced. Overall, both techniques can effectively reduce TLB noise. Preloading TLB is necessary when multithread traverse is not used and huge pages is generally beneficial for all scenarios.

Generally speaking, increasing the number of retries improves the success rate, as shown in Figure 12. However, it also prolongs the time of a single trial and thus raises the time to success for some cases. It looks like, setting the retry limit to four times of the number of ways provides comparatively good results. In our experiments, the best settings are 48, 48 and 96 for i7-3770, Xeon-4110 and i7-8700 respectively.

Table 2: Space of Parameters

Parameter	Good Setting	Proposed by
prune algorithm	Algorithm 3	this paper
repeat parameter	$b = 4, d = 4$	[7, 26]
multithreading	enable	this paper
traverse function	$round(1)$	[18, 26]
rollback	$rb(2)$	[26]
reuse failed set	$ru(1)$	this paper
TLB preload	enable	[5]
huge page	enable	[26]
retry limit	$rt(4w)$	this paper
candidate set	$n = s \cdot w / 64$	this paper
split parameter	$l = w$	this paper

The experiments with differently sized candidate sets and split parameters show similar trends with the results on the ideal cache as shown in Figure 6a and 6b in Section 3.3. The optimal sizes of candidate sets and split parameters (n, l) are found to be (2700, 12), (3500, 9) and (4500, 12) for i7-3770, Xeon-4110 and i7-8700 respectively.

4.4 Best Practice and Summary

Table 2 lists all the parameters that can be tuned to improve the speed and accuracy of finding minimal eviction sets. Obviously, this is a large parameter space to search, especially for a new processor. To reduce the search effort, the “Good Setting” column provides a known good or nearly optimal setting for each parameter. Note that the size of the candidate set is provided assuming eviction sets are found at the page granularity (constant page offset). For finding eviction sets at the granularity of cache blocks, $n = s \cdot w$ is a good start.

In this paper, we have done a manual search of the optimal settings for finding eviction sets on the three processors under evaluation and Table 3 presents the best results we have achieved so far. When finding eviction sets at the granularity of pages, we seek to find the best setting for the shortest time to success for four different scenarios (combination of the availability of huge pages and multithreading). On Xeon-4110, we failed to find minimal eviction sets when multithread traverse is not used. We consider the scenario with multithread traverse but without huge pages as the common case. For the common case, it is possible to find minimal eviction sets in just 0.085s, 0.170s and 0.095s on i7-3770 (IvyBridge), Xeon-4110(SkyLake) and i7-8700 (Coffee Lake) respectively.

We have also estimated the highest success rate achievable by increasing the number of reuses to 50 (except for i7-8700 as it is already high). It is shown that the highest rates are 99.2%, 97.0% and 95.4% on i7-3770, Xeon-4110 and i7-8700 respectively.

To our best knowledge, we are the first to successfully find minimal eviction sets at the granularity of cache blocks, which means doing the search with totally random addresses. The results are also provided in Table 3 with *granularity* set as

Table 3: The Current Best Results of Finding Minimal Eviction Sets

	Granularity	Multithread	Huge Page	Candidate Size	Repeat Parameter (b, d)	Traverse Function	Retry (rb)	Split Parameter (l)	rollback (rb)	Reuse Failed Set (ru)	TLB Preload	Success rate	Time of a Single Trial	Time to Success
i7-3770	4KB	N	N	2700	(4, 2)	<i>random</i> (16)	48	16	2	0	Y	0.340	0.051s	0.150s
i7-3770	4KB	N	Y	2700	(4, 2)	<i>random</i> (16)	48	16	2	0	Y	0.508	0.046s	0.091s
i7-3770	4KB	Y	N	2700	(4, 3)	<i>round</i> (4)	48	16	2	0	Y	0.646	0.055s	0.085s
i7-3770	4KB	Y	N	2700	(4, 3)	<i>round</i> (4)	48	16	2	50	Y	0.992	0.116s	0.117s
i7-3770	4KB	Y	Y	2700	(5, 3)	<i>round</i> (4)	64	16	6	3	Y	0.960	0.058s	0.060s
i7-3770	64B	Y	N	162000	(5, 7)	<i>round</i> (4)	80	16	2	10	Y	0.390	43.98s	1.88m
Xeon-4110	4KB	Y	N	3500	(2, 14)	<i>round</i> (1)	48	9	2	1	Y	0.758	0.129s	0.170s
Xeon-4110	4KB	Y	N	3500	(2, 14)	<i>round</i> (1)	48	9	2	50	Y	0.970	0.320s	0.330s
Xeon-4110	4KB	Y	Y	3500	(2, 14)	<i>round</i> (1)	48	9	2	1	Y	0.760	0.102s	0.134s
Xeon-4110	64B	Y	N	281600	(5, 18)	<i>round</i> (1)	48	9	2	50	Y	0.980	1.70m	1.74m
i7-8700	4KB	N	N	3500	(4, 3)	<i>round</i> (4)	80	14	0	1	Y	0.782	0.158s	0.202s
i7-8700	4KB	N	Y	3500	(4, 3)	<i>round</i> (4)	64	14	0	1	Y	0.860	0.106s	0.123s
i7-8700	4KB	Y	N	3500	(4, 3)	<i>round</i> (1)	64	16	0	2	Y	0.954	0.091s	0.095s
i7-8700	4KB	Y	Y	3500	(4, 3)	<i>round</i> (1)	64	16	2	2	Y	0.966	0.059s	0.061s
i7-8700	128B	Y	N	112000	(4, 3)	<i>round</i> (4)	64	16	2	10	Y	0.100	63.3s	10.6m

Table 4: Improvement against the State-of-the-Art [26]

	Huge Page	Traverse Function	Number of repeat (b)	State-of-the-Art [26]			This Paper			Improvement		
				Success rate	Time of a Single Trial	Time to Success	Success rate	Time of a Single Trial	Time to Success	Success rate	Time of a Single Trial	Time to Success
i7-3770	N	round	7	0.475	0.226s	0.477s	0.646	0.055s	0.085s	36.0%	-75.7%	-82.1%
i7-3770	Y	round	1	0.530	0.116s	0.219s	0.960	0.058s	0.060s	81.1%	-50.0%	-72.6%
i7-8700	N	round	1	0.617	0.151s	0.244s	0.954	0.091s	0.095s	54.6%	-39.7%	-61.1%
i7-8700	Y	round	1	0.500	0.093s	0.186s	0.966	0.059s	0.061s	93.2%	-36.6%	-67.2%

64B. We have managed to succeed on both i7-3770 and Xeon-4110, and achieved a surprisingly high success rate of 98% on Xeon-4110. We believe the number of cores on the server level processors (Xeon-4110) contributes to the high success rate because they allow more threads to traverse concurrently compared with the desktop level processors.

Unfortunately, we failed on i7-8700. The best result is to find eviction sets at the granularity of 128B (2 cache blocks) at a success rate of 10%. The intermediate data show that the pruning algorithm tends to remove the first group of elements (G_1 in Algorithm 3) with abnormally high rates when the size of the candidate set is large. This indicates that the rate of false positive errors in the initial rounds of pruning is too high for the algorithm to tolerate. Although the traverse function *round*(4) achieves the best success rate, the size of the residue sets is rather large (thousands of elements) when it fails. Using *random*(16) results in much smaller residue sets (less than a hundred) but slightly lower success rate and longer latency. We think there might be new changes to the replacement policies inside the latest Coffee Lake processors. The re-accessing of the target address during the eviction

test (line 4 in Algorithm 4) might be recognized as a scan leading to its amateur eviction. Improving the techniques in re-accessing the target address and further optimizing traverse functions should bring down the rate of false positive errors. These are our future works.

To evaluate the improvement we have made against the state-of-the-art method [26], we have rerun it on the same platforms used in this paper. Table 4 reveals the comparison results. Since the state-of-the-art method fails to find eviction sets on Xeon-4110, no comparison is made on this platform. On i7-3770 and i7-8700, evaluations are made both with and without huge pages.

We have manually tested and selected the best traverse function and the best number of repeats (b) for both platforms. The size of the initial candidate set is set to 4000, as same as in [26], while all other parameters are set to their optimal values (after manual tuning). Note that both the *success rate* and the *time to success* are significantly better than the reported rate (around 20%) and latency (0.75s) in [26]. According to our discussion with Vila, the reported low success rate was collected when no optimization was applied, while the re-

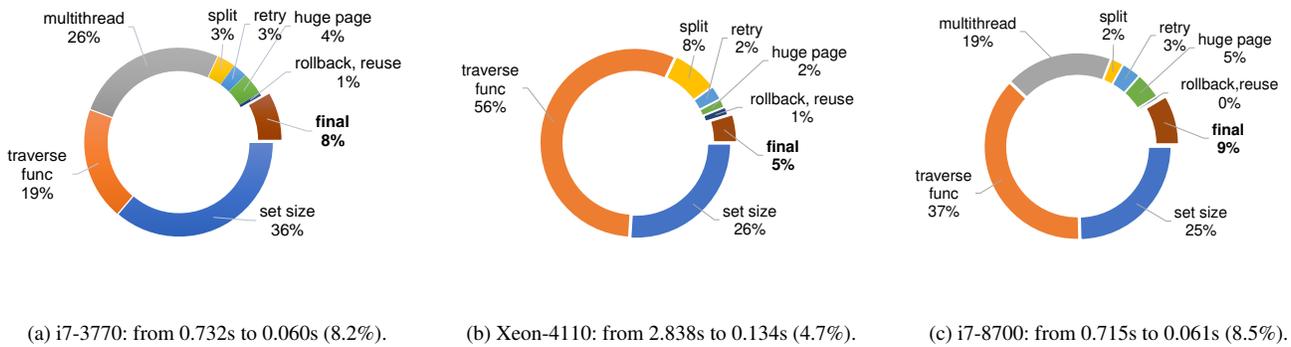


Figure 13: Contribution of individual optimization techniques. Configuration for the initial case: *granularity*= 4KB, *multithread* disabled (except for Xeon-4110), *huge page* disabled, *candidate size*= 4000, *traverse function*= *list*(1, 1), $(b, d) = (4, 4)$, *retry*= 32, *split parameter*= 32, *rollback* disabled, *reuse failed set* disabled and *TLB preload* enabled. The order of optimization: *candidate size*; *traverse function* and (b, d) ; *multithread*; *split parameter*; *retry*; *huge page*; *rollback* and *reuse failed set*.

ported long latency was due to a very conservative setting of 50 to the number of repeats. The performance boost obtained from optimizing the state-of-the-art method clearly demonstrates that we can significantly reduce the overall latency by tuning parameters.

Even compared with the boosted results of the state-of-the-art method, the improvement achieved by using the techniques proposed in this paper is still significant. As shown in Table 4, *time to success* has been reduced by more than 60% in all scenarios while the success rate has been greatly boosted.

We have also evaluated the contribution of individual optimization techniques towards the reduction of *time to success*. The results of all platforms are shown in Figure 13. It is found that the order used in tuning parameters has a strong impact on the contribution. Optimization techniques are not independent with each other and some techniques are closely related. From our observation, the repeat parameter (b, d) is closely related to the traverse function. They are thus tuned together. Reducing the split parameter too early while the success rate is low might actually hurt performance. The split parameter is also related to the retry parameter. Overall, applying the various optimization methods reduces the *time to success* by more than 90% on all platforms.

We tried to find eviction sets on an AMD Ryzen 3 2200G processor but both ours and the state-of-the-art method [26] failed. According to a recent research [30] on the side-channel attacks on non-inclusive LLCs, the recent AMD processors are suspected to use a snoopy-based cache coherence protocol. Since evicting a cache block from a non-inclusive LLC does not purge the block from L1 caches, the assumption of $a = w$, as used in Equation 5, no longer holds and none of the pruning algorithms described in this paper can reduce the size of the candidate set to w . Dynamically finding eviction sets targeting non-inclusive LLCs is still an open question, especially when snoopy-based coherence protocols are used.

5 Related Work

Search algorithms: It was found by Hund *et al.* [10] that accessing a large enough collection of virtual addresses is sufficient to evict any data from the LLC. Liu *et al.* [18] and Oren *et al.* [21] proposed the first pruning algorithm which prunes a large eviction set into a minimal one in $O(n^2)$ memory accesses. Eviction sets found in this way were used to construct eviction sets without fully reversing the virtual to physical address mapping [18, 20], and launch attacks inside a sandbox [5, 7, 21] or an SGX enclave [25] without huge pages. Recently Vila *et al.* [26] managed to reduce the upper bound of the pruning algorithm to $O(w^2n)$. We reduce it further to $O(wn)$ in this paper.

Traverse functions: Starting from the Intel IvyBridge, thrashing resistant cache replacement policies are adopted [12, 29]. Simply repeatedly and sequentially accessing an eviction set no longer guarantees an eviction. A *dual pointer chase* method was found effective on IvyBridge processors [29]. Gruss *et al.* [7] generalized the method with *eviction strategies*. The *round trip* method was first introduced by Liu *et al.* [18]. *Random traverse* and *multithread traverse* are introduced in this paper.

TLB noise: TLB noise was explained by Genkin *et al.* [5]. They discovered that *preloading the TLB entry* [5] can effectively reduce such noise. Also allocating candidate sets from huge pages is effective as well [26].

Existing defenses: *Cache partitioning* is a promising defense against cache side-channel attacks [4, 14, 17]. However, it cannot defend the cases when the target and the eviction set cannot be separated by partitions, such as reversing the complex addressing scheme [13], rowhammer inside a sandbox [25], and circumventing the user level or kernel space ASLR [6, 10]. *Randomizing the cache layout* is another type of effective defenses, which hinders the computing of minimal eviction sets by randomizing the cache set index. Wang *et*

al. [27, 28] proposed to apply randomization on the L1 caches, which is ineffective to thwart attacks against the LLC. Qureshi recently proposed CEASER [23], which dynamically randomizes the LLC using a low latency block cipher. If adopted, this would be a strong defense against all conflict-based attacks against the LLC.

6 Conclusion

In this paper, we have reduced the upper bound of the latency in finding minimal eviction sets from $O(w^2n)$ to $O(wn)$ and shown that recent defenses have significantly over-estimated such latency. Practical experiments are produced on three modern processors. Using multiple new techniques proposed in this paper, including using concurrent multithread execution to circumvent the thrashing resistant cache replacement policies, we demonstrate that minimal eviction sets can be found within a fraction of a second on all processors, including a latest Coffee Lake one. We also show that it is possible to find minimal eviction sets with totally random addresses without fixing the page offset bits.

Acknowledgments

This work was supported by the CAS Pioneer Hundred Talents Program, the National Natural Science Foundation under grant No. 61802402, and internal grants from the Institute of Information Engineering, CAS. The authors would like to thank the anonymous reviewers for their invaluable comments, Kaveh Razavi for kindly shepherding this paper, and Pepe Vila for a detailed explanation of his paper. The authors also express gratitude to Wenhao Wang, Xiaoxin Li, Xiaofei Fu and Yifei Sun for discussions related to cache randomization and cache eviction.

Availability

The ideal cache model described in in Section 3.1 can be accessed at <https://github.com/comparch-security/cache-model>. The test programs on actual processors can be accessed at <https://github.com/comparch-security/smart-cache-evict>.

References

- [1] MFSA 2018-01: Speculative execution side-channel attack (“Spectre”), Jan. 2018. <https://www.mozilla.org/en-US/security/advisories/mfsa2018-01/>.
- [2] Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’14)*, pages 141–142. IEEE Computer Society, Mar. 2014.
- [3] Christoph Berg. PLRU cache domino effects. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET’06)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. <http://drops.dagstuhl.de/opus/volltexte/2006/672>.
- [4] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4):35:1–35:21, 2012.
- [5] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS’18)*, pages 83–102. Springer, 2018.
- [6] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *Proceedings of the Network and Distributed System Security Symposium (NDSS’17)*. Internet Society, 2017.
- [7] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA’16)*, pages 300–321. Springer, 2016.
- [8] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC’17)*, pages 299–312. USENIX Association, Jul. 2017.
- [9] Lars T. Hansen. Shared memory and atomics for ECMAscript, Feb. 2017. https://github.com/tc39/ecmascript_sharedmem.
- [10] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P’13)*, pages 191–205. IEEE, May 2013.
- [11] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S\$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P’15)*. IEEE, May 2015.

- [12] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the International Symposium on Computer Architecture (ISCA'10)*, pages 60–71. ACM, Jun. 2010.
- [13] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the Annual Design Automation Conference (DAC'16)*. ACM Press, 2016.
- [14] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the USENIX Security Symposium (Security'12)*, pages 189–204. USENIX Association, Aug. 2012.
- [15] Paul Kocher, 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 *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*, pages 19–37, May. 2019.
- [16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the USENIX Security Symposium (Security'16)*, pages 549–564. USENIX Association, Aug. 2016.
- [17] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'16)*, pages 406–418, 2016.
- [18] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*. IEEE, May 2015.
- [19] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)*, pages 48–65. Springer, 2015.
- [20] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'17)*, Mar. 2017.
- [21] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM Press, 2015.
- [22] Colin Percival. Cache missing for fun and profit, 2005. https://papers.freebsd.org/2005/cperciva-cache_missing/.
- [23] Moinuddin K. Qureshi. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (Micro'18)*, pages 775–787. IEEE, 2018.
- [24] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'17)*, pages 247–267. Springer, Jan. 2017.
- [25] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*, pages 3–24. Springer, Jul. 2017.
- [26] Pepe Vila, Boris Köpf, and Jose Morales. Theory and practice of finding eviction sets. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 2019.
- [27] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the International Symposium on Computer Architecture (ISCA'07)*, pages 494–505. ACM, Jun. 2007.
- [28] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*, pages 83–93. IEEE, 2008.
- [29] Henry Wong. Intel Ivy Bridge cache replacement policy, Jan. 2013. <http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/>.
- [30] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side-channel attacks in a non-inclusive world. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 2019.

- [31] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the USENIX Security Symposium (Security'14)*, pages 719–732. USENIX Association, 2014.
- [32] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on ARM and their implications for Android devices. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, pages 858–870. ACM Press, Oct. 2016.
- [33] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'12)*, pages 305–316. ACM, Oct. 2012.

A Proof of Propositions

Proposition 4. Algorithm 1 prunes a candidate set of n elements into a minimal eviction set in $O(n^2)$ memory accesses:

$$T_{\text{base}}(n) \sim O(n^2)$$

Proof. For a candidate set with n elements, at most one element is removed in each iteration of the **foreach** loop. In total $n - w$ elements are removed. The minimal number of memory accesses is reached when one element is removed for the first $n - w$ iterations while the maximal is reached when the first

w elements are found irremovable.

$$\begin{aligned} \sum_{i=w+1}^n i + w^2 &\leq T(n) \leq wn + \sum_{i=1}^{n-w} (i+w) \\ \sum_{i=1}^n i &< T(n) < \sum_{i=1}^n n \\ \frac{n^2+n}{2} &< T(n) < n^2 \end{aligned}$$

Since $n \gg w > 1$, $T_{\text{base}}(n) \sim O(n^2)$. \square

Proposition 5. Assuming $l > w$, the number of memory accesses using Algorithm 2 to prune a candidate set of n elements has an upper bound:

$$T_{\text{split}}(n) < \frac{l(l-1)n}{l-w}$$

Proof. In the worst case, the w elements of the minimal eviction set is evenly distributed in w of the l groups; therefore, only $l - w$ groups are removed in each iteration. $l - 1$ groups are tested in the worst case because only $l - w - 1$ groups are found removable in the $l - 1$ tests. The upper bound of $T(n)$ can be calculated as:

$$\begin{aligned} T(n) &\leq (l-1)n + (l-1)n\frac{w}{l} + \dots + (l-1)n\left(\frac{w}{l}\right)^k \\ &= (l-1)n \sum_{i=0}^k \left(\frac{w}{l}\right)^i \end{aligned}$$

where the termination condition is $n\left(\frac{w}{l}\right)^{k+1} \leq w$.

$$T(n) \leq (l-1)n \frac{1 - \left(\frac{w}{l}\right)^{k+1}}{1 - \frac{w}{l}} < \frac{l(l-1)n}{l-w}$$

\square

Time and Order: Towards Automatically Identifying Side-Channel Vulnerabilities in Enclave Binaries

Wubing Wang Yinqian Zhang Zhiqiang Lin
Department of Computer Science and Engineering
The Ohio State University

Abstract

While Intel SGX provides confidentiality and integrity guarantees to programs running inside enclaves, side channels remain a primary concern of SGX security. Previous works have broadly considered the side-channel attacks against SGX enclaves at the levels of pages, caches, and branches, using a variety of attack vectors and techniques. Most of these studies have only exploited the “order” attribute of the memory access patterns (e.g., sequences of page accesses) as side channels. However, the other attribute of memory access patterns, “time”, which characterizes the interval between two specific memory accesses, is mostly unexplored. In this paper, we present ANABLEPS, a tool to automate the detection of side-channel vulnerabilities in enclave binaries, considering both order and time. ANABLEPS leverages concolic execution and fuzzing techniques to generate input sets for an arbitrary enclave program, constructing extended dynamic control-flow graph representation of execution traces using Intel PT, and automatically analyzing and identifying side-channel vulnerabilities using graph analysis.

1 Introduction

Intel Software Guard eXtension (SGX) is a hardware addition that is available in recent Intel processors. It offers both integrity and confidentiality to application software running in a shielded execution environment—a secure enclave—even when the entire operating system is untrusted. Recent work has explored the use of Intel SGX for a variety of applications such as secure cloud data analytics [25], smart contracts [44], anonymity network [18], game hacking protection [7], and unmodified code execution [8, 31], which have outlined a promising future of SGX’s broad adoption in both server-end and client-side computation.

Computer micro-architecture related side channels are not new. Side-channel attacks that exploit micro-architectural resources shared by mutually distrusting computing entities (e.g., processes or threads) date back to the era of Pentium

4 [23, 24]. A malicious program or a virtual machine may manipulate the shared micro-architectural resources, such as CPU caches, branch prediction units, or function units, to learn the pattern with which these resources are used by the victim program and thereby infer secrets that dictate such a usage pattern. Over the past decades, computer micro-architecture has evolved drastically, but the issues of side channels remain. What differ in the SGX context are two fold: First, as SGX is designed to protect the confidentiality of applications that demand high levels of security, side channels become a major security threats. Second, because the adversary against SGX enclaves is assumed to have OS system-level privileges, a wider range of attacks are enabled. Particularly, over the past a few years, researchers have demonstrated that secrets can be leaked from a variety of attack vectors, such as branch prediction units [20], CPU caches [17], paging structures [35, 38, 43], and DRAM row buffers [38].

Completely eliminating side channels from CPU chips is unrealistic. Admitting this decades-old security concern, Intel recommends developers take special care to avoid side-channel vulnerabilities when writing enclave code [6]. However, developers are not experts of side channels and relying on regular program developers to solve side-channel issues is less promising. Moreover, there is no tool available that helps the developers automatically identify improper coding patterns in their enclave binaries.

In this paper, we aim to explore principles and techniques that automatically identify side-channel vulnerabilities in enclave binaries that allow a side-channel attacker who is able to observe execution traces of the *control flow* of an enclave program to infer sensitive information inside the enclave. The root cause of the vulnerability is the secret-dependent control flows that are inherent in the enclave code. More specifically, since side-channel attacks observe the runtime behavior of the enclave programs, an intuitive approach for the vulnerability identification would be to find a large set of secret values (e.g., input of the enclave program), run the enclave program with these secret values, and collect the enclave’s execution traces with respect to the control flow transfers (CFTs). The diversity

of the collected execution traces for different secret values is a viable indicator of the side-channel vulnerabilities—if all secret values correspond to the same execution trace, the enclave code is not vulnerable. With respect to the execution traces, there are both spatial (*i.e.*, order) and temporal (*i.e.* time) differences. A comprehensive solution should include both.

However, it is non-trivial to develop such a comprehensive approach for a number of reasons. First, how to generate the valid secret values (*e.g.*, program input) to expose the execution traces at different granularity (*e.g.*, branch, cache, or page). Second, how to collect the execution traces, especially the temporal information associated with the traces. We cannot use static analysis as it will not be able to resolve secret-dependent CFTs, and meanwhile cannot collect the precise time information. While we can use dynamic analysis, we still need to solve the coverage issues. Third, how to represent the execution traces and perform the cross-comparison, especially when there are multiple execution traces. Finally, how to quantitatively analyze the information leakage due to the detected vulnerabilities. Fortunately, we have addressed these challenges and built a tool dubbed ANABLEPS, by leveraging concolic execution and fuzzing techniques to generate input sets for an arbitrary enclave program, constructing extended dynamic control-flow graph representation of execution traces using Intel PT, and automatically analyzing and identifying side-channel vulnerabilities using graph analysis.

We have tested ANABLEPS with 8 programs and libraries, including text rendering, image processing, gnomonic processing, and deep learning. Our tool has discovered numerous input leakage execution points for these programs. Our study also suggests automated tools can identify the side-channel vulnerabilities based on syntactic inputs and execution traces. However, the semantics (*i.e.*, the meaning) of the input is also of critical importance especially for the exploitation of the side-channel vulnerabilities.

Contributions. To summarize, the contributions of this paper are as follows:

- A novel and comprehensive approach to detecting both time-based and order-based control-flow side-channel vulnerabilities for enclave binaries.
- A practical implementation integrating fuzzing, symbolic execution, and hardware supported execution tracing.
- The first large-scale analysis of sensitive control-flow vulnerabilities for real world enclave binaries.

Roadmap. The rest of the paper is organized as follows. §2 presents necessary background knowledge including related works to facilitate our discussion of the problem and our motivation. In §3, we present the problem statement and a running example to highlight our key insights. We detail our design of ANABLEPS in §4. Then, we present how we implement

ANABLEPS and evaluate its effectiveness in §5. We also made a number of case studies to understand the exploitability of the vulnerabilities in §6. §7 discusses the limitation of the approach and future research directions. Finally, §8 concludes the paper.

2 Background and Related Work

Intel SGX. At a high level, Intel SGX is a set of new instructions for the x86 architecture. These instructions allow application developers to protect sensitive code and data by utilizing a secure container called enclave [13]. The trusted hardware establishes an enclave by protecting isolated memory regions within the existing address space called Processor Reserved Memory (PRM) to assure confidentiality and integrity against other non-enclave memory accesses, including kernel, hypervisor, and other privileged code. The confidentiality of regions outside the PRM is protected by the memory encryption engine (MEE). Enclave programs with memory footprints larger than that is allowed by RPM can make use of memory regions outside the PRM via page swapping. Memory pages swapped out of the RPM need to be encrypted by MEE.

SGX Side-Channel Attacks. Side channels are the Achilles' Heel of Intel SGX's confidentiality guarantees. In the past few years, a variety of side-channel attacks have been demonstrated against SGX enclaves, particularly from the CPU's memory management perspective. For instance, it has been demonstrated that by controlling the *present* flag or the *reserved* flags of the page table entries (PTEs) [29, 43], the adversary could force the enclave program to trigger page faults when accessing a memory page, thus extracting sufficient amount of secrets (*e.g.*, image contours, user input, cryptographic keys). Most recently, it was shown that the page table access patterns can also leak the enclave secrets without actively triggering the page fault [35, 38], which can be achieved by monitoring the *accessed* flag of the PTEs.

Other micro-architectural side-channel attack vectors that have been studied on traditional hardware have also been found exploitable in SGX. It has been demonstrated that cache-based side-channel attacks can be migrated on SGX [9, 15, 17, 26], which can be more powerful than non-SGX settings. Branch prediction units have been demonstrated to leak the branch history inside the enclaves [20]. DRAM row buffer contention has been exploited to steal secrets from enclaves [38].

Most recently, Spectre [19], Meltdown [21], Foreshadow [32], and SGXPectre [10] attacks have been demonstrated to leverage speculative execution and out-of-order execution to read memory content protected by MMU isolation. These attacks are out of scope of this paper as they are micro-architecture vulnerabilities which cannot be solely addressed from software.

Existing Defenses. A number of enclave hardening techniques have been proposed to mitigate these side-channel attacks. To defeat page-level side-channel attacks, T-SGX [28] uses the Transactional Synchronization Extensions (TSX), Déjà Vu [12] relies on the execution time of the enclave program path, SGX-LAPD [14] explores the internal enclave data structures. To guard against cache side channels, Gruss *et al.* [16] encapsulates snippets of enclave code into hardware-supported memory, HyperRace [11] implements contrived data races. Varys [22] also proposes to reserve physical cores for secure enclave computation.

Closely related works to ours are Stacco [42], MicroWalk [40], and DATA [39], all of which detect side-channel vulnerabilities due to secret-dependent control flows. Particularly, Stacco [42] uses Intel Pin tools to detect vulnerabilities in SSL/TLS implementations, and it manually generates input to the SSL libraries, and MicroWalk [40] focuses on vulnerabilities in Intel IPP and Microsoft CNG. Similarly, DATA [39] only focuses on differential address trace analysis for cryptographic primitives. In contrast, as ANABLEPS works on arbitrary enclave binary, it must generate the large volume of input automatically and conduct vulnerability analysis without known semantics. These new design challenges differentiate our work and Stacco, DATA and MicroWalk. Outside the SGX context, CacheD [37] is also relevant to our work. However, in contrast to these works, ours considers more attack vectors.

3 Overview

3.1 Problem Statement and Definitions

The key objective of this work is to automatically identify the side-channel vulnerabilities caused by the secret-dependent control-flow transfers in the enclave programs. As enclave programs are typically shipped to the hosting services in the form of plaintext binary code, we anticipate the primary secret that the enclave developer would like to hide is the input to the enclave code. Therefore, the goal of the attacks is to learn, through a variety of side channels (*e.g.*, page accesses [29, 35, 38, 43], cache eviction [9, 15, 17, 26, 33], and branch prediction [20]), the input to the enclave programs.

However, most of these prior studies on SGX side channels only consider the *order* attribute of memory access patterns, *i.e.*, which memory page (or cache set) has been accessed and in what order. Few has exploited the *time* of memory accesses as a side-channel vector. In fact, the first observation that *time* and *order* are the two key attributes of a side (and covert) channel can date back to the early 1990s [41]. As such, in our work, we consider both, and broadly define that an enclave program is vulnerable to side-channel attacks if different input can lead to different traces from either the executing *order* of each execution unit (*e.g.*, an instruction) or the *timing* at which each unit is visited.

Defining Side-Channel Vulnerabilities. More formally, given an enclave binary program p , a concrete input to p will lead to a concrete execution trace r , which is defined as $[(m_0, t_0), (m_1, t_1), (m_2, t_2), \dots, (m_k, t_k)]$, where m_j is the address of the j^{th} execution unit and t_j is its timestamp relative to the beginning of the execution. When the memory addresses are normalized to be free of effects of randomization, for each input, there is a corresponding trace r .

Definition 1 Given an enclave program p and an input I_i , the mapping function $\mathcal{E}(p, I_i) = r_i$, where r_i is the execution trace of p under the input I_i . Similarly, for a set of input I , we define the mapping function $\mathcal{E}(p, I) = \{r_i | r_i = \mathcal{E}(p, I_i), \forall I_i \in I\}$. The entire input space is denoted I_{space} . Therefore, the entire space of execution traces $R = \mathcal{E}(p, I_{\text{space}})$.

The mapping function \mathcal{E} generates a program's execution trace under a specific input or a set of inputs, which allows us to define side-channel vulnerabilities as follows.

Definition 2 Given an enclave program p and a set of input I , the program is considered to be vulnerable to side-channel attacks (under the input set I) if and only if $|\mathcal{E}(p, I)| > 1$; the input set can be completely leaked through the side channels if and only if $|\mathcal{E}(p, I)| = |I|$.

Informally, we define an enclave program p is vulnerable to side-channel attack if not all the input maps to the same trace. That is, the enclave program's execution is not input oblivious. However, even though the program is vulnerable to side-channel attack, the amount of leaked information can be different. The complete leakage captures the case that every input can be uniquely identified from the execution trace. It is worth noting that the set of input I is a subset of the entire input space I_{space} , *i.e.*, $I \in I_{\text{space}}$. In most practical scenarios, it is impossible to obtain I_{space} . Therefore, the definition of side-channel vulnerabilities is only meaningful when the program and its input set is fixed. In this paper, we consider two types of input set I : $I_{\text{syntactic}}$, the set of input generated automatically from program analysis, and I_{semantic} , the set of input provided by developers that are semantically meaningful.

Representing Execution Traces. To facilitate cross comparison of execution traces and directly pinpoint the secret-dependent control flow transfer (CFT) that leaks the information through side channels, execution traces need to be represented in proper data structures. String, in the form of linear trace $[(m_0, t_0), (m_1, t_1), (m_2, t_2), \dots, (m_k, t_k)]$, however, is not an optimal choice as it will be quite challenging to identify the alignment (*i.e.*, anchor) point from the string. In our design, we choose to use a graph representation of the linear traces.

Definition 3 An extended dynamic control-flow graph (EDCFG) of a program p under input $I_i \in I$ is defined as a directed graph $\mathcal{G} = \langle N, E \rangle$, where $n_i \in N$ is a node of the graph that represents a basic block of the CFG; and $e_i \in E$ is a directed edge of the graph connecting two nodes that represents the dynamic CFT when p is executed with the input I_i .

Also, each edge ($e_i \in E$) has a counter w_i (i.e., weight) to indicate how many times the edge is executed. The information of the program's execution order and time is embedded in each node $n_i \in N$. Each $n_i \in N$ has two ordered lists: $Order = [n_1^i, n_2^i, \dots, n_k^i]$, where n_j^i is the j^{th} successor of node n_i during the execution of p with input I_i ; $Time = [t_1^i, t_2^i, \dots, t_k^i]$, where t_j^i is the execution time to reach node n_j^i .

An ED-CFG of an enclave program uniquely specify the execution trace of the program under a given input. More specifically, \mathcal{G}^i represents the execution trace in a graph representation for the input I_i .

Execution Units in Side-Channel Attacks. An execution unit in the context of a side-channel attack is defined as the minimal single execution trace observable by attackers. For the enclave program execution, an attacker can mostly achieve the minimal execution unit at either cache level, or at page level. Typically, it is hard to observe the single instruction execution or basic block execution, but an attacker might be able to do so at certain scenario (e.g., the branch shadowing attack [20] and the Nemesis attack [34]). Therefore, in our work we focus on the execution unit at page level (address aligned with 4K bytes), at cache level (address aligned with 64 bytes)¹, and at branch level.

Definition 4 A page-level ED-CFG, \mathcal{G}_p , is a variant of \mathcal{G} , where each node of \mathcal{G}_p contains the page execution unit (i.e., all the executed instructions that belong to a particular page, aligned with 2^{12} bytes), and each edge connects the CFTs between the pages. Similarly, we define the cache-level ED-CFG, \mathcal{G}_c , where each node contains the cache execution unit and edge captures the CFTs at cache level.

Therefore, eventually for each input I_i , we will build \mathcal{G}^i first, from which to derive \mathcal{G}_p^i and \mathcal{G}_c^i . To detect the vulnerabilities, we will then cross compare \mathcal{G}^i , \mathcal{G}_p^i , or \mathcal{G}_c^i , respectively, for all input $I_i \in I$. If a trace is different (in terms of time or order of the specific execution units) among different user input, we conclude the enclave program is vulnerable to the corresponding side-channel attacks at different levels such as at branch, cache, or page. Further analysis can be performed on the graphs to quantify the vulnerability, or to identify the leaking code segments.

3.2 A Running Example

Next, we would like to use a simple running example to illustrate how to use \mathcal{G}_p to detect the time and order side-channel vulnerabilities at the page granularity for the software running inside the SGX enclave. Detecting basic block-granularity and cacheline-granularity vulnerabilities is similar when given \mathcal{G}_c . In particular, we use the code snippet shown in Figure 1(d) as

¹In this work, we simply model cache-based side-channel attacks on SGX assuming that the attacker is able to monitor the execution of the enclave program at the granularity of a 64-byte memory block. Interested readers can refer to Wang *et al.* [38] for more detailed discussion on attack techniques.

a running example. This code snippet is a simplified version of a barcode image processing function.

We notice in Figure 1(d) that this program takes three types of inputs: character '1', '2', or an illegal input. The program outputs two types of barcode, or an error message, accordingly. More specifically, function `main()` calls function `DrawBar()` if the input character is '1' or '2', otherwise returns an error (and `exit`). Function `DrawBar()` is used to draw a barcode on the canvas, and the weight of the canvas is decided by the length of the barcode. Then for each column of the barcode, it calls function `DrawLine()`, which calls the function `Paint()` in a loop if the given position is to draw a line.

Trace Construction. By providing input I_1 with '1', I_2 with '2', and an invalid input I_{invalid} , we get the corresponding execution traces $\mathcal{E}(p, I_1)$, $\mathcal{E}(p, I_2)$, and $\mathcal{E}(p, I_{\text{invalid}})$, from which to build \mathcal{G}^1 , \mathcal{G}^2 , and $\mathcal{G}^{\text{invalid}}$. As shown in Figure 1(a)(b)(c), each node represents the executed basic block, and each edge represents the CFT between the basic blocks. We also assigned an index for each node for easier locating them in the graph (e.g., n_1 and n_2). Two ordered lists, `Order` and `Time`, associated with each node record the successor nodes (in execution order) and the execution time (in nanosecond *ns*) to reach them during execution. For instance, in Figure 1(a), the `Order` list of node n_6 is $[n_4, n_4, \dots, n_7, \dots, n_7, \dots]$, which suggests that the execution of the program will first follow the edge from $n_6 \rightarrow n_4$ multiple times, then follow the edge from $n_6 \rightarrow n_7$. The first element of the `Time` list suggests the mean execution time to reach node n_4 for the first time is $0.8ns$.

The corresponding page-level ED-CFGs (\mathcal{G}_p^1 , \mathcal{G}_p^2 , and $\mathcal{G}_p^{\text{invalid}}$) are illustrated in Figure 1(e)(f)(g). For instance, the ED-CFG in Figure 1(a) can be converted to the page-level ED-CFG in Figure 1(e) in the following steps: First, node n_1 and n_7 of the original ED-CFG are both placed on page `0x804a`, they are merged to a single node n_1 in the page-level ED-CFG. Similarly, node n_2 , n_4 , n_5 , and n_6 are merged into node n_2 in page-level ED-CFG. Edges between nodes of the same page are removed in the page-level ED-CFG; those crossing page boundaries are preserved or merged. For instance, the edge $n_2 \rightarrow n_3$ becomes the new edge $n_2 \rightarrow n_3$ in \mathcal{G}_p^1 , and the edges $n_3 \rightarrow n_6$ and $n_3 \rightarrow n_4$ merges into the new edge $n_3 \rightarrow n_2$ in \mathcal{G}_p^1 . We point out that it is not always straightforward to convert ED-CFG to page-level ED-CFG. Some basic blocks in ED-CFG may cross the page boundary. Dealing with these pages require additional efforts, which we will discuss in more details in §4.

Vulnerability Identification. By comparing the \mathcal{G}_p s (\mathcal{G} s or \mathcal{G}_c s), one can easily identify the side-channel vulnerabilities. For instance, by comparing Figure 1(e) and Figure 1(f), it can be seen that the two input values, '1' and '2', leads to different page-level execution orders: the sequence of $n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1$ is repeated one more time when the input is '1'. Figure 1(g) is very different from the other

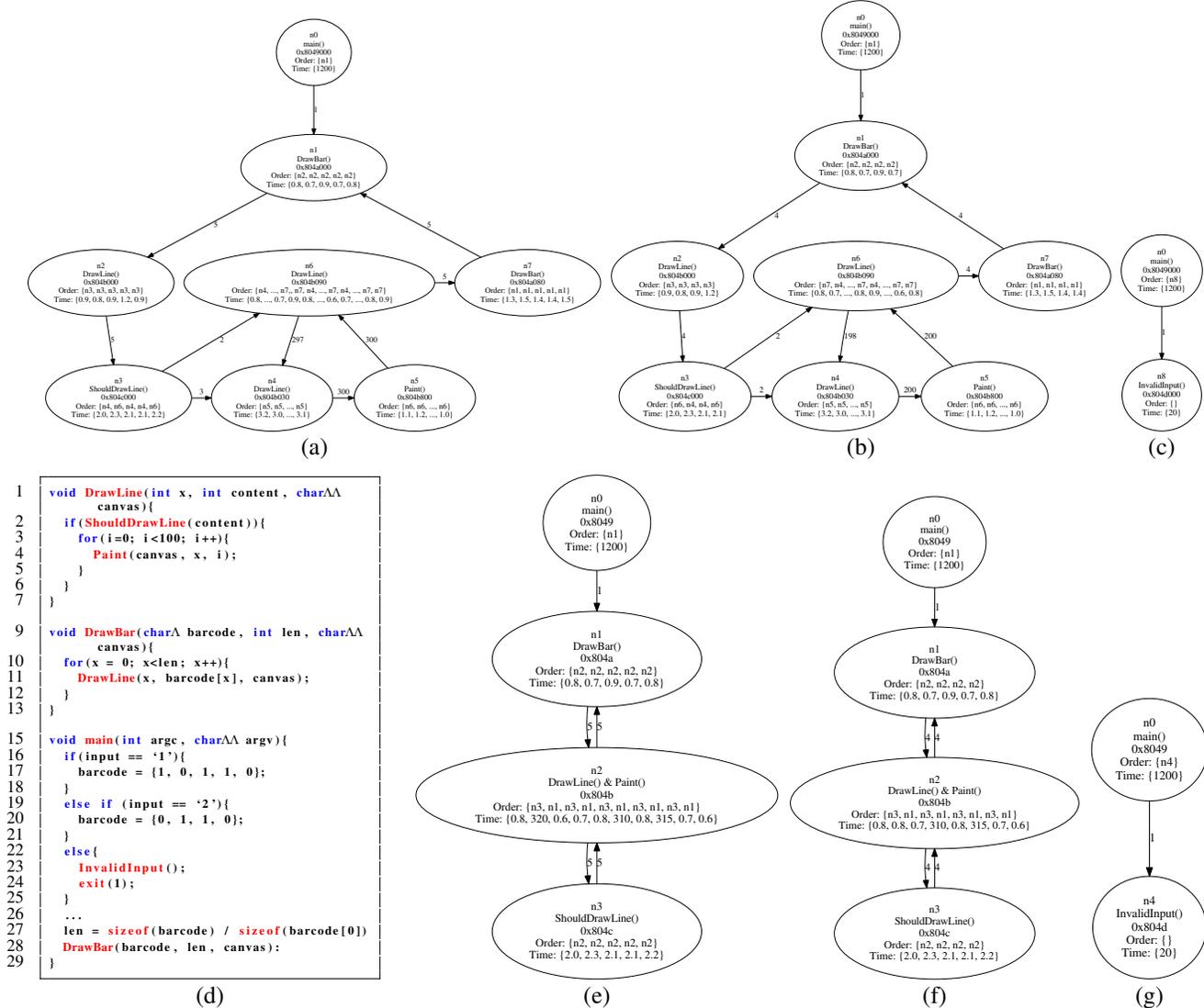


Figure 1: (a) \mathcal{G}^1 , (b) \mathcal{G}^2 , (c) $\mathcal{G}^{invalid}$, (d) Code snippet of our running example, (e) \mathcal{G}_p^1 , (f) \mathcal{G}_p^2 , and (g) $\mathcal{G}_p^{invalid}$

two \mathcal{G}_p s, easily differentiating $I_{invalid}$ from other input. We can validate this vulnerability by scrutinizing the code in Figure 1(d): Function `DrawLine()` is called five times when input value is ‘1’, but four times when the value is ‘2’; the `main()` exits directly with invalid input.

Interestingly, an adversary can also infer more useful knowledge about whether a column in the barcode is a black line or white line. More specifically, according to the implementation of function `DrawLine()`, it will call function `Paint()` 100 times to draw a black line. Therefore, the execution time of function `DrawLine()` is much longer when it draws a black line in given position that of drawing a white line. With this information, the adversary can successfully recover the content of barcode by collecting the execution time of the page node on which function `DrawLine()` is placed. This vulnerability can be detected by scrutinizing the `Time` list of node

n_2 . Let $n_2.Time[k]$ denote the k^{th} element of the `Time` list of node n_2 (the index of an element starts with 1). The execution time to reach node n_1 from n_2 , i.e., $n_2.Time[2]$, $n_2.Time[6]$, $n_2.Time[8]$ of graph \mathcal{G}_p^1 are significantly larger ($> 300 ns$) than $n_2.Time[4]$ and $n_2.Time[10]$ ($< 1ns$). Therefore, it can be inferred that the painted barcode is `[1,0,1,1,0]`, which correspond to input ‘1’.

3.3 Threat Model, Scope, and Assumptions

We assume the knowledge of at least the enclave binary code, especially the code layout and mapping. We assume there is no address space layout randomization (ASLR) with the enclave binaries (such as SGX-Shield [27]). We assume the adversary is capable of launching, resetting, and terminating the targeted enclaves, and is in control of the entire operating

system. This threat model is consistent to the controlled side-channel attacks [43] and also many other side-channel attacks against SGX enclaves [9, 15, 17, 26, 33].

Not all of the side channels are of our focus in this paper. In particular, we focus on identifying the side channels through branch, cache, page access behaviors, and timing information. Other side channels such as hardware architecture caused side channels (e.g., Meltdown [21] and Spectre [19]) are out of the scope. Also, we focus on the side channels from code access, and data access pattern caused side channel is out of scope.

4 Design

4.1 Input Generation

Since ANABLEPS uses dynamic analysis, it is important to generate the concrete input that covers as much as possible of the input space I_{space} . Fortunately, we are not the first to encounter such a problem, and many of the existing vulnerability identification tools all faces similar challenge. The state of the art is to combine both concolic execution (a.k.a, dynamic symbolic execution) and evolution fuzz testing (e.g., AFL [1]) together to generate the best set of $I_{\text{syntactic}}$ (e.g., Driller [30]). Therefore, when design ANABLEPS, we use the Driller approach and extend it for our purpose.

In particular, to start our analysis, we first use AFL [1] to execute the enclave program. The input generated by AFL is called I_{fuzz} . When fuzzing gets stuck and cannot explore the program path further, we use concolic execution to solve the path constraints and generate new input, which is called I_{concolic} . With the new input, we again let fuzzing execute first and only when fuzzing gets stuck, we invoke the concolic execution. When both fuzzing and concolic execution cannot explore the program path further, we terminate the input generation analysis.

During this stage execution, we have collected as many as possible of the program traces, which are the best effort to approximate R in the state of the art. and the minimal possible concrete input $I_{\text{syntactic}} = I_{\text{fuzz}} \cup I_{\text{concolic}}$ we use to expose these traces, denoted as $\mathcal{E}(p, I_{\text{syntactic}})$, where $\mathcal{E}(p, I_{\text{syntactic}}) \subseteq R$. At this stage, for each $I_i \in I_{\text{syntactic}}$, we have a corresponding $r_i \in \mathcal{E}(p, I_{\text{syntactic}})$. While we do know $|I_{\text{syntactic}}|$ (since each I_i is unique), we do not know $|\mathcal{E}(p, I_{\text{syntactic}})|$ yet, since we do not know whether each r_i is unique or not.

4.2 Trace Construction

Next, we describe how the concrete execution traces $\mathcal{E}(p, I_{\text{syntactic}})$ were collected when running the enclave program with each given input $I_i \in I_{\text{syntactic}}$, and also describe how we construct various ED-CFG representations (e.g., \mathcal{G}^i , \mathcal{G}_p^i , and \mathcal{G}_c^i) that are suitable for the vulnerability identification from $\mathcal{E}(p, I_{\text{syntactic}})$.

Trace Collection. ANABLEPS requires collecting information regarding both the execution order and time of an execution trace. There are a variety of approaches to collecting these traces, such as using Intel Processor Trace (PT) and Last Branch Records (LBR). The issue with LBR is that it only has limited number of entries and branch records can be lost if not collected in timely manners. Therefore, we use Intel PT to conduct dynamic analysis. Intel PT is a hardware feature available on recent Intel processors (i.e., Broadwell or later families) to facilitate program debugging and performance profiling. It collects the information of the CFTs of a program with very small performance overhead. A useful feature of PT is that it also records timestamps together with the CFTs, and thus it perfectly fits the purpose of our design.

Although PT provides timestamps information of control flow transfers, it does not provide fine-grained time information of each execution unit, e.g., an instruction. Moreover, because the Cycle Count (CYC) packets are generated right before the event packets such as Taken NotTaken (TNT) packets, which may include taken or not taken information of up to 6 consecutive conditional branches, precisely recording execution time is not even possible at the basic-block granularity. As such, ANABLEPS only approximates the execution time in its construction of ED-CFGs, which will be detailed later. Also, ANABLEPS sets the memory buffer large enough so that no packet is lost during the dynamic analysis. The recorded packets are then parsed and recorded in a log file, which will be used for ED-CFG construction.

ED-CFG Construction. We generate the ED-CFG \mathcal{G}^i , for a given input I_i and trace r_i , based on the execution order and time of each basic block tracked in the trace files by Intel PT. Eventually, a D-CFG will be firstly built according to the PT trace file, where each node represents a basic block, each edge represents the CFT between the blocks, and the weight of each edge represents how many times the corresponding CFT has been executed.

Next, we add the execution order and time information into D-CFG to make it become ED-CFG, namely \mathcal{G}^i , for input I_i . More specifically, in each node, we use two lists to record the execution order and time for every basic block. The order list records the next node to jump to, and time list records the execution time every time when current node gets executed. The execution order is acquired by traversing the PT trace file again. However, for the execution time of each basic block each time when it gets executed, we have to approximate it (get a lower bound and upper bound) since PT does not offer fine-grained time recording for each basic block.

Resolving the execution time for each basic block. To get the execution time for basic blocks, we have to rely on the CYC packet, which is generated before each Mini Timestamp Counter (MTC) packet, TNT and Target IP (TIP) packet. However, not all of CFTs between basic blocks will generate a CYC packet as one TNT packet can capture up to six ba-

asic block execution. Therefore, we have to approximate the execution time for each basic block.

We take the following strategies to estimate the upper bound and lower bound of the CPU cycles for a basic block. The upper bound is an over-estimated execution time for each basic block with the CPU cycles recorded in the CYC packet, and the lower bound is the shortest CPU cycles in theory.

- **Upper Bound.** The upper bound of a basic block execution time is the CPU cycles recorded in the CYC packet, regardless of the number of basic blocks the CYC packet has covered.
- **Lower Bound.** The lower bound of a basic block execution time is the sum of the latency of the instructions that belong to the basic block. The latency for each individual instruction is acquired from [4].

While we cannot provide precise estimate of the execution time for each basic block, fortunately, we will get the precise PT recorded information for many of the basic blocks when we merge them to generate \mathcal{G}_p^i and \mathcal{G}_c^i , based on page or cache level execution unit if the basic blocks recorded by the TNT packets actually belong to these execution units.

\mathcal{G}_p^i and \mathcal{G}_c^i Generation. Once we have built \mathcal{G}^i for each input I_i , next we would like to derive \mathcal{G}_p^i and \mathcal{G}_c^i such that our vulnerability identification can be performed. Since the difference between page level execution unit and cache level execution unit is only the address alignment is different (2^{12} vs. 2^8), in the following we just describe how we convert \mathcal{G}^i to \mathcal{G}_p^i (\mathcal{G}^i to \mathcal{G}_c^i is similarly converted).

The conversion is straightforward, we need to combine all the basic block nodes that belong to the same page into just a single page node, and add the corresponding edges when there is a CFT between the pages. Also, we have to split the basic block that crosses two pages. To make our algorithm simple, we just first get all of the page numbers for all of the executed basic blocks by traversing \mathcal{G}^i , and then we traverse \mathcal{G}^i again to add the edges between the pages, and to add orders and timing on the page node. Especially, for timing information, we discard our lower and upper bound timing estimation for each basic block that was captured by the TNT packet if they all belong to the same page.

We take a two step approach to convert \mathcal{G}^i to \mathcal{G}_p^i . The first step is to generate the corresponding node and edge for \mathcal{G}_p^i by traversing \mathcal{G}^i , and the second step is to traverse \mathcal{G}^i again to generate the execution order and timing information.

- **Generating nodes and edges.** We design an algorithm shown in algorithm 1 to illustrate this. At a high level, we need to combine all the basic block nodes that belong to the same page into just a single page node, and add the corresponding edges when there is a control flow transfer between the pages. Also, we have to split the basic block that crosses two pages. To make our algorithm simple,

Algorithm 1: Generating the nodes and edges for \mathcal{G}_p^i from \mathcal{G}^i

```

begin
   $\mathcal{G}_p^i.N \leftarrow \emptyset$ 
   $\mathcal{G}_p^i.E \leftarrow \emptyset$ 
  foreach  $n \in \mathcal{G}^i.node()$  do
     $pg_{num} \leftarrow n.StartAddr() / 4096$ 
     $\mathcal{G}_p^i.N \leftarrow \mathcal{G}_p^i.N \cup \{pg_{num}\}$ 
    if  $n.StartAddr() / 4096 \neq n.EndAddr() / 4096$  then
       $pg_{next} \leftarrow n.EndAddr() / 4096$ 
       $\mathcal{G}_p^i.N \leftarrow \mathcal{G}_p^i.N \cup \{pg_{next}\}$ 
       $\mathcal{G}_p^i.E \leftarrow \mathcal{G}_p^i.E \cup \{ \langle pg_{num} - 1, pg_{num} \rangle \}$ 
       $w(pg_{num} - 1, pg_{num}) \leftarrow w(pg_{num} - 1, pg_{num}) + 1$ 
    end
  end
   $N_{tmp} \leftarrow \{ \mathcal{G}^i.Entry() \}$ 
  repeat
     $n \leftarrow head(N_{tmp})$ 
     $pg_{num} \leftarrow n.StartAddr() / 4096$ 
    foreach  $n_s \in n.successor()$  do
       $pg_{next} \leftarrow n_s.StartAddr() / 4096$ 
      if  $pg_{num} \neq pg_{next}$  then
         $\mathcal{G}_p^i.E \leftarrow \mathcal{G}_p^i.E \cup \{ \langle pg_{num}, pg_{next} \rangle \}$ 
         $w(pg_{num}, pg_{next}) \leftarrow w(pg_{num}, pg_{next}) + 1$ 
      end
    end
     $N_{tmp} \leftarrow N_{tmp} \setminus \{n\}$ 
     $N_{tmp} \leftarrow N_{tmp} \cup \{n.successor()\}$ 
  until  $N_{tmp} \neq \emptyset$ ;
  return  $\mathcal{G}_p^i$ 
end

```

we just first get all of the page numbers for all of the executed basic blocks by traversing \mathcal{G}^i , and then we traverse \mathcal{G}^i again to add the edges between the pages. The weights are updated accordingly when there is a cross-page control flow transfer.

- **Generating the order and timing.** Once we have generated the nodes and edges for \mathcal{G}_p^i , we then generate the order and timing information. The algorithm works similar to algorithm 1 with the differences that we need to record the new page order information, based on the original order recorded in \mathcal{G}^i while traversing \mathcal{G}^i . Also, for timing information, we will accumulate the recorded timing information of the basic blocks that belong to the same page based on the execution order. We will discard our lower and upper bound timing estimation for each basic block that was captured by the TNT packet if they all belong to the same page.

4.3 Vulnerability Identification

ANABLEPS detects both order-based and time-based side-channel vulnerabilities by cross comparing the corresponding ED-CFGs. More specifically, comparing \mathcal{G}_p s reveals vulnerabilities at the page-level, which can be exploited by an adversary that monitors the enclave program's page accesses (through page faults or page table entry updates). Comparing \mathcal{G}_c s reveal vulnerabilities at the cache-level, which can be ex-

exploited by an adversary that monitors the enclave program’s cache accesses. Directly comparing \mathcal{G} s reveal vulnerabilities at the basic-block level, which can be exploited by monitoring the branch prediction units [20]. In the following, we use \mathcal{G}_p as examples to illustrate the process of vulnerability detection.

Order-based Vulnerability Detection. We compare every \mathcal{G}_p^i with each other, the program is not vulnerable to page level attack if the order information of every edge been accessed in all \mathcal{G}_p^i s are the same. Otherwise, the attacker can infer the secret based on the differences. The algorithm for graph comparison is straightforward: $\mathcal{G}_p^i = \mathcal{G}_p^j$ if and only if the sets of node and edges are identical, including the `Order` list in each node, and the execution counts in the edges. In Figure 1(d)(e)(f), with different input, the execution order of the nodes are different. For instance, by comparing nodes n_1 in \mathcal{G}_p^1 and \mathcal{G}_p^2 , the length of their `Order` lists is different, which can clearly differentiate the two graphs.

Time-based Vulnerability Detection. When any two graphs \mathcal{G}_p^i and $\mathcal{G}_p^j, \forall i, j \in I$, are not vulnerable to order-based side channels. ANABLEPS needs to further investigate time-based vulnerabilities, by comparing the `Time` lists of the corresponding nodes. The comparison of the `Time` lists is as follows: The k^{th} element of $n_l.\text{Time}$ in node n_l in graph \mathcal{G}_p^i is compared with the k^{th} element of $n_l.\text{Time}$ in graph \mathcal{G}_p^j . However, unlike comparison of the `Order` lists, where any difference can directly conclude the comparison, comparing the `Time` lists is more subtle. The execution time of a program can be influenced by many reasons, such as on-demand paging, caching, interrupts, *etc.*. In practice, each $n_l.\text{Time}[k]$ is a 2-tuple (t_{mean}, t_{std}) , rather than a single value. The first element of the 2-tuple is the mean execution time to reach the successor node from multiple runs and the second element is the one standard deviation. With enough number of samples, the impact from side effects can be reduced.

To generate (t_{mean}, t_{std}) for the list `Time` of each node, the program is executed with the same input $I_i \in I$ L times; so each $n_l.\text{Time}[k]$ (the k^{th} element of n_l) is also executed L times. The mean and standard deviation are calculated using these L execution time between node n_l and its k^{th} successor. In our implementation, $L = 10$.

Determining the Input Space for \mathcal{G}^i . Since the edge in \mathcal{G}_p^i (and \mathcal{G}_c^i) can correspond to the jumps in different locations in the program, we can only use the one-to-one mapping relationship between \mathcal{G}^i and I_i to determine the input space for \mathcal{G}^i . In particular, for each concrete input $I_{\text{syntactic}} \in \{I_{\text{fuzz}} \cup I_{\text{concolic}}\}$, we run the concolic execution with this seed input again, but we also track the corresponding path constraints for this seed input. Once we have collected the path constraints, we then use a constraint solver to solve the constraints. If no other input satisfies (or the execution time of the solver takes too much time to solve.²), it means the input is unique (I_i is

completely leakable). Otherwise, we have to use application-specific knowledge to determine the leakage.

5 Evaluation

We have implemented ANABLEPS to detect the side-channel vulnerabilities for x86 and x86-64 ELF binaries by integrating and extending a number of open source tools. In particular, we extend Driller [30], which is built atop of AFL [1] and concolic execution, for *Input Generation*, and we use `perf` to configure Intel PT and dynamically collect the runtime information of each input. We built the PT packets decoder based on the open source library, `libipt` [3]. The ED-CFG construction and cross-comparison tool is built using python scripts by analyzing the PT packets, and matching the decoded address to the binary code with `pyelftools` library [5]. To quantify the input space for a given trace, we extended `angr` [36], an easily extensible python-based symbolic execution tool, to negate the constraints of the input we provide and calculate the input space. The prototype of ANABLEPS will be public available at github.com/OSUSecLab/ANABLEPS.

In this section, we present our evaluation results. We first describe how we set up the experiment in §5.1, and then describe the experimental results in §5.2. All of our evaluations are performed in Ubuntu Desktop 16.04LTS, running atop Intel i7-7700 CPU, with 32G physical memory.

5.1 Experiment Setup

Benchmark Selection. Ideally we would like to use the SGX programs for the test. However, there are not that many SGX programs available, and therefore we run the legacy applications with library OS (*e.g.*, Grephane-SGX [2]) support for the evaluation. In particular, we selected 8 programs from a variety of applications such as data analytics and machine learning, image processing, and text processing. The name of these programs is presented in the first column of Table 1.

Functionality Under Test. Each of the tested benchmark program contains quite sophisticated functionalities. Certainly, we cannot test all of their functionalities; we only tested the functionality of our interest (shown in the 2nd column of Table 1), based on our best understanding with the benchmarks. For instance, when testing `Genometools`, we know the genomic related program usually takes two types of input: `bed` format and `gff3` format. Converting between these two formats is a widely used operation in genomes. Therefore, we test the genome library `libgenometools.so` by converting `bed` format to `gff3` format.

Input Generation. To launch each of the testing program with Driller [30], we provide the seed inputs based on our best understanding of the program. Even with both AFL and concolic execution, we still cannot explore all the program paths. We therefore configure Driller [30] to run 48 hours for each

²We currently set up this time to be 90 minutes.

of the testing program. The number of syntactic inputs eventually generated are presented in the 3rd column of Table 1.

Trace Collection. With the input generated above, we run the tested program traced by Intel PT. The tested program is run outside of SGX in a debug mode. The execution time would be similar to that of executing inside enclaves, because instructions executed in the enclave-mode and non-enclave-mode have the same timing constraints (the main timing difference happens at ECalls/OCalls). Each input generated a separate trace file. The total size of the decoded PT trace file for each program is presented in the 4th column of Table 1. Depending on the size and input to the program, this size varies from a few Gigabytes to several hundreds of Gigabytes.

5.2 Experimental Results

Next, we present how ANABLEPS detects the branch level, page level, and cache level side-channel vulnerabilities based on each individual trace and their corresponding input. As we have described, from each input (and its corresponding execution trace), we first built their ED-CFGs, namely \mathcal{G}^i s, which are used to detect the branch level side channels. The total number of such ED-CFGs is presented in the 5th column of Table 1. Compared to the 3rd column of Table 1, we can notice that except for three benchmarks (namely Freetype, QRcodegen, and Genometools), the total number of unique \mathcal{G}^i s are all smaller than the total number of the syntactic inputs generated by ANABLEPS.

Detecting Order-based Side Channels. To detect order-based side channels, we first cross-compare all of the \mathcal{G}^i s (\mathcal{G}_p^i s or \mathcal{G}_c^i s) to detect whether there is any unique I_i that maps to a particular \mathcal{G}^i (\mathcal{G}_p^i or \mathcal{G}_c^i). As we are detecting order-based side channels, only `Order` of the \mathcal{G}^i s (\mathcal{G}_p^i s or \mathcal{G}_c^i s) are used in the comparison. Many inputs have such a one-to-one mapping $\mathcal{G}^i \leftrightarrow I_i$ ($\mathcal{G}_p^i \leftrightarrow I_i$ or $\mathcal{G}_c^i \leftrightarrow I_i$), which suggests that no other input I_j maps to the same \mathcal{G}^i . The branch-level, page-level and cache-level statistics for this mapping is reported in the 6th column of Table 1, the 3rd column of Table 2, and the 8th column of Table 2, respectively. From the table, we can notice that compared to the branch-level vulnerabilities, less one-to-one mappings are detected in page-level and cache-level. For instance, while all inputs of `dA` in deep learning can be recovered by branch-level side channel, they cannot be recovered by page-level side channels.

As the traces are dynamically collected, the node or edge which can differ any two \mathcal{G}^i s (\mathcal{G}_p^i s or \mathcal{G}_c^i s) must leak some secret of interest. It is possible that many nodes or edges only leak a partial secret. However, for some program, a set of vulnerable nodes can be used together to leak the entire secret (e.g., the Deep Learning case uses two nodes to leak the entire secret). Moreover, it is also possible that part of leaked secret can be used to infer the entire secret (e.g., the padding oracle

attack for crypto algorithms only need to know if the padding is correct or not).

Detecting Time-based Side Channels. For those that have multiple inputs corresponding to the same trace, i.e., one-to-N mappings ($\mathcal{G}^i \rightarrow I_{is}$, $\mathcal{G}_p^i \rightarrow I_{is}$ or $\mathcal{G}_c^i \rightarrow I_{is}$), their statistics are reported in the 8th column of Table 1, the 4th column of Table 2, and the 9th column of Table 2, respectively. Next, we use the timing information to further differentiate \mathcal{G}^i (\mathcal{G}_p^i and \mathcal{G}_c^i) and see whether there is still one-to-one mapping (i.e., $\mathcal{G}^i \leftrightarrow I_i$) after considering the timing differences. That is, we hope to determine whether there are time-based side-channel vulnerabilities when the program is not vulnerable to order-based side channels. In practice, only large enough time differences can be used to differentiate two traces. Therefore, thresholds are defined from empirical results. We report under three different threshold settings (i.e., with $t_1 = 2ns$, $t_2 = 10ns$, and $t_3 = 20ns$), the number of such one-to-one mappings, and these results are reported in the last three columns of Table 1, the columns 5 to 7 and columns 10 to 12 of Table 2. We notice that it is relative hard to differentiate inputs based on timing information at branch level. However, many inputs can be further differentiated after applying time information at page or cache level.

Determining Input Spaces. Previous experiments are based on generated inputs I_{is} . However, not all inputs in the whole inputs set are generated. Therefore, we would like to know whether there is only one input I_i in the whole inputs set that can map to a particular \mathcal{G}^i , that is, if $|\{I_j | \mathcal{E}(p, I_j) = \mathcal{G}^i, \forall j \in I\}| = 1$, which can be determined by using concolic execution. If so, then the input I_i can be differentiated by order-based vulnerabilities. The total number of symbolic execution determined input $I_{deterministic}$ is reported in the 7th column of Table 1. We can see that for some applications, such as QRcodegen and Deep learning, $I_{deterministic}$ is non-zero, meaning at branch-level some inputs of these programs can be uniquely identified by execution traces. For some applications, $I_{deterministic}$ is zero, indicating by the constraint solver that there are other inputs that all have the same execution traces with generated inputs, e.g., function `Sort` in `gsl`, although $|\mathcal{G}^i \leftrightarrow I_i|$ is non-zero (120 for `gsl`).

However, the concolic execution cannot finish for five programs (marked with \times in the Table), including Hunspell, PNG, and Freetype, because of the limitation in either computation power or physical memory space. For these programs, ANABLEPS cannot answer if these execution traces will completely leak the information of the input.

5.2.1 Performance Overhead

We also measured the performance of ANABLEPS, though it is an offline analysis tool. We report the execution time for each of the key component of ANABLEPS in Table 3. More specifically, during the Input Generation (IG) phase,

Benchmark Program	Functionality under Test	$ I_{\text{syntactic}} $	Trace Size (GB)	$ G^i $	Detecting Branch Side Channel					
					$ G^i \leftrightarrow I_i $	$ I_{\text{deterministic}} $	$ G^i \rightarrow I_i s $	$ G^i \leftrightarrow I_i $		
								t_1	t_2	t_3
Deep Learning	dA	214	76.8	214	214	214	0	-	-	-
	SdA	176	384.2	176	176	176	0	-	-	-
	DBN	152	139.0	152	152	152	0	-	-	-
	RBM	187	225.9	55	16	0	39	56	43	0
	LogisticRegression	198	25.1	41	18	0	23	31	5	0
gsl	Sort	220	2.8	154	120	0	34	0	0	0
	Permutation	200	3.0	135	100	×	35	0	0	0
Hunspell	Spell Checking	231	307.2	168	157	×	11	1	0	0
PNG	Image Render	294	82.3	135	120	×	15	0	0	0
Freetype	Character Render	206	352.6	206	206	×	0	-	-	-
Bio-rainbow	Bioinfo Clustering	128	51.3	119	118	0	1	0	0	0
QRcodegen	Generate QR Code	204	17.9	204	204	204	0	-	-	-
Genometools	bed to gff3 conversion	201	382.4	25	12	×	13	0	0	0

Table 1: The benchmark programs, their concrete input size, the corresponding PT trace size, and the result of branch level side channel detection

Benchmark Programs	Functionality Under Test	Detecting Page Side Channel						Detecting Cache Side Channel					
		$ G_p^i \leftrightarrow I_i $	$ G_p^i \rightarrow I_i s $	$ G_p^i \leftrightarrow I_i $			$ G_c^i \leftrightarrow I_i $	$ G_c^i \rightarrow I_i s $	$ G_c^i \leftrightarrow I_i $				
				t_1	t_2	t_3			t_1	t_2	t_3		
Deep Learning	dA	127	12	65	52	9	214	0	-	-	-		
	SdA	112	5	33	28	0	176	0	-	-	-		
	DBN	128	9	15	11	0	152	0	-	-	-		
	RBM	28	15	56	43	0	55	16	56	43	0		
	LogisticRegression	6	9	82	24	0	18	23	82	24	0		
gsl	Sort	17	12	0	0	0	33	16	0	0	0		
	Permutation	100	2	0	0	0	100	2	0	0	0		
Hunspell	Spell Checking	156	11	5	2	2	157	11	7	7	7		
PNG	Image Render	103	25	1	1	1	111	22	10	6	0		
Freetype	Character Render	206	0	-	-	-	206	0	-	-	-		
Bio-rainbow	Bioinfo Clustering	39	9	1	0	0	118	1	0	0	0		
QRcodegen	Generate QR Code	204	0	-	-	-	204	0	-	-	-		
Genometools	bed to gff3 conversion	5	8	5	5	3	5	8	5	5	3		

Table 2: The page level and cache level vulnerability detection results for the tested benchmark programs

Benchmark Programs	Functionality Under Test	IG (h)	TC (h)	VI (m)	CS (h)
Deep Learning	dA	48	26.1	7.9	31.3
	SdA	48	187.2	61.7	132.1
	DBN	48	63.3	43.8	82.3
	RBM	48	110.1	13.2	45.9
	LogisticRegression	48	7.2	1.8	8.4
gsl	Sort	48	0.62	0.2	2.5
	Permutation	48	0.57	0.2	-
Hunspell	Spell Checking	48	68.2	4.4	-
PNG	PNG Image Render	48	19.8	1.6	-
Freetype	Character Render	48	87.4	19.8	-
Bio-rainbow	Clustering bioinformatics	48	14.2	20.9	58
QRcodegen	Generate QR Code	48	8.89	22.4	126
Genometools	bed to gff3 conversion	48	192.6	15.2	-

Table 3: Performance overhead for running each component of ANABLEPS the tested programs. IG stands for Input Generation, TC stands for Trace Construction, VI stands for Vulnerability Identification, and CS stands for Constraint Solver

we configured ANABLEPS to run 48 hours for all of the benchmarks. Then, our Trace Construction (TC) component decodes the trace, builds each G^i , G_p^i , and G_c^i . For the Vulnerability Identification (VI), ANABLEPS just performs the cross-comparison with the graphs we have built. Only when detecting the branch-level side channel, we invoke Constraint Solver (CS) to determine whether there is a unique input for a specific trace. This execution time is reported in

the last column of Table 3. For certain programs that concolic execution cannot finish (marked with ‘-’ in the Table), we cannot evaluate their performance overhead. We can notice that the bottleneck of the ANABLEPS is Trace Construction and Constraint Solver, which are affected by the size of execution trace files and computation power.

6 Exploitability of the Vulnerability

So far, we have discussed the design, implementation and evaluation of ANABLEPS in automatically detecting order and time based side-channel vulnerabilities. However, automated tools can only provide *syntactic*-level analysis. Oftentimes, such analysis cannot be directly translated into exploitability of the program, especially when the input space of interest (to the attackers) cannot be automatically determined. In this section, we discuss how ANABLEPS can be used by enclave program developers to analyze the exploitability of the vulnerabilities by providing the proper input, locating and exploiting the vulnerabilities.

6.1 Developer-assisted Vulnerability Analysis

Developer-supplied Input. While the automated syntactic analysis has provided a large number of inputs, not all of them are of interest to attackers. For instance, in the PNG example, not all input correspond to valid images; it is not very interesting to determine the errors in the PNG file formats. In practice, only software developers are able to identify the true secretive set of input that they would like to make indistinguishable. This is called *semantic-level analysis*. Developers may select the set of inputs I that she wishes to be indistinguishable from the execution traces and use ANABLEPS to analyze $\mathcal{E}(p, I)$.

The steps to perform such an analysis is similar to automated analysis described in §4. The only difference is that the *Input Generation* step and “determining input spaces for \mathcal{G}^i ” of the *Vulnerability Identification* step can be skipped, as the set of input of interest is now provided by the developers. The output of the analysis would be $|\mathcal{G}^i \leftrightarrow I_i|$ that can be differentiated by order or time information of the execution traces.

Locating Vulnerabilities. Given a secretive set of input I , if $|\mathcal{E}(p, I)| > 1$, we would like to find the set of nodes in \mathcal{G} that can be used to learn the inputs. That is, we would like to locate the vulnerabilities (*i.e.*, vulnerable node) in the graph and the program. With the method discussed in §4, we can differentiate *order-based vulnerable nodes* and *time-based vulnerable nodes*. The capability to easily locate vulnerabilities is one benefit of adopting ED-CFG to represent execution traces.

With the input set of $I_{\text{syntactic}}$, the statistics of the vulnerable nodes are shown in Table 4. The cache-level statistics are listed in the column 3 to 5, and the page-level statistics are reported in column 6 to 8. The total numbers of nodes in \mathcal{G}_c^i and \mathcal{G}_p^i are shown in column 3 and 6; the numbers of order-based vulnerable nodes are listed in column 4 and 7; and the numbers of time-based vulnerable nodes are listed in column 5 and 8, respectively. In the Table 4, the time-based vulnerable nodes are mutually exclusive with the order-based vulnerable nodes. According to the results presented in Table 4, ANABLEPS narrows down the number of nodes to be examined for side-channel vulnerabilities dramatically. On average, the number of order-based vulnerable nodes is only 18% of all nodes in \mathcal{G}_c , and 37% of all nodes in \mathcal{G}_p ; the number of time-based vulnerable nodes is only 6% of all nodes in \mathcal{G}_c , and 13% of all nodes in \mathcal{G}_p . The fraction of vulnerable nodes can be further reduced with a developer-supplied input set that is of interest.

6.2 Case Studies of the Exploitability Analysis

In this section, we briefly summarize three interesting cases to show how ANABLEPS can help enclave developers identify side-channel vulnerabilities that can be exploited to extract sensitive information.

```
1 int binomial(int n, double p) {
2     ...
3     for(i=0; i<n; i++) {
4         r = rand() / (RAND_MAX + 1.0);
5         if (r < p) c++;
6     }
7     ...
8 }
11 void dA_get_corrupted_input(dAA this, int Ax, int Atilde_x, double p)
12 {
13     int i;
14     for(i=0; i<this->n_visible; i++) {
15         if(x[i] == 0) {
16             tilde_x[i] = 0;
17         } else {
18             tilde_x[i] = binomial(x[i], p);
19         }
20     }
21 }
```

Figure 2: The deep learning vulnerable code

6.2.1 Deep Learning Algorithms

According to Table 2, there are 214 different inputs for algorithm dA that has unique \mathcal{G}_c , *i.e.*, $\mathcal{G}_c^i \leftrightarrow I_i$. Therefore, potentially the vulnerabilities in dA may lead to exploitable information leakage. In order to start analyzing the vulnerabilities in dA algorithm, we first manually selected inputs that might be of interest to attackers: a set of $|I|$ training data that differ only in values. Then, we feed these inputs to ANABLEPS. The output of ANABLEPS indicates that all selected inputs have unique cache-level execution traces, *i.e.*, $|\mathcal{E}(p, I)| = |I|$.

After locating the vulnerable nodes and some manual effort to examine the identified vulnerable nodes, we find the leakage primarily comes from function `dA_get_corrupted_input()`, which has a for loop that enumerates every element of array `x` and calls function `binomial()` if the element is not 0. The code snippet is shown in Figure 2.

The execution of `dA_get_corrupted_input()` and `binomial()` may be exploited to leak training data information. Whether or not function `binomial()` is called by `dA_get_corrupted_input()` reveals the value of array `x`. The function call sequence can be learned through cache-level side channels. The two functions are located in the same page but different cachelines. After compilation, the for loop in `dA_get_corrupted_input()` is compiled into two cachelines, denoted m_1 and m_2 , function `binomial()` is compiled into two consecutive cachelines. We denote the first cacheline as m_3 . Therefore, if the i^{th} element of array `x` is 0, the order of the executed cachelines is $[m_1, m_2]$; otherwise, the execution order becomes $[m_1, m_2, m_3, m_2]$. This order-based side-channel vulnerability on the cache-level can completely leak the training data of the deep learning algorithm.

6.2.2 Freetype Font Engine

According to Table 2, there are 206 inputs that have unique \mathcal{G}_p^i . To validate the page-level vulnerability, we generated some printable characters as input and fed them to ANABLEPS. The result indicates that every input corresponds to a unique \mathcal{G}_p^i .

Programs	Functionalities Under Test	Cache Level			Page Level		
		#Nodes	#Order-Based Vulnerable Nodes	#Time-Based Vulnerable Nodes	#Nodes	#Order-Based Vulnerable Nodes	#Time-Based Vulnerable Nodes
Deep Learning	dA	69	9	4	13	2	3
	SdA	109	12	21	22	3	3
	DBN	126	17	81	14	3	10
	RBM	68	8	27	13	2	7
	LogisticRegression	48	2	16	11	0	7
gsl	Sort	31	12	0	11	5	0
	Permutation	99	30	0	29	15	0
Hunspell	Spell checking	302	48	9	47	27	10
PNG	PNG Image Render	640	170	90	53	39	2
Freetype	Character Render	1054	263	18	82	20	13
Bio-rainbow	Bioinfo Clustering	214	16	0	24	2	1
QRcodegen	Generate QR	176	32	18	15	6	3
Genometools	bed to gff3 conversion	1901	231	9	147	53	5

Table 4: Locating vulnerable nodes in \mathcal{G}_c and \mathcal{G}_p

```

1 static void psh_glyph_interpolate_strong_points (...) {
2     ...
3     for (; count>0; count--, point++){
4         ...
5         if (psh_point_is_edge_min(point))
6             point->cur_u = ...;
7         else if (psh_point_is_edge_max(point))
8             point->cur_u = ...;
9         else {
10            data = ...;
11            if (delta <= 0)
12                point->cur_u = FT_MulFix(...) + ...;
13            else if (delta >= hint->org_len)
14                point->cur_u = FT_MulFix(...) + ...;
15            else
16                point->cur_u = FT_MulDiv(...) + ...;
17        }
18        psh_point_set_fitted(point);
19    }
20 }

```

Figure 3: The freetype vulnerable functions

ANABLEPS has helped us identify the vulnerable nodes. In fact, there are more than one vulnerable nodes. To illustrate these vulnerabilities, we explain the leakage through function `psh_glyph_find_strong_points()` at the page level. The code snippet is shown in Figure 3. `psh_glyph_interpolate_strong_points()` includes a loop to interpolate every strong point into the glyph. Adversaries can recover the strong points position according to the page sequence. More specifically, function `psh_point_is_edge_min()` is placed in page m_1 . Functions `FT_MulFix()` and `FT_MulDiv()` are placed in another page, denoted m_2 . The page of function `psh_glyph_interpolate_strong_points()` is denoted m_3 . The access order of these pages leaks information of the interpolated point: When a point is not marked as a strong point, the order of page access is $[m_3]$; when the strong point is located in the edge, the order of page access is $[m_3, m_1, m_3, m_1, m_3]$; otherwise, the sequence would be $[m_3, m_1, m_3, m_2, m_3, m_1, m_3]$. Given the sequence of this function, the attacker can learn whether each point is strong or not. Though the example does not completely leak the content of the data, it illustrates how leakage can be identified.

```

1 8050920 <get_parser>:
2 ...
3 8050b05: 89 1c 24      mov  %ebx,(%esp)
4 8050b08: 83 c3 02      add  $0x2,%ebx
5 8050b0b: e8 e0 63 00 00 call _unguarded_linear_insert+
6 8050b10: 39 de        cmp  %ebx,%esi
7 8050b12: 75 f1        jne  8050b05 <get_parser+0x1e5>
8 ...
11 8056ef0 <_unguarded_linear_insert>:
12 ...
13 8056f28: 89 c2        mov  %eax,%edx
14 8056f2a: 89 d8        mov  %ebx,%eax
15 8056f2c: 0f b7 18     movzwl (%eax),%ebx
16 8056f2f: 66 89 1a     mov  %bx,(%edx)
17 8056f32: 0f b6 50 ff  movzbl -0x1(%eax),%edx
18 8056f36: 8d 58 fe     lea  -0x2(%eax),%ebx
19 8056f39: 0f b6 70 fe  movzbl -0x2(%eax),%esi
20 8056f3d: c1 e2 08     shl  $0x8,%edx
21 8056f40: 01 f2        add  %esi,%edx
22 8056f42: 66 39 ca     cmp  %cx,%dx
23 8056f45: 77 e1        ja   8056f28 <
...
24 _unguarded_linear_insert+0x38>

```

Figure 4: The assembly code of `std::sort`

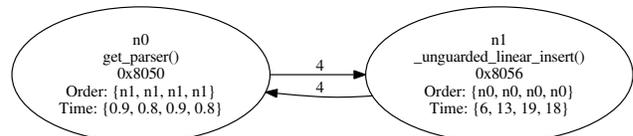


Figure 5: A subgraph of \mathcal{G}_p for function `std::sort`

6.2.3 Hunspell

Hunspell is a popular spell checker. Xu *et al.* identified that Hunspell is vulnerable to page-level controlled channel attacks due to its input-dependent access pattern to data pages [43]. But its control flow was considered immune to side-channel attacks. However, as shown in Table 2, ANABLEPS identifies various control-flow side-channel vulnerabilities that may be exploited by attackers.

With the help of ANABLEPS, we narrow down our attention to the `get_parser()` function of Hunspell, in which the function `std::sort(vector.begin(), vector.end())` is called to sort the data in the vector. We found this function both have cache-level and branch-level order-based vulnerability and page-level time-based vulnerability. This is a function implemented in C++ standard library. After compilation,

the linear insertion algorithm is used in this sort function with the snippet of assembly code in Figure 4. According to the code snippet of function `get_parser()`, the function `_unguarded_linear_insert()` is called when an element in the unsorted vector is to be inserted into the sorted vector. As such, by monitoring the execution sequence that involves this function, the attacker is able to learn the number of elements to be sorted in page-level, cache-level and branch-level. Moreover, function `_unguarded_linear_insert()` contains a loop to compare the element to be inserted with elements already in the sorted vector. According to the insertion sort algorithm, the number of loops in function `_unguarded_linear_insert()` reflects the number of comparisons during the insertion, which can be used to infer the location of an element after the insertion.

Such leakage can be easily identified in \mathcal{G}_p with time-based vulnerability. A subgraph of \mathcal{G}_p^i of a particular input I_i is shown in Figure 5. The edge $n_0 \rightarrow n_1$ is executed 4 times, which reflects that four elements are being sorted. The elements of `Time` list in node n_1 reveals the number of comparisons in function `_unguarded_linear_insert()`: the first element corresponds to no comparison, the second element corresponds to 1 comparison, the third and fourth elements correspond to 2 comparisons. Therefore, the page level order-based vulnerability in Hunspell, or more precisely the `sort` algorithm implemented in the standard C++ library, can only leak the number of elements to be sorted; however, the time-based vulnerability can be exploited to leak the list to be sorted if sorting result is known. We specially tested the `sort` algorithm by providing a set of $|I|$ unsorted lists that correspond to the same sorted list after sorting. As expected, ANABLEPS reports $|\mathcal{E}(p, I)| = |I|$ for this set of inputs.

7 Limitations and Future Work

Although we have demonstrated that ANABLEPS is capable of identifying side-channel vulnerabilities in enclave binaries, we only made a first step and there are a number of avenues for future works. First, the currently design only considers side-channel vulnerabilities due to secret-dependent control flows. Leakages due to secret-dependent data accesses are out of scope currently. Interestingly, the differences in the data access pattern caused by divergence in the control flow can actually be identified by ANABLEPS's control-flow based vulnerability analysis. What is missed by ANABLEPS is memory pointers or array indexes that are determined by the secret values. One of the future works is to extend ANABLEPS in handling of these vulnerabilities.

Second, while ANABLEPS has integrated the state-of-the-art input generation tools such as fuzzing and concolic execution, it still cannot generate the complete set of input. Currently, we rely on developers' knowledge to remediate this limitation since developers have the best understanding of the semantic of the enclave program and its input space. Certainly,

any advances in the research of test case generation itself will improve ANABLEPS.

Third, the capability of the constraint solver is limited. Given an input to a program, ANABLEPS relies on symbolic execution to collect constraints. These constraints are solved by a constraint solver to determine the size of \mathcal{G}^i 's input space. However, not all the constraints can be solved (e.g., hash functions). Also, a solver may take too much time to solve a constraint. Currently, ANABLEPS requires the solver to return the result in 90 minutes. Otherwise, it considers unsolvable. Any advancement of constraint solver will make ANABLEPS more efficient.

8 Conclusion

In conclusion, we designed and implemented ANABLEPS, a software tool for automatically vetting side-channel vulnerabilities in SGX enclave programs. ANABLEPS is the first side-channel vulnerability analysis tool that considers both time and order of a program's memory access patterns. It leverages concolic execution and fuzzing techniques to generate input sets for an arbitrary enclave program, constructs extended dynamic control-flow graph representation of execution traces using Intel PT, and automatically analyzes and identifies side-channel vulnerabilities using graph analysis. With ANABLEPS, we have uncovered a large number of side channel leaks in enclave binaries we tested. Our experimental results also demonstrate ANABLEPS can be used by both security analysts and software developers to identify the side-channel vulnerabilities for enclave programs.

Acknowledgments

We would like to thank the anonymous reviewers for their very helpful comments. This work was supported in part by the NSF grants 1750809, 1718084, 1834213, 1834215, and 1834216 as well as a research gift from Intel.

References

- [1] american fuzzy lop. <http://lcamtuf.coredump.cx/afl/>. (Accessed on 04/28/2018).
- [2] Graphene / graphene-SGX library os - a library os for linux multi-process applications, with intel SGX support. <https://github.com/oscarlab/graphene>.
- [3] libipt - an intel(r) processor trace decoder library. <https://github.com/01org/processor-trace>.
- [4] Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. https://www.agner.org/optimize/instruction_tables.pdf.
- [5] pyelftools - parsing elf and dwarf in python. <https://github.com/eliben/pyelftools>.

- [6] Intel® software guard extensions enclave writer's guide. <https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf>, 2017. Revision 1.02, Accessed May, 2017.
- [7] E. Bauman and Z. Lin. A case for protecting computer games with SGX. In Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16), December 2016.
- [8] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. ACM Transactions on Computer Systems (TOCS'15), 2015.
- [9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostianen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In USENIX Workshop on Offensive Technologies, 2017.
- [10] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Stealing intel secrets from SGX enclaves via speculative execution. In Proceedings of the 2019 IEEE European Symposium on Security and Privacy, June 2019.
- [11] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin. Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races. In 2018 IEEE Symposium on Security and Privacy (SP'18). IEEE, 2018.
- [12] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In 12th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17). ACM.
- [13] V. Costan and S. Devadas. Intel SGX explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 20 16. <http://eprint.iacr.org>.
- [14] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'17), September 2017.
- [15] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel SGX. In 10th European Workshop on Systems Security (EuroSec'17). ACM, 2017.
- [16] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In USENIX Security Symposium (USENIX Security'17). USENIX Association, 2017.
- [17] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In 2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, 2017.
- [18] S. M. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing security and privacy of tor's ecosystem by using trusted execution environments. In (NSDI'17), 2017.
- [19] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. ArXiv e-prints, Jan. 2018.
- [20] 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. In 26th USENIX Security Symposium (USENIX Security'17). USENIX Association, 2017.
- [21] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Melt-down. ArXiv e-prints, Jan. 2018.
- [22] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, 2018.
- [23] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In 6th Cryptographers' track at the RSA conference on Topics in Cryptology, 2006.
- [24] C. Percival. Cache missing for fun and profit. In 2005 BSDCan, 2005.
- [25] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using SGX. In 2015 IEEE Symposium on Security and Privacy (SP'15). IEEE, 2015.
- [26] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using SGX to conceal cache attacks. Springer International Publishing, 2017.
- [27] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-shield: Enabling address space layout randomization for SGX programs. In In Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), 2017.
- [28] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS'17), 2017.
- [29] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS'16). ACM, 2016.
- [30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In (NDSS'16), 2016.
- [31] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library os for unmodified applications on SGX. In Proceedings of the USENIX Annual Technical Conference (ATC'17), 2017.
- [32] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In 27th USENIX Security Symposium (USENIX Security'18). USENIX Association, 2018.
- [33] J. Van Bulck, F. Piessens, and R. Strackx. SGX-step: A practical attack framework for precise enclave execution control. In Proceedings of the 2Nd Workshop on System Software for Trusted Execution, (SysTEX'17), 2017.

- [34] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18). ACM, 2018.
- [35] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In Proceedings of the 26th USENIX Security Symposium (USENIX Security'17). USENIX Association, 2017.
- [36] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17), 2017.
- [37] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu. Cached: Identifying cache-based timing channels in production software. In 26th USENIX Security Symposium (USENIX Security'17). USENIX Association, 2017.
- [38] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bind-schaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, (CCS'17). ACM, 2017.
- [39] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. DATA – differential address trace analysis: Finding address-based side-channels in binaries. In 27th USENIX Security Symposium (USENIX Security'18). USENIX Association, 2018.
- [40] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar. Microwalk: A framework for finding side channels in binaries. In Proceedings of the 34th Annual Computer Security Applications Conference. ACM, 2018.
- [41] J. C. Wray. An analysis of covert timing channels. J. Comput. Secur., 1992.
- [42] Y. Xiao, M. Li, S. Chen, and Y. Zhang. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, (CCS'17). ACM, 2017.
- [43] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP'15). IEEE, 2015.
- [44] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16). ACM, 2016.

Application level attacks on Connected Vehicle Protocols

Ahmed Abdo
*Department of Electrical
and Computer Engineering
University of California,
Riverside*
Email: aabdo003@ucr.edu

Sakib Md Bin Malek
*Department of Computer
Science and Engineering
University of California,
Riverside*
Email: sbin003@ucr.edu

Zhiyun Qian
*Department of Computer
Science and Engineering
University of California,
Riverside*
Email: zhiyunq@cs.ucr.edu

Qi Zhu
*Department of Electrical
and Computer Engineering
Northwestern University*
Email: qzhu@northwestern.edu

Matthew Barth
*Department of Electrical
and Computer Engineering
University of California,
Riverside*
Email: barth@ee.ucr.edu

Nael Abu-Ghazaleh
*Department of Computer
Science and Engineering
University of California,
Riverside*
Email: naelag@ucr.edu

Abstract

Connected vehicles (CV) applications are an emerging new technology that promises to revolutionize transportation systems. CV applications can improve safety, efficiency, and capacity of transportation systems while reducing their environmental footprints. A large number of CV applications have been proposed towards these goals, with the US Department of Transportation (US DOT) recently initiating three deployment sites. Unfortunately, the security of these protocols has not been considered carefully, and due to the fact that they affect the control of vehicles, vulnerabilities can lead to breakdowns in safety (causing accidents), performance (causing congestion and reducing capacity), or fairness (vehicles cheating the intersection management system). In this paper, we perform a detailed analysis of a recently published CV-based application protocol, Cooperative Adaptive Cruise Control (CACC), and use this analysis to classify the types of vulnerabilities that occur in the context of connected Cyber-physical systems such as CV. We show using simulations that these attacks can be extremely dangerous: we illustrate attacks that cause crashes or stall emergency vehicles. We also carry out a more systematic analysis of the impact of the attacks showing that even an individual attacker can have substantial effects on traffic flow and safety even in the presence of message security standard developed by US DOT. We believe that these attacks can be carried over to other CV applications if they are not carefully designed. The paper also explores a defense framework to mitigate these classes of vulnerabilities in CV applications.

1 Introduction

The United States Department of Transportation (US DOT) has been developing next-generation Intelligent Transporta-

tion Systems (ITS) [2] where vehicles and transportation infrastructure communicate and collaborate towards goals such as improving safety, increasing traffic flow capacity, supporting driver assistance functionality, and reducing overall carbon footprint [16]. Some of these technologies are already installed across the country such traffic signal coordination, transit signal priority, and traveler information systems.

One widely deployed early example of such functionality is Intelligent Traffic Signal Systems (I-SIG), which have been deployed in several cities, reducing the average traffic delay by 26.6% [24]. While I-SIG involves only making the infrastructure intelligent, another class of ITS applications involves vehicles communicating to coordinate with other vehicles and the infrastructure intelligently. The subset of ITS applications that involves vehicles communicating to each other (V2V) and the Infrastructure (V2I) are called *Connected Vehicles* (CV) applications. Many of the CV applications are starting to be prototyped and have reference implementations [25]. The US Department of Transportation (US DOT) has started testing applications in three deployment sites. Other experimental projects incorporating platooning are starting to emerge: e.g., a consortium of companies, universities and the Flemish government are building a test bed to experimentally test automated CV driving [1]. Tesla is also working on self-driving electric trucks that can move in platoons behind a designated lead vehicle [6].

In these initial stages where researchers and engineers are developing early prototypes of CV applications, security is not being considered deeply. CVs expose a large attack surface as an open systems with many participants and complex functionality: attacks may target application protocols, networking, sensing and vehicle control, with the potential to cause accidents, traffic delays and other harm to the system. A message security standard, the Secure Certificate Manage-

ment System (SCMS), has been defined by USDOT but it only ensures that cars and road side units have certificates that enable them to participate in communication [20].

Vulnerability and Attack Analysis: It is essential to understand the threats faced by CV protocols to understand how to design them securely. Towards this goal, this paper explores the vulnerabilities that arise at the application level of CV applications. We show that even when an attacker does not spoof or modify messages, it does not stop a malicious actor from obtaining a certificate, or a compromised participant with a valid certificate, from using it to falsify information in its messages. We present the threat model in Section 2. We conduct this analysis in the context of an important CV application called Cooperative Adaptive Cruise Control (CACC). CACC is used to group nearby cars into a platoon and adaptively control their speed. The vehicles in a platoon are subjected to reduced air drag as well as improvements in overall traffic flow, driving safety, capacity, and fuel economy. Section 3 introduces CACC. The application logic is complex, having to consider cases such as cars joining and leaving a platoon, merging and splitting of platoons, lane changes, and platoon leaders leaving. These maneuvers are triggered and coordinated through messages. An attacker can exploit this protocol by sending messages with false information leading to a number of possible attacks that reduce the safety, and performance of the system. In Section 4 we introduce five general classes of vulnerabilities that we believe that can be applied to networked cyber physical systems. We describe specific attacks against CACC in Section 5, showing a number of successful attacks even in the presence of SCMS.

Attack Demonstration and Evaluation: As CV systems are not deployed and/or generally available for public experimentation, to evaluate these attacks, we use a previously developed implementation of CACC in a state of the art vehicular simulator, VENTOS [9], that is widely used by practitioners and developers. We show scenarios where the vulnerabilities can be exploited to cause safety breakdowns or to interfere with an emergency vehicle. We define metrics for evaluating the attack impact that measures mobility (traffic throughput) and safety (average separation between cars). We show that attacks can substantially interfere with the operation of CACC leading to increased vehicular speeds and reduced safety margins. We present our results in Section 6.

Potential Mitigation: Having established these attacks on the CACC application level, we need to consider a mitigation framework in Section 7. We use the classification of the five vulnerability types we introduce to guide the design of the mitigation steps that either eliminate or interfere with them. We show that the defense indeed mitigates the vulnerabilities we identified in CACC without substantially harming performance.

2 Threat Model

We assume a CV application using Security Credentials Management System (SCMS) [7]. SCMS became available to coincide with the full-scale deployment of devices at three US DOT CV pilot sites (New York, Tampa, and Wyoming) [10–12]. The current implementation is a proof-of-concept Certificate-Based Authentication system that uses a Public Key Infrastructure [20] for certificate management. Pseudonym Certificates (PCs) are used and rotated to enable message authentication and validation without exposing the privacy of a vehicle by having a permanent certificate. A vehicle can enroll in the system by submitting an enrollment request to US DOT. PC can be obtained by vehicles for a short term, ranging from 5 minutes to few days, and is used for basic safety message (BSM) authentication. On Board Equipment (OBE) uses identification certificates to authenticate itself in V2I applications. However, none of the V2I applications we reviewed require encryption by the OBE at the application level.

SCMS prevents an attacker from falsifying messages from another vehicle as each message gets signed with a certificate. However, SCMS can not prevent a malicious actor from obtaining a certificate and participating in the protocol through replaying the messages while they are valid, or sending its own message, with fabricated data, using its certificate. Although it is currently unclear how well SCMS can function since it is not open source, we assume that it introduces no significant latency. In general, we do not consider message delays, jamming, physical attacks on sensors or controllers, DoS attacks, or any similar attacks to be part of our threat model since our focus is on application level exploitation. It is clear that such attacks are possible, and perhaps can be used in conjunction with application level attacks to amplify their damage. We also do not consider attacks exploiting bugs in the software stack of any of the existing components running on the infrastructure components, or other cars which we consider to be orthogonal to our threat model. We also do not consider physical attacks on the sensors of the vehicles or any sensors deployed by the infrastructure.

In some attacks, we assume that the attacker is a compromised vehicle which uses a radio that is capable of reaching cars farther away than typical vehicular radios and is capable of authenticating itself to the SCMS as a regular vehicle, then applying its attacks in the application level. We assume that the attacker knows the application logic and crafts its actions to manipulate this logic.

3 Cooperative Adaptive Cruise Control

In this section, we introduce the Cooperative Adaptive Cruise Control (CACC) application to provide background necessary to understand its potential security vulnerabilities. In CACC, a group of vehicles, with a close spacing between

them, can form a platoon if they are traveling in the same direction. Once created, vehicles in the platoon co-operate to travel at the same speed and make decisions as a group, maintaining reduced clearance gaps between each other, allowing for more efficient use of the highway and reducing the air drag compared to vehicles traveling individually. A Platoon Management Protocol (PMP) controls platoon operations and maneuvers. The leading/front vehicle acts as the coordinator and controls platoon decisions such as the speed, lane changes, and merging with other platoons. Vehicles communicate typically through Dedicated Short Range Communication (DSRC/IEEE 802.11p [21]), although eventually they may use 5G instead [22]. Road Side Units (RSUs) [19] are infrastructure units that are used to coordinate behavior or maneuver across cars, or to maintain shared certain state. Each vehicle has On Board Unit (OBU) that can use Basic Safety Messages (BSMs) to send some periodic information such as speed and location and receive event messages such as those informing of traffic conditions in an area they are entering.

In our experiments we use PMP, which was proposed and developed by Amoozadeh et al [15]. PMP supports a number of maneuvers representing different operations that platoons could potentially perform. This section introduces some of the primary maneuvers.

Joining a new Platoon (or forming a new platoon): If a vehicle receives a beacon message sent from a vehicle ahead of it, it will evaluate the position, speed, acceleration, and other relevant information to determine whether or not to join the platoon. Beacon messages also contain a Platoon Id, which is a locally distinct number used to distinguish the various platoons in the area.

Split Maneuver: Split maneuver is always initiated by the platoon leader. When the platoon size exceeds the optimal platoon size, the maneuver can be used to break the platoon into two, at a specific position. First, a SPLIT_REQ message is sent to the vehicle where the split should occur. If the request is accepted, a SPLIT_ACCEPT message is sent back to the leader. Subsequently, the leader sends a unicast CHANGE_PL to the potential leader of the new platoon resulting from the split. Finally, the original leader will report split end by sending SPLIT_DONE message.

Merge Maneuver: In this maneuver, two platoons, traveling in the same lane and close to each other, merge to form one platoon. If the leader of the rear platoon discovers another platoon in front of it with capacity to merge, the leader sends a unicast MERGE_REQ to the front platoon leader. Once the front leader accepts the merge request, it sends back a MERGE_ACCEPT message. On receiving this message, the rear platoon leader starts a catch-up maneuver. Upon reaching the front platoon, the rear platoon leader sends CHANGE_PL to all its followers to change the platoon leader to the front leader. Now the followers start listening to the front leader and eventually the rear leader changes its state from leader to follower after sending a MERGE_DONE message.

	Vulnerability	Explanation
V1	Fake message contents	Attacker sends messages with false information
V2	Insufficient information	Critical data not communicated
V3	Inadequate identifier binding	Incorrect binding of physical object to logical object
V4	Incomplete or unsafe protocol logic	Protocol does not consider all scenarios
V5	Trust delegation	Decisions delegated to possibly malicious participant

Table 1: Vulnerability Classification in Networked Cyber-physical Systems

Leave Maneuver: The departing vehicle initiates the process by sending a LEAVE_REQ message. The leader sends a LEAVE_ACCEPT message and then split process starts. Once the leaving vehicle changes lane, a GAP_CREATED message is broadcast. A merge process begins to reduce the gap until the platoon has the target gap distance between each car.

Change Lane Maneuver: In this maneuver, the platoon leader decides that the platoon needs to change lane. A platoon might need to change lanes if the platoon need to exit the highway or if it has been given instruction from the RSU due to lane congestion. The platoon leader sends CHANGE_LANE instruction to all the other vehicles in the platoon and they perform the maneuver together following the leader’s lane change. After that, all the followers send an ACK message to the leader, if they changed the lane successfully.

4 Vulnerability Analysis and Classification

It is tempting to consider networked cyber-physical systems such as CV as simply another networked system from the perspective of security, and indeed this is the case with respect to the vulnerability vectors. However, these systems differ in two important aspects with profound implications on vulnerabilities and defenses. The systems are (1) cooperative: they coordinate to accomplish a combined outcome; and (2) constrained by physics: protocol logic, as well as misbehavior outcomes are defined with respect to their impact on the system in the physical world, for example, considering both space and time.

The factors, outlined above, lead to vulnerability classes that are tied to the protocol logic and the physical system. Based on our analysis of multiple CV applications, we identified a number of vulnerability classes, which we believe generalize to other networked cyber-physical systems as well. These vulnerabilities arise even if vehicles have a certificate, which, to begin with, is not that difficult to obtain.

The first vulnerability class (V1) relies on the ability of the attacker to generate messages with malicious content (e.g.,

a fake location). By manipulating the information shared to other participants, the protocol logic can be exploited leading to safety or performance compromises. A related class of vulnerability (V2) concerns protocols where information that is critical to a sound decision is not considered, perhaps because it is not available, or is not exchanged. For example, the vehicles' lane position and platoon identification number are important parameters that we discovered were not considered when initiating a merge.

A third class of vulnerability (V3) relates to ambiguities that arise in *binding identifiers to vehicles*, the act of associating a detected physical information with a moving object such as a vehicle or pedestrian that is known through communication messages. Specifically, sensors can detect physical signals such as proximity to an object and mistakenly associate it with a different object in the message identifier space. For example, an attacker may pretend to be a platoon leader while a vehicle is attempting to join the platoon, a different vehicle may be mistakenly identified as the attacker/leader.

The next vulnerability class (V4) relates to under-specified or incomplete protocol logic. The application logic fails to consider corner cases such as the sudden loss of a platoon leader. In the reference CACC implementation [15], follower cars drive aimlessly if the platoon leader does not communicate with them. Ensuring the robustness of the protocol algorithm is essential for secure application.

The final vulnerability class (V5) arises when one object in the system delegates decisions to a malicious or compromised object, thus safety can be compromised. For example, trust is delegated to the platoon leader in CACC which enables arbitrary dangerous maneuvers that can cause crashes and blocking emergency vehicles.

5 Application level attacks on CACC

In this section, we present application layer attacks that attempt to exploit the functionality of the PMP implementation of CACC. These attacks were identified from a detailed code review of the PMP implementation. In each attack, we start with explaining the maneuver functionality and consider an attacker that participates in the protocol, sending messages in a way that passes the certificate based authentication and the application logic but results in disrupting the operation of one or more vehicles. We demonstrate the impact of these attacks in later sections.

5.1 Attack 1: Merge over large distances

If two platoons are traveling in the same lane and they are close enough while exchanging messages with each other, the PMP application allows them to merge to form one platoon for added efficiency. The application checks prerequisite conditions for the merge, such as, ensuring that the resulting

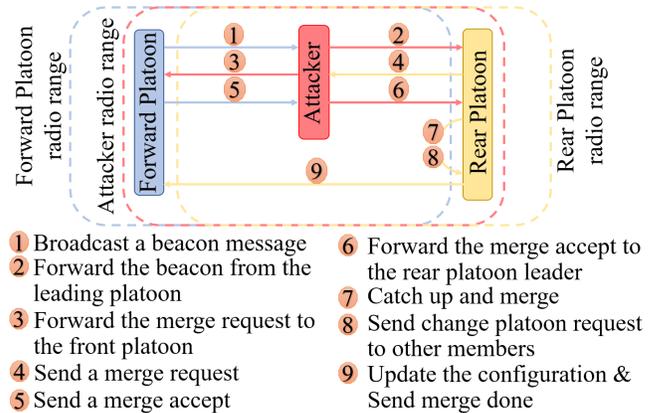


Figure 1: Attack scheme of distant merge attack

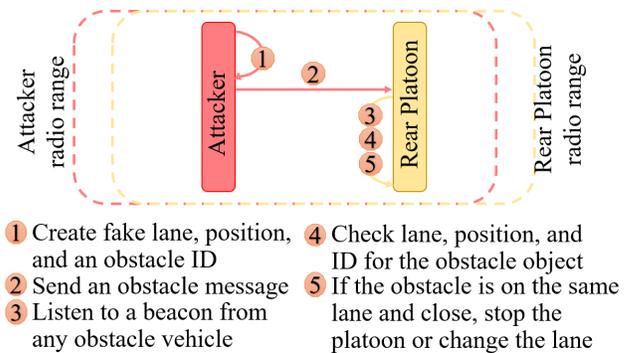


Figure 2: Attack scheme of the fake obstacle attack

combined platoon does not exceed the size limit. In our experiments, we found out that for two platoons to merge, the rear platoon must receive beacon messages from the front platoon. Then, it measures a certain distance to the last member of the front platoon using its ranging sensor. In our attack scenario, the attacker takes advantage of fake message contents (V1) and insufficient information (V2) vulnerabilities to target two platoons that are not within the communication range of each other. The attacker in this scenario is located between two platoons such that it can communicate with both platoons simultaneously and deceive ranging sensor by pretending that it is a member of the front platoon. For a farther distance, the attacker can have a sophisticated radio that can send and receive messages for a longer range.

The attack (Fig. 1) begins when the attacker replays the front platoon beacon messages to the rear platoon; since they are merely instantaneous replaying messages, the credentials on these messages are considered valid by the receiving vehicles. Upon receiving these beacons, the leader of the rear platoon will check to see if a platoon exists ahead by using its local sensors to look for a car from the front platoon in the lane ahead, which will be in this case, our malicious vehicle. The rear platoon will then speculate that the front platoon is

approaching and initiates merging if the new platoon size is under the predefined permissible threshold (i.e., size of the combined platoon is less than the maximum platoon size).

The rear platoon leader extracts the platoon ID of the front platoon from the beacon and sends a unicast merge request message to the front platoon (which is again relayed by the attacker). The front platoon leader, if it accepts the request, sends a unicast merge accept message, which the attacker then transmits back to the rear platoon. Upon receiving it, the rear platoon leader reduces its time-gap by increasing the speed of the whole platoon to the maximum limit to catch up. At this point, the attack impact shows up when the rear platoon increases its speed for a large distance degrading both safety and economy. Once the inter-platoon spacing becomes small, the rear platoon leader sends change platoon message to all its followers to change the platoon leader to the front platoon leader. Finally, the rear platoon leader sends a merge done message to front platoon leader and changes its state from leader to follower.

5.2 Attack 2: Fake Obstacle Attack

A platoon may have automatic incident detection enabled; with this option, the platoon can receive and rapidly react to an obstacle message. Upon encountering an obstacle or accident in its lane, a vehicle will come to a stop and send an obstacle message with its position to any oncoming vehicles, allowing them to stop or change their lanes when they arrive at the location of the incident. In this scenario, the malicious vehicle exploits the fake content (V1) vulnerability and creates a false obstacle message with a specific location in the lane, forcing incoming platoons to slow down until they stop or change lanes. The attack scheme is shown in Fig. 2. The fake obstacle attack affects the speed of the platoon and this rapid deceleration can affect safety. The presence of an obstacle is impossible to validate by a distant platoon. Note, that it is possible to combine this attack with *Attack 1* to attempt to create an accident by first speeding up the cars and then forcing them to stop quickly.

5.3 Attack 3: Merge across different lanes

In this scenario, we attack two platoons, within the communication range of each other, that are traveling in separate lanes. Critical variables such as lane number and other surroundings information for each vehicle are neither communicated nor checked (V2 and V4 vulnerabilities). The attacker can look for a slow platoon in front and try to merge it with a faster platoon from a different lane to slow down traffic flow.

The attack (Fig. 3) starts when the malicious vehicle is in front of the rear platoon, and sends messages pretending to be a part of the other platoon (in another lane). This can be done by manipulating the platoon ID parameter in Basic safety message. The rear platoon will see the attacker vehicle

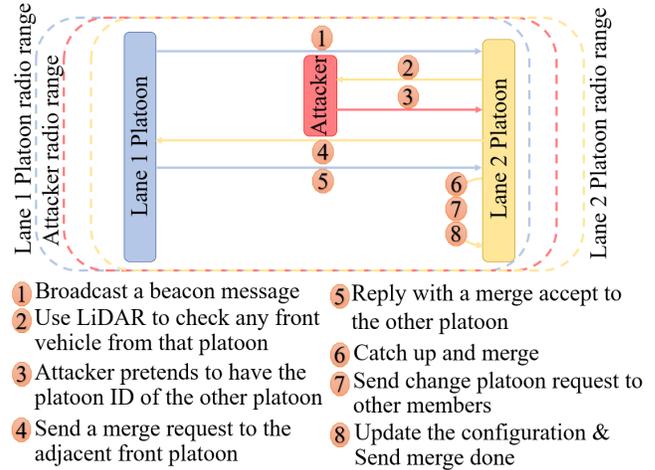


Figure 3: Attack scheme of merging across lanes attack

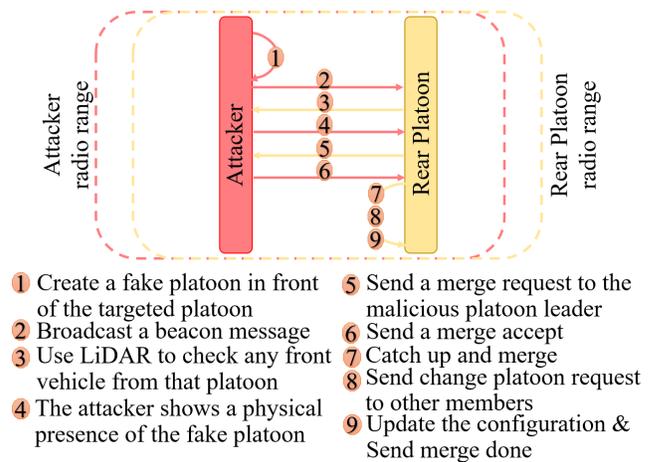


Figure 4: Attack scheme of platoon takeover attack

using its LiDAR sensor and assumes that the attacker is part of the platoon (V3). Information such as Lane ID is neither communicated nor checked. It then begins a merging maneuver. As consequence, the adjacent leading platoon leader sends a merge accept message. As a result, the rear platoon leader increases the speed of the platoon to catch up. Afterwards, the attacker leaves its location and the rear platoon leader sends change platoon to all its followers.

5.4 Attack 4: Platoon Takeover

This attack is conceptually similar to the *Attack 3* except that there is only one platoon (the rear platoon), with the attacker attempting to become its leader. The attacker counts on different vulnerabilities but mainly on the fake message contents (V1) vulnerability by pretending to be the leader of the fictitious front platoon by generating any logically consistent description of the front platoon such as the locations and

speeds of a fake platoons' members in front of the victim platoon. The attacker transmits the fake messages for each false vehicle of the fake platoon. The rear platoon leader will notice the attacker through the LiDAR sensor and initiate a merging maneuver since it believes that this is the platoon in front of it that it listens to. The attacker responds to all requests from the rear platoon. This leads to the completion of the merging process. The platoon is now under the attackers' control and can be manipulated in a dangerous manner as we show in Section 6.1, exploiting the trust delegation (V5) vulnerability. We show the steps of this attack are shown in Fig. 4.

6 Experimental Attack Scenarios and Results

In this section, we first describe the simulation set up used in the experiments. We then present an experimental evaluation of the proposed attacks and evaluate their impact on the traffic system with respect to safety and performance. Given the limited availability of deployed CV applications, and the closed nature of these systems, we elected to evaluate the attacks using simulation. We used VENTOS (VEhicular NeTwork Open Simulator), an extension of Veins [27]. Veins integrates a C++ simulator for studying vehicular traffic flows, collaborative driving, and interactions between vehicles and infrastructure with another simulator which models communication through a DSRC-enabled wireless communication. Veins combines two widely used simulators, Simulation of cars/physics simulator (SUMO) [5] and OMNET++ [3]. SUMO is an open-source road traffic simulator developed by the Institute of Transportation Systems at the German Aerospace Center and serves as the traffic flows physics simulator. This framework has been used in hundreds of studies from academia, industry, and the government (a partial list can be found on the project [8]). VEINS uses SUMO's Traffic Control Interface, TraCI, to communicate simulation commands to it. OMNET++ is an open-source simulation package and carries out the wireless communication simulation. We configure it to use the models for the IEEE 802.11p [21] protocol, a standard adopted for V2V communication. We use Wave Short Message Protocol (WSMP) to carry beacon and micro-command messages. These messages are directly sent to the data-link layer which uses continuous channel access based on IEEE 1609.4.

6.1 Dangerous Attack Demonstrations

First, we demonstrate the potential impacts of the attacks using two specific scenarios, one causing a collision and the second interfering with and delaying an emergency vehicle.

Causing a Collision: In this attack, the followers of a platoon that is controlled by a compromised leader, fail to see and stop for stationary or slower vehicles. The malicious car may have acquired leadership of the platoon using the platoon takeover attack. The attacker can suddenly veer out

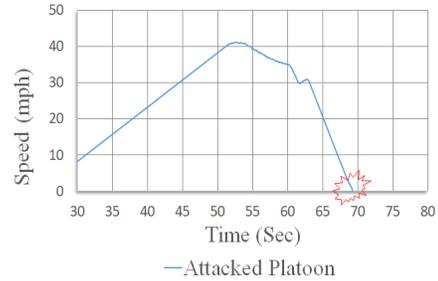


Figure 5: Speed profile in collision attack

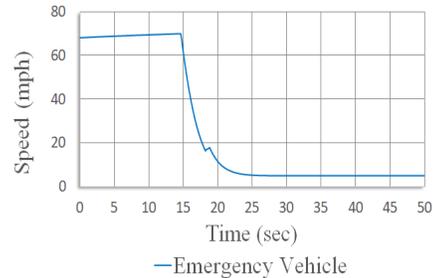


Figure 6: Speed profile in Emergency Vehicle Attack. The attacker slows down the emergency vehicle from 70 to 5 mph

of a lane without informing the followers to slow down or change lanes. The followers' braking systems may not be able to stop if an obstacle appears immediately in their path. We can see the sudden stop then collision at the time 60s for the victim vehicles in Fig. 5. After investigating this scenario in detail, we discovered that vehicles in the platoon were not keeping a safe distance between each other. Instead, they were delegating trust (V5) to the platoon leader (the attacker), trusting that the leader will maintain safe separation from any obstacles.

Emergency Vehicle Interference: We again start with the attacker using the *Platoon Takeover* attack, described in Section 5.4. The attack is comprised of Attack vector V5 (untrusted delegation). The attacker slows down the whole platoon then makes some followers move to another lane. If an emergency vehicle (police or ambulance) is coming fast in that lane, a slow vehicle on the same lane will make it much slower or even stop it, as shown in Fig. 6. This can cause catastrophic slowdowns in real life (e.g., potential loss of life). Other approaches to delay an emergency vehicle can be devised, for example, using the merge across different lanes attack.

6.2 Isolated Attack Scenarios

In this set of experiments, we investigate vehicle performance after implementing the four different attacks described in Section 5 isolating the impact on just one or two targeted

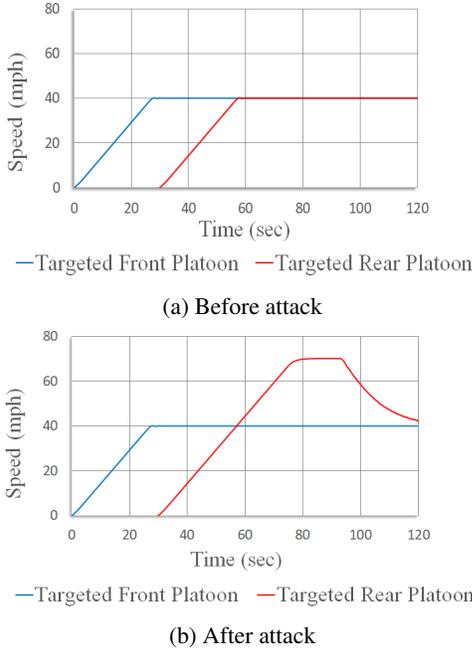


Figure 7: Speed profile in Attack 1 (distant merge)

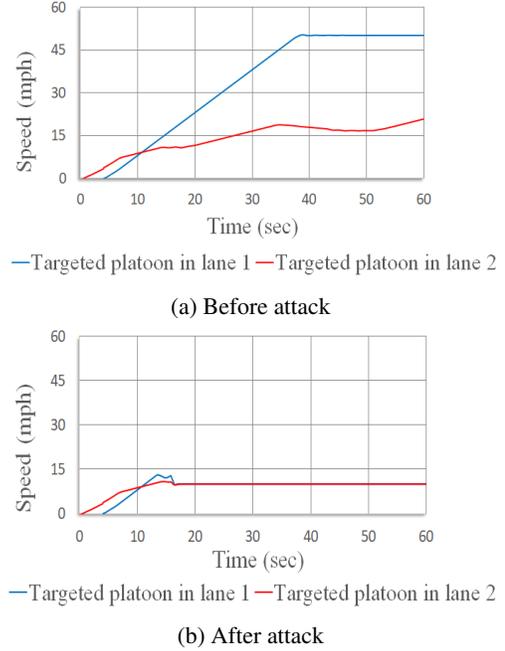


Figure 9: Speed profile in attack 3 (merging across lanes)

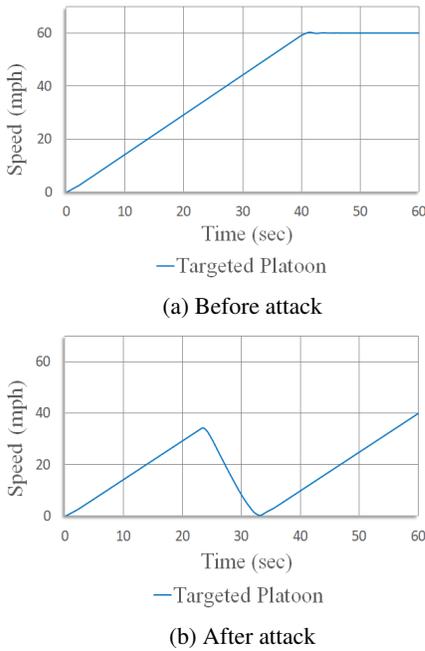


Figure 8: Speed profile in Attack 2 (fake obstacle attack)

platoons. These scenarios allow us to evaluate the isolated impact of the attacks.

Attack 1– Distant Merging attack: Our intention in this attack is to make some platoons go to the catch-up process where they speed up abnormally for some time potentially degrading both safety and efficiency. Fig. 7a shows the aver-

age speed of two platoons in the scenario in the absence of an attack. The rear platoon starts a little later, but both platoons accelerate to 40mph before cruising at that speed. Fig. 7b shows the behavior of the platoons in the presence of the attack. In this case, the rear platoon accelerates aggressively, reaching the maximum velocity, in an effort to catch up with the front platoon.

Attack 2– Fake Obstacle attack: From Fig. 8a, we see a platoon of 3 cars accelerating to 60mph. After initiating the attack starting around time 20s, we can notice how the platoon suddenly comes to a halt as shown in Fig. 8b. This occurs for a certain time then the platoon changes the lane and accelerates again, but the attack can be repeated.

Attack 3– Merging platoons across lanes: In this scenario, two platoons travel on different lanes where the front platoon is slower than the rear one. The attacker realizes that both platoons are close to each other and locates itself in front of the rear platoon. Next, the attacker initiates the merge maneuver as described in Attack 3. When the attack succeeds, all the members of the rear platoon will follow the front platoon (despite being in a different lane) and travel according to its speed as shown in Fig. 9. In this case, the lower speed platoon slows down the traffic flow. In another case, the rear platoon may be tricked to go faster than the optimal speed for the lane, compromising safety.

Attack 4– Platoon Takeover Attack: The attacker starts with sending different beacon messages pretending that they come from a front platoon. Once the platoon finds that the leading vehicle on the same lane is the last platoon member that it listens to (through its LiDAR sensor), it will then start

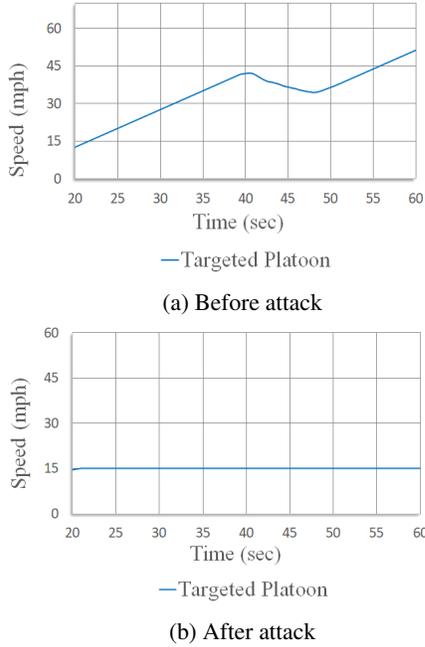


Figure 10: Speed profile in attack 4 (platoon takeover attack)

the merging process. After the merging succeeds, the attacker now acts as a platoon leader and controls this platoon in any way it desires within the platoon operational parameters. For this example attack, the attacker decreases the platoon velocity and then repeatedly changes the lane of the platoon in order to affect as many lanes as many as possible. Fig. 10 shows the platoon speed changes.

6.3 Attacks within traffic scenarios

Next, we evaluate the impact of the attacks when applied as part of an active traffic scenario. We use different metrics to quantitatively analyze the effects of the attacks on *Mobility* and *Safety*. For mobility, we use two metrics: (1) **Average speed** of vehicles is a common metric for mobility; and (2) **Flow** of traffic, is defined as the number of vehicles passing a point on the road in a given time. To measure safety, we also use two metrics: (1) **Average speed difference** between consecutive vehicles measures the differences in speed among vehicles. This metric is known to correlate with the onset of collisions and near-collisions; and (2) **Time-to-Collision (TTC)** [23] is metric for safety which measures the time taken for a vehicle to collide with the vehicle in front of it, should they maintain the same speed. TTC of vehicle i at instant t can be calculated as follows,

$$TTC_i(t) = \frac{V_i(t) - V_{i-1}(t) - l_i}{V_i(t) - V_{i-1}(t)}$$

here, $V_i(t)$ stands for the speed of the vehicle i at instant t and l_i is the length of the vehicle i .

California Department of Transport provides real time traffic condition through Performance Measurement System [4] by using various sensors installed in the state's most highways sections. We use data from a section of the highway I-5 in south California and generate scenarios with vehicles entering stochastically following the observed distribution. Each scenario, ran for 5 minutes, simulates the entrance of traffic into a highway section of length 6 miles. We assume that all vehicles are CV enabled to avoid making assumptions on the interactions of CV and non-CV vehicles. We configure about 25% of the vehicles to form platoons of different sizes. The maximum speed for the road is 70 mph. The communication range for each vehicle is 300 meters. Each road has five lanes and approximately evenly spaced road side units (RSU) such that all points in the highway are in range with at least one RSU.

Attack 1–Distant Merging Attack: Fig. 11a shows the effects of attack 1 on the average speed, flow, average speed difference, and average TTC for the scenario. The attack causes an increase in average speed and flow of traffic. Even though the flow of vehicle increases by a small amount, the attack causes vehicles under attack to travel at a much higher speed, thus compromising safety, which is reflected by the increased average speed difference and reduced TTC. Even though the flow of vehicle increases by a small amount, distant merge attack causes vehicles under attack to travel at a much higher speed, thus compromising safety.

Attack 2– Fake Obstacle Attack: Fake obstacle attack causes the traffic to slow down potentially abruptly, similar to the slow down due to road site construction. Thus, it has slight adverse effect on safety, with increased average speed difference and TTC, but a large effect on the mobility, with decreased average speed and flow, as depicted in Fig. 11b.

Attack 3– Merging across lanes: In this attack, the attacker connects the flow of traffic of two or more lanes, forcing a faster platoon to slow down. The effect of the attack is shown in Fig. 11c. Average speed difference increases only slightly, while TTC increases, leading to a marginal impact on safety. However, the flow of the traffic is severely hindered which is shown by the steep drop in average speed and flow.

Attack 4– Platoon takeover: In this attack, the attacker takes over the control of a platoon and can control it fully. This is the most dangerous form of attack that the attacker can carry out. Although different arbitrary maneuvers are possible once the attacker controls the platoon, we went with a speed reduction and repeated lane change maneuvers. Both safety and mobility metrics are highly affected by this attack, as seen in Fig. 11d.

7 Potential Mitigation

Our eventual goal is to develop a defense approach that is automated and can mitigate the vulnerability classes we identified in Table 1, thus making the protocol logic more secure in a principled way. The general defense approach relies on

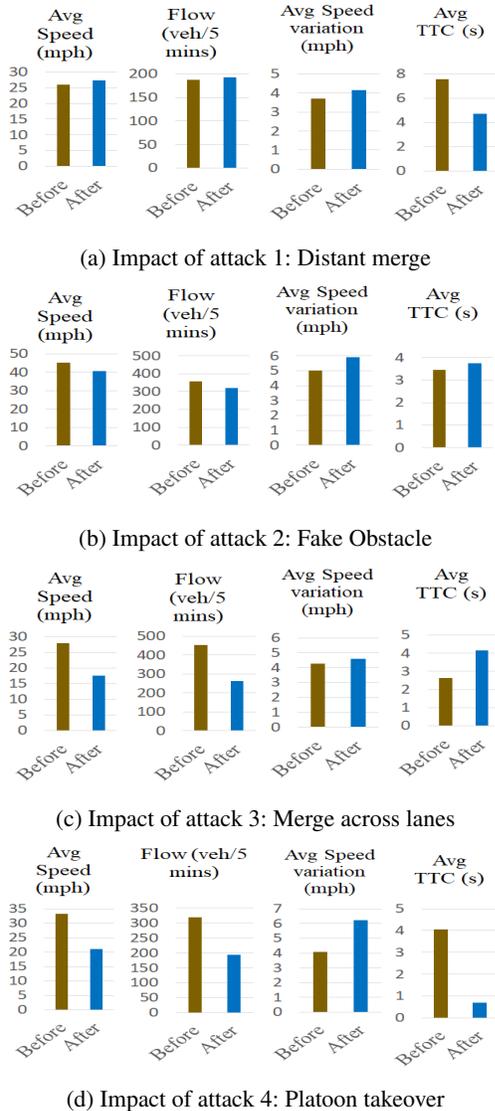


Figure 11: Impact of attacks

augmenting the information available to vehicles with a redundant source of information that enables detection of incorrect or malicious information, and makes the protocol logic more robust. If such a source of redundant information is available, the veracity of the exchanged messages can be checked before conducting critical actions within a maneuver, thus addressing **V1** and **V2** vulnerabilities. To give an example, if a merge is attempted with a far-away platoon, the requested platoon should check if the distance of the front platoon is within the merge range; previously, this was assumed from the fact that the messages were received from the front platoon, an assumption that can be exploited by an attacker that replays a message (effectively extending its reach) or to use higher power radio to increase its range. Moreover, this defense substantially reduces the opportunities for **V3** attacks

since it becomes more difficult to create wrong bindings between message sources and other physical objects. This check would defeat the replay attack that allows the adversary to initiate a merge. **V4** can be addressed by in depth protocol testing and analysis. Finally, **V5** can be addressed by either avoiding trust delegation or verifying delegated decisions.

We collect complementary information through a reliable sensory system to protect against fake message contents (**V1**). Validating protocol components by linking message contents and redundant sensor data is also desirable for a reliable decision. The consistency of the application and environment constraints using a robust algorithm need to be considered to prevent a message with clearly unfeasible information to be acted on and ensure that the resulting action is consistent with the protocol logic. If everything checks out, a final decision will be assigned to protocol controller to lead the required action.

7.1 Preliminaries and Assumptions

RSU: Defense components infrastructure: The main component that we rely on in our scheme is the road side unit (RSU), where its hardware and software components are specified by US DOT [26]. The RSU is a more sophisticated and more protected component of the system deployed and managed by the infrastructure provider, making it an attractive component to root defenses. It is expected to operate unattended in harsh outdoor environments for extended periods of time (typical Mean Time Between Failures of 100,000 hours). It detects and auto-recovers from minor software failures, transient power spikes, and power interruptions. We consider a case where RSUs are reachable from any point on the highway as a proof of concept, but the protocol can be made to act conservatively in Safe mode when RSU are not reachable.

We note that without relying on the RSU, the alternative is to reach consensus between the different cars which is an interesting possibility. A naive implementation could be too costly to achieve on-demand, and therefore we elected to root our defenses in the RSU.

Safe mode and functionality of RSU: We identify a safe operation mode for platoons with respect to any maneuver or protocol state. The goal of the safe mode is to be used as a cautious behavior when protocol exchanges are in progress, or when a decision cannot be made. For example, the platoons could either maintain their speed or slow down and wait for confirmation after sending a maneuver request. The defense proceeds by having the RSU check the proposed action against the configuration of the platoon (e.g., the location of each member of the relevant platoons from all basic safety messages (BSMs) it collects). The RSU uses, as a source of redundant information, a video tracking system to track the vehicle locations. The system also maps any incoming messages to vehicles based on the geographic information to check the consistency of messages being sent by any particular vehicle.

Algorithm 1 Pre-Approval protocol for Platoon Leader

```
1: procedure PRE-APPROVALPROTOCOL
2:   SendManeuverRequestToRSU()
3:   Change to SAFE mode    ▷ Wait for RSU response
4:   loop:
5:   if Disapproval Received then return AbortManeuver()
6:   if Approval Received then return StartManeuver()
7:   if Time Out Exceeded then return Exit-loop
8:   goto loop           ▷ Time Out NOT Exceeded
9:   StartBackupProcedure()
```

Other sources of redundancy are also possible, for example, exchange of past information from nearby RSUs for vehicle tracking, or alternative real time sensors. Our proposed video tracking system is feasible: many vehicles tracking systems using video cameras have been proposed [29], [28]. We would next see how the defense would work for the previous attacks.

7.2 Defense overview

Defense against Merging attacks: For *Attacks 1, 3, and 4*, the defense starts by allowing the back platoon to send a merging request to RSU. After receiving the maneuver request, the RSU verifies the relevant information. Then, it tests if the merge process is applicable or not by inspecting the constraints between the platoons such as making sure that the distance between them is within the permissible range. If all checks pass, an approval reply is sent to the two platoons to start merging. If the maneuver confirmation is received and leader, for any reason does not exist, the platoon members can start a voting process where they study the collected BSMs and check its neighbor vehicles through its sensors to choose their leader to control the maneuver.

Defense against Obstacle attacks: For *Attacks 2*, RSU carries out the same steps regarding requesting a maneuver. For this scenario, it checks specifically if the obstacle and the incoming platoon are in the same lane or not and, if yes, the distance between them. Then, the RSU will send an approval reply to stop the coming platoon or change its lane. In the meantime, the traveling platoon leader will go to the safe mode where it moves within the safety speed limit which we defined here to be below 20 mph. Generally, it is sufficient to ensure the ability to stop in case the obstacle message is confirmed. If the platoon does not receive any confirmation for the obstacle maneuver until the obstacle location, it can start the backup protocol where it can stop or change lane. Fig. 12 shows the general protocol for the RSU. Algorithm 1 shows the steps for the platoon leader.

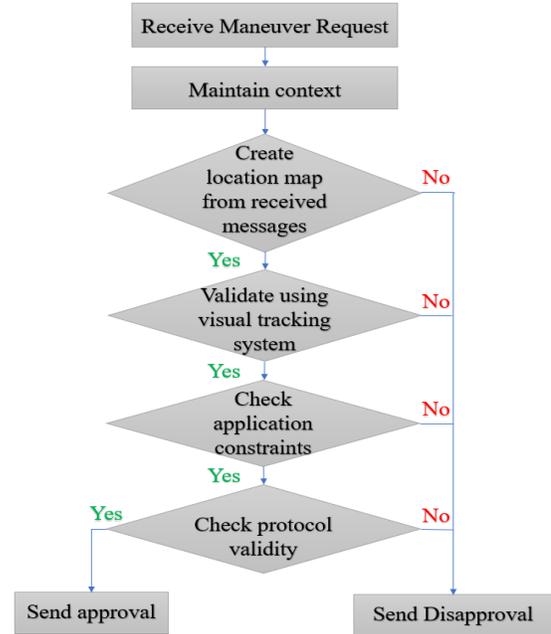


Figure 12: Pre-Maneuver Protocol process for RSU

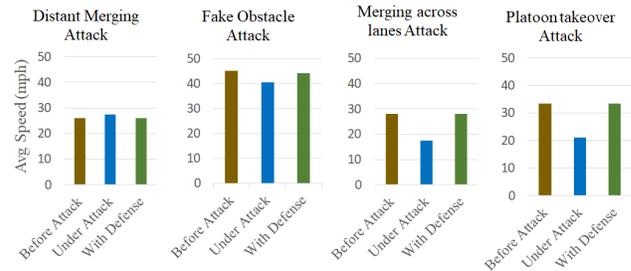


Figure 13: Effect of defense on the studied attacks

7.3 Evaluation

We implemented the defense logic within the simulator. We emulate the video tracking by using the ground truth value of the location and adding Gaussian noise to it with a mean of 2 meters. We augmented the application with the defense by following the mitigation steps discussed above. Fig. 13 demonstrates that with the defense in place, the attack impact is mitigated from all attacks other than the fake obstacle attack where it has a minor effect. The effect is due to the delay in confirmation from the RSU, during which the safe mode reduces performance whether there is a real obstacle or not. This mitigation's approach will also be able to stop the dangerous attacks described in Section 6.1. This is due to the fact that those attacks are bases on the basic attacks demonstrated in Section 5 but used in specific scenarios.

7.4 Discussion

A concern with any defense strategy that requires additional operations is delays in making decisions, while information is validated. However, we believe that the redundant information can be prepared proactively so that the check is often local. Moreover, it is critical to deploy the safe backup operation while decisions are being taken in any cyber-physical system, prioritizing safety over performance.

The approach heavily relies on the visual tracking system and sensors for more reliable decision, which may not be available in all vehicles and in some areas. Thus, we accept that it is a strong assumption on our part to assume that such redundant data will always be available to the decision making system. We can see from Fig. 13 that safe mode does not significantly degrade performance in CACC application. Nevertheless, we will carry out analysis and performance measurements on other CV applications to justify this statement in the future. In the future work, robust algorithms may be employed to detect all the different attacks in the early stages.

8 Vulnerabilities in other protocols

In this section, we analyze other protocols and classify the vulnerabilities using the attack vectors defined in Table 1. We performed code reviews of two protocols available on the US DOT open source CV protocol repository [13]: (1) Intelligent Intersection Management and (2) Eco-Traffic Signal Timing.

Intelligent Intersection Management has shown great potential in improving transportation efficacy especially for autonomous vehicles where it connects with them wirelessly and schedules their intersection crossing steps. In [17], they proposed to use existing infrastructure-side sensors to stop malicious messages. For example, vehicle detectors buried underneath the stop bar of each lane can be used to measure aggregated traffic information. After analyzing the scheme, we found out that malicious messages can still be sent to manipulate the application and increase total delay time. This is due to inadequate identifier binding (V3) vulnerability, where sensors do not correlate the messages with the vehicles and do not give the exact location for each vehicle.

Eco-Traffic Signal Timing application aims to improve traffic signals delays thus reducing environmental impact. It processes real-time and historical CV data at signalized intersections to reduce fuel consumption and overall emissions. In this application, we discovered that vehicle trajectory data can be subjected to fake message contents (V1) and inadequate identifier binding (V3) vulnerabilities. We were able to implement exploits to manipulate the timing phase for any lane based on sending malicious vehicles information. For both applications, the defense principles we introduced can be adapted to mitigate these vulnerabilities.

9 Related work

Chen et al. [17] performed a security analysis on a system called Intelligent Traffic Signal System (I-SIG). In this system, a real-time vehicle trajectory data is sent through the dedicated short-range communications (DSRC) technology that is acquired by any CV. This data is then used to control the sequence and duration of traffic signals. The system that was attacked includes real deployments at road intersections in some cities such as Anthem, AZ, and Palo Alto, CA. In these deployments, it enhanced the traffic by reducing total vehicle delay by 26.6%. The paper presented an attack that can cause the traffic mobility to be 23.4% worse than that without adopting I-SIG. The attack consists of a packet spoofed to tell the I-SIG of a vehicle approaching from a long distance, causing the traffic light to wait for it, while holding up traffic from other directions. The authors suggested a possible defense that considers scheduling over multiple periods.

Amoozadeh et al. [14] performed a security and vulnerabilities risks analysis related to the VANET communication in CV in specific applications including cooperative adaptive cruise control application (CACC). They focused mainly on how to attack a single platoon. They considered a CACC vehicle stream that is moving in a straight single-lane highway where all the vehicles use a simple one vehicle look-ahead communication scheme. They did not consider the security credential management system (SCMS) in their simulation. In their work, they examined existing countermeasures, and explored the limitations of these methods and possible ways to lighten negative effects.

Dominic et al. [18] presented a risk assessment framework for autonomous and cooperative automated driving was conducted to define a reference scheme for automated vehicles, and to describe the new attack surfaces and data flow. They used recent automotive threat models and introduced a novel application based threat enumeration and analysis approach that is able to address different automated vehicles applications across all levels of automation. They also established a framework with an example application assessment. In their work, they concluded that their results would guide the design of the secured automated driving architectures that will certainly and quickly become necessary.

The Intelligent Transportation Systems Joint Program Office (ITS JPO) [7] worked with its partners with a fund nearly \$25 million to support a foundational vehicle cyber security threat assessment for CV applications. Their work includes designing, developing, and operating the security credential management system (SCMS) for the CV Safety Pilot evaluations were conducted in some cities such as Ann Arbor, Michigan, as well as the current US DOT CV pilot studies in NYC [11], Tampa [12], and Wyoming [10]. They developed certification practices to check equipment prior to implementation in the Safety Pilot to ensure that they meet cyber security requirements. The program is also working to ease provid-

ing the certification services for different industries. Finally, one of the goals is to improve the best practices for handling foundational electronics control and reliability cyber threat information for existing vehicle fleets.

10 Concluding Remarks

Connected Vehicles (CVs) is an emerging field in transportation that is garnering interest from the US DOT because it promises to bring about a new age of transportation where vehicles and transportation infrastructure are all interconnected wirelessly. However, many applications are still not considering their security vulnerabilities. This paper shows that one of the most complete reference implementations of a CVs protocol (for Cooperative Automatic Cruise Control) is vulnerable to attacks of many types, even under a threat model that considers the state-of-the-art SCMS certificate based security standard being developed for these applications. These attacks that exploit the vulnerabilities of these communication protocols, may lead to a complete reversal of the benefits made by CVs, and as such, they have a ways to go before it is reliably safe from attacks. We demonstrated these attacks in simulation and showed their impact on safety, performance, and economy of the traffic.

It also introduces a defense scheme that places vehicles in a safe mode while they check the consistency of received information against an estimate of the local traffic state constructed through video analytics at the road side unit. Finally, we showed that the proposed defenses are able to mitigate all the attacks we introduced, making it a promising approach to support security in CV applications.

Acknowledgements

This material is partially supported by the National Science Foundation (NSF) grant CNS-1646641 (CNS-1839511) and CNS-1724341. It is also partially supported by UC Lab Fees grant LFR-18-548554. All opinions and statements reported here represent those of the authors.

References

- [1] Coordination of Automated Road Transport Deployment for Europe. Accessed Dec 2016 from https://connectedautomateddriving.eu/wp-content/uploads/2017/02/20161216_CARTRE_MS_Workshop_v1.1-1.pdf.
- [2] CV Pilot Deployment Program. Accessed from https://www.its.dot.gov/pilots/cv_pilot_apps.htm.
- [3] OMNeT++. Accessed from <https://www.omnetpp.org/>.
- [4] PeMS - Caltrans Performance Measurement System. Accessed Aug 2018 from <http://pems.dot.ca.gov/>.
- [5] SUMO - Simulation of Urban Mobility. Accessed from <http://sumo.dlr.de/index.html>.
- [6] The Tesla Semi-trailer truck as It relates to platooning. Accessed Dec 2017 from <http://adamkwitko.com/the-tesla-semi-as-it-relates-to-platooning/>.
- [7] USDOT: Security Credential Management System (SCMS). Accessed from https://www.its.dot.gov/factsheets/pdf/CV_SCMS.pdf.
- [8] VEINS - Vehicles in Network Simulation. Accessed from <http://veins.car2x.org/>.
- [9] VENTOS - Vehicular Network Open Simulator. Accessed from <http://maniam.github.io/VENTOS/>.
- [10] Connected Vehicle Pilot Deployment Program, Wyoming, 2018. Accessed August 2018 from https://www.its.dot.gov/pilots/pilots_wydot.htm.
- [11] Connected Vehicle Pilot Project, New York City, 2018. Accessed August 2018 from <http://www.cvp.nyc>.
- [12] Connected Vehicle Pilot Project, Tampa, 2018. Accessed August 2018 from <https://www.tampacvpilot.com/>.
- [13] The Open Source Application Development Portal for Federal Highway Administration, USDOT, 2018. Accessed Aug 2018 from <https://www.itsforge.net/index.php/community/explore-applications/for-search-results#/30/63>.
- [14] M. Amoozadeh, A. Raghuramu, C. n. Chuah, D. Ghosal, H. M. Zhang, J. Rowe, and K. Levitt. Security vulnerabilities of connected vehicle streams and their impact on cooperative driving. *IEEE Communications Magazine*, 53(6):126–132, June 2015.
- [15] Mani Amoozadeh, Hui Deng, Chen-Nee Chuah, H Michael Zhang, and Dipak Ghosal. Platoon management with cooperative adaptive cruise control enabled by vanet. *Vehicular communications*, 2(2):110–123, 2015.
- [16] Matthew J Barth, Guoyuan Wu, and Kanok Boriboonsomsin. Intelligent transportation systems and greenhouse gas reductions. *Current Sustainable/Renewable Energy Reports*, 2(3):90–97, 2015.
- [17] Qi Alfred Chen, Yucheng Yin, Yiheng Feng, Z Morley Mao, and Henry X Liu. Exposing congestion attack on emerging connected vehicle based traffic signal control. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS'18), San Diego*, 2018.

- [18] R. Eustice D. Ma Di D. Dominic, S. Chhawri and A. Weimerskirch. Risk assessment for cooperative automated driving. *Proceedings of the 2Nd ACM Workshop on Cyber-Physical Systems Security and Privacy*, pages 47–58, 2016.
- [19] Fluidmesh Networks LLC. DSRC Roadside Unit. Accessed March 2019 from <https://www.fluidmesh.com/dsrc-roadside-unit/>.
- [20] John Harding, Gregory Powell, Rebecca Yoon, Joshua Fikentscher, Charlene Doyle, Dana Sade, Mike Lukuc, Jim Simons, Jing Wang, et al. Vehicle-to-vehicle communications: readiness of V2V technology for application. Technical report, United States. National Highway Traffic Safety Administration, 2014.
- [21] Hannes Hartenstein and LP Laberteaux. A tutorial survey on vehicular ad hoc networks. *IEEE Communications magazine*, 46(6), 2008.
- [22] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.
- [23] Michiel M Minderhoud and Piet HL Bovy. Extended time-to-collision measures for road traffic safety assessment. *Accident Analysis & Prevention*, 33(1):89–97, 2001.
- [24] University of Arizona, SCSC Econolite University of California PATH Program, Savari Networks Inc, and Volvo Technology. Multi-Modal Intelligent Traffic Signal Systems (MMITSS), Concept of Operations, 2012. Accessed May 2018 from http://www.cts.virginia.edu/wp-content/uploads/2014/05/Task2.3._CONOPS_6_Final_Revised.pdf.
- [25] U.S. Department of Transportation. The Connected Vehicle Reference Implementation Architecture. Accessed March 2019 from <https://local.iteris.com/cvria/html/applications/applications.html>.
- [26] Frank Perry, Kelli Raboy, Ed Leslie, Zhitong Huang, Drew Van Duren, et al. Dedicated Short-Range Communications RoadSide Unit Specifications. Technical report, United States. Department of Transportation, April 2017.
- [27] Christoph Sommer, Reinhard German, and Falko Dressler. Bidirectionally coupled network and road traffic simulation for improved ivc analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, January 2011.
- [28] Yao-Jan Wu, Feng-Li Lian, and Tang-Hsien Chang. Traffic monitoring and vehicle tracking using roadside cameras. *2006 IEEE International Conference on Systems, Man and Cybernetics*, 6:4631–4636, 2006.
- [29] Jingxin Xia, Wenming Rao, Wei Huang, and Zhenbo Lu. Automatic Multi-Vehicle Tracking using Video Cameras: An improved CAMShift approach. *KSCE Journal of Civil Engineering*, 17:1462–1470, 09 2013.

S3: A DFW-based Scalable Security State Analysis Framework for Large-Scale Data Center Networks

Abdulahakim Sabur, Ankur Chowdhary, and Dijiang Huang
{asabur, achaud16, dijiang}@asu.edu
Arizona State University

Myong Kang, Anya Kim, and Alexander Velazquez
{myong.kang, anya.kim, alexander.velazquez}@nrl.navy.mil
US Naval Research Lab

Abstract

With an average network size approaching 8000 servers, data-center networks need scalable security-state monitoring solutions. Using Attack Graph (AG) to identify possible attack paths and network risks is a common approach. However, existing AG generation approaches suffer from the state-space explosion issue. The size of AG increases exponentially as the number of services and vulnerabilities increases. To address this issue, we propose a network segmentation-based scalable security state management framework, called S3, which applies a *divide-and-conquer* approach to create multiple small-scale AGs (i.e., sub-AGs) by partitioning a large network into manageable smaller *segments*, and then merge them to establish the entire AG for the whole system. S3 utilizes SDN-based *distributed firewall* (DFW) for managing service reachability among different network segments. Therefore, it avoids reconstructing the entire system-level AG due to the dependencies among vulnerabilities.

Our experimental analysis shows that S3 (i) reduces AG generation and analysis complexity by *reducing* AG's density compared to existing AG-based solutions; (ii) utilizes SDN-based DFW to provide a granular security management framework, by incorporating security policies at the level of individual *hosts* and *segments*. In effect, S3 helps in limiting targeted slow and low attacks involving lateral movement.

1 Introduction

With the surge in cloud infrastructure and new technology such as containers and server-less applications, the attack surface has increased significantly. In order to have a better understanding of the security situation of a system, scalable and effective *attack representation methods* (ARMs) are required. The security administrator can use information derived from ARMs and apply the appropriate countermeasure on the identified critical path [6].

Graphical security states management and analysis solutions, e.g., *Attack Graphs* (AGs), is such a tool to fulfill the

purpose of security state analysis. AGs are defined as a data structure, used to model all possible critical attack paths and vulnerabilities of a system, which an adversary can exploit in order to achieve his/her attacking goals. However, generating and analyzing AG in a security system requires a significant generation and analysis overhead, an issue addressed in this paper, that has not been effectively addressed by existing solutions. The AG generation and attack path searching when performing AG-based attack scenario analysis can be an NP-hard problem as noted by Durkota *et. al.* [12], which depends on the density of a given AG. In a large network system, AGs are often incomprehensible to a user due to its complex interdependence among vulnerabilities. The identification of information regarding vulnerability dependencies becomes increasingly difficult as the number of services and vulnerabilities are increasing in the network system. Amman *et. al.* [1] proposed an AG generation approach with the scalability of the order $O(N^6)$. MulVAL [27] reduces the AG generation and analysis complexity from $O(N^6)$ to $O(N^2) - O(N^3)$, where N is the number of network hosts¹. Nevertheless, the generation of AG by using MulVAL still takes an order of minutes for a few hundreds of hosts.

The second problem S3 framework considers in this work is *lateral movement*. This sophisticated attack is conducted by highly expert individuals or organizations, which allow for multiple exploits and movement from one system node to another. AG can be used to identify the critical paths that can be exploited by an expert attacker. AG-based security analysis can identify the dependencies between services in a network and minimize the security issues that will allow an adversary to compromise critical services by lateral movement along *east-west* network using exhaustive trials method [8] over different attack paths. Moreover, AG has been widely used in identifying the least expensive countermeasure solution [6].

¹MulVAL ignored the scenarios when multiple vulnerabilities exist on the same host. In current virtualized environments, vulnerabilities can be interdependent within a given host and thus contribute to the complexity of AG generation and analysis. In this paper, we consider N is the measurement of the number of given vulnerabilities.

Thus, to identify and detect lateral movement, a well-modeled security analysis tool is needed. S3 framework focuses on the scalability of AG, which in return can be used to identify critical systems and help in the detecting complex attack scenarios.

In this paper, we propose to leverage SDN-based Distributed Firewall (DFW) [13] rules to partition the large-scale network into small network segments in order to efficiently compute an AG in a data-centric network. We utilize the SDN controller to inspect network events and obtain a global view of the network and service reachability. SDN controller interacts with distributed firewall's *Security Policy Database* (SPD) at the application plane to obtain an updated list of *security policies*. The data-plane is based on OpenFlow switches. SDN controller installs DFW rules on connected switches at data-plane layer to limit traffic across different logical segments. We apply micro-segmentation [3,21] in order to divide the large data center into small segments based on granular security policies. This helps in restricting the lateral movements along *east-west* communication paths among network segments.

In general, S3 framework follows a *divide-and-conquer* approach to generate AG for each segment (*sub-AG*) after obtaining vulnerability and reachability information from vulnerability analysis tools. Finally, the result of divide-and-conquer approach (*sub-AGs*) are merged based on the $|DFW|$ rules to obtain the system AG, or we call it the Composite Attack Graph (CAG). The key contributions of S3 are as follows:

- Our proposed S3-based micro-segmentation algorithm is able to generate a scalable AG by utilizing DFW rules. The algorithm complexity We achieved is $O((\frac{N}{K})^2) + O(K \log(K))$, where N is the total number of vulnerabilities and K is the number of established segments. The AG generation time achieved using S3 is much faster compared to state-of-the-art AG generation tools, i.e., S3 takes ~ 20 sec for generating AG for a network with services over 6000 services, as shown in Section 6, compared to over 1 hour taken by MulVAL [27].
- We propose a granular *divide-and-conquer* based security state management approach for large data center network *micro-segmentation*, which utilizes DFW to significantly reduce the AG density and number of attack paths. Consequently, our approach is able to generate a scalable AG by leveraging SDN capabilities and utilizing DFW to obtain real-time security state policies.

The rest of the paper is organized as follows, Section 2 provides a literature review of existing AG generation methods and DFW frameworks, along with their limitations. We discuss the threat model, S3 *architecture* and the details about *AG formalism* in the Section 3. We discuss the SDN-based S3 system and *API architecture* in Section 4. The description of

S3-based *micro-segmentation* approach, graph segmentation *algorithm*, optimal number of micro-segments, and a proof of scalability of *micro-segmentation* method has been provided in the Section 5. The evaluation of AG *scalability*, SDN *controller overhead*, and experimental details on *optimal number of micro-segments* is discussed in Section 6. In Section 7, we discuss related issues such as cycles in directed graphs, alternative segmentation heuristics, and possible security policy conflicts that can be induced by the *micro-segmentation*. Finally we conclude the research paper in Section 8, and provide details on the future work.

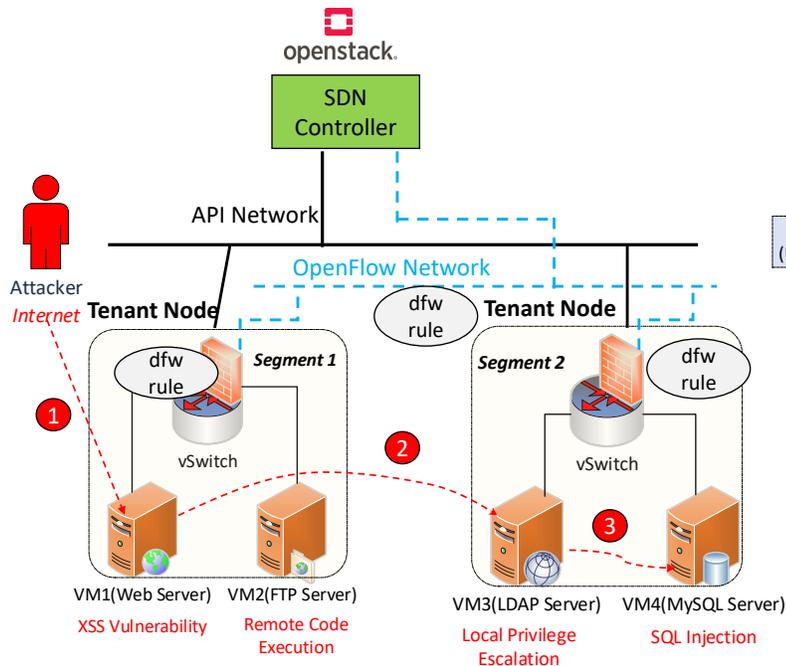
2 Related Work

The Scalability of Attack Graphs: Generation of scalable attack graphs has been a popular area of research. Amman *et al.* [1] presented a scalable solution in comparison with prior modules [32] by assuming *monotonicity*. This assumption allowed them to achieve scalability of $O(N^6)$ [19]. To mitigate the state space explosion problem, most of the existing solutions try to *reduce the dependency* among *vulnerabilities* by using logical representation [27]. Hong *et al.* [18] apply a *hierarchical strategy* to reduce the computing and analysis complexity of constructing and using AGs by *grouping and dividing* the connectivity of the system into hierarchical architecture. The performance time is, however, $\sim 50s$ for 50 services, whereas our framework generates scalable attack graph of similar scale in 2.2s.

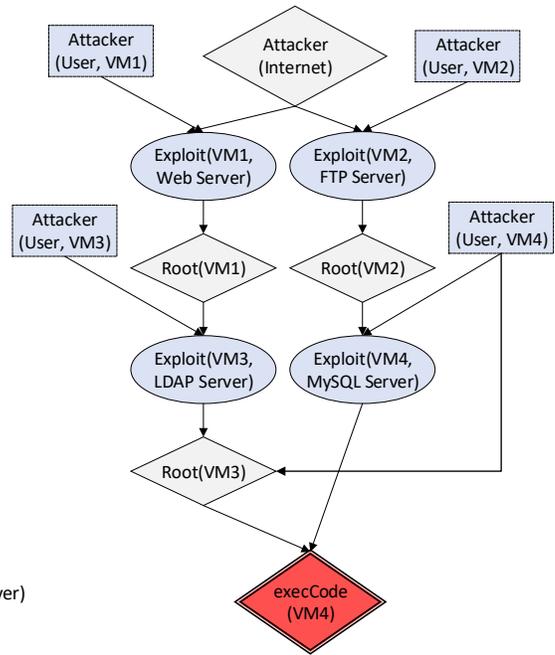
Kaynar and Sivrikaya [23] proposed a framework for distributed AG generation that utilizes a shared memory approach. The graph generation time is of order 2-3 minutes for 450 hosts, which *cannot be used for real-time security analysis*. Chowdhary *et al.* [5], use distributed hypergraph partitioning for Attack Graph generation. The research work has however not considered application of DFW for further optimizing the size of Attack Graph. Cao *et al.* [2] proposed an approach to compute AG in parallel. The division is based on the privileges inside the hosts. The experimental analysis in this work shows that the required generation time for ~ 500 hosts is ~ 20 sec, while in our work it takes only ~ 6.5 sec.

Mjihil *et al.* [25] used a parallel graph decomposition approach. The evaluation in this research work tests the effect of the number of vulnerabilities on the AG generation time, which is not reliable since it does not explain how the number of vulnerabilities is related to each service in the system as we did in this paper. The research work tested a maximum of 50 vulnerabilities in which they obtained an AG in ~ 10 sec, while in our work we obtain an AG in ~ 2 sec.

Distributed Firewall: Some researchers have addressed DFW in SDN [20] [30] [28]. Yet, they only consider stateless firewall which does not leverage the full advantage of both SDN and DFW. VMware has proposed a distributed firewall for their *NSX model*, by using a central object that manages the distributed firewall's policies [26]. Unfortunately, this ar-



(a) Multi-tenant Cloud network with attacker's lateral movement



(b) Attack graph with attack goal VM4 (MySQL Server)

Figure 1: Representation of vulnerability information and corresponding attack graph in a multi-tenant data-center network. Distributed Firewall (DFW) can be implemented at each tenant/segment.

architecture is only applicable to the NSX model and cannot be adopted to OpenFlow standards because NSX comprises of both *stateful* and *stateless* components. The firewall rules of the host machines are also controlled by the NSX manager, whereas the OpenFlow protocol based SDN framework implements a *stateless firewall*. There are no existing research works, which attempt to control the security state explosion in attack representation methods using better security policy design and management framework. In S3 framework, we utilize design principles based on VMWare NSX architecture, and the programming flexibility afforded by the SDN, in order to *limit the connections* between different logically separated regions of a data-center.

3 Background

3.1 Threat Model

In this section, we describe threat model, AG model, and AG scalability challenge in order to motivate the need for a scalable attack representation framework and allow the reader to have a comprehensive understanding of AG scalability issue.

To explain the attack graph more clearly, we show Figure 1 (a), which illustrates a simple multi-tenant data-center based on Openstack framework. The network in this example con-

sists of 4 virtual machines (VM1-4), but this general model can be applied to large scale cloud networks. The OpenStack management framework can be used by the security administrator in order to insert DFW policies or monitor the status of each VM present on tenant nodes.

The attacker is located on the *Internet*. The *attack model* here is vulnerability exploitation to achieve privilege escalation. There are several attack paths the attacker may take to achieve their *goal*, which is to ex-filtrate data from the *MySQL* server, by obtaining *root privileges* on *VM4*, i.e., *execCode(VM4)*.

Lateral Movement: The scope of security enforcement offered by a traditional firewall is limited to *north-south* traffic, i.e., firewall serves as a sentry between trusted and untrusted networks. Once the attacker has managed to breach the security restrictions at the network edge, they can laterally move inside the network (*east-west* traffic) and exploiting key resources virtually unchecked. Centralized firewalls do not protect networks from *multi-stage* attacks using *lateral movement*. Since everyone on the internal networks is trusted and the traffic within these trusted networks is not rigorously inspected by the traditional firewall based defense mechanisms.

The volume of *east-west* traffic in the datacenter environment is around 76%, as compared to *north-south* traffic - 17% [7]. As shown in Figure 1 (a), if the attacker can compromise Web Service on VM1 in Step (1) of the multi-stage

attack, they can use this attack as a pivot for compromising VM3, and VM4 in steps (2) and (3). The lateral movement is hard to detect and prevent using traditional security architecture since in most cases its intended purpose is to defend the network system against outsider adversaries.

3.2 Attack Graphs and Scalability Challenge

Data-center networks are scaling up at a fast pace. For example, *Amazon Web Services (AWS)* has on average between 50,000 servers, to 80,000 servers, according to [22]. An efficient security analysis of such data-center is expected to be scalable and granular. Hence, there is a need for scalable security analysis, and AG serves this purpose to module the critical paths in the system.

In this paper, our research uses the *exploit dependency graph* [4], since it directly models dependencies between the vulnerabilities in a computer networked system. Also, all services for application-based on networked-based attacks are related in this graph and it shows what are the pre-requisites (pre-conditions) and post-conditions for those attacks. *Nodes* in such an AG are not the network states, but rather, they are *vulnerabilities*. AG can be formally defined as follows:

Definition 1. (*Attack Graph (AG)*) An attack graph is represented as a graph $G = \{V, E\}$, where V is the set of nodes and E is the set of edges of the graph G , where

1. $V = N_C \cup N_D \cup N_R$, where N_C denotes the set of conjunctive or exploit nodes (pre-condition), N_D is a set of disjunctive nodes or result of an exploit (post-condition), and N_R is the set of a starting nodes of an attack graph, i.e. root nodes.
2. $E = E_{pre} \cup E_{post}$ are sets of directed edges, such that $e \in E_{pre} \subseteq N_D \times N_C$, i.e., N_C must be satisfied to obtain N_D . An edge $e \in E_{post} \subseteq N_C \times N_D$ means that condition N_C leads to the consequence N_D .

An example of *vulnerability* information, network service information, and *Host Access Control List (HACL)* represented in *datalog* format is shown below:

```
vulExists(ipaddr, cve-id, service)
networkServiceInfo(ipaddr, service, prot, port)
hacl(srcip, dstip, prot, port)
```

Attack graph uses HACL tuples to model network and firewall configurations, in which it uses a general rule to test and specify reachability information (i.e., any host can access any host using any port and protocol).

Attack Graph Scalability Challenge: As can be seen in the Figure 1 (b), a network consisting from 4 hosts resulted in a graph of 13 nodes. Large data-center networks have thousands of services, servers, and VMs. The expected AG size of such system is huge due to representing network state using a

conditional or combination of conditional and exploit representation of security situation, which will lead to huge number of nodes and edges in the AG [24, 34]. Therefore, an efficient and scalable methodology is needed to help the administrator in representing and analyzing the security situation in the system.

4 System Model and Architecture

4.1 S3 System Architecture

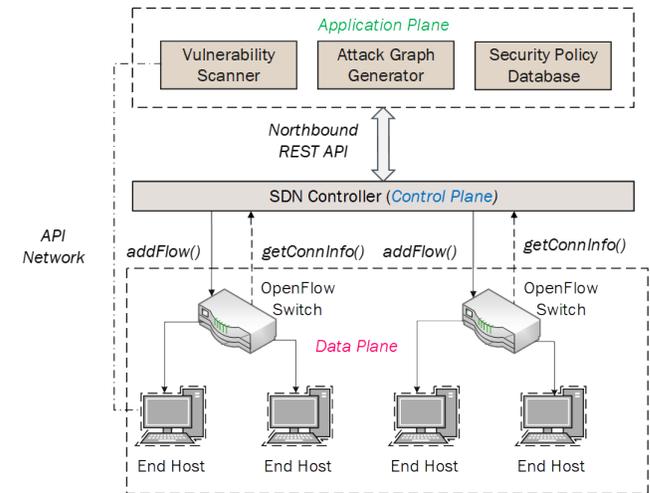


Figure 2: S3 System Architecture and operating layers

We consider the cloud infrastructure shown in Figure 2 as the architecture for framework S3, where the networking infrastructure is based on *Software Defined Networking (SDN)* solutions. SDN is an emerging technology aiming to enhance the current networking protocols by separating control-plane from data-plane. The *Application Plane* comprises of *vulnerability scanner* (Nessus) which collects vulnerability information from each network host. The vulnerability scanner, on the other hand, interacts with the individual hosts at *data plane* using *API network*.

The *Security Policy Database (SPD)*, as shown in Figure 2, creates security policies to define micro-segmentation policies. These policies dictate how segments are created. The traffic between segments is regulated using security policies defined by SPD. The SPD interacts with SDN controller using *northbound REST APIs* to update *security policy* information.

The *Attack Graph Generator* module, which is a wrapper program we developed for the generation of *sub-AGs* at the level of each segment. This module interacts with a vulnerability scanner and SPD to create segments using an algorithm, and finally, utilize *merge* algorithm to merge segments into fully connected AG.

S3 framework utilizes OpenFlow *southbound APIs* to provide flexible and programmable micro-segmentation archi-

Algorithm 1 Segmentation and Scalable AG Generation Algorithm

```
1: procedure SEGMENT ESTABLISHMENT
2:  $V \leftarrow \{1, n\}$   $\triangleright$  list of vulnerabilities
3:  $R \leftarrow \{1, m\}$   $\triangleright$  distributed firewall rules
4:  $S \leftarrow \{1, 1\}$   $\triangleright$  network services
5:   for all service  $\in S$ , vuln  $\in V$ , rule  $\in R$  do
6:     sort(service, type)
7:     if connected(servicei, servicej, rulek) & vuln
       $\in$  (servicei, servicej) then
8:       update(segment, servicei, servicej)
9:     else if type(servicei) == type(servicej) then
10:      update(segment, servicei, servicej)
11:    end if
12:  end for
13: procedure ATTACK GRAPH GENERATION
14:   for all Segments do
15:     Compute sub-AG for segmenti
16:   end for
17:   for all subAGs do
18:     if connected(subAGi, subAGj, rulek) then
19:       merge(subAGi, subAGj)
20:     end if
21:   end for
```

about the existing vulnerabilities residing in the system and provides a methodology to build *logical segments* containing services based on their vulnerability and reachability information. The flow diagram of S3 is presented in Figure 4. The network services are first analyzed and sorted according to the service type and connectivity between the services based on the $|DFW|$ rules. If the services are of the same type, then we append them to segments. If the services are connected by $|DFW|$ rule, and there is a vulnerability in the connected services, then we also append the services into different separate segments. We separated the segments because we want to keep services of same type in one segment and separated from other ones having different type. After that, we compute the sub-AG of all the segments and check for connectivity between the sub-AG based on the $|DFW|$ as well. Finally, the sub-AG from individual segments are merged and *Composite Attack Graph* (CAG) is created. The graph is *frequently updated*, based on changes in network and vulnerability configurations, which are *monitored* by SDN controller. Next, we present an Algorithm 1, which explains more on how the segmentation is achieved for the of building scalable CAG.

5.1.1 Algorithm Analysis

S3 segmentation and AG generation approach is shown in Algorithm 1. The algorithm first starts by obtaining all vulnerabilities, running services, and $|DFW|$ rules in the system lines 2-4. Next, the algorithm span over all of these gathered

information and sorts the services by their type as shown in line 5. After that, the algorithm checks for services that are of the same type and put them in the same logical segment. Also, if there are two services connected to each other by a $|DFW|$ rule such that there is a vulnerability in service_i allowing the attacker to exploit a vulnerability in service_j, then these two services are aggregated in the same segment. To avoid redundancy, if the services have been already added to an existing segment, the algorithm continues. After sorting and aggregating services, vulnerabilities, and $|DFW|$ rules, we now have a number of segments that contain services of similar type, or vulnerable services connected by $|DFW|$ rule. The next step is to compute an AG for each of these segments to check and the critical paths inside each of these segments - lines 13-15. Next, the sub-AG needs to be merged together to allow the system administrator to see the relationship between these segments and how an adversary can move from one segment to another one to achieve their final goal. The merge procedure, as shown in line 16-19 is based on the connectivity of the separated segments.

5.1.2 Complexity Analysis

The sort operation over services in algorithm 1 can be performed using quick sort algorithm [9] which has average complexity of $O(S \log(S))$, where S is the number of services. Computing the sub-AG is a *linear* time operation as the computation is being performed in parallel with the help of S3 SDN controller. The merge operation requires searching among the segments and their connectivity, thus the search can be done using *divide and conquer* approach which has the complexity of $O(K \log(K))$ [9], where K is the number of segments. As a result, the complexity of computing the global view of AG is based on the total number of vulnerabilities in the system N , divided by the number of segments K , plus the complexity for the merge operations, a total of $O((\frac{N}{K})^2) + O(K \log(K)) + O(S \log(S)) \sim O((\frac{N}{K})^2)$ when $N \gg K$ and $N \gg S$.

To make the full picture more clear about the resulting AG, we present the following abstraction representation definitions of AG:

Definition 2. Composite Attack Graph (CAG) is a tuple $CAG = \{S, E, N\}$.

- S denotes the set of all segments. Each segment has a sub attack graph (sub-AG), i.e., $sub-AG_1, sub-AG_2 \in S$. If there is a connectivity ($|DFW|$ rule) from one service in segment s to service present in segment s' , $s \neq s'$, where this connection is the one required to exploit a vulnerability present on service in s' , then this information (post and pre-condition, vulnerability and connectivity) are appended and concatenated in a segment.
- N denotes the nodes the set of all nodes present in the CAG. A node in an individual segment s can be denoted

by N^s . The nodes can be conjunct nodes N_C^s , disjunct node N_D^s or root node N_R^s . A link from segment s to segment s' indicates reachability to a vulnerability in the target segment s' , or a $|DFW|$ rule that exists based on the Service Level Agreement (SLA). In other words, this link is the result from exploiting segment s which we call post-condition that is needed for the attacker to reach and exploit s' . Hence the post-condition from s becomes a pre-condition s' .

- $E \subseteq E_{pre}^s \cup E_{post}^s$ is the edges present across all segments. If an edge from segment (sub-AG) s creates a post-condition in segment (sub-AG) s' , we denote the post-condition using edge $E_{post}^{s'} = N_C^s \times N_D^{s'}$.

5.2 Optimal Micro-Segmentation and Validation

In a large cloud network, when creating segmentation, the natural question is to identify the *quality* of segmentation. The network administrator needs to identify the *optimal number* (K) of segments and the basis of segmentation. We use a heuristic approach based on service similarity, vulnerability weight, and DFW rules, in order to validate segment quality and optimal number (K). We define segmentation properties used in S3 framework as follows.

1. **Segment Compactness:** Evaluates how closely the services in the same segment are related to each other. For instance, all *Web Server* (*http/https*) services can be put in the same segment. We represent this using variable s_{com} .
2. **Separation:** The separation can be decided based on the number of $|DFW|$ rules currently present between two segments. The higher the number, the higher the distance will be between the segments. Variable s_d is used for representing separation measure.
3. **Connectivity:** The connectivity depends on dependencies between vulnerabilities in the same segment as described in Definition 2. The connectivity is denoted using variable s_{con} .

The segmentation procedure aims at finding segmentation with high separation across segments, and high connectivity between nodes in the same segment. We utilize *Segmentation Index* (SI), an indexing measure based on *Dunn Index* [11], often used in *K-Means* clustering [15] approach in order to validate the quality of our micro-segmentation. Using the variables enumerated above, we define segmentation index as,

$$SI = \frac{\alpha \times \{s_{com} + s_{con}\}}{\beta \times s_d}, \alpha, \beta \in (0, 1], \quad (1)$$

where α and β are indexing parameters, their values are chosen based on the administrator's need. This indexing equation can help the system administrator to decide the optimal number of segments based on the desired requirements. If $\alpha > \beta$, it means the SI places higher weight on connectivity between services in the same segment. On the other hand if $\alpha < \beta$, the SI places higher weight on $|DFW|$ rules between the segments. High segmentation index value indicates optimal number of segments, since this equation is derived from *Dunn Index*, which also aims at finding the maximum distance between clusters [11]. We show in the Evaluation section 6.4 through experimental results how SI has been used for identifying optimal number of segments.

5.3 DFW Dynamic Traffic Match and Flow Update

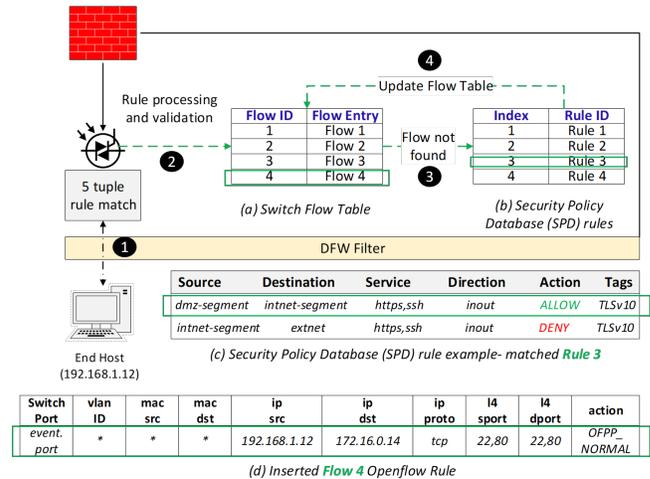


Figure 5: Distributed Firewall (DFW) security policy rule match and flow table update.

The DFW utilizes OpenFlow and REST API network to match the traffic based on five tuples, i.e., $\{srcip, dstip, sport, dstport, protocol\}$. We use the example Figure 5, to illustrate DFW traffic match and rule update operations.

- **Step 1:** The *end-host* (192.168.1.12) from *intranet-segment*, attempts to send *http* traffic to port 80 and *ssh* traffic to port 22 of host (172.16.0.14) situated in another segment *dmz-segment*.
- **Step 2:** Initially, when flow table is checked using *table_lookup*, there is no rule present for the matching traffic rule. The flow table only has rules with *Flow ID* {1-3} - Figure 5 (a). The packet is sent to the controller using *action=OFPP_CONTROLLER*.
- **Step 3:** The controller checks the security policies defined by the *Security Policy Database* (SPD) rules

present in application plane, using northbound REST API. The traffic pattern matches the *Rule ID {3}* - Figure 5 (b), (c). The action defined in the SPD for this traffic is *ALLOW*.

- **Step 4:** The flow table is updated with new OpenFlow rule - *Flow ID {4}*. The fields corresponding to layer 3,4 are updated and layer 2 fields are wildcarded - Figure 5 (d). Thus communication is enabled between two hosts. If there is no match for the traffic, in either flow table or SPD, the traffic is discarded based on whitelisting policy.

5.4 Scalable Attack Graph Generation Cost Analysis

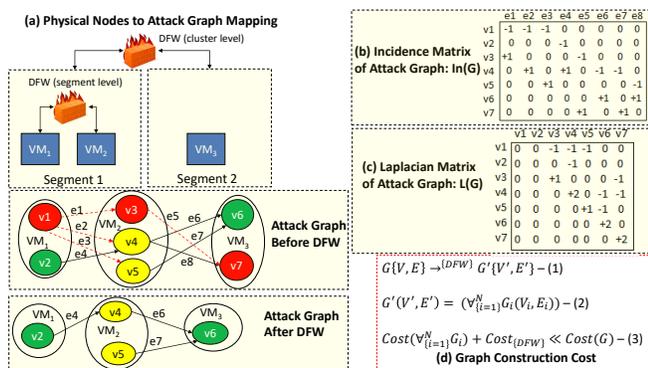


Figure 6: Distributed Firewall-based Multi-Level AG Generation.

We consider the mapping between the physical network and virtual network shown in Figure 6 (a). The physical topology consists of two segments, i.e., *Segment 1* and *Segment 2*, with $VM_1, VM_2 \in Segment 1$ and $VM_3 \in Segment 2$. Each VM consists of a number of services such as apache2, mysql, etc. The connectivity relation between the VMs are used to determine the AG for the entire network. For instance, if the firewall rules are defined between VMs and segments in a coarse grained manner, the AG will be huge in size as shown in *Before DFW* case in the above Figure 6. The traffic across each segment according to whitelisting policy might be limited, whereas, if we enforce white-listing policy at segment and service (SSH, MySQL) level as shown in Figure 5 (c), the amount traffic can be scale of attack graph generated can be finite for security analysis. Once the DFW is enforced at different levels of network, i.e., at the granularity of per-VM, per-segment, or entire network, we obtain a sparse AG, as shown in *After DFW* in Figure 6 (a). We define the *Incidence* and *Laplacian* matrices for the attack graph G below,

Definition 3. Incidence Matrix The incidence matrix $In(G)$ of graph $G\{V, E\}$ is a $|V| \times |E|$ matrix, as shown in the Figure 6 (b), with one row for each node and one column for each edge. For each edge $e(i, j) \in E$, column entry e of $In(G)$ is zero,

except for i^{th} and j^{th} entries, which are $+1$ and -1 , respectively (if there is an edge from i to j , the value is $+1$, whereas it is -1 if there is an edge from j to i in the graph, the value is zero if there is no edge $e(i, j)$).

Definition 4. Laplacian Matrix The Laplacian matrix $L(G)$ of graph $G\{V, E\}$ as shown in the Figure 6 (c), is a $|V| \times |V|$ symmetric matrix, with one row and column for each node. It is defined by

- $L(G) (i, i)$: is the degree of node I (number of incident edges).
- $L(G) (i, j)$: -1 if $i \neq j$ and there is an edge (i, j) .
- $L(G) (i, j)$: 0 otherwise.

The application of DFW at different levels of the physical and logical network increases graph sparsity. The aggregated graph has reduced state space compared to the original AG.

5.5 Sparse Graph Connectivity using DFW

The incidence graph $In(G)$ and laplacian graph $L(G)$ have the following properties.

- $L(G)$ is symmetric, i.e., eigenvalues of $L(G)$ are real and its eigenvectors are real and orthogonal. For example, let $e = [1, \dots, 1]^T$ be a column vector. Then $L(G) \times e = 0$.
- Matrices are independent of signs chosen for each column of $In(G)$, $In(G) \times In(G)^T = L(G)$.
- Let $L(G) \times v = \lambda \times v$ and $\lambda \neq 0$, where v is eigenvector and λ is eigenvalue of $L(G)$,

$$\lambda = \frac{\|In(G)^T - v\|^2}{\|v\|^2}$$

$$\lambda = \frac{\sum_{e(i,j) \in E} (v(i) - v(j))^2}{\sum_i v(i)^2} \quad (2)$$

- Eigenvalues of $L(G)$ are non-negative, i.e., $0 = \lambda_1 \leq \lambda_2 \dots \leq \lambda_n$.
- The number of connected components of G is equal to number of λ_i equal to 0. In particular, $\lambda_2 \neq 0$ if & only if G is connected.

Using the properties defined above, we check the algebraic connectivity of two graphs G and G' , which can be compared to check the density reduction. The graph $G'\{V', E'\}$ obtained in the case of *After DFW* scenario, is composed on sub attack graphs (sub-AGs), G_1, G_2, \dots, G_n , i.e., $G'\{V', E'\} = \cup_{i=1}^N G_i$, as shown in Figure 6 (d). Since $G'\{V', E'\}$ is obtained from $G\{V, E\}$ after collapsing vertices and edges at different layers using a multi-level DFW, it naturally follows that G' is a subgraph of G , i.e., $G' \subseteq G$. We utilize an important corollary from spectral bisection algorithm [33] and the properties of

laplacian matrix discussed in this subsection to derive the equation $\lambda_2(L(G')) \leq \lambda_2(L(G))$.

Result: $G'\{V', E'\} \subseteq G\{V, E\} \rightarrow \lambda_2(L(G')) \leq \lambda_2(L(G))$, i.e., on application of DFW, the algebraic connectivity, and in effect, density of the AG reduces. Thus, our approach, helps in creating *scalable AGs* (CAG) in a multi-tenant cloud network.

Cost Analysis: *Upped bound* on the cost can be obtained by considering that graph $G\{V, E\}$ is *fully connected*, in which case, the *micro-segmentation* will not be able to achieve noticeable benefit. The cost of generating the full AG in the absence of DFW, $Cost(G)$, is much higher than in the case of using DFW. The goal of *micro-segmentation*, however, is to ensure that the graph is sparsely connected based on white-listing approach.

Consequently, $Cost(G') = \forall_{i=1}^N Cost(G_i) + Cost(DFW)$ - Figure 6 (d) and $Cost(G') \ll Cost(G)$ since the effort for generation of graphs G_1, \dots, G_i is computed in parallel with the help of SDN controller. The only additional effort $Cost(DFW)$ is needed for checking DFW rules, and maintaining synchronization between different DFW agents present on individual *segments*.

6 Performance Evaluation

In order to evaluate and measure the performance of our proposed approach. First, we created the system is shown in Figure 2, which is a *OpenStack* [31] based system that is running SDN controller and a number of virtual machines (VMs) connected to OpenFlow switches. Our evaluation consists of evaluating the scalability of AG when the number of vulnerabilities increases, in which S3 proved to have a reduced number of nodes and edges compared to not using S3. Our second experimental evaluation is to measure the AG generation time when the number of services increase, taking into account the generated number of segments as shown in Table 2. Moreover, since S3 is utilizing SDN computing capabilities, we conducted experiments to check how much overhead our algorithm and AG module consume from the SDN controller, which turned out not exceed 11% from the overall SDN bandwidth and an optimal number of segments in a scalable AG.

6.1 Attack Graph scalability Evaluation

As we mentioned in the introduction, the number of vulnerabilities have a direct influence on the AG solubility due to the overhead of managing and analyzing all the security state those vulnerabilities cause. To show how scalable S3 is, we simulated a system with a different number of vulnerabilities as shown in Figure 7, the vulnerabilities in the OpenStack based cloud system. The Figure 7 emphasizes on the relationship between the number of vulnerabilities and the size of the resulted AG in terms of nodes and edges, where the x-axis shows the total number of vulnerabilities in the entire system,

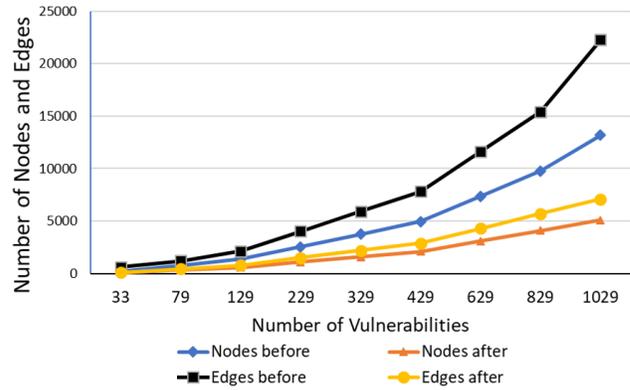


Figure 7: Comparison for the number of Nodes and Edges before and after Using S3. Marked reduction in density achieved using DFW

and the y-axis shows the number of nodes and edges in the AG. The black and blue lines show the number of nodes and edges in the AG before using our approach (which is equivalent to MulVAL's [27] approach), respectively. The red and yellow lines show the number of nodes and edges in the AG after using our approach, respectively. The total number of nodes and edges before using S3 when the system has over 1000 vulnerabilities is about 13k nodes and 22k edges. This is due to the absence of $|DFW|$ rules affecting the reachability between the individual components in the system. After using S3, where the exact reachability information that is being enforced by the $|DFW|$ is stated, the number of nodes drop to about 5k and the number of edges is 7k, respectively - Figure 7. This is a significant reduction compared to an AG without any $|DFW|$ rules.

6.2 Attack Graph Generation Time and density Reduction Evaluation

It is crucial to generating AG in a timely manner. We created several test cases to test the time required to generate the AG when we have a different number of segments, and a different number of services in each of those segments. We first started to measure the generation time of AG in a system that contains 50-100 services, we inserted a mixture of vulnerabilities in the hosts such that we obtain the provided number of segments shown in Table 2. Thus, in the first experiment, we are testing how much time is needed to generate an AG for a system having 50-100 services with a various number of vulnerabilities on those services that resulted in 5 segments. Moreover, we are measuring the graph density of the resulted AG using the following formula [10]:

$$Density = \frac{|E|}{|V|(|V|-1)}, \quad (3)$$

Table 2: Sub-AG generation time, graph density, and the number of nodes and edges for each sub-AG when increasing the number of services.

# Services	50-100 Services				100-200 Services				200-300 Services				300-500 Services			
	5	10	15	20	5	10	15	20	5	10	15	20	5	10	15	20
#Segments	5	10	15	20	5	10	15	20	5	10	15	20	5	10	15	20
Time (sec)	2.22	3.88	5.925	8.22	2.386	4.93	7.2112	10.229	3.56	7.15	10.6	13.96	6.46	11.05	15.91	19.7
# Edges	6552	12186	18990	27450	14400	28494	40698	52956	18819	44100	63918	88065	34242	65922	93117	128580
# Nodes	5829	10842	16895	24420	12805	25338	36191	47092	18101	39210	57951	79668	32533	60698	85623	116304
Density	19.3E-05	10.4E-05	6.6E-05	4.6E-05	8.8E-05	4.44E-05	3.1E-05	2.4E-05	5.7E-05	2.9E-05	1.9E-05	1.4E-05	3.2E-05	1.8E-05	1.3E-05	9.5E-06

where $|E|$ is the total number of edges for the AG, and $|V|$ is the total number of nodes or vertices in the AG. Our evaluation and approach show scalable AG generation. For instance, in the last case in Table 2 where the system has 300-500 services and it was divided based on the vulnerabilities in the system into 20 segments, the AG generation time is about 20 seconds, which is rational time for such a large system. In Table 3, we show the average time for AG generation and the standard deviation for the 5, 10, 15, and 20 segments cases respectively.

To prove the effectiveness of our DFW-based segmentation approach, S3, we conducted additional experiments to examine the generation time by not considering the segmentation and using a *Firewall* (centralized one); and segmentation by DFW. Table 4 shows the AG generation time with and without segmentation, for the specified number of hosts where the vulnerabilities are simulated to give the shown number of segments. The results when using $|DFW|$ are significantly better than when not using segmentation and using a centralized firewall. This is due to the absence of east/west traffic among running services, which did not specify reachability information between running services. Hence, AG is computing centrally and resulted in a magnificent time.

Table 3: Mean and standard deviation for the AG generation time for the displayed number of segments in Table 2.

#Segments	5	10	15	20
Mean time	3.66	6.75	9.91	13.03
standard deviation	1.96	3.17	4.46	5.04

Table 4: sub-AG scalability generation time by using DFW, and both with and without Segmentation

# Services	# Segments	Generation Time without Segmentation (sec)	Generation time using Segmentation (sec)
750	5	3.51	0.872
1450	10	17.344	2.083
2490	15	4980	4.468
3360	20	6720	10.027

6.3 SDN Controller Overhead

Since S3 is based on an SDN-managed data-center network, evaluating the overhead of computing AG using SDN controller is necessary to ensure that the AG generation does not overwhelm the SDN controller, since this may result in service disruption for end users. Specifically, our proposed algorithm 1 line 15 relies on the SDN controller to compute the sub-AGs for all the obtained segments. Hence, we measure the effect of this operation to the SDN controller bandwidth.

To do this, we used the first case in Table 2, where the system has 50-100 services (~ 4000 vulnerabilities) running and emulated the scenario. We utilized network throughput measurement tool *iperf* to assess end-to-end bandwidth. Figure 8 shows a comparison of the SDN controller bandwidth overhead before the AG computation takes place and during computation. The evaluation results are an average of *three runs*. The network throughput for a network with 5 segments was around ~ 11.3 Gbps without micro-segmentation. On incorporating micro-segmentation, the throughput decreases to 9.95 Gbps. Similarly, for the case with 10, 15 and 20 network segments, the throughput drops slightly, as expected.

This drop can be explained as the overhead induced by AG generation in each network segment, and the computation required to merge individual segments into full AG. The worst-case throughput impact on SDN controller was $\sim 10\%$ (20 segment case). This experiment shows that on an average the scalable AG generation process will not impact the SDN controller's performance in a large data-center network.

6.4 Optimal Number of Segment Experiments

We conducted a simulation experiment to identify an optimal number of segments in a large network with 50 services and 50 vulnerabilities. We varied the number of segments from 5 to 25, with an increasing number of DFW rules (3 in case of 5 segments, 17 in case of 25 segments), induced by the increase in the number of segments.

In section 5.2, we showed and discussed a heuristic approach to evaluate what is the optimal number of segmentation based on the derived equation for *Segmentation Index* (SI) 1. The equation showed, depending on the system administrator requirement, how to obtain the optimum number

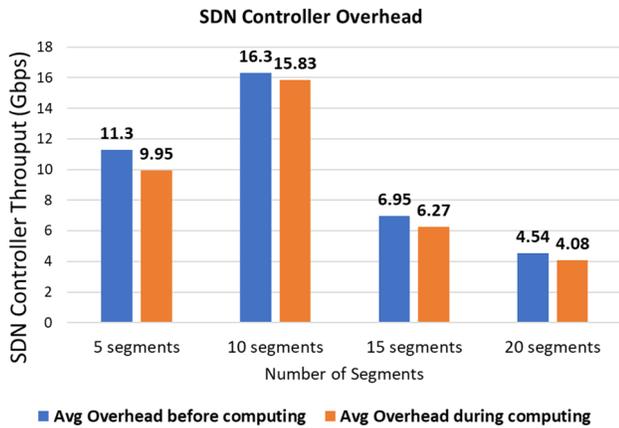


Figure 8: Evaluation of SDN Controller overhead when before computing segmented AG and during computation shows limited overhead

of segments, *i.e.* whether connectivity is more important or $|DFW|$ rule. The Figure 9 shows our experiments, where the blue line indicates the segments have *high connectivity*, *i.e.* $s_{con} + s_{com} > s_d$ ($\alpha = 0.877, \beta = 0.105$) between services in one segment, the orange line indicates *higher separation* between the segments based on the number of $|DFW|$ rules, *i.e.*, $s_{con} + s_{com} < s_d$ ($\alpha = 0.4, \beta = 0.877$). Finally, the black line shows an *equal weight* for *connectivity* and number of $|DFW|$ rules, *i.e.*, $s_{con} + s_{com} = s_d$ ($\alpha = 0.5, \beta = 0.5$).

For a small number of segments (5), the connectivity influences the segmentation index (SI=8.0 high connectivity, SI<1 when DFW rules are dominant). This is because of the high degree of intra-segment traffic. As the number of segments, increase (15) the connectivity becomes much less of a factor and drops drastically (SI=2.0 for 15 segments).

From Figure 9, it is shown that the *optimal number of segments turned out to be 20* since the SI > 10 for high connectivity and SI also increases steadily for cases where firewall rules weight is high. This can be explained by the fact that dependencies between vulnerabilities in each segment are reduced, using traffic regulation provided by DFW. Finally, increasing the number of segments more (25) turned out to have a low SI value for all 3 lines, which indicates segments are disconnected from each other.

7 Discussion

Cycle Detection: The dependencies between services in a network can cause cycles in the directed AG. Homer *et. al.* [17] discussed the problems of cycles that can limit the scalability of AGs. The research work takes about 150 ms for cycle detection over a network with 10 hosts and 46 vulnerabilities. S3 utilizes the network connectivity and vulnerability dependency information to detect any cycles present in the final directed AG. We use parallel *nested Depth First Search* [16]

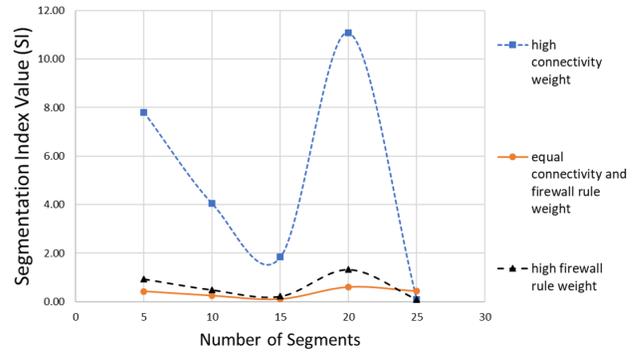


Figure 9: Optimal Number of Segments Evaluation shows 20 segments to have high Segmentation Index (SI)

over each *sub-AG* in order to identify the cycles present within each segment. The algorithm scales linearly with the number of vulnerabilities present in each network segment. We omit details on *cycle detection* in the paper for the sake of brevity.

Segment Validation and Segmentation Heuristics: We utilized a *Segmentation Index* based sub-AG (segment), that has a validation heuristic approach. The algorithm provides information about the appropriate size of each segment, such that not only the complexity concerns for AG generation are addressed, but also each segment is highly *cohesive* (has the same type of services and vulnerabilities). This will help in the application of security patches to the entire segment. There are other segmentation heuristics, classified under *graph clustering* algorithms, *e.g.*, *k-spanning tree*, which creates k-groups of non-overlapping vertices, *shared nearest neighbor* (SNN) graph. We plan to compare the optimal segmentation heuristic discussed in Section 5.2 with other state-of-the-art graph segmentation heuristics in future work.

Policy Conflicts and SLA Impact: It is important taking into account the *Service Level Agreement (SLA)* that states the relationship between a service provider and client. This *SLA* will have an impact on the $|DFW|$ rule that the system administrator will enforce to create segments and isolate vulnerable services from protected ones. After applying segmentation and deriving a new $|DFW|$ rules, a conflict might exist between the *SLA* and the $|DFW|$ rules. In effect, the $|DFW|$ rule might cause a service disruption for users. Security policy conflict [14, 29] handling, however, is another area of research that will be considered as a part of future work.

8 Conclusion

Attack graph scalability and granular security enforcement are key problems in data-centric networks today. We provide a SDN-based *micro-segmentation* approach using *S3* framework for addressing these issues. *S3* enforces granular security policies in the data-center network to deal with threats such as *lateral movement* of the attack. *S3* reduces the number

of security states in the network by reducing attack graph density and generation time as evident from section 6.2. The *micro-segmentation* approach is capable of establishing and generating a scalable AG for a large network - Section 6.1. Moreover, the impact on the SDN controller because of *micro-segmentation* is limited, as proved from the experimental analysis in Section 6.3. We also identified optimal number of segments using *Segmentation Index* (SI) method, which can ensure high-quality (cohesive) segments with fine-grained access control policies across segments - Section 5.2. The current research work doesn't identify the security policy conflicts that can be induced by co-dependency between *micro-segmentation* policies. Additionally, we have not compared our segmentation method with a diverse set of graph segmentation/clustering heuristics. In the future, we plan to address these limitations.

Acknowledgment

All authors are thankful for research grants from Naval Research Lab N00173-15-G017, N0017319-1-G002 and National Science Foundation US DGE-1723440, OAC-1642031, SaTC-1528099. Special thank to Jim Kirby from NRL for the valuable feedback on the paper. Also, Abdulhakim Sabur is a scholarship recipient from Taibah University through Saudi Arabian Cultural Mission (SACM).

References

- [1] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224. ACM, 2002.
- [2] Ningyuan Cao, Kun Lv, and Changzhen Hu. An attack graph generation method based on parallel computing. In *International Conference on Science of Cyber Security*, pages 34–48. Springer, 2018.
- [3] Ramaswamy Chandramouli and Ramaswamy Chandramouli. Secure virtual network configuration for virtual machine (vm) protection. *NIST Special Publication*, 800:125B, 2016.
- [4] Ioannis Chochliouros, Anastasia S Spiliopoulou, and Stergios P Chochliouros. Methods for dependability and security analysis of large networks. In *Encyclopedia of Multimedia Technology and Networking, Second Edition*, pages 921–929. IGI Global, 2009.
- [5] Ankur Chowdhary, Sandeep Pisharody, and Dijiang Huang. Sdn based scalable mtd solution in cloud network. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, pages 27–36. ACM, 2016.
- [6] Chun-Jen Chung, Pankaj Khatkar, Tianyi Xing, Jeongkeun Lee, and Dijiang Huang. Nice: Network intrusion detection and countermeasure selection in virtual network systems. *IEEE transactions on dependable and secure computing*, 10(4):198–211, 2013.
- [7] Cisco. Trends in Data Center Security. url = <https://blogs.cisco.com/security/trends-in-data-center-security-part-1-traffic-trends>, May 2014. Online; accessed 20 Dec 2018.
- [8] Eric Cole. Detect, contain, and control cyberthreats. *SANS Institute, June*, 2015.
- [9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms mit press. *Cambridge, MA*, page 819, 2001.
- [10] Reinhard Diestel. *Graduate texts in mathematics*. Springer-Verlag New York, Incorporated, 2000.
- [11] Joseph C Dunn. Well-separated clusters and optimal fuzzy partitions. *Journal of cybernetics*, 4(1):95–104, 1974.
- [12] Karel Durkota, Viliam Lisý, Branislav Bosanský, and Christopher Kiekintveld. Optimal network security hardening using attack graph games. In *IJCAI*, pages 526–532, 2015.
- [13] Massimo Ferrari. Release: Vmware nsx 6.1. 2014.
- [14] Hazem Hamed and Ehab Al-Shaer. Taxonomy of conflicts in network security policies. *IEEE Communications Magazine*, 44(3):134–141, 2006.
- [15] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.
- [16] Gerard J Holzmann, Doron A Peled, and Mihalis Yannakakis. On nested depth first search. *The Spin Verification System*, 32:81–89, 1996.
- [17] John Homer, Xinming Ou, and David Schmidt. A sound and practical approach to quantifying security risk in enterprise networks. *Kansas State University Technical Report*, pages 1–15, 2009.
- [18] Jin B Hong and Dong Seong Kim. Performance analysis of scalable attack representation models. In *IFIP International Information Security Conference*, pages 330–343. Springer, 2013.
- [19] Jin B Hong, Dong Seong Kim, Chun-Jen Chung, and Dijiang Huang. A survey on the usability and practical applications of graphical security models. *Computer Science Review*, 26:1–16, 2017.
- [20] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. Flow-guard: building robust firewalls for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 97–102. ACM, 2014.
- [21] Dijiang Huang, Ankur Chowdhary, and Sandeep Pisharody. *Software-Defined Networking and Security: From Theory to Practice*. CRC Press, 2018.
- [22] Pierr Johnson. With The Public Clouds Of Amazon, Microsoft And Google, Big Data Is The Proverbial Big Deal. url=<https://www.forbes.com/sites/johnsonpierr/2017/06/15/with-the-public-clouds-of-amazon-microsoft-and-google-big-data-is-the-proverbial-big-deal/409452d02ac3>, 2017. Online; accessed 3 Mar 2019.
- [23] Kerem Kaynar and Fikret Sivrikaya. Distributed attack graph generation. *IEEE Transactions on Dependable and Secure Computing*, 13(5):519–532, 2016.
- [24] Peter Mell and Richard Harang. Minimizing attack graph data structures.
- [25] Oussama Mjihil, Dijiang Huang, and Abdelkrim Haqiq. Improving attack graph scalability for the cloud through sdn-based decomposition and parallel processing. In *International Symposium on Ubiquitous Networking*, pages 193–205. Springer, 2017.
- [26] Nilesh Mojidra. Stateful vs. Stateless Firewalls. url=<https://www.cybrary.it/0p3n/stateful-vs-stateless-firewalls/>, 2016. Online; accessed 20 Sep 2018.
- [27] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. Mulval: A logic-based network security analyzer. In *USENIX Security Symposium*, pages 8–8. Baltimore, MD, 2005.
- [28] Justin Gregory V Pena and William Emmanuel Yu. Development of a distributed firewall using software defined networking technology. In *Information Science and Technology (ICIST), 2014 4th IEEE International Conference on*, pages 449–452. IEEE, 2014.
- [29] Sandeep Pisharody, Janakarajan Natarajan, Ankur Chowdhary, Abdullah Alshalan, and Dijiang Huang. Brew: A security policy analysis framework for distributed sdn-based cloud environments. *IEEE Transactions on Dependable and Secure Computing*, 2017.

- [30] Dhaval Satasiya, Rupal Raviya, and Hires Kumar. Enhanced sdn security using firewall in a distributed scenario. In *Advanced Communication Control and Computing Technologies (ICACCCT), 2016 International Conference on*, pages 588–592. IEEE, 2016.
- [31] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.
- [32] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. Automated generation and analysis of attack graphs. In *null*, page 273. IEEE, 2002.
- [33] Horst D Simon. Partitioning of unstructured problems for parallel processing. *Computing systems in engineering*, 2(2):135–148, 1991.
- [34] Su Zhang, Xinming Ou, and John Homer. Effective network vulnerability assessment through model abstraction. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 17–34. Springer, 2011.

Container-IMA: A privacy-preserving Integrity Measurement Architecture for Containers

Wu Luo, Qingni Shen[†], Yutang Xia, Zhonghai Wu[†]
Peking University, China
{lwyluo, qingnishen, ytxia, wuzh}@pku.edu.cn
[†]Corresponding author

Abstract

Container-based virtualization has been widely utilized and brought unprecedented influence on traditional IT architecture. How to build trust for containers has become an important security issue as well. Despite the fact that substantial efforts have been made to solve this issue, there are still some challenges to be handled, i.e. how to prevent from exposing information of the underlying host and other users' containers to a remote *verifier*, how to measure the integrity status of a designated container along with its reliant services in the underlying host and generate a hardware-based integrity evidence. None of the current solutions can counter these challenges and guarantee efficiency simultaneously.

In this paper, we present Container-IMA, a novel solution to cope with these challenges. We firstly analyze the essential evidence to validate the integrity of a designated container. Afterwards we make a division of the traditional Measurement Log (ML), which ensures privacy and decreases the latency of attestation. A container-based Platform Configuration Register (cPCR) mechanism is introduced to protect each ML partition with a hardware-based Root of Trust. The attestation mechanism is proposed as well. We implement a prototype based on Docker. The experiment results demonstrate the effectiveness and efficiency of our solution.

1 Introduction

Container-based virtualization technologies, e.g. Docker [18], LXC [25] and rkt [39], have become more and more prevalent, as they offer a light-weight virtualization approach to run multiple environments using the host kernel [20, 38, 42, 53]. However, since all containers share the same operating system (OS) kernel, the rapidly developing container technologies have introduced many security issues [11, 43], such as insecure production system configuration, vulnerabilities inside the images, and vulnerabilities directly linked to Docker, etc. Building trust for containers is a promising countermeasure to enhance security, as it can notify a remote verifier whether a container has genuinely enforced proclaimed

security-enhancement components and other unnecessary or adverse components have not loaded.

The major existing mechanisms [9, 15, 22, 26, 50] to build trust are based on Trusted Computing technology [37]. Trusted Computing technology provides a hardware-based solution to validate the integrity of physical platforms. The *Chain of Trust (CoT)* is built from the *Root of Trust (RoT)*, which contains a chip called Trusted Platform Module (TPM) [5], embedded in the target platform (*prover*) and is trusted by default. When the *prover* powers on, all components in the boot time will be extended to *CoT*, including BIOS, GRUB and OS kernel. With the help of Integrity Measurement Architecture (IMA) [46], the *CoT* finally reaches the application layer and measures all software components loaded in *prover*. Any measured component is not only recorded into the Measurement Log (ML), but also aggregated into Platform Configuration Registers (PCRs) inside a TPM. Furthermore, the remote user (*verifier*) can perform Remote Attestation to collect ML and validate it against PCRs. If it matches, the *verifier* compares each entry with his expectation to determine the *prover*'s integrity.

Recently, researchers have made substantial efforts to build trust for containers. The mainstream mechanisms among them [7, 9, 15, 22, 26, 50] are based on the following projects or technologies: vTPM [10], IMA, rkt and SGX [6]. Trusted Computing is the bedrock for the first three projects, while SGX is designed to provide isolated areas (i.e. enclave) for applications. However, none of them is sufficient to deal with the challenges that container-based virtualization faces.

Firstly, it is very important to ensure the privacy of underlying host and other users' containers. One overarching theme of building trust for containers is to utilize IMA, since it measures the integrity of a designated platform. For example, both Tao et al. [50] and Benedictis et al. [15] measure containers' integrity through adding an additional item of IMA to indicate which container the process belongs to. Since IMA measures all components into a particular PCR and records them into a single ML, each *verifier* has to collect the entire ML when validating the integrity status of containers. Hence, every *veri-*

stores this *measure* into ML and extends it into PCR10 via *PCR_Extend*. ML is a supplement to PCR, since the size of PCR is fixed and it is impossible to recover the list of stored values backwards from the current content of a PCR. ML records the detailed list of the measurement values and necessary metadata for the software components, representing for the integrity status of the platform.

IMA has defined *Remote Attestation* to enable a *verifier* to validate the integrity of *prover*. When receiving an attestation request, the Attestation Agent in *prover* collects integrity evidence, including PCR values, the signature signed by TPM and the ML. This signature is well defined in TCG specifications [5], and the specifications ensure that the private key (i.e. Attestation Identity Key, AIK) can only be used inside a specific TPM attached to a specific platform. The PCR values are included in the signature. Hence, a valid signature represents for the identity of *prover* and the trustworthiness of the transferred PCR values. The genuineness of ML is further determined by simulating *PCR_Extend* and matching the simulated result with trusted value of PCR10. If the validation result is positive, the *verifier* searches his expected values and compares them with the trusted ML. The *verifier*'s expectations are stored into a Reference Manifest Database, which is established through collecting information from the original source: the software and hardware manufacturers.

However, transmitting the entire ML to *verifier* violates privacy in a container setting. A *verifier* can be the owner of a designated container. He should not obtain the unnecessary information of other containers and the underlying host. It is significant to ensure privacy during Remote Attestation.

3 Motivation and Architecture Overview

3.1 Scenario and Threat Model

Figure 2 shows a use case in a container setting. We define *prover* as the platform hosting quite a few containers for multiple users. For example, user A and user B each has two containers running in *prover*. *Verifier* is defined as a remote user, concerning about the integrity status of his containers running in *prover*. User A wants to know whether his container (e.g. mysql) is running as he expects, such as booting from a correct image and loading benign softwares.

Prover in Figure 2 shows a simplified container architecture. The container management services, e.g. Docker Daemon, are responsible for spawning and managing containers. All other components running in the underlying host are classified into *host applications*. When a client requests to run a container, the container management services locate and load the container image with specified configurations, such as configuring network and setting isolation environment. Linux namespace mechanism [33] is utilized for containers to establish isolation environments. This mechanism divides the system resources into many different in-

stances in several aspects, including mount, hostname, IPC, PID and network [11]. Each container usually has a unique namespace, such that one container is isolated from others¹.

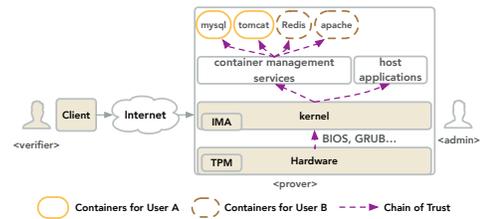


Figure 2: Use Case in a Container Setting

We assume that *prover* possesses trusted hardware, e.g. TPM, to support Trusted Computing, and trusted boot [35] ensures the integrity of OS kernel. We do not consider physical attacks and the attacker cannot get TPM's ownership. As for the capability of adversary, we focus on the *Local Adversary* and *Remote Adversary* [4]. A local adversary is sufficiently near *prover* to be capable of eavesdropping on, and interfering with, the *prover*'s communication. A remote adversary can remotely infect *prover* with malware, e.g. modifying files or integrity evidence a container relies on, affecting the attestation mechanism or even impersonating as container management services. Finally, we assume that only an authorized *verifier* can perform Remote Attestation, as a cluster should have an effective user management module.

Similar to other existing mechanisms based on Trusted Computing [9, 15, 22, 50], the run time memory attacks are not considered. This limitation can be mitigated by leveraging existing mechanisms, such as address space layout randomization [51] and control flow attestation [3], etc.

3.2 Notations

Measurement Event (ME): represents for an event which triggers a *measure*. MEs are defined by IMA through a policy. The default IMA policy measures all system sensitive files, including executables, mmaped libraries, and files opened for read by root [1]. Our work adheres to this definition.

Measurement Log (ML): each entry in ML represents for the measurement result and related metadata. Equation 1 shows details, where $measure(boot)$ refers to measure components in the *prover*'s boot time by trusted boot [35], and $measure(ME)$ is the result of measuring ME by IMA. At least the path and hash value of this ME are included into the measurement results. To protect each entry's integrity, IMA calculates the digest value of this entry, and appends this digest value ($node_{hash}$) into $measure(ME)$.

$$\begin{aligned}
 ML &= measure(boot) \cup \{measure(ME)\} \\
 &= measure(boot) \cup \{ \langle e.path, e.hash, e.node_{hash} \rangle \mid e \in ME \}
 \end{aligned}
 \tag{1}$$

¹Sharing namespaces among containers is discussed in Section 6.4.

3.3 Analysis

Traditionally, trusted boot [35] enables to measure components during a platform’s boot time. With a *measure-before-loading* mechanism, a *CoT* is constructed and trust is transferred from *RoT* to the OS kernel. The measurement mechanism is changed when trust reaches to the application layer. All software components in the application layer are measured by IMA and written into ML according to the time of loading. Consequently, the sequence of loaded software components is conceived to the *CoT* in the application layer. The entire *CoT* can be decomposed into the following partitions:

1. *Integrity of prover’s Boot Time* (I_{boot}^{Pro}): starts from powering on *prover* and ends with successfully loading OS kernel. It includes BIOS, GRUB and OS kernel.
2. *Integrity of Containers’ Dependencies* (I_{dep}^{Con}): refers to the container management services and files or libraries they require.
3. *Integrity of a Container’s Boot Time* (I_{boot}^{Con}): refers to the images and boot configurations when the container management services launch a container.
4. *Integrity of a Container’s Applications* (I_{app}^{Con}): starts from container management services successfully launching a container and ends with shutting down it. It includes all components running inside a container.
5. *Integrity of Host Applications* (I_{app}^{Host}): starts from successfully launching OS kernel and ends with shutting down *prover*. The container management services and all containers are not included in this partition.

Figure 2 shows the containers’ *CoT*. From a container’s perspective, its direct dependency is the container management service. Other host applications are isolated from containers through namespaces. Hence a container’s *CoT* only includes I_{boot}^{Pro} , I_{dep}^{Con} , this container’s I_{boot}^{Con} and I_{app}^{Con} . When a *verifier* requests to attest a container, the *prover* can aggregate these partitions to convince its trustworthiness. And components which do not belong to the *CoT* of a designated container should not be revealed to a *verifier*.

Hence the privacy requirement considered in this paper is: *the integrity evidence, that the prover sends back to verifier, should not expose the information of I_{app}^{Host} , other containers’ I_{app}^{Con} and I_{boot}^{Con}* . Note that a strong privacy requirement is that a *verifier* cannot distinguish whether his container is the only container running on *prover*. We will discuss it in Section 6.1.

To meet this privacy requirement, we need to divide the traditional ML into the above partitions. One of the challenges is how to make a division of ML automatically. Kernel should be empowered to know which partition a ME belongs to. Through adding additional items in the IMA kernel module, existing mechanisms [15, 50] can differentiate I_{app}^{Host} and I_{app}^{Con} .

But they cannot handle I_{dep}^{Con} and I_{boot}^{Con} . It is also insufficient for mechanism based on *rkt* [22], since it can only record I_{boot}^{Con} .

Another challenge is how to ensure the integrity of ML’s each partition with a hardware-based *RoT*. The traditional IMA constructs a single ML and binds this ML to PCR10. In a container setting, launching N containers introduces N more I_{app}^{Con} s. It is not feasible to adopt the traditional IMA’s strategy, i.e. binding each ML partition to a unique PCR. So schemes [15, 50] adhering to the traditional IMA cannot protect I_{app}^{Host} and I_{app}^{Con} with hardware-based *RoT* separately.

3.4 Architecture Overview

Figure 3 depicts our architecture. Compared with the traditional IMA, our architecture introduces three additional components, including the Split Hook, the Namespace Parser and the container-PCR (cPCR) Module. Since the container-based virtualization utilizes the namespace mechanism to isolate system resources from others, each container should possess a unique namespace. Split Hook inspects the syscall to create a new namespace, and notifies kernel a event to split ML. The Namespace Parser extracts the namespace number of the current process to identify which partition the current ME belongs to. cPCR module is liable for protecting each ML partition with TPM by maintaining cPCRs. cPCR is a data structure in kernel simulating for PCR.

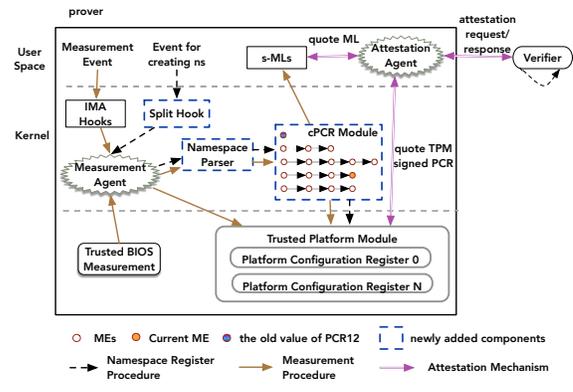


Figure 3: Overview Architecture of Container-IMA

Our architecture contains two parts: *measurement mechanism* (Section 4) and *attestation mechanism* (Section 5).

The *measurement mechanism* accomplishes its task with the help of these three newly added components. On one hand, Namespace Register Procedure contributes to splitting ML for each namespace. It starts from when the Split Hook is triggered and ends with creating and maintaining corresponding partition of ML and cPCR. On the other hand, Measurement Procedure is responsible for measuring MEs, storing them into corresponding partition of ML and protecting this partition with TPM via the assist of cPCR.

The *attestation mechanism* allows a remote *verifier* to request an integrity evidence of *prover*, for the purpose of

checking whether a designated container running as he expects. When receiving such a request, the modified Attestation Agent in *prover* locates the related ML partitions to ensure privacy, and retrieves the signed aggregated PCR from TPM. Since the *measurement mechanism* ensures that all partitions are protected by physical PCRs, the trustworthiness of the received ML partitions can be determined.

4 Measurement Mechanism

This section elaborates the measurement mechanism. Note that components in the *prover*'s boot time, i.e. I_{boot}^{Pro} , are measured by *trusted boot* and protected by PCR0-7. They are inherently separated from other ML partitions. Hence we do not describe them in the remaining of this section.

4.1 Basic Namespace Register Procedure

Distinguishing different containers from the kernel's perspective is the basic problem we need to solve. Since a container should have a distinct namespace, kernel can distinguish containers through parsing the namespace number of the running process when performing the integrity measurement. And hence the I_{app}^{Host} and I_{app}^{Con} s can be separated.

Namespace Register Procedure is designed to empower kernel to make a division of ML. Kernel in our architecture has to maintain multiple double-linked lists to represent for the separated measurement logs (s-MLs) from ML. Equation 2 defines all s-MLs, where ME_{ns} refers to a ME whose namespace is *ns*, and *n* is the number of ML's partitions.

$$s-MLs = \{ \langle value, ns \rangle \}_n = \{ \langle \{ measure(ME_{ns}) \}, ns \rangle \}_n \quad (2)$$

In order to protect the integrity of these s-MLs, we simulated a set of PCRs, which we call as container-based PCRs (cPCRs). Each cPCR has its value, a unique *secret* and the corresponding namespace number, i.e. *ns*. *Secret* is a random value generated by TPM and is utilized to hidden other containers' cPCRs for a given *verifier* (See Section 4.2.1).

$$cPCR-list = \{ cPCR \}_n = \{ \langle value, ns, secret \rangle \}_n \quad (3)$$

The workflow of Namespace Register Procedure is shown in Figure 3. When a container is launched, an event for creating namespace is generated. Split Hook captures the corresponding syscall, and the Measurement Agent knows this event and notifies Namespace Parser to parse the number of this new namespace, e.g. *ns*. Once receiving *ns*, cPCR Module will firstly request TPM to generate a random value as this cPCR's *secret*. Next, cPCR Module creates a new cPCR and a new s-ML, as the Equation 4 shows, where *AllZero* refers to the initialized value of a cPCR, i.e. all bytes are set to zero.

$$\begin{aligned} cPCR-list &:= cPCR-list \cup \{ AllZero, ns, secret \} \\ s-MLs &:= s-MLs \cup \{ \langle \{ \}, ns \rangle \} \end{aligned} \quad (4)$$

Namespace Register Procedure creates separated s-MLs for containers (namespaces), yet I_{dep}^{Con} and I_{boot}^{Con} are not divided. Section 4.3 presents the solutions.

4.2 Basic Measurement Procedure

The measurement procedure is responsible for measuring and recording MEs. Generating a ME triggers the IMA hooks. The Measurement Agent is notified and measures this ME. The measurement results ($measure(ME)$) and the namespace number of the current process (ns_{num}) parsed by Namespace Parser will be passed to the cPCR module. cPCR module afterwards locates the target cPCR and s-ML (Equation 5).

$$\begin{aligned} target-cPCR &= \{ cPCR \mid cPCR.ns == ns_{num} \cap cPCR \in cPCR-list \} \\ target-ml &= \{ s-ML \mid s-ML.ns == ns_{num} \cap s-ML \in s-MLs \} \end{aligned} \quad (5)$$

Secondly, the cPCR module extends the current ME into the target cPCR and appends it into the target s-ML (Equation 6). $ME.node_{hash}$ refers to the $node_{hash}$ in $measure(ME)$.

$$\begin{aligned} target-cPCR.value &:= HASH(target-cPCR.value, ME.node_{hash}) \\ target-ml.value &:= target-ml.value \cup measure(ME) \end{aligned} \quad (6)$$

The extend operation is the same as PCR_Extend provided in TPM specifications. The above steps ensure that the integrity of each s-ML is protected by the related cPCR, that is, we can check the genuineness of each s-ML by matching the related cPCR with the result of simulating Equation 6 with s-ML.

4.2.1 Bind cPCRs to Hardware-based RoT

After extending the target cPCR and appending the target s-ML, the cPCR module binds cPCRs into a physical PCR through the following steps.

1. records the history value of a specific PCR (*historyPCR*) which is not used in the current system. In our prototype based on TPM 1.2, we choose PCR12. We provide a GRUB parameter for user to change this PCR index.
2. records the digest of all cPCRs. *tempPCR* is initialized as $cPCR_0.value$ xored with $cPCR_0.secret$. For all other cPCRs in cPCR-list, Equation 7 is performed to extend them into *tempPCR*. Finally, *tempPCR* will save the digest value of all cPCRs' values.

$$\begin{aligned} tempValue &:= cPCR_i.value \text{ xor } cPCR_i.secret \\ tempPCR &:= HASH(tempPCR \parallel tempValue) \end{aligned} \quad (7)$$

3. extends the physical PCR12 with the final *tempPCR*.

$$PCR12 := PCR_Extend(PCR12, tempPCR) \quad (8)$$

The *secret* field is essential to hidden other containers' cPCRs to a particular *verifier*. Given a user owning container

whose s-ML is extended into $cPCR_u$, a remote attestation means transferring a *nonce* to *prover* which is utilized to defend against replay attacks. The *prover* afterwards sends back at least the following values to the remote user (more details shown in Section 5): *historyPCR*, *sendcPCRs*, $\text{Sign}(\text{AIK}, \text{nonce} \parallel \text{related PCR})$, where *sendcPCRs* are values extended into PCR12 (Equation 9).

$$\text{sendcPCRs} = \{cPCR_i.\text{value} \text{ xor } cPCR_i.\text{secret} \mid cPCR_i \in cPCR\text{-list}\} \quad (9)$$

In this case, the *verifier* can validate TPM's signature and afterwards get the genuine *nonce* and PCR12. Next, *verifier* validates *sendcPCRs*' genuineness via recalculating a simulated value, i.e. performing Equation 7 and extending *historyPCR* with the returned digest value. If the calculated result equals the decrypted PCR12, the trustworthiness of *sendcPCRs* is determined. Finally, if the *verifier* has obtained $cPCR_u.\text{secret}$, he can restore the $cPCR_u.\text{value}$ by *xoring* $cPCR_u.\text{secret}$ with corresponding entry in *sendcPCRs*. In our current prototype, when a user successfully launches a container, this container's *secret* is fed back to user, and from that moment the *prover*'s kernel will not expose this *secret* to the user space. We will further discuss the relevant privacy issues in Section 6.1.

In summary, a one-by-one protection chain is established, i.e. TPM protects PCR12 which further protects cPCRs, and cPCRs protect s-MLs.

4.3 Extensions

Currently we haven't record the *Integrity of Containers' dependencies* (I_{dep}^{Con}) and *Integrity of Containers' Boot Time* (I_{boot}^{Con}) separately. The former does not create a new namespace, so the above mechanism fails to identify them. For the latter, a container's images and boot configurations are upper concepts in the user-space. Thus it is hard to let kernel parse and record them. This section gives our solutions.

4.3.1 Integrity of Containers' Dependencies (I_{dep}^{Con})

To reuse the Namespace Register Procedure, we add a new special program named *bootstrap program*, whose task is to create a new namespace for an application. Therefore, when launching these dependencies with *bootstrap program*, the Namespace Register Procedure will be triggered. Note that the container's dependencies are extended into $cPCR_0$, as they run ahead of any other containers. Besides, all containers rely on these dependencies, so we choose $cPCR_0.\text{secret}$ to be a well-known value, e.g. *AllZero*.

In our prototype, we choose *unshare* to be the *bootstrap program*, because *Docker-ce* [16] utilizes the *unshare* syscall to allocate a new namespace for a container. We have to enable the *verifier* to check: ① the integrity of *bootstrap program*, and ② whether his containers are indeed launched with services generated by *bootstrap program*.

The former issue can be addressed by adding the process which creates new namespace (hereafter we call this process as *createProcess*) into the related s-ML. For instance, the *createProcess* of containers' dependencies refers to *bootstrap program*. Since s-MLs are finally protected by PCR, the integrity of *bootstrap program* is protected as well. Thus, Equation 4 should be modified to:

$$\begin{aligned} cPCR\text{-list} &:= cPCR\text{-list} \cup \\ &\quad \{OP_{\text{extend}}(\text{AllZero}, \text{measure}(\text{createProcess})), ns, \text{secret}\} \\ s\text{-MLs} &:= s\text{-MLs} \cup \{< \{ \text{measure}(\text{createProcess}) \}, ns >\} \end{aligned} \quad (10)$$

where $OP_{\text{extend}}(\text{AllZero}, \text{measure}(\text{createProcess}))$ indicates that this cPCR is extended with the measurement results of *createProcess* with an initialized value *AllZero*.

A simple way to deal with the latter issue is to record *pid* of *createProcess* and its ancestor processes' *pids*. We define these *pids* as *createProcess.pids*. If a container is launched with the genuine dependencies, the *pid* of dependencies' *createProcess* can be found in *pids* of this container. And hence a *verifier* can determine whether the s-ML for containers' dependencies is genuine. The measurement of *createProcess*, i.e. P , is changed to:

$$\text{measure}(P) = < P.\text{path}, P.\text{hash}, P.\text{pids}, \text{node}_{\text{hash}} > \quad (11)$$

Figure 4 gives an example. File *ascii_runtime_measurements* records I_{app}^{Host} , which is irrelevant to containers. File 4026532222 and 4026532238 refer to s-MLs for containers' dependencies (I_{dep}^{Con}) and the launched container (I_{app}^{Con}), respectively. Each entry of s-ML comprehends four elements: *index* (*pcr index* or *namespace number*), *template-hash* (i.e. $\text{node}_{\text{hash}}$), *template-name*, *file-hash* (the digest value for measured file) and *file-path*. We omit the *template-hash* in Figure 4. For instance, the second entry in file 4026532222 means that the measured file is */usr/bin/dockerd* in namespace 4026532222.

In Figure 4, the first entries in file 4026532222 and 4026532238 refer to the corresponding *createProcess*. It means that containers' dependencies are launched by */usr/bin/unshare* whose *pid* is 1990. The container is launched by */usr/local/sbin/runc* whose *pid* is 2358 and it is a child of process 1990. Therefore, file 4026532222 indeed represents for this container's dependencies.

4.3.2 Integrity of All Containers' Boot Time (I_{boot}^{Con})

The integrity of containers' boot time includes the image, configuration and parameters to bootstrap a container. These information cannot be obtained by kernel as they are concepts in application layer. Considering that the containers' dependencies have been genuinely recorded and protected by *RoT*, we can let them collect this information.

In our prototype, we modify *runc* [2] to do the measurement and extend $\text{node}_{\text{hash}}$ into another physical PCR, e.g.

```

==> 4026532222 <==
4026532222 ... ima-ng sha1:38919a201521117fb8bf907ab5d41bb31eb29a39 1990->1980->1907->1447->1249->1071->1->0_4026532222:/usr/bin/unshare
4026532222 ... ima-ng sha1:a348d30d0774ed4d84945da0f10d6319da4cd9ac 4026532222:/usr/bin/dockerd
4026532222 ... ima-ng sha1:80a5ea753fe069ecc8f5bdf857d4af9d8aef39b 4026532222:/usr/bin/docker-containerd
4026532222 ... ima-ng sha1:126ee56dd59433f8a488cf873d0fefea2c3f91a 4026532222:/var/lib/docker/tmp/docker-default618280113
4026532222 ... ima-ng sha1:a00d3061edf0ff271e7e7395d2d9676736f95477 4026532222:/lib/modules/3.13.11-ckt39/kernel/ubuntu/aufs/aufs.ko

==> 4026532238 <==
4026532238 ... ima-ng sha1:d5442f82ce6ee98f17b4a21e49fdd326e4d1c6ae 2358->2354->2346->2001->1990->1980->1907->1447->1249->1071->1->0_4026532238:/usr/local/sbin/runc
4026532238 ... ima-ng sha1:611a59c515074dbb376713fd19040c10a0c8e5e2 4026532238:/bin/bash
4026532238 ... ima-ng sha1:b43aec6bd95cab18f2bcedf1dcf01462f7e088d 4026532238:/lib/x86_64-linux-gnu/ld-2.23.so
4026532238 ... ima-ng sha1:eaee87c3d507be4634db12ab562c9f1cb243e764 4026532238:/etc/ld.so.cache
4026532238 ... ima-ng sha1:2bd9389f52439de7a42efcb8a3c3f8d4c881e8b2 4026532238:/lib/x86_64-linux-gnu/libtinfo.so.5.9

==> asci_runtime_measurements <==
10 ... ima-ng sha1:27d6a1e1108a90c19003d919f325c7bd0f4acb10 boot_aggregate
10 ... ima-ng sha1:a5e65f1ca3a779cea954a015bde7ec5daa3f7612 4026531840:/init
10 ... ima-ng sha1:dc3e621c72cde19593c42a7703e143fd3dad5320 4026531840:/bin/sh
10 ... ima-ng sha1:67c253d8ea7089253719cad7f952fb4c22240f27 4026531840:/lib64/ld-linux-x86-64.so.2
10 ... ima-ng sha1:fac553d7706114a2ed0ef587aa748f59e19f381a 4026531840:/etc/ld.so.cache

==> docker-boot <==
"... [4026532238] sha256:7aa3602ab41ea3384904197455e66f6435cb0261bd62a06bd1d8e76cb8960c42 ... /var/lib/docker/containers/.../config.v2.json" c952b062e8be3cbc407242cb2ebc27c8111b489

```

Figure 4: An example for the first 5 entries of each ML partition after we bootstrapped dockerd (Docker Daemon) through unshare command and afterwards set up a new container via docker run -ti ubuntu:16.04.

PCR11. Runc is responsible for spawning and running containers. File `docker-boot` in Figure 4 gives an example for s-ML related to container’s boot time. Each entry in this file is composed of six elements: `id` (the container’s id), `ns` (the namespace number), `HASH(image)` (the digest value of its image), `HASH(config)` (the digest value of its configuration), `PATH(config)` (the absolute path of its configuration file), and `node_hash`. In Figure 4, we replace the `id` and `HASH(config)` with symbol `'...'` to have a better display.

One privacy issue is that we cannot transfer the entire `docker-boot` to `verifier` because it records all container’s boot information. Instead, we only transfer the `node_hash` for other containers in our prototype.

5 Attestation Mechanism

This section shows how to enable a remote `verifier` attesting the integrity of a designated container and its dependencies. Our attestation mechanism derives from the traditional Remote Attestation. Note that we assume that there is an effective user management system in a cluster, e.g. kubernetes [32], to prevent unauthorized `verifier` from launching attestation mechanism. Hence we do not consider identifying `verifiers`.

Message Transferring When verifying the integrity of a container running in `prover`, a remote `verifier` should send a request: `<nonce, containerID>`, where `nonce` is a random number generated by `verifier`, and `containerID` refers to the target container. When receiving this request, the Attestation Agent in the `prover` collects:

1. the related PCR values and the signature of TPM via performing `TPM_Quote`, i.e. `Sign{AIK, nonce || PCRs}`, where PCRs refer to PCR0-7, PCR11 and PCR12.
2. `sendcPCRs` (See Equation 9) and the history value of PCR12 (i.e. `historyPCR`).

3. s-ML for `prover`’s boot time and the containers’ dependencies, i.e. I_{boot}^{Pro} and I_{dep}^{Con} .
4. s-ML for containers’ boot time components (I_{boot}^{Con}), which is described in Section 4.3.2. For the target container, the entry in this s-ML includes `id`, `ns`, `HASH(image)`, `HASH(config)`, `PATH(config)` and `node_hash`. For other containers, it only contains `node_hash`.
5. s-ML for the target container’s applications, i.e. I_{app}^{Con} .

Workflow of Verifier Firstly, `verifier` validates the identity of TPM. Each TPM is written a unique endorsement certification by manufacture. The endorsement key (EK) can be used to identify a TPM. AIK is constructed from EK and generates signatures. Matching the received PCR values with a correct TPM signature confirms the correctness of PCR values. The trustworthiness of `nonce` can be determined as well, and `verifier` knows the freshness of this response.

Secondly, based on PCR values which are determined as trusted, the integrity of s-MLs for bootstrapping `prover` (I_{boot}^{Pro}) and the target container (I_{boot}^{Con}) can be confirmed. These s-MLs are extended into physical PCRs, i.e. PCR0-7 for I_{boot}^{Pro} and PCR11 for I_{boot}^{Con} . The genuineness of these s-MLs can be checked by simulating the PCR_Extend operation and comparing the simulation result with the trusted PCRs.

Thirdly, the trusted PCR12 can be used to verify the genuineness of `sendcPCRs`. The `verifier` calculates `tempPCR` through Equation 7 with the received `sendcPCRs`. Afterwards, a simulated PCR value (`sPCR`) can be calculated by extending the `historyPCR` with `tempPCR`. If the final `sPCR` equals to PCR12, `verifier` gets a trusted `sendcPCRs`, otherwise some attacks have happened. With the genuine `sendcPCRs` and the `secret`, `verifier` can get his concerned container’s cPCR (e.g. `cPCRu.value`) and `cPCR0.value`.

Finally, the trustworthiness of I_{app}^{Con} and I_{dep}^{Con} can be determined by simulating PCR_Extend and matching the results with `cPCRu.value` and `cPCR0.value`, respectively. The `veri-`

verifier should also confirm whether I_{dep}^{Con} is indeed his container's dependency, as we mentioned in Section 4.3.1.

If all aforementioned validation procedures are positive, all related s-MLs are trusted. Then the verifier can validate s-MLs against his expectations. These expectations are defined as the same as the traditional Trusted Computing, which are collected from the software and hardware manufacturers.

6 Implementation and Evaluation

This section presents our implementation details, analyzes the privacy and security achievements, and illustrates the experimental results. A prototype is implemented to evaluate our solution. The prover runs Ubuntu 14.04 with our modified kernel based on kernel 3.13.11 to support Measurement Mechanism, along with 16GB memory, 4 processor cores. A TPM1.2 chip is equipped in prover. Docker-ce [16] is installed in prover with version 17.06.1-ce. The verifier is Ubuntu 14.04 with the default kernel and 16GB memory.

Measurement Mechanism requires to hook action which creates a new namespace. We hook the unshare syscall to perform Namespace Register Procedure. Meanwhile, if a ME does not have a namespace number or its namespace number can not be found in s-MLs, this event belongs to host applications in prover. In this case, the measurement result is extended into PCR10 as traditional IMA does.

The implementation of Attestation Mechanism is based on the OpenAttestation (OAT) [27] project, which is an open-source project to support PCR-based report schema. We modified the message transferring protocol and the validation procedure of OAT to support our Attestation Mechanism.

6.1 Privacy

The privacy achievements comprise two aspects. Firstly, the host applications' information (i.e. I_{app}^{Host}) are not necessarily sent back to verifier. In our solution, the s-ML related to host applications are recorded into a separate file. This s-ML is extended into PCR10 as traditional IMA. Both this s-ML and PCR10 are not required during the remote attestation for containers. Secondly, other container's boot time information (i.e. I_{boot}^{Con}) and applications (i.e. I_{app}^{Con}) are not exposed. Although all containers' boot time are recorded into the same s-ML, the information transferred to verifier only contains the corresponding $node_{hash}$. Since the container id is a random value, the verifier cannot obtain meaningful information from $node_{hash}$. We can even add more unguessable value into I_{boot}^{Con} to confuse the malicious verifier. Meanwhile, other containers' I_{app}^{Con} s are not transferred for remote attestation.

A privacy issue is related to $cPCR.value$. A standardized container has a well-known boot hash. Knowing $cPCR.value$ may allow the attacker to deduce which container runs. In our solution, other containers' $cPCR.values$ are xored with corresponding $cPCR.secrets$. The results of xor operation

(i.e. $sendcPCRs$) are sent to verifier. Since other containers' $cPCR.secrets$ are unknown to a verifier, he cannot restore their $cPCR.values$. An issue is how to ensure the privacy of $cPCR.secret$. In our prototype, we feed back $secret$ to user once a container is successfully launched and afterwards ensure that $secret$ will not be exposed to the user space. This strategy requires a trusted communication between user and prover at this moment, which we think is reasonable in a container cluster. An alternative strategy is to let prover generate a key pair and bind the private part (sk) into TPM in the initialization procedure. All users obtain the public part (pk) and perform a remote attestation by encrypting $nonce$ with pk . Since only prover's TPM can get sk , $nonce$ cannot be leaked to attackers. Afterwards, prover can xor $nonce$ with the target $secret$. The xored result is utilized as the previous remote attestation's $nonce$ to get TPM's quote. In this case, verifier can obtain a genuine $nonce xor secret$ through attestation response. By xoring with the $nonce$ the verifier sent before, the $secret$ can be restored. A malicious verifier cannot obtain other users' secrets without nonces, so the privacy of secrets is ensured. We will implement this strategy in the future.

Finally, as we mentioned in Section 3.3, a much stronger privacy requirement is that any verifier cannot distinguish whether his container is the only container running on prover. Having this knowledge helps an attacker decide whether to launch a co-resident attack [21]. In Container-IMA, a malicious verifier can get the current number of containers running on prover by counting the size of $sendcPCRs$. We can use some obfuscated strategies to solve this problem. For instance, we can add a GRUB parameter to configure prover to run at most max containers. The unused cPCRs are filled with unmeaningful values. Since all xored cPCRs are sent to verifier, the size of $sendcPCRs$ is not a privacy sensitive data. Another similar problem is that the attacker can perform remote attestation several times to inspect $historyPCR$. If this value is changed, attacker knows that another container performs some operations. We can let prover update PCR12 when a host application's ME occurs. This operation confuses the verifier whether the change of $historyPCR$ is caused by host applications or containers' events. Considering loading additional host applications may be rare after a prover is deployed, we can let kernel do some timely unaffected measures and update PCR12. In summary, although our prototype does not enforce obfuscated strategies, it is feasible to solve these issues.

6.2 Security

The security achievements include two aspects: the integrity evidence ① covers a container's entire CoT , and ② is protected by hardware-based RoT . Container-IMA achieves ML partition and protection for each partition. The I_{boot}^{Pro} is measured by trusted boot and is recorded into PCR0-7. The I_{dep}^{Con} and I_{app}^{Con} are recorded into cPCRs, and are further protected by PCR12. The I_{boot}^{Con} is extended into PCR11. Hence the

evidence is well protected and its completeness is achieved. Note that the mechanism based on *rkt* [22] does not cover the entire *CoT*, as it only considers the container’s boot phase. Similarly, it is not enough to ensure container security by simply blacklisting of known vulnerable container images.

Compared with integrity mechanisms based on vTPM, our approach does not require an additional layer in the user space. The typical integrity mechanisms to leverage vTPM are [26] and [9]. The former presents two possible architectures to build trust for containers through vTPM. One architecture is to put the vTPM instances in the host OS kernel. The container manager in the user-space must be responsible for asking the kernel to create a new vTPM and assigning the device to a container. Another architecture is to put vTPM and vTPM manager in one of the privileged container, which is similar to Dom0 in Xen [8]. Both these two architectures require a user-space component to bind vTPMs into physical TPM. The latter mechanism [9] implements a vtpm proxy driver in the Linux kernel that enables to spawn a TPM device with a client TPM character device and a server side file descriptor. The client device is moved into the container by creating a character device, while the server side file descriptor is passed to the TPM emulator in the user space. All these mechanisms require additional trusted components in the user space. Once these components are affected by attacker, the integrity evidence is not protected. Our solution relies on cPCR module, but it is inside the OS kernel, and the integrity of kernel is measured by *trusted boot*. Finally, all these two mechanisms do not elaborate the measurement and attestation mechanisms for containers.

6.3 Efficiency

6.3.1 Performance of Containers

This section shows our evaluation on the performance of containers. In each case, we test two kernels: One is *Default* referring to the default kernel 3.13.11 with IMA enabled, the other is *Container-IMA* which implements our Measurement Mechanism based on the default kernel 3.13.11. IMA is enabled by adding *ima_tcb* in GRUB command line.

Influence on Basic Environment We run a Ubuntu 16.04 container in *prover* to evaluate the overhead of our modified kernel. The benchmark tool is *LMbench3* [36], a well-known tool for performance analysis. For each kernel, we run *LMbench3* for 20 times and record their average usages. Table 1 shows the experiment results. The percent overhead calculation against case *Default* is shown in case *Container-IMA*. As Table 1 shows, for most measurements, the evaluation results of case *Container-IMA* cost a little more than case *Default*. It is reasonable, since a container’s ME leads to extending cPCR and binding cPCRs into physical PCR, while the *Default* kernel just extends PCR10.

Table 1: Influence on Container

Type	Default	Container-IMA
Processor, Processes (microseconds) - smaller is better		
null call	0.199	0.199(0.00%)
null I/O	0.27	0.27(0.00%)
stat	2.311	2.328(0.74%)
signal install	0.2675	0.267(-0.19%)
signal handle	1.04	1.04(0.00%)
exec process	371.5	374.85(0.90%)
sh process	1510.4	1514.1(0.24%)
Local Communication bandwidths in MB/s - bigger is better		
Pipe	4518.25	4503.45(-0.33%)
AF Unix	13.8K	13.9K(0.72%)
File reread	9150.215	9148.97(-0.01%)
Bcopy(libc)	10.K5	10.5K(0.00%)
Bcopy(hand)	7566.39	7562.99(-0.04%)
Mem read	14K	14K(0.00%)
Mem write	11K	11K(0.00%)

Influence on Container Applications In this experiment, we choose MySQL to be the tested application. We run a MySQL container and utilize *Sysbench* [29] to create a test table with 2,000,000 rows of data. We increase the number of used threads from 5 to 100 with the step of 5. In each case we record the transactions per second (TPS), the read/write responses per second (RPS) and the average response time for more than 95% requests. The experiment results are shown in Figure 5(a), 5(b) and 5(c) respectively. When the number of threads increases, TPS, RPS and the response time tend to be increased. Compared with case *Default*, the influence of our *Container-IMA* kernel is negligible. The default IMA policy measures the binary programs, the dynamic link libraries, the kernel modules and files opened by *root*. Operating MySQL does not trigger much of these measurements. Note that most containers are developed as designated applications. These containers trigger little measurements once they become stable, such that the introduced overhead is negligible.

Influence on Spawning Containers Kubernetes provides *Docker Micro Benchmark* [30] to evaluate Docker operations. We utilize this tool to benchmark the overhead of spawning containers via *benchmark.sh -o*. This tool launches 100 *go routines* keep starting containers until the latency is larger than 50s. We gradually increase the Queries Per Second (QPS) and record the related Container Started Per Second (CPS). QPS represents for the limited query rate of all *go routines*. Figure 6(a) shows the results. For example, when QPS is set to 20, our solution creates 0.4448 containers per second, while Docker Daemon running in the default kernel creates 0.6185 containers per second. In case *Container-IMA*, creating a container means creating cPCR and s-ML, measuring the binary program, extending cPCR and binding cPCRs into physical PCR. It is reasonable that our solution introduces some overhead towards spawning containers.

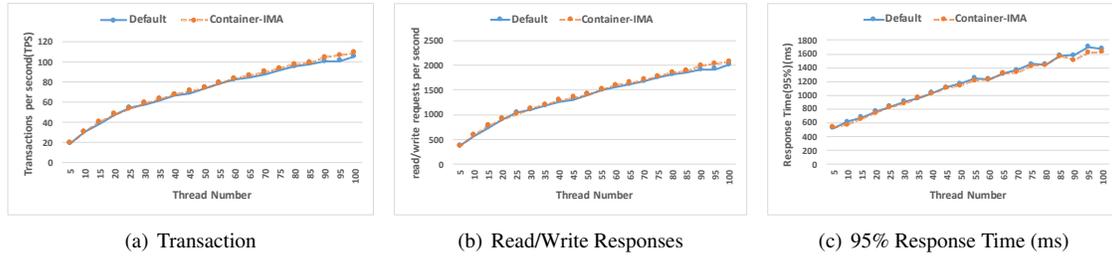


Figure 5: Experiments for MySQL Running in a Container

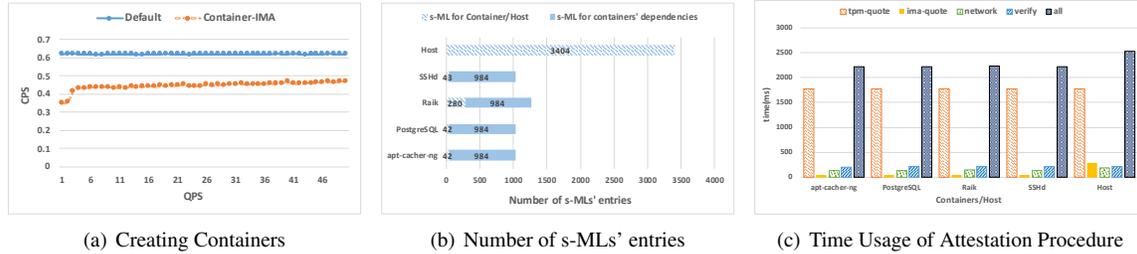


Figure 6: Experiments for Spawning Containers and Attestation Procedure

6.3.2 Performance of Attestation Procedure

Docker community gives some sample applications [17] to show how to run popular software using Docker. We choose a part of them to evaluate our solution, including *apt-cacher-ng*, *PostgreSQL*, *Riak* and *SSHd*. After starting up *prover* and launching above sample containers, we firstly record the number of MEs for each s-MLs. Figure 6(b) shows the details, where label *Host* refers to the s-ML for *host applications*, and other labels of the vertical axis refer to the s-MLs of sample containers. For example, container *Riak* possesses 280 MEs for its run time components and 984 MEs for its dependencies, while the *Host* has 3404 MEs for *host applications*.

We then test the cost of attestation. For each case, we record the time usage within 20 minutes and calculate their average value for five phases: *tpm-quote* (*prover* collects signature for PCR via TPM_Quote), *ima-quote* (*prover* collects other evidence except PCR's signatures), *network* (transferring attestation request and integrity evidence), *verify* (the *verifier* validates the trustworthiness of integrity evidence), and *all* (the whole attestation procedure). Figure 6(c) shows the experiment results. For example, with respect to *PostgreSQL*, the average time for these five phases are 1768.76ms, 52.72ms, 141.50ms, 214.78ms, 2217.33ms respectively.

The time usages for attesting containers are nearly in the same order. During the attestation procedure, the *prover* collects *sendcPCRs*, s-MLs for this container, containers' dependencies and the signature of TPM for PCRs. The size of *sendcPCRs* and the number of quoted PCRs are the same value for attesting containers. And the number of a container's s-MLs, which is shown in Figure 6(b), is also nearly the same order of magnitude. Consequently, attesting them takes nearly the same time. Attesting *Host* differs from containers. It does

not require to collect and validate cPCRs. However, since the number of MEs for *Host* is much higher than a container's, it takes more time to collect (phase *ima-quote*) and transfer (phase *network*) these MEs. Hence attesting *Host* requires more time than attesting a designated container.

Note that the traditional IMA records all MEs into a single ML, including the *Host's* and containers'. When a *verifier* attests a container, the *prover* should response with all MEs, which will be larger than a container's s-MLs in our solution. Because of the larger size of MEs, the cost of attesting a *prover* through the traditional IMA is larger than our solution.

6.4 Discussion and Limitation

During a container's life cycle, the user may need to stop/restart or pause/unpause it for the sake of usability. These operations do not change the container's namespace, such that Container-IMA is able to support them. However, a container may be migrated in a cluster. Container-IMA does not cover the container migration. Solving this problem will be our future work.

Besides, containers are able to share namespaces in some container clusters. For example, Kubernetes provides Pod for containers with shared namespaces and volumes [31]. A Pod usually hosts containers which are relatively tightly coupled, meaning that these containers commonly need to be considered as a whole. Although Container-IMA does not separate s-MLs for them, transferring these s-MLs does not hinder the privacy. Additionally, a sophisticated attacker may run his container in the same Pod as the victim's container, e.g. through tricking the cluster scheduler. In this case, the attacker's operations will be recorded into the same s-ML

with the victim container. The victim user can be aware of the attacker's existence through the remote attestation. However, if a container cluster itself improperly schedules containers from different users with shared namespaces, there should be a privacy breach. We will consider this problem in the future.

7 Related Work

Trusted Computing Trusted Computing contributes to verifying the trustworthiness of *prover*. Based on IMA, binary remote attestation enables a *verifier* to determine the current integrity state of *prover*. Stumpf et al. [48] propose the Timestamped Hashchain-based Attestation to compensate the deficiency of remote attestation. Considering the large cost of transmitting and calculating ML, Jaeger et al. [28] present Policy-Reduced IMA (PRIMA) which leverages the information flow and SELinux to measure the code and data related to the target application. This limits the scope of the attestation target applications, and hence reduces the size of ML. Some other works [34, 41] also achieve ML reduction.

Virtual TPM (vTPM) [10] devotes to building trust for the hypervisor-based virtualization environment. Through constructing multiple separated *RoTs* for VMs by vTPM, Remote Attestation works for VMs. Several types of vTPM are concluded in [52], such as software-based vTPM [10], hardware-based vTPM [47], para-vTPM [19] and property-based vTPM [45]. Additionally, TCG group has released the TPM 2.0 specification [24]. Chen et al. [13] present a new digital signature primitive in TPM 2.0 with provable security. Raj et al. [40] describe a software-only implementation of TPM 2.0 used in millions of mobile devices. Camenisch et al. [12] research on the TPM 2.0 interfaces and propose a revision to obtain better security and privacy guarantees which requires only minimal changes to the current TPM 2.0 commands.

Container Security Container-based virtualization offers a light-weight virtualization approach to run multiple environments using the host kernel, yet it faces many security issues. Some typical usages of Docker are presented in [14], and the authors discuss Docker's security implications and point out its vulnerabilities. Various mechanisms have been proposed to enhance the security of Docker. The whitepaper of NCC Group [23] explores various security mechanisms for containers, and enumerates strong security recommendations to counter deployment weaknesses. Harbormaster [54] is proposed to enforce policy checks and implement the principle of least privilege for Docker.

Substantial efforts are made to measure the integrity status of containers and then build trust for containers. The mainstream mechanisms are based on the following technologies: vTPM, IMA, *rkt* and SGX.

Some researchers propose solutions based on vTPM [9, 26]. Hosseinzadeh et al. [26] present two possible architectures

to build trust for containers through vTPM. Both these two architectures require a user-space component to bind vTPMs into physical TPM. Berger et al. [9] implement a *vtpm proxy driver* in the Linux kernel that enables to spawn a TPM device with a client TPM character device and a server side file descriptor. The server side file descriptor is passed to the TPM emulator in the user space. Compared with them, our approach does not require an additional layer in the `user space` to manage and coordinate multiple vTPM instances.

Leveraging IMA is another way to build trust for containers. IMA utilizes the hash value of binary code to identify a software component, and records its file path and hash value to ML. Meanwhile, container-based virtualization, e.g. Docker, introduces the `mount namespace` to isolate containers. Different containers may have some files with the same path, e.g. `/bin/ls`, yet IMA regards them as the same software component. To address this problem, researchers in [15, 50] modified IMA to add an additionally item indicating which container the process belongs to, thus it empowers IMA to measure the integrity of containers. However, each *verifier* has to collect the entire ML when validating the integrity status of *prover*, regardless of a container or the underlying host. The privacy requirements thus cannot be achieved. Sun et al. [49] propose `security namespace` for IMA, which allows containers to have their own MLs and IMA policies. The IMA policy is able to specify which PCR the ML is extended into [44]. Once the number of containers is bigger than the number of PCRs, some containers will share a same PCR. And these containers' MLs should all be sent back to *verifier* during the Remote Attestation. The privacy issue still exist. In our work, introducing cPCR is capable of solving this problem. Besides, they did not consider the integrity of container's dependencies.

Based on *rkt*, CoreOS team presents a mechanism [22] to measure the boot phase of containers, such as images and configurations, yet the components loaded in the run time of containers and their dependencies are ignored. And hence it is not enough to meet the security requirement. SCONE [7] leverage SGX trusted execution support to protect container processes from outside attacks. However, the performance overhead is more than expected with regards to the service running in native container environment.

8 Conclusion

In this work, we presented Container-IMA to build trust in a container setting. We firstly analyzed the essential evidence to validate the integrity of a designated container from the perspective of a remote *verifier*. Based on the analysis, we described how the kernel achieves ML partition to ensure privacy and how to utilize cPCR to protect each partition with a hardware-based *RoT*. The corresponding attestation mechanism is proposed as well. Our prototype and the evaluation results demonstrate that our architecture can satisfy the privacy, security and efficiency requirements.

Acknowledgments

This work was supported by National Natural Science Foundation of China under Grant No. 61672062 and No. 61232005.

References

- [1] Integrity measurement architecture (ima). <https://sourceforge.net/p/linux-ima/wiki/Home/>.
- [2] Runc. <https://github.com/opencontainers/runc>.
- [3] Tigest Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *ACM Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.
- [4] Tigest Abera, N Asokan, Lucas Davi, Farinaz Koushanfar, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. Invited-things, trouble, trust: on building trust in iot systems. In *Proceedings of the 53rd Annual Design Automation Conference*, page 121. ACM, 2016.
- [5] Trusted Computing Platform Alliance. Main specification. *Version*, 1:1–284, 2000.
- [6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.
- [7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O’Keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, pages 689–703, 2016.
- [8] 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. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177. ACM, 2003.
- [9] Stefan Berger. Virtual tpm proxy driver for linux containers. <https://www.kernel.org/doc/html/v4.10/security/tpm/index.html>, 2016.
- [10] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS’06, Berkeley, CA, USA, 2006. USENIX Association.
- [11] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.
- [12] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One tpm to bind them all: fixing tpm2. 0 for provably secure anonymous attestation. *Proceedings of IEEE S&P 2017*, 2017.
- [13] Liqun Chen and Jiangtao Li. Flexible and scalable digital signatures in tpm 2.0. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 37–48. ACM, 2013.
- [14] 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.
- [15] Marco De Benedictis and Antonio Lioy. Integrity verification of docker containers for a lightweight cloud environment. *Future Generation Computer Systems*, 97:236–246, 2019.
- [16] Docker. Docker-ce. <https://docs.docker.com/install/linux/docker-ce/ubuntu/>.
- [17] Docker. Sample applications for docker. <https://docs.docker.com/samples/>.
- [18] Docker. Docker containers. <http://docker.com/>, 2016.
- [19] Paul England and Jork Loeser. Para-virtualized tpm sharing. *Trusted Computing-Challenges and Applications*, pages 119–132, 2008.
- [20] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.
- [21] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. Containerleaks: Emerging security threats of information leakages in container clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 237–248. IEEE, 2017.
- [22] Matthew Garrett. What trusted computing means to users of coreos and beyond. <https://coreos.com/blog/coreos-trusted-computing.html>, 2015.

- [23] Aaron Grattafori. Understanding and hardening linux containers. *Whitepaper, NCC Group*, 2016.
- [24] Trusted Computing Group. Tpm 2.0 library specification. <https://trustedcomputinggroup.org/resource/tpm-library-specification/>.
- [25] Matt Helsley. Lxc: Linux container tools. *IBM developerWorks Technical Library*, page 11, 2009.
- [26] Shohreh Hosseinzadeh, Samuel Laurén, and Ville Leppänen. Security in container-based virtualization through vtpm. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 214–219. ACM, 2016.
- [27] Intel. Openattestation (oat). <https://01.org/OpenAttestation>, 2014.
- [28] Trent Jaeger, Reiner Sailer, and Umesh Shankar. Prima: policy-reduced integrity measurement architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28. ACM, 2006.
- [29] Alexey Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012.
- [30] Kubernetes. Docker micro benchmark. <https://github.com/kubernetes/contrib/tree/master/docker-micro-benchmark>.
- [31] Kubernetes. Sharing namespaces in pod. <https://kubernetes.io/docs/concepts/workloads/pods/pod/>.
- [32] Kubernetes. Kubernetes. <http://kubernetes.io/>, 2016.
- [33] Linux. Linux namespace mechanism. <http://man7.org/linux/man-pages/man7/namespaces.7.html>.
- [34] Wu Luo, Wei Liu, Yang Luo, Anbang Ruan, Qingni Shen, and Zhonghai Wu. Partial attestation: Towards cost-effective and privacy-preserving remote attestations. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, pages 152–159. IEEE, 2016.
- [35] Richard Maliszewski, Ning Sun, Shane Wang, Jimmy Wei, and Ren Qiaowei. Trusted boot (tboot), 2015.
- [36] Larry W. McVoy and Carl Staelin. Imbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.
- [37] Chris Mitchell. *Trusted computing*, volume 6. Iet, 2005.
- [38] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 2007.
- [39] A Polvi. Coreos is building a container runtime, rkt. *Retrieved*, 4:2015, 2014.
- [40] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. ftpm: A software-only implementation of a tpm chip. In *USENIX Security Symposium*, pages 841–856, 2016.
- [41] Tobias Rauter, Andrea Höller, Nermin Kajtazovic, and Christian Kreiner. Privilege-based remote attestation: Towards integrity assurance for lightweight clients. In *Proceedings of the 1st ACM Workshop on IoT Privacy, Trust, and Security*, pages 3–9. ACM, 2015.
- [42] Nathan Regola and Jean-Christophe Ducom. Recommendations for virtualization technologies in high performance computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 409–416. IEEE, 2010.
- [43] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N Asokan. Security of os-level virtualization technologies. In *Nordic Conference on Secure IT Systems*, pages 77–93. Springer, 2014.
- [44] Eric Richter. Specifying pcr in the ima policy. <https://git.kernel.org/pub/scm/linux/kernel/git/zohar/linux-integrity.git/commit/security/integrity/ima?h=next-namespacing-experimental&id=0260643ce8047d2a58f76222d09f161149622465>.
- [45] Ahmad-Reza Sadeghi, Christian Stübke, and Marcel Winandy. Property-based tpm virtualization. *Information Security*, pages 1–16, 2008.
- [46] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004.
- [47] Frederic Stumpf and Claudia Eckert. Enhancing trusted platform modules with hardware-based virtualization techniques. In *Emerging Security Information, Systems and Technologies, 2008. SECURWARE'08. Second International Conference on*, pages 1–9. IEEE, 2008.
- [48] Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the scalability of platform attestation. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 1–10. ACM, 2008.
- [49] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security namespace: making linux security frameworks available to containers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1423–1439, 2018.

- [50] Su Tao. *Trust and integrity in distributed systems*. PhD thesis, Politecnico di Torino, 2017.
- [51] PaX Team. Address space layout randomization (aslr). <http://pax.grsecurity.net/docs/aslr.txt>, 2003.
- [52] Xin Wan, Zhiting Xiao, and Yi Ren. Building trust into cloud computing using virtualization of tpm. In *Multi-media Information Networking and Security (MINES), 2012 Fourth International Conference on*, pages 59–63. IEEE, 2012.
- [53] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.
- [54] Mingwei Zhang, Daniel Marino, and Petros Efstathopoulos. Harbormaster: Policy enforcement for containers. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 355–362. IEEE, 2015.

Fingerprinting SDN Applications via Encrypted Control Traffic

Jiahao Cao^{1,2,3,4}, Zijie Yang^{1,2,4}, Kun Sun³, Qi Li^{2,4},
Mingwei Xu^{1,2,4}, and Peiyi Han⁵

¹*Department of Computer Science and Technology, Tsinghua University*

²*Institute for Network Sciences and Cyberspace, Tsinghua University*

³*Department of Information Sciences and Technology, George Mason University*

⁴*Beijing National Research Center for Information Science and Technology, Tsinghua University*

⁵*School of Computer Science, Beijing University of Posts and Telecommunications*

Abstract

By decoupling control and data planes, Software-Defined Networking (SDN) enriches network functionalities with deploying diversified applications in a logically centralized controller. As the applications reveal the presence or absence of internal network services and functionalities, they appear as black-boxes, which are invisible to network users. In this paper, we show an adversary can infer what applications run on SDN controllers by analyzing low-level and encrypted control traffic. Such information can help an adversary to identify valuable targets, know the possible presence of network defense, and thus schedule a battle plan for a later stage of an attack. We design deep learning based methods to accurately and efficiently fingerprint all SDN applications from mixed control traffic. To evaluate the feasibility of the attack, we collect massive traces of control traffic from a real SDN testbed running various applications. Extensive experiments demonstrate an adversary can accurately identify various SDN applications with a 95.4% accuracy on average.

1 Introduction

As a promising network paradigm, Software-Defined Networking (SDN) has attracted much attention from both industry and academia. It is being widely deployed in real-world environments, such as cloud networks [5], data centers [28], and next-generation mobile networks [7]. SDN separates control and data planes with a logically centralized SDN controller managing the whole network. A wide range of innovative applications are deployed in the controller to enable diversified network functionalities, such as load balancing [10], denial-of-service (DoS) attacks detection [24, 71], and network security forensics [61]. They call *high-level* application programming interfaces (APIs) provided by the controller to build their control logic. The controller translates the API calls into *low-level* control traffic, e.g., OpenFlow [8] traffic, to enforce network policies in SDN switches. To prevent potential attacks, control traffic

is usually encrypted with the transport layer security (TLS) protocol [8].

SDN applications provide various network services and functionalities in the network and appear as black-boxes by design to switches. Therefore, network users do not know what applications are running on controllers. It is critical for attackers to be aware of what applications are running on the controller before launching their attacks. Attackers may leverage this information to identify valuable targets, understand the presence of network defense, and develop a battle plan for a future attack. For example, if attackers know there is no TopoGuard [27] security application in SDN, a topology poisoning attack can be directly launched to hijack network flows [27]. In contrast, if attackers detect the presence of TopoGuard, they can customize their attack plan to bypass the defense, e.g., leveraging *Port Amnesia* [56].

In this paper, we show that what applications are running on SDN controllers can be inferred by analyzing low-level control traffic even if the traffic is encrypted. The key insight behind our inference attack is that different SDN applications call APIs with different behaviors, which results in diverse patterns of control traffic. For example, Anonymous Communication [42] periodically rewrites action fields of flow rules with FLOW_MOD control messages, while Traffic Monitor [6] periodically collects flow statistics from flow rules with STATS_REQUEST and STATS_REPLY control messages. The number of packets, the length of packets, and the ratio of incoming and outgoing packets for the control traffic of the two applications are all significantly dissimilar. Such patterns still exist though the control traffic is encrypted. To our best knowledge, inferring applications running on controllers has not been considered so far as a potential attack vector in SDN. Previous studies [12, 21, 32, 39, 52, 57] focus on fingerprinting SDN networks, host communication patterns, and composition of flow rules by actively sending probing packets. Our work here fingerprints SDN applications by passively analyzing control traffic without sending any packets.

Nevertheless, we face two challenging problems to successfully fingerprint SDN applications as follows:

- How to accurately characterize the pattern of control traffic for an application?
- How to efficiently identify multiple applications with mixed control traffic?

For the first problem, the key challenge is that high-level SDN applications generate massive, encrypted, and low-level network control packets, which results in labor-intensive, time-consuming, and difficult manual analysis for characterizing the patterns of control traffic. Particularly, complicated SDN applications call many types of APIs, which results in overlaps of API calls between different SDN applications. For example, Load Balancer [4] generates `STATS_REQUEST` and `STATS_REPLY` control messages to calculate the throughput of a flow, and leverages `FLOW_MOD` control messages to determine the port for forwarding the flow. However, the above three types of control messages are also partly generated by Traffic Monitor [6] and Anonymous Communication [42]. Moreover, there are some identical control packets for different applications, which further increases the difficulty to characterize the patterns of control traffic.

To address the problem, we transform network control packets into a time series and apply deep learning to automatically extract patterns for different applications from it. We try to maintain raw information of control traffic in the time series as much as possible to improve the accuracy of pattern extraction. Specifically, each element in the time series denotes a packet, and the value of an element is the packet length. When a packet is sent from controllers to switches, the corresponding element is multiplied by -1. Besides, the order of elements in the time series is consistent with the order of packets appearing in control traffic. Consequently, most raw information of control traffic is naturally encoded into the time series, such as the lengths of packets, the directions of packets, etc. Thus, the time series can be directly fed into deep neural networks for accurate and automatic feature extraction. Although the contents and delays of packets are missed, they are unhelpful to characterize the patterns of control traffic considering that the packets are encrypted and their delays are usually changeable.

For the second problem, the key challenge is that one single TCP connection between a controller and a switch contains control packets generated by multiple applications concurrently running on the controller. An adversary cannot separate the control traffic of an application from the mixed control traffic to infer what it is. A naive method is to train a deep neural network with all possible combinations of mixed traffic to build a classifier that gives the compositions of applications. However, the number of combinations exponentially grows with more applications. Thus, the deep neural

network quickly becomes exceedingly complicated for classifying the exponential combinations of applications. It is extremely time-consuming and not scalable to train such a complicated deep neural network.

Fortunately, we can solve the problem by dividing it into several subproblems. We train multiple classifiers for multiple SDN applications. Each classifier solves a 2-class classification problem, i.e., whether mixed control traffic contains traffic of an application or not. The training samples for each classifier are two types of mixed control traffic that includes or excludes traffic of an application. Thus, the structure of deep neural networks is simplified and each classifier can be trained in parallel, which significantly reduces the training time. By merging the output results of all classifiers, we know what applications run on controllers.

We conduct experiments in a real SDN testbed consisting of commercial hardware switches and a popular open source controller. We deploy 10 SDN applications on the controller, ranging from network performance optimization to network monitor and network security enhancements. We collect about 6,000,000,000 control packets with different combinations of applications and translate them into many time series of equal lengths. We systematically explore three state-of-the-art deep learning models, i.e., Convolutional Neural Network (CNN) [34], Long Short-Term Memory (LSTM) [26], and Stacked Denoising Autoencoder (SDAE) [59], to train classifiers for fingerprinting SDN applications with time series. The results show that CNN performs the best, which achieves an average accuracy of 95.4% to fingerprint different SDN applications. Besides, we find that the accuracy can be further improved by increasing the length of a time series.

We summarize our key contributions as follows:

- We uncover a new attack vector in SDN, which allows an adversary to infer what applications are running on an SDN controller by analyzing low-level and encrypted control traffic.
- We develop techniques to accurately and efficiently fingerprint SDN applications with mixed control traffic.
- We collect a large dataset of control traffic from a real SDN testbed and systematically evaluate the feasibility of fingerprinting SDN applications with it.

The rest of the paper is organized as follows: Section 2 introduces background on SDN and deep learning. Section 3 provides our techniques to fingerprint SDN applications. Section 4 describes data collection methods and datasets. Section 5 evaluates the effectiveness of fingerprinting SDN applications. Section 6 discusses our current limitations and possible countermeasures against fingerprinting SDN applications. Section 7 reviews related work and Section 8 concludes the paper.

2 Background

In this section, we briefly introduce the necessary background of SDN and deep learning.

2.1 SDN

Software-Defined Networking (SDN) is an emerging programmable network framework that decouples control and data planes. As shown in Figure 1, SDN consists of three main layers: an application layer, a control layer, and a data plane layer. Multiple applications concurrently run in the application layer. They obtain a highly abstracted view of the network and make network policies by calling APIs provided by the control layer. The control layer manages installed network applications and establishes connections with network switches in the data plane layer. It translates API calls of applications into low-level control messages to tell switches on how to forward and process packets.

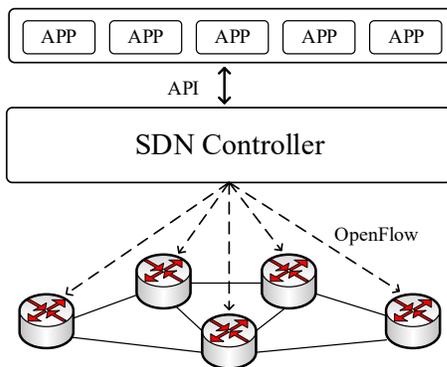


Figure 1: The framework of SDN.

The standardized communication protocol between the control layer and the data plane player is OpenFlow [8]. OpenFlow also specifies functions of SDN switches and enables controllers to manage switches in an open, vendor-neutral, and interoperable way. It defines various control messages to enable diversified functionalities, such as device capabilities advertisement, packet forwarding control, flow statistics reporting, and network events notification. We summarize main control messages and their functionalities in Table 1. Furthermore, as control messages contain sensitive network information and critical network decisions, they are usually encrypted with the TLS protocol.

2.2 Deep Learning

Deep learning has made amazing achievements in many aspects, such as speech recognition, natural language processing, and face recognition. With the support of sufficient data, models with deep structure fit data well and thus can be

Table 1: Main Control Messages in OpenFlow.

Category	Message	Functionality
State Modification	FLOW_MOD	Modify rules in different tables to control packet forwarding and processing.
	GROUP_MOD	
	METER_MOD	
Statistics Collection	FLOW_STAT* †	Collect various flow statistics measured by rules in different types of tables and ports of switches.
	GROUP_STAT*	
	METER_STAT*	
	PORT_STAT*	
Device Configuration	SWITCH_CONFIG	Set and query configuration parameters in switches.
	TABLE_CONFIG	
Capability Announcement	HANDSHAKE	Identify SDN switches and query their capabilities that different tables support.
	TABLE_FEATURE	
	GROUP_FEATURE	
	METER_FEATURE	
Event Notification	PACKET_IN	Notify network events to controllers, e.g., new flows arriving, and send data packets to switches.
	PORT_STATUS	
	FLOW_REMOVED	
	PACKET_OUT	
Liveness Verification	ECHO_REQUEST	Verify liveness and conduct customized measurements.
	ECHO_REPLY	

* A pair of request and reply messages.

† FLOW_STAT can also be used to know all flow rules in switches.

applied to multiple tasks, such as classification and prediction. Compared to traditional machine learning that requires designing a sophisticated feature extractor with expert experience, deep learning adopts a universal learning method to automatically extract features from massive data, which avoids the heavy workload of manually designing features. Different types of deep neural networks (DNNs) have been designed by researchers for different purposes. Out of all existing types of DNNs, we explore three popular types of DNNs to fingerprint SDN applications.

Convolutional Neural Network (CNN). CNN [34] has been widely used in computer vision systems. It contains an input layer, an output layer, and multiple hidden layers that are convolutional layers, pooling layers, and fully-connected layers. Convolutional layers perform a convolution operation to the input and create feature maps that contain abstract features. Pooling layers reduce the dimensions of data by downsampling. CNN typically contains several convolutional and pooling layers to extract more abstract features. Fully-connected layers perform final classification with output feature maps. CNN can well characterize the spatial relationship of data and search for the most important local features. As the positions of SDN control packets in a network flow have strong space relationship due to control logic of applications and may have evident local features, CNN is suitable to characterize patterns of control traffic.

Long Short-Term Memory (LSTM). LSTM [26] is a variant of Recurrent neural network (RNN) that uses feedback connections to store representations of recent input events. LSTM improves RNN for learning long-term dependency in-

formation of sequences and avoiding the problem of vanishing gradient. A common LSTM unit consists of a memory cell, an input gate, an output gate, and a forget gate. The cell is responsible for remembering information over arbitrary time intervals so that it can keep track of the dependencies between the elements in the input sequence. The three gates regulate the flow of information into and out of the cell, deciding whether to let the information in, whether to produce the output, and whether to forget the information. Due to the network structure, it captures temporal dependencies between data. LSTM may be suitable to process control packets of SDN applications since control packets naturally have temporal dependencies.

Stacked Denoising Autoencoder (SDAE). Autoencoder (AE) is a special feedforward neural network, consisting of an input layer, a hidden layer, and an output layer. The input layer and the hidden layer act together as an encoder that compresses data from the input layer into a low-dimensional representation. The hidden layer and the output layer act together as a decoder that reconstructs the data back, i.e., decompressing the representation into something that closely matches the original data. Stacked Denoising Autoencoder (SDAE) [59] stacks multiple AEs together to form a deep network architecture and adds noise to the input data, which makes the network robust. SDAE learns meaningful data representations. Particularly, we may get low-dimensional and highly compressed representations of control traffic to fingerprint SDN applications with SDAE.

3 Fingerprinting SDN Applications

In this section, we first present the threat model and the key insight for fingerprinting SDN applications. We then introduce practical challenges and design methods to solve them.

3.1 Threat Model

In our threat model, we consider control traffic between an SDN controller and a switch is protected with TLS/SSL. We assume an adversary can eavesdrop control traffic between the controller and a switch. Attackers may eavesdrop SDN control traffic in different ways [13, 15–17, 33, 49, 68], such as conducting ARP poisoning for switches and controllers to make control traffic first pass a listening host [17], placing a device between controllers and switches to intercept control traffic [16]¹, intercepting a forwarding link to eavesdrop control traffic [13], and dumping control traffic through listening mode of switches [15]. Particularly, Yoon et al. [68] demonstrate the feasibility of eavesdropping control traffic with real experiments. We do not require an adversary know payloads of control packets that are usually encrypted.

¹SDN control traffic may be carried by an inherently adversarial Internet Service Provider (ISP) [16].

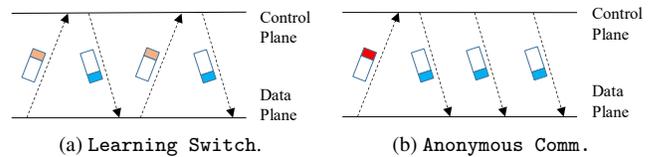


Figure 2: Patterns of control packets for Learning Switch and Anonymous Communication.

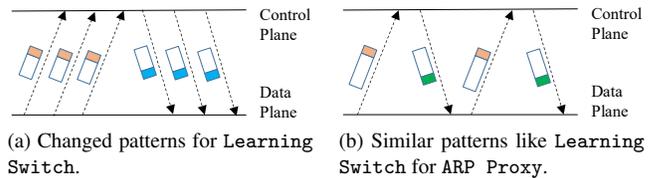


Figure 3: Examples to illustrate that accurately characterizing patterns of control packets for an application is difficult.

Moreover, an adversary may not insert, modify, delay or drop control packets. We do not assume SDN controllers, applications, or switches are compromised by an adversary.

3.2 Key Insight and Challenges

We first give intuitive examples to illustrate our key insight on fingerprinting SDN applications, i.e., different applications generate different patterns of control traffic due to their inherent control logic. Figure 2a and Figure 2b show the patterns of control packets for Learning Switch and Anonymous Communication. The patterns for the two applications are significantly different. Learning Switch receives a PACKET_IN message (orange packets) to analyze a packet for a flow and sends back a FLOW_MOD message (blue packets) to install flow rules on how to forward the packets for the flow. Consequently, the control traffic of Learning Switch consists of multiple pairs of PACKET_IN and FLOW_MOD messages. However, Anonymous Communication periodically inspects all flow rules in a switch with a FLOW_STAT message (red packets). After that, it sends multiple FLOW_MOD messages to rewrite actions of flow rules to modify packet headers for anonymous communication. Different control logic of the two applications results in different patterns of control traffic in many aspects, i.e., packet lengths, directions of packets, relative orders between packets, etc. The patterns still exist even if controllers encrypt control traffic with TLS/SSL. Therefore, an adversary can fingerprint SDN applications by analyzing patterns of control traffic.

However, there are two key challenges in real SDN environments. The first challenge is how to accurately characterize the pattern of the control traffic for an application. The control traffic is low-level and encrypted, which leads to a hard description of patterns of control traffic for differ-

ent applications. Particularly, patterns of control traffic for some applications are mutable due to different network flows in switches. As shown in Figure 3a, the pattern of control traffic for Learning Switch changes if massive new flows come quickly. The reason is that the application is busy processing the burst PACKET_IN messages. It takes some time to respond to the messages. Besides, different applications may have similar patterns of control traffic. For example, Figure 3b shows that control traffic for ARP Proxy consists of multiple pairs of PACKET_IN and PACKET_OUT messages (green packets). It looks similar to control traffic in Figure 2a since we cannot know the content of the encrypted packets. We just see there are many pairs of uplink and downlink packets both in Figure 2a and 3b. The only difference here is that ARP Proxy has larger uplink and downlink control messages. We need a method that can generalize well to accurately characterize the patterns of control traffic for different applications.

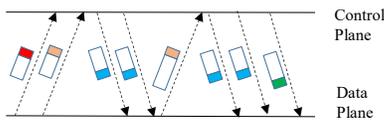


Figure 4: Mixed control packets with Learning Switch, Anonymous Communication, and ARP Proxy.

The second key challenge is that control traffic for multiple SDN applications is mixed in a single TCP connection between a controller and a switch. We cannot easily divide mixed control traffic into multiple types of pure control traffic for identifying each application in turn. Figure 4 shows the mixed control packets with three applications. Actually, the types of packets (colors in packets) are not known from the adversary’s view due to encryption. It is difficult to infer which packets belong to an application especially when there are some identical packets, i.e., blue packets in Figure 4. Furthermore, control traffic becomes more complicated with more applications running on controllers. Although we may infer the compositions of applications for one time without dividing mixed control traffic, the number of the compositions exponentially grows with the number of applications. We need a method that can efficiently identify multiple applications with mixed control traffic.

3.3 Methodology

3.3.1 Packet Transformation

To accurately characterize the pattern of control traffic for an application, we apply deep learning since it can automatically extract features and conduct classifications from enough datasets. Moreover, classification models trained by deep learning can achieve a good generalization ability. However, we cannot directly feed SDN control packets into

neural networks. The reason is two-fold. First, each bit in control packets is encrypted, which does not maintain the original information. Feeding full packets into neural networks may significantly reduce the accuracy of fingerprinting applications. Second, the size of control packets can be large, e.g., up to 12144 bits in Ethernet-based networks. Training deep neural networks with massive large packets is time-consuming.

In order to efficiently train an accurate classifier, we try to maintain useful information and remove unnecessary information in control packets. We transform control packets into a time series that can be the raw input for deep neural networks to automatically extract features and build classifiers. Formally, consider \mathbf{a}_i is the i -th control packet in a packet series $\mathbf{S} = [\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m]$. We transform \mathbf{S} into $S = [f(\mathbf{a}_1), f(\mathbf{a}_2), \dots, f(\mathbf{a}_m)]$. Here, $f(\mathbf{a}_i)$ is a transformation function that maps a control packet into a real number. It is defined as follows:

$$f(\mathbf{a}_i) = \begin{cases} |\mathbf{a}_i|, & \text{if } \mathbf{a}_i \text{ is sent to controllers} \\ -|\mathbf{a}_i|, & \text{if } \mathbf{a}_i \text{ is sent to switches} \end{cases} \quad (1)$$

Here, $1 \leq i \leq m$ and $|\mathbf{a}_i|$ denotes the length of the packet \mathbf{a}_i . Although the transformation process is simple and fast, useful information for SDN application classification is naturally encoded into the time series. The lengths of control packets are denoted by the absolute values of the numbers in the time series, the directions of packets are denoted by the signs of the numbers, and the relative orders of packets are denoted by the positions of the numbers. Thus, we can directly feed each time series into deep neural networks to conduct pattern extraction. Although the encrypted payloads of packets and inter-packet delays are lost in the time series, we consider they are little helpful for identifying an SDN application.

3.3.2 Task Decomposition

Our task is to identify multiple applications that concurrently run on SDN controllers with mixed control traffic. As the mixed control traffic cannot be split, a naive method is to train deep neural networks with all possible combinations of control traffic to build a multi-class classifier that gives the compositions of applications running on SDN controllers. Formally, assume that there are n possible applications running on a controller and control traffic trace is denoted by \mathbf{t} , we aim to give a classifier C that assigns a label c to \mathbf{t} , where $c \in [0, 1, \dots, 2^n - 1]$, i.e., possible combinations of n SDN applications. As shown in Figure 5a, if we directly infer the compositions of possible n applications running on a controller, a classifier should output 2^n types. We need to build a deep neural network with a large amount of parameters to classify the types of exponential scales. Therefore, the neural network quickly becomes exceedingly com-

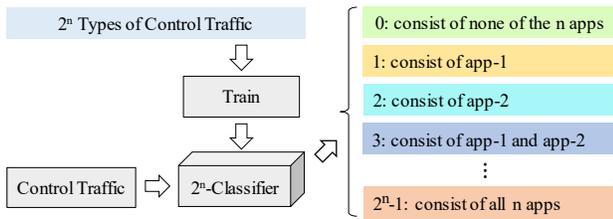
Table 2: SDN Applications in Our Testbed and Corresponding Control Messages.

SDN Applications		Main Control Messages							
		FLOW_MOD	GROUP_MOD	FLOW_STAT	GROUP_STAT	PORT_STAT	PACKET_IN	PACKET_OUT	ECHO
Basic Network Functionalities	Topology Discovery* [11]	×	×	×	×	×	✓	✓	×
	Learning Switch† [3]	✓	×	×	×	×	✓	×	×
	ARP Proxy† [1]	×	×	×	×	×	✓	✓	×
Network Monitor	Traffic Monitor† [6]	×	×	✓	×	✓	×	×	×
	Link Delay Monitor* [31]	×	×	×	×	×	✓	✓	✓
Network Opt.	Load Balancer† [4]	✓	✓	✓	×	✓	✓	×	×
Security and Privacy Enhancement	TopoGuard* [27]	×	×	×	×	×	✓	✓	×
	DoS Detection* [71]	✓	✓	✓	✓	×	✓	✓	×
	Anonymous Comm* [42]	✓	×	✓	×	×	×	×	×
	Scan Detection*‡ [41]	✓	×	×	×	×	✓	✓	×

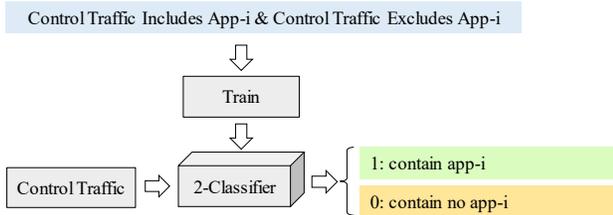
* For applications without source code, we implement and run them on Floodlight according to papers.

† For applications with source code, we directly run them on floodlight.

‡ We implement Scan Detection with the TRW-CB algorithm in the paper [41].



(a) A model of deep learning for the original task.



(b) A model of deep learning for a subtask.

Figure 5: Task Decomposition.

plicated with many applications. Training such a classifier is time-consuming and not scalable.

Thus, in order to efficiently identify multiple applications, we divide the original task into several subtasks. We train n binary classifiers $[C_1, C_2, \dots, C_n]$ for n applications. Each classifier C_i ($1 \leq i \leq n$) only answers if the mixed control traffic \mathbf{t} contains control traffic of the i -th application. As shown in Figure 5b, we feed two types of control traffic, i.e., control traffic including and excluding the i -th application, to train the classifier C_i . Thus, the architecture of neural networks is simplified since a classifier only outputs two classes, i.e., containing control traffic of the i -th application or otherwise.

Moreover, each classifier is independent and can be trained in parallel to reduce total training time. By testing mixed control traffic in each well-trained classifier, we can know what applications run on controllers.

4 Data Collection

To successfully build an accurate classifier, enough training data is required for deep learning to learn underlying patterns and enable good generalization to unseen samples. As far as we know, there are no public traces of SDN control traffic. In this section, we provide the method of collecting the traces and introduce the dataset.

4.1 Data Collection Methodology

We build a real SDN testbed with five commercial hardware SDN switches, Edgecore AS4610-54T, and a popular open source controller, Floodlight. We deploy the controller on a server with a quad-core Intel Xeon CPU E5504 and 32GB RAM. We attach one host on each switch. Each host has a dual-core Intel i3 CPU and 4GB RAM. All hosts in our experiments run Ubuntu 16.04 LTS. In order to generate real data traffic in our testbed, we inject real traffic traces from CAIDA [2] with TCPReplay.

We deploy 10 SDN applications on the controller, ranging from basic network functionalities, advanced network performance optimization, network monitor to security and privacy enhancements. We list these applications and their main control messages in Table 2, which implement representative network functionalities. Topology Discovery dynamically discovers switches and links between the switches and the controller. Learning Switch learns the mappings be-

tween MAC addresses and switch ports, and forwards packets according to the mappings, which makes SDN switches act as layer-2 switches. ARP Proxy provides a MAC address of a host to answer an ARP query for an IP address. Traffic Monitor and Link Delay Monitor monitor network throughput and link delays to provide necessary information for other applications, respectively. Load Balancer optimally schedules the workloads across multiple computing resources. The above six applications are bundled applications in most controllers, including Floodlight, ONOS, and OpenDaylight. Thus, we choose them as typical applications to evaluate the effectiveness of our method on fingerprinting applications. The other four applications are to enhance the security and privacy of SDN [27, 41, 42, 71]. Topoguard fixes a vulnerability of topology poisoning that widely exists in SDN controllers. DoS Detection applies SDN based methods to detect the DoS attack that is one of the most powerful attacks to disrupt a company or an organization. Anonymous Communication provides strong anonymity guarantees for communications in SDN. Scan Detection enables prominent traffic anomaly detection algorithms with SDN to effectively identify malicious activities of hosts.

The types of control messages between these applications are overlapped. However, the control traffic still has different underlying patterns, such as packet length, contexts between packets, etc. We consider the applications as suitable tests for deep learning both in the coverage of different applications and diversified control traffic. We write a shell script to automatically combine different applications to run on controllers. We leverage `tcpdump` on the controller's host to capture TCP packets with the 6653 port (OpenFlow port) between the Floodlight controller and a switch. Note that we only leverage the above method to collect control packets for training deep learning models. In the attacking phase, since it is almost impossible for an attacker to run `tcpdump` on the controller to collect control packets, an attacker should collect them using methods mentioned in Section 3.1. We save each captured control traffic for one combination of SDN applications into a text file. We write a Python program to automatically label all control packets in each text file according to the combination of the applications. Due to storage constraints, we only save extracted metadata from traces of control traffic. The metadata consists of the capture time of packets, the directions of packets, and the lengths of packets. We discard encrypted payloads of packets since they have little value for an adversary. We remove TCP acknowledgment (ACK) packets containing no control messages.

4.2 Dataset

Our dataset contains 6,000,000 control packets for each combination of the 10 applications. Each type of control traffic for one combination is saved in a separated text file. Totally,

there are 6,144,000,000 control packets and 1024 text files. Our current dataset only contains control packets between one switch and the controller. Although collecting more control flows between multiple switches and the controller may help to improve the accuracy of fingerprinting applications, we aim to study the accuracy of fingerprinting in a generalized case since eavesdropping one control flow for an adversary is easy.

5 Evaluation

In this section, we conduct comprehensive experiments to verify the feasibility of fingerprinting SDN applications. We first evaluate the accuracy, precision, and recall rate for 10 applications with three popular deep learning models. Then, we explore how the effectiveness changes with different split lengths of control traffic and different number of datasets. Finally, we evaluate the training time for building a classifier to fingerprint an SDN application.

5.1 Experiment Setup

We implement three models, i.e., SDAE, LSTM, and CNN, for each of the 10 applications with Keras in Python. We train each model on a server equipped with one Intel Xeon Silver 4116 CPU (12 cores), 128 GB RAM, 1TB SSD, and NVIDIA Quadro P4000 GPUs. To train a model for an application, we divide all traces of mixed control traffic into two classes: the first contains control traffic of the application and the second contains no control traffic of the application. We randomly select 60% samples from both the classes as the training set, 20% samples as the test set, and 20% samples as the validation set. We initially define a sequence of 150 control packets as one sample. Moreover, we change the length of one sample to explore how different split lengths affect the effectiveness of fingerprinting SDN applications.

In order to accurately fingerprint SDN applications, hyperparameters of each model should be well tuned so that models have the best classification performance and generalize well to unseen traffic traces. Although conducting an exhaustive grid search or other search algorithms is effective, it is computationally expensive. In our experiment, we semi-automatically tune hyperparameters. We first conduct a grid search with a small dataset, i.e., one-tenth of the original dataset, to know the impacts of each hyperparameter. We next manually adjust parameters with the original dataset based on our experience and experimental results. We list our final parameters in Appendix A.

5.2 Effectiveness

Effectiveness with Different Models. We initially set the split length of a sample as 150. Table 3 shows the accuracy, recall rate, and precision of fingerprinting SDN ap-

Table 3: The Effectiveness of Fingerprinting SDN Applications with different DNN Models.

SDN Applications	SDAE			LSTM			CNN		
	Accuracy	Recall	Precision	Accuracy	Recall	Precision	Accuracy	Recall	Precision
Topology Discovery	90.8%	90.6%	94.1%	92.7%	95.7%	92.5%	94.2%	89.0%	99.3%
Learning Switch	96.4%	97.3%	99.6%	98.3%	92.7%	88.8%	96.4%	90.3%	99.5%
ARP Proxy	87.1%	83.7%	83.9%	92.2%	95.7%	88.1%	94.3%	92.8%	90.8%
Traffic Monitor	94.4%	96.5%	92.5%	92.8%	94.1%	91.7%	90.6%	93.0%	97.9%
Link Delay Monitor	93.4%	92.9%	94.6%	93.2%	98.5%	84.5%	96.5%	99.3%	95.2%
TopoGuard	94.8%	94.8%	97.5%	95.4%	98.5%	83.5%	95.7%	92.0%	98.7%
Load Balancer	93.1%	91.9%	87.2%	90.6%	93.1%	85.4%	97.1%	95.3%	97.3%
DoS Detection	89.6%	90.2%	88.7%	94.3%	90.0%	90.3%	97.8%	97.9%	96.3%
Anonymous Comm	98.2%	97.4%	97.5%	98.1%	94.9%	83.8%	94.7%	89.3%	92.6%
Scan Detection	95.6%	94.5%	87.7%	94.2%	96.6%	94.2%	96.8%	94.0%	98.7%
Average Value	93.3%	93.0%	92.3%	94.2%	95.0%	88.3%	95.4%	93.3%	96.6%
Standard Deviation	3.2%	4.0%	5.0%	2.4%	2.5%	3.7%	2.0%	3.3%	2.8%

plications. For an application, different models perform differently. For example, the accuracy for identifying ARP Proxy is 87.1%, 92.2%, and 94.3% for SDAE, LSTM, and CNN, respectively. The difference between the highest accuracy and the lowest accuracy is 7.2%. The recall rate and precision also change with different models. Moreover, SDAE performs best for Anonymous Communication with a 98.2% accuracy, LSTM performs best for Learning Switch with a 98.3% accuracy, and CNN performs best for DoS Detection with a 97.8% accuracy. Our interpretation is that different models have different capabilities to characterize underlying patterns of applications. Besides, different applications have unique patterns that may be more suitable for extraction with some deep learning model.

Among the three models, LSTM performs the best for the recall rate with an average value of 95.0%. However, it achieves a low precision, i.e., 88.3% on average. Particularly, there is only an 83.8% precision for Anonymous Communication. CNN performs the best both for the accuracy and the precision, which achieves a 95.4% accuracy and a 96.6% precision on average. 7 of the 10 applications have the highest accuracy and 8 of the 10 applications have the highest precision with CNN compared to the other two models. Moreover, CNN achieves an acceptable recall rate of 93.3% on average. SDAE achieves a 93.3% accuracy, a 93.0% recall rate, and a 92.3% precision on average. It performs the worst for the accuracy and the recall rate.

We evaluate the stability of the three models on fingerprinting different applications with the standard deviation. SDAE has the highest standard deviations of accuracy, recall rate, and precision. LSTM has the lowest standard deviation of recall rate and the moderate standard deviations of accuracy and precision. CNN outperforms the other two models both in the standard deviations of accuracy and precision and

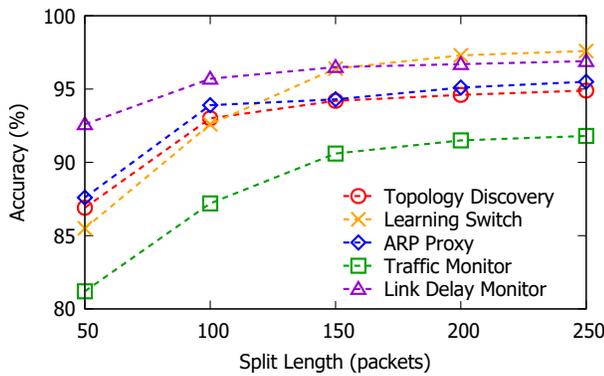
has a moderate standard deviation of recall rate, which is the most stable deep learning model.

Overall, by comparing the three models with each other, we observe that CNN is the most effective and stable for an adversary to fingerprint different SDN applications, especially in classification accuracy and precision.

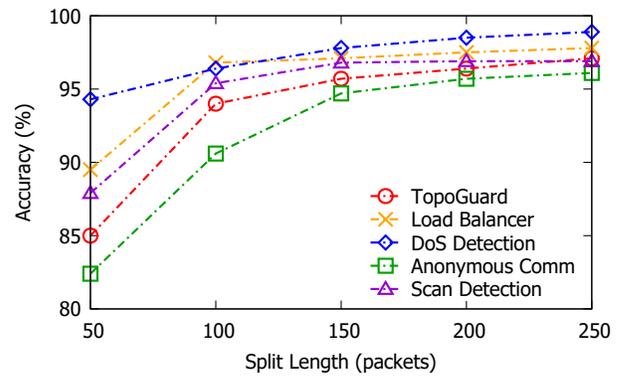
Effectiveness with Different Split Lengths. The effectiveness of fingerprinting SDN applications may change with different lengths of samples. Thus, we divide sequences of control packets into different lengths to train and test deep neural networks. Because CNN performs best, we here explore its accuracy, recall rate, and precision of fingerprinting applications with different lengths of samples.

Figure 6 shows the accuracy for fingerprinting different applications with various split lengths. The results show the accuracy of fingerprinting an application goes up with the split length. When the split length is 50, fingerprinting most applications achieves a low accuracy that is less than 90%. Particularly, fingerprinting Traffic Monitor only reaches an accuracy of 81.2%. When the split length is increased to 250, the accuracy reaches more than 95% for 9 of the 10 applications. The accuracy of fingerprinting Learning Switch, Traffic Monitor, Topoguard, and Anonymous Communication increases by more than 10%. The reason is that more packets in a sample give more underlying patterns. Although the accuracy goes up with the split length, the growth rate of the accuracy gradually slows down. When we increase the split length from 200 to 250, the accuracy is increased less than 1% for most applications and tends to converge.

Figure 7 shows the recall rate for fingerprinting different applications with various split lengths. Similar to the accuracy, the recall rate goes up with split lengths. When the split length is increased from 50 to 150, the recall rate for fin-

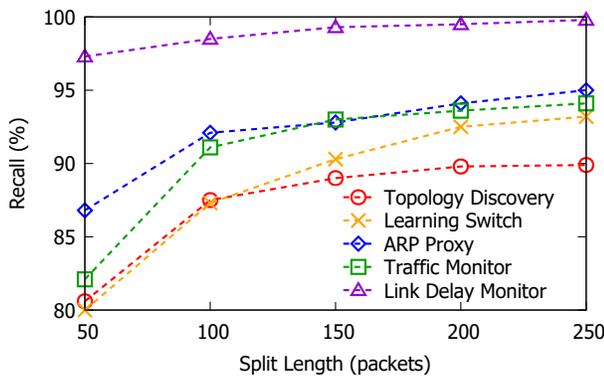


(a) Accuracy of First Five Apps.

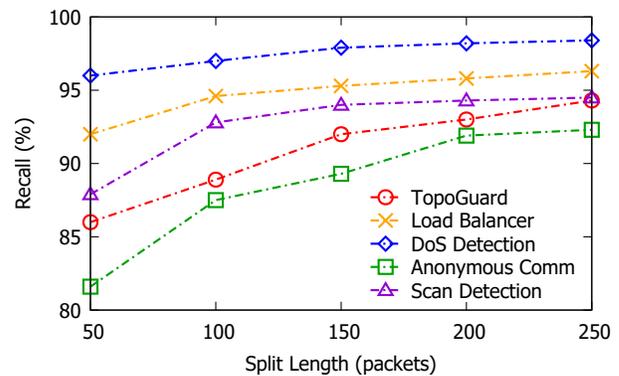


(b) Accuracy of Last Five Apps.

Figure 6: Accuracy of Fingerprinting SDN Applications with Different Split Lengths.



(a) Recall Rate of First Five Apps.



(b) Recall Rate of Last Five Apps.

Figure 7: Recall Rate of Fingerprinting SDN Applications with Different Split Lengths.

gerprinting most applications increases significantly. For instance, the recall rate for fingerprinting Learning Switch increases by 13.2%. There are two exceptions of SDN applications, i.e., Link Delay Monitor and DoS Detection. The recall rate of fingerprinting the two applications is already more than 90% even with a small part of control traffic, i.e., 50 packets, and improves slightly with more control packets in a sample. When the split length is greater than 150, the recall rate stops significant improvement.

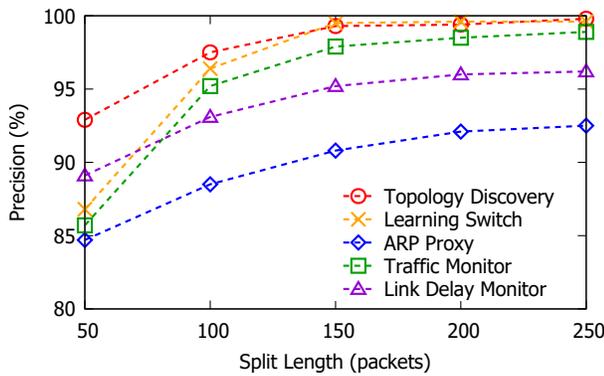
Figure 8 shows the precision for fingerprinting different applications with various split lengths. The precision gradually increases with the split length, following a similar trend like the accuracy and the recall rate. When the split length is increased from 50 to 250, the precision for fingerprinting Learning Switch improves the most, i.e., a 12.8% increase, and the precision for fingerprinting DoS Detection improves the least, i.e., a 5.3% increase. Moreover, the precision for fingerprinting most applications does not significantly improve when the split length exceeds 150.

According to the above results, we conclude that an adver-

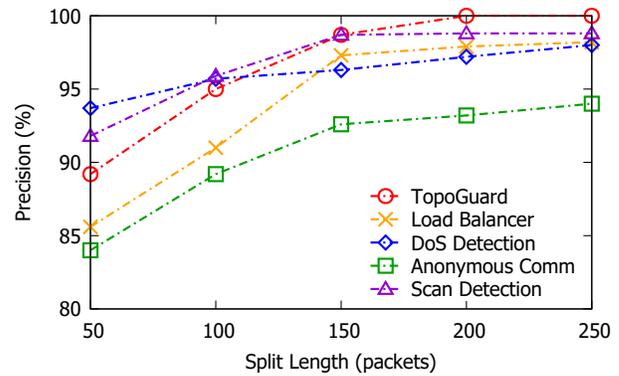
sary can well fingerprint SDN applications with more than 150 encrypted control packets.

Effectiveness with Different Number of Applications. We explore how the effectiveness of fingerprinting an application changes with different number of applications running on controllers. We train and test CNN models for fingerprinting ARP Proxy with five datasets². The five datasets contain control traffic of at most 6, 7, 8, 9, and 10 SDN applications, respectively. As shown in Figure 9, the accuracy, recall rate, and precision slightly decrease with the number of applications. When the number of applications increases from six to ten, the accuracy drops by 1.9%, the recall rate drops by 1.8%, and the precision drops by 2.0%. The results demonstrate that the effectiveness of fingerprinting applications is not significantly affected by the number of applications. Our main conclusion here is that deep learning based classifiers are capable of extracting stable patterns of control

²We also test the effectiveness of fingerprinting other applications with different number of applications. The results are similar to those in Figure 9. For simplicity and due to space constraints, we do not present the results.



(a) Precision of First Five Apps.



(b) Precision of Last Five Apps.

Figure 8: Precision of Fingerprinting SDN Applications with Different Split Lengths.

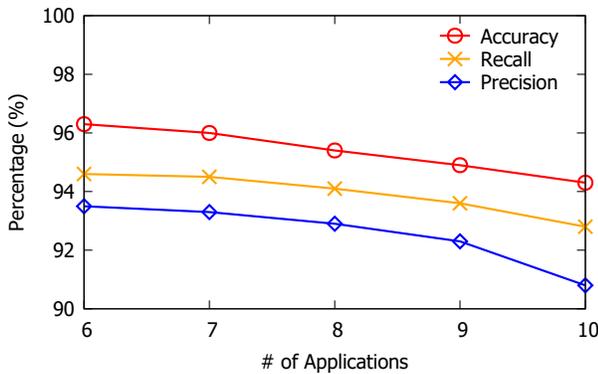


Figure 9: Effectiveness of Fingerprinting an SDN Application with Different Number of Applications.

traffic, which allows an adversary to fingerprint SDN applications with a high success rate.

5.3 Performance

We evaluate the runtime for building a classifier to fingerprint an SDN application. As the runtime of classifiers with different applications changes slightly (less than 1% differences), we list the average runtime for different applications in Table 4. CNN runs fastest among the three models with 2.4 min runtime since it has fewer learnable parameters and most computations in CNN happen in parallel. LSTM performs the slowest due to its recurrent structure where the subsequent processing steps depend on the previous ones.

Table 4: Runtime of Different DNN Models

DNN Models	SDAE	LSTM	CNN
Average Runtime	6.9 min	350.1 min	2.4 min
Loss	0.179	0.192	0.125

6 Discussion

In this section, we discuss the limitations of our current work and possible countermeasures to mitigate the attack.

6.1 Limitations

Model Effectiveness on Traffic and Settings Changes.

Our experiments contain more than 10,000 real network flows to train the deep learning models. Results show that enough training data makes the models generalize well for different flows containing different numbers and sizes of packets. However, since deep learning models learn patterns from data, they cannot classify unseen patterns that training data does not contain. Thus, if network traffic or setting of applications changes significantly, the classification accuracy for fingerprinting certain applications may decrease unless we provide more diverse data to train models. According to our analysis, among the tested ten applications, the accuracy of fingerprinting four applications, i.e., Topology Discovery, Traffic Monitor, Link Delay Monitor, and Anonymous Communication, is sensitive to settings changes but not to traffic changes since they hardly generate network events based on network traffic. For fingerprinting the other six applications, the accuracy may decrease when either network traffic or settings significantly change if there lacks enough training data to cover the changed patterns.

Classifying SDN Applications with Control Traffic of Multiple Switches.

Our threat model currently assumes that an adversary eavesdrops control traffic between one SDN controller and one switch, which is common in practice. Although we demonstrate many applications can be identified with control traffic by deep learning models, we admit a few applications cannot be well classified without further information from control traffic between the controller and other switches. It is because a few applications perform similar

behaviors on one local switch but have different behaviors on multiple switches. Therefore, if we assume a stronger threat model, i.e., an adversary can eavesdrop control traffic of multiple switches, more applications may be classified.

We elaborate this with an example. Considering the two SDN applications: *Learning Switch* and *Reactive Routing*. Any of them running on controllers receives a `PACKET_IN` message to analyze a new flow and then installs a flow rule into the ingress switch with a `FLOW_MOD` message. The `PACKET_IN` messages are same between the two applications for same flows and the `FLOW_MOD` messages between them can also be same if the two applications set same match fields and actions in the flow rules. Thus, the patterns of control traffic for the two applications are same in all aspects, such as packet lengths, relative orders of packets, directions of packets, etc. An adversary cannot classify which application running on the controller only with control traffic between the controller and the ingress switch. However, *Reactive Routing* paves a routing path for a flow in many switches, i.e., installing multiple `FLOW_MOD` messages into each switch along the path once receiving a `PACKET_IN` message from the ingress switch. Instead, *Learning Switch* performs per-hop forwarding, i.e., receiving a `PACKET_IN` message from each switch and installing a `FLOW_MOD` message into each corresponding switch. The patterns of the two applications are different from the view of multiple switches. Thus, it is possible for an adversary to classify the two applications by analyzing control traffic from many switches. However, how to effectively leverage the context between control traffic of multiple switches to fingerprint SDN applications is challengeable. We leave it as future work.

Fingerprinting SDN Applications that generate little control traffic. Although most applications running on controllers continuously generate much control traffic, a few applications only generate little control traffic at some time when network administrators actively change the policy of the applications. For example, *REST Firewall* [9] in the *Floodlight* controller generates no control messages most of the time. However, if a network administrator updates network security policies by commanding the application with *REST API*, the application will install flow rules with specified match fields and actions into switches to enable new security policies. The control traffic of the application is little most of the time, which is difficult for deep learning models to train effective classifiers to identify the application. Moreover, the patterns of control traffic highly depend on how a network administrator command the application, which is extremely mutable. Deep learning models may not extract universal patterns to generalize well to identify the application. One possible but ineffective solution is to manually summarize useful patterns with enough SDN background knowledge to identify them. Our future work will focus on how to efficiently solve the problem.

6.2 Possible Countermeasures

Reducing Differences for Control Traffic. To the best of our knowledge, there are no public SDN defense systems that can mitigate our attack. However, as an adversary fingerprints SDN applications mainly based on different patterns of control traffic, one straightforward mitigation is to reduce the difference between control traffic of various applications. As we mentioned in Section 3.2, the main difference exists in the packet lengths, the packet directions, the relative orders of packets, and the number of packets. Thus, we may encapsulate control messages to normalize them. To normalize the packet lengths, both controllers and switches can reshape different packets by splitting one big packet into several packets or adding padding in small packets so that the packet lengths are equal. Since different packets cannot be identified without knowing their real lengths, the relative orders of packets are also hidden. Moreover, to eliminate the differences in the packet directions and the number of packets, controllers and switches can morph packets into fixed bursts, i.e., breaking each traffic pattern into small bursts of packets consisting of a fixed number of consecutive outgoing packets followed by a fixed number of consecutive incoming packets. By normalizing control packets, deep learning models may thus identify SDN applications with a low accuracy. However, one main disadvantage is that it requires many modifications in switches, controllers, and the *OpenFlow* protocol. Applying the countermeasure in real SDN environments may take a long time and bring some costs.

Adding Adversarial Examples. Another interesting defense strategy worthy of being further studied is to mislead deep neural networks by deliberately generating adversarial examples. They are specially crafted instances with small and intentional feature perturbations to fool deep learning models into false classifications or predictions. Previous studies [40, 65] have demonstrated that adversarial examples can successfully fool deep learning for computer vision and pattern recognition. We may explore how an SDN application can generate adversarial examples of control packets to mislead fingerprinting SDN applications. For example, *ARP Proxy* may periodically generate control packets that simulate the patterns of another application, such as *Learning Switch*, to mislead the classification of deep learning models. It may effectively decrease the accuracy of fingerprinting SDN applications. This defense requires to modify the SDN applications.

7 Related Work

Fingerprinting and Probing in SDN. There are many previous studies on fingerprinting and probing information in SDN. Shin et al. [52] designed a scanning tool to remotely fingerprint networks that deploy SDN by measuring response delays of probing packets. Klöti et al. [32] provided a prob-

ing technique to fingerprint aggregated flow rules by timing TCP setup. Cui et al. [21] demonstrated that an adversary can acquire knowledge on which flow rules installed on switches by analyzing the packet-pair dispersion of data packets. Achleitner et al. [12] presented SDNMap to reconstruct the detailed composition of flow rules by actively sending probing packets with different network protocols. Liu et al. [39] developed a Markov model to infer if a target flow occurred recently by sending optimized probing packets, in the face of rule expiration and eviction. John et al. [57] presented a sophisticated attack to infer host communication patterns, network access control and network monitoring policies by timing processing delays of controllers. Azzouni et al. [14] fingerprinted SDN controllers by timing timeouts of flow rules as well as processing time of controllers. Although the above studies effectively probe many types of information in SDN, none of them show how to fingerprint SDN applications with control traffic. Our work reveals a new attack vector in SDN.

Security Research in SDN. Recently, many SDN security issues have been studied. They cover attacks and security enhancements in all layers of SDN. In the application layer, studies focus on cross-app poisoning [58], malicious applications abusing [36], and secure system for permission control of SDN applications [46, 63]. A wide range of studies focus on the control layer security. Various attacks are presented, including flooding controllers [53, 60], disrupting control channels [18], attacking information mismanagement in SDN-datastores [23], poisoning network typologies [27] and identifiers of network stack [30], generating harmful race conditions in controllers [66], and subverting SDN controllers [48]. Extensive security enhancement systems [22, 30, 51, 56, 60] are designed to mitigate the attacks. Other studies present attacks on data plane, such as low-rate flow table overflow [19], attacking SDN switches with control plane reflection [69], and security policies violation [45]. To fortify SDN data plane, intrusion detection and abnormal data plane diagnose systems [38, 50] are provided. Moreover, automatic vulnerability discovery and security assessment tools [29, 35] are designed to understand the possible attack surface of SDN. In contrast to existing work, we show a new threat to SDN and existing defense systems cannot defend it.

Encrypted Network Traffic Analysis. Analyzing encrypted network traffic to infer possible information based on packet sizes, timing and other side channel leaks has been extensively studied. Li et al. [37] fingerprinted personas from WI-FI traffic by analyzing meta-data information on interactions through HTTPS connections with machine learning. Chen et al. [20] showed detailed sensitive information can be leaked out from encrypted network traffic of web applications. Wright et al. [64] designed an attack to identify the phrases spoken within a call by analyzing lengths of encrypted VoIP packets. Zhang et al. [70] pro-

vided HoMonit to monitor smart home applications from encrypted wireless traffic. Moreover, a series of previous studies [25, 43, 44, 47, 54, 55, 62] focuses on website fingerprinting with the onion router (Tor) that preserves anonymity for Internet users. They reveal which website Tor users are visiting by analyzing Tor traffic with machine learning. However, all these studies make a single page assumption, i.e., the collected traffic always belongs to a single page from a website and contains no mixed traffic from other pages. One study [67] relaxes the assumption and provides a multi-tab website fingerprinting attack on partially mixed traffic. It provides a split algorithm to extract a small initial chunk of packets of the first page, which is not overlapped with the packets of the following pages. Different from these studies, we concentrate on fingerprinting what applications run on SDN controllers via encrypted control traffic. Particularly, control packets of different SDN applications are mixed in a single TCP connection. Since all applications concurrently send control messages to switches, control packets are tightly coupled and totally mixed and thus can not be split using the algorithm in the study [67]. We provide novel techniques to accurately and efficiently fingerprint all applications from mixed control traffic.

8 Conclusion

In this paper, we present a new attack on SDN that fingerprints SDN applications with low-level and encrypted control traffic. It exploits different patterns of control traffic caused by different behaviors of applications to infer what applications run on SDN controllers. In order to characterize the underlying patterns, we transform network packets into the time series and apply deep learning to automatically learn the patterns to fingerprint SDN applications. We divide the task of fingerprinting multiple SDN applications into several subtasks to improve the efficiency of training deep learning models. We collect massive traces of control traffic from a real SDN testbed. Extensive experiments demonstrate that an adversary can effectively fingerprint SDN applications with a high accuracy.

Acknowledgments

The research is partly supported by the National Natural Science Foundation of China under Grant 61625203, 61832013, 61572278, and U1736209, ONR grants N00014-16-1-3214, N00014-18-2893, and ARO grant W911NF-17-1-0447. Mingwei Xu and Qi Li are corresponding authors.

References

- [1] ARP Proxy. <https://github.com/mbredel/floodlight-proxyarp/>. [Online].

- [2] CAIDA Passive Monitor: Chicago B. http://www.caida.org/data/passive/trace_stats/chicago-B/2015/?monitor=20150219-130000. UTC. [Online].
- [3] Learning Switch. <https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/learningswitch/>. [Online].
- [4] Load Balancer. <https://github.com/floodlight/floodlight/blob/master/src/main/java/net/floodlightcontroller/loadbalancer/>. [Online].
- [5] Microsoft Azure and Software Defined Networking. https://docs.microsoft.com/en-us/windows-server/networking/sdn/azure_and_sdn/. [Online].
- [6] Network Traffic Monitor. <https://github.com/floodlight/floodlight/blob/master/src/main/java/net/floodlightcontroller/statistics/>. [Online].
- [7] NGMN - 5G White Paper. <https://ngmn.org/5g-white-paper/5g-white-paper.html>. [Online].
- [8] OpenFlow Specification v1.5.1. <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>. [Online].
- [9] REST Firewall. <https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/firewall/>. [Online].
- [10] Software Load Balancing (SLB) for SDN. <https://docs.microsoft.com/en-us/windows-server/networking/sdn/technologies/network-function-virtualization/software-load-balancing-for-sdn>. [Online].
- [11] Topology Discovery. <https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/topology/>. [Online].
- [12] Stefan Achleitner, Thomas La Porta, Trent Jaeger, and Patrick McDaniel. Adversarial network forensics in software defined networking. In *ACM SOSR'17 (2017)*.
- [13] Ahmad Aseeri, Nuttapon Netjinda, and Rattikorn Hewett. Alleviating eavesdropping attacks in software-defined networking data plane. In *ACM CISRC'17 (2017)*.
- [14] Abdelhadi Azzouni, Othmen Braham, Thi Mai Trang Nguyen, Guy Pujolle, and Raouf Boutaba. Fingerprinting openflow controllers: The first step to attack an sdn control plane. In *IEEE GLOBECOM'16 (2016)*.
- [15] Kevin Benton, L Jean Camp, and Chris Small. Openflow vulnerability assessment. In *ACM HotSDN'13 (2013)*.
- [16] Kevin Benton, L Jean Camp, and Chris Small. Openflow vulnerability assessment (extended abstract). https://benton.pub/research/openflow_vulnerability_assessment.pdf.
- [17] Michael Brooks and Baijian Yang. A man-in-the-middle attack against opendaylight sdn controller. In *ACM SIGITE/RIIT'15 (2015)*.
- [18] Jiahao Cao, Qi Li, Xie Renjie, Kun Sun, Guofei Gu, Mingwei Xu, and Yuan Yang. The crosspath attack: Disrupting the sdn control channel via shared links. In *USENIX Security'19 (2019)*.
- [19] Jiahao Cao, Mingwei Xu, Qi Li, Kun Sun, Yuan Yang, and Jing Zheng. Disrupting sdn via the data plane: a low-rate flow table overflow attack. In *SecureComm'17 (2017)*.
- [20] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *IEEE S&P'10 (2010)*.
- [21] Heng Cui, Ghassan O Karame, Felix Klaedtke, and Roberto Bifulco. On the fingerprinting of software-defined networks. *IEEE Transactions on Information Forensics and Security*, 11(10):2160–2173, 2016.
- [22] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. Sphinx: Detecting security attacks in software-defined networks. In *NDSS'15 (2015)*.
- [23] Vaibhav Hemant Dixit, Adam Doupé, Yan Shoshitaishvili, Ziming Zhao, and Gail-Joon Ahn. Aim-sdn: Attacking information mismanagement in sdn-datastores. In *ACM CCS'18 (2018)*.
- [24] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *USENIX Security'15 (2015)*.
- [25] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security'16 (2016)*.
- [26] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

- [27] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS'15 (2015)*.
- [28] Jain, Sushant et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.
- [29] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, Richard Skowyra, and Sonia Fahmy. Beads: automated attack discovery in openflow-based sdn systems. In *RAID'17 (2017)*.
- [30] Samuel Jero, William Koch, Richard Skowyra, Hamed Okhravi, Cristina Nita-Rotaru, and David Bigelow. Identifier binding attacks and defenses in software-defined networks. In *USENIX Security'17 (2017)*.
- [31] Seong-Mun Kim, Gyeongsik Yang, Chuck Yoo, and Sung-Gi Min. Bfd-based link latency measurement in software defined networking. In *IEEE CNSM'17 (2017)*.
- [32] Rowan Klöti, Vasileios Kotronis, and Paul Smith. Openflow: A security analysis. In *IEEE ICNP'13 (2013)*.
- [33] Diego Kreutz, Fernando Ramos, and Paulo Verissimo. Towards secure and dependable software-defined networks. In *ACM HotSDN'13 (2013)*.
- [34] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [35] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip A Porras. Delta: A security assessment framework for software-defined networks. In *NDSS'17 (2017)*.
- [36] Seungsoo Lee, Changhoon Yoon, and Seungwon Shin. The smaller, the shrewder: A simple malicious application can kill an entire sdn environment. In *ACM SDN-NFV Security 2016*.
- [37] Huaxin Li, Zheyu Xu, Haojin Zhu, Di Ma, Shuai Li, and Kai Xing. Demographics inference through wi-fi network traffic analysis. In *IEEE INFOCOM'16 (2016)*.
- [38] Qi Li, Yanyu Chen, Patrick PC Lee, Mingwei Xu, and Kui Ren. Security policy violations in sdn data plane. *IEEE/ACM Transactions on Networking (TON)*, 26(4):1715–1727, 2018.
- [39] Liu, Sheng et al. Flow reconnaissance via timing attacks on sdn switches. In *IEEE ICDCS'17 (2017)*.
- [40] Jiajun Lu, Theerasit Issaranon, and David Forsyth. Safetynet: Detecting and rejecting adversarial examples robustly. In *IEEE ICCV'17 (2017)*.
- [41] Syed Akbar Mehdi, Junaid Khalid, and Syed Ali Khayam. Revisiting traffic anomaly detection using software defined networking. In *RAID'11 (2011)*.
- [42] Roland Meier, David Gugelmann, and Laurent Vanbever. itap: In-network traffic analysis prevention using software-defined networks. In *ACM SOSR'17 (2017)*.
- [43] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. Deepcorr: Strong flow correlation attacks on tor using deep learning. In *ACM CCS'18 (2018)*.
- [44] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website fingerprinting at internet scale. In *NDSS'16 (2016)*.
- [45] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *ACM HotSDN'12 (2012)*.
- [46] Phillip A Porras, Steven Cheung, Martin W Fong, Keith Skinner, and Vinod Yegneswaran. Securing the software defined network control layer. In *NDSS'15 (2015)*.
- [47] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In *NDSS'18 (2018)*.
- [48] Christian Röpke and Thorsten Holz. Sdn rootkits: Subverting network operating systems of software-defined networks. In *RAID'15 (2015)*.
- [49] Arash Shaghghi, Mohamed Ali Kaafar, Rajkumar Buyya, and Sanjay Jha. Software-defined network (sdn) data plane security: Issues, solutions and future directions. *arXiv preprint arXiv:1804.00262*, 2018.
- [50] Arash Shaghghi, Mohamed Ali Kaafar, and Sanjay Jha. Wedgetail: An intrusion prevention system for the data plane of software defined networks. In *ACM AsiaCCS'17 (2017)*.
- [51] Gao Shang, Peng Zhe, Xiao Bin, Hu Aiqun, and Ren Kui. Flooddefender: Protecting data and control plane resources under sdn-aimed dos attacks. In *IEEE INFOCOM'17 (2017)*.
- [52] Seungwon Shin and Guofei Gu. Attacking software-defined networks: A first feasibility study. In *ACM HotSDN'13 (2013)*.

- [53] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *ACM CCS'13 (2013)*.
- [54] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *ACM CCS'18 (2018)*.
- [55] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *ACM CCS'18 (2018)*.
- [56] Richard Skowyra, Lei Xu, Guofei Gu, Veer Dedhia, Thomas Hobson, Hamed Okhravi, and James Landry. Effective topology tampering attacks and defenses in software-defined networks. In *IEEE DSN'18 (2018)*.
- [57] John Sonchack, Anurag Dubey, Adam J Aviv, Jonathan M Smith, and Eric Keller. Timing-based reconnaissance and defense in software-defined networks. In *ACSAC'16 (2016)*.
- [58] Benjamin E Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. Cross-app poisoning in software-defined networking. In *ACM CCS'18 (2018)*.
- [59] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12):3371–3408, 2010.
- [60] Haopei Wang, Lei Xu, and Guofei Gu. Floodguard: A dos attack prevention extension in software-defined networks. In *IEEE DSN'15 (2015)*.
- [61] Haopei Wang, Guangliang Yang, Phakpoom Chinpruthiwong, Lei Xu, Yangyong Zhang, and Guofei Gu. Towards fine-grained network security forensics and diagnosis in the sdn era. In *ACM CCS'18 (2018)*.
- [62] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *USENIX Security'14 (2014)*.
- [63] Xitao Wen, Bo Yang, Yan Chen, Chengchen Hu, Yi Wang, Bin Liu, and Xiaolin Chen. Sdnshield: Reconciling configurable application permissions for sdn app markets. In *IEEE DSN'16 (2016)*.
- [64] Charles V Wright, Lucas Ballard, Scott E Coull, Fabian Monrose, and Gerald M Masson. Spot me if you can: Uncovering spoken phrases in encrypted voip conversations. In *IEEE S&P'08 (2008)*.
- [65] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Yuyin Zhou, Lingxi Xie, and Alan Yuille. Adversarial examples for semantic segmentation and object detection. In *IEEE ICCV'17 (2017)*.
- [66] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. Attacking the brain: Races in the sdn control plane. In *USENIX Security'17 (2017)*.
- [67] Yixiao Xu, Tao Wang, Qi Li, Qingyuan Gong, Yang Chen, and Yong Jiang. A multi-tab website fingerprinting attack. In *ACM ACSAC'18 (2018)*.
- [68] Changhoon Yoon, Seungsoo Lee, Heedo Kang, Taejune Park, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Flow wars: Systemizing the attack surface and defenses in software-defined networks. *IEEE/ACM Transactions on Networking (TON)*, 25(6):3514–3530, 2017.
- [69] Menghao Zhang, Guanyu Li, Lei Xu, Jun Bi, Guofei Gu, and Jiasong Bai. Control plane reflection attacks in sdns: new attacks and countermeasures. In *RAID'18 (2018)*.
- [70] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. Homonit: Monitoring smart home apps from encrypted traffic. In *ACM CCS'18 (2018)*.
- [71] Jing Zheng, Qi Li, Guofei Gu, Jiahao Cao, David KY Yau, and Jianping Wu. Realtime ddos defense using cots sdn switches via adaptive correlation analysis. *IEEE Transactions on Information Forensics and Security*, 13(7):1838–1853, 2018.

A Hyperparameters

Table 5: Hyperparameters of Different DNN Models

Hyperparameter	DNN Models		
	SDAE	CNN	LSTM
optimizer	RMSProp	Adam	RMSProp
learning rate	0.001	0.001	0.001
decay	0.0	0.0	0.0
batch size	64	64	64
training epoch	5 .. 10	5 .. 10	15 .. 25
number of layers	4	10	4
input units	50 .. 250	50 .. 250	50 .. 250
hidden layer units	100, 50, 20	-	32, 32, 32
dropout	0.1	0.5	-
activation	tanh	tanh	-
pretraining optimizer	RMSProp	-	-
pretraining learning rate	0.001	-	-
kernel size	-	100, 200	-
kernel size	-	10	-
pool size	-	2	-

Exploring Syscall-Based Semantics Reconstruction of Android Applications

Dario Nisi¹, Antonio Bianchi², and Yanick Fratantonio¹

¹EURECOM

²Purdue University

dario.nisi@eurecom.fr, antoniob@purdue.edu, yanick.fratantonio@eurecom.fr

Abstract

Within the realm of program analysis, dynamic analysis approaches are at the foundation of many frameworks. In the context of Android security, the vast majority of existing frameworks perform API-level tracing (i.e., they aim at obtaining the trace of the APIs invoked by a given app), and use this information to determine whether the app under analysis contains unwanted or malicious functionality. However, previous works have shown that these API-level tracing and instrumentation mechanisms can be easily evaded, regardless of their specific implementation details. An alternative to API-level tracing is syscall-level tracing. This approach works at a lower level and it extracts the sequence of syscalls invoked by a given app: the advantage is that this approach can be implemented in kernel space and, thus, it cannot be evaded and it can be very challenging, if not outright impossible, to be detected by code running in user space. However, while this approach offers more security guarantees, it is affected by a significant limitation: most of the semantics of the app's behavior is lost. These syscalls are in fact low-level and do not carry as much information as the highly semantics-rich Android APIs. In other words, there is a significant *semantic gap*.

This paper presents the first exploration of how much it would take to bridge this gap and how challenging this endeavor would be. We propose an approach, an analysis framework, and a pipeline to gain insights into the peculiarities of this problem and we show that it is much more challenging than what previously thought.

1 Introduction

In the realm of malware analysis for Android apps, dynamic analysis approaches and instrumentation techniques are at the foundation of virtually all existing analysis frameworks, developed by both academia and industry [2, 6, 10, 14, 17–19, 21, 31, 34]. While dynamic analysis approaches can take many forms, they all share one key aspect: given an application, the goal is to “capture” all actions it performs during its

execution. To this end, these apps are run in an instrumented environment, which records a trace of the app's behavior.

API-level tracing. In Android, most of these approaches aim at producing a list of high-level API calls performed by the app under analysis. These high-level API calls are Java methods exposed by the Android framework, a vast extension of the Java SDK. These methods include standard Java methods (e.g., string operations, networking primitives), as well as a large corpus of Android-specific methods, such as APIs dealing with building Android user interfaces, inter-app interactions, reading values from device's sensors, sending and receiving text messages. Having access to an accurate trace of invoked APIs is of great importance. In fact, these APIs capture the high-level, *semantic-rich* behavior of an app and allow both human analysts and automated approaches to detect and characterize both malicious and unsafe actions the app could perform.

These approaches work by heavily instrumenting the app itself or the execution environment (e.g., by function hooking). Unfortunately, to date, *all* current API-level instrumentations can be easily detected and bypassed [8]. The key problem is that these instrumentation mechanisms all introduce visible instrumentation artifacts, which co-exist within the same security boundary as the app itself. Acquiring these high-level traces require heavy instrumentation, which is hard to implement efficiently and it is trivially detectable by malware, which could decide not to show any malice when instrumentation is detected [33]. Android apps can also contain components written in native code, whose behavior cannot be captured if the app's instrumentation is only performed at the Java API level. More importantly, previous works have shown that the mere presence of native code can make the results of the analysis of the Java layer not only detectable and evadable, but even misleading [8]. In fact, since native code components and Java code run within the same security boundary, native components could surreptitiously modify the intended functionality of Java components making high-level recordings of an app's behavior completely un-

reliable. These issues severely affect the reliability of acquiring high-level API-based traces.

Syscall-level tracing. A different approach consists in capturing the actions performed by an app by recording its low-level interactions with the operating system, specifically, by recording the system calls (syscalls) it invokes [11, 30]. This approach is not affected by the limitations mentioned above. In fact, regardless of the language used to implement the different app’s components, to interact with the operating system, the app *needs* to eventually invoke a system call. Moreover, this kind of instrumentation is harder to detect, since it can be easily implemented entirely by code running in kernel mode, not visible to the analyzed malicious app.

Bridging the semantic gap. Given the security guarantees that approaches based on syscall-level analysis would get us, it is clear that, ideally, this approach should be preferred. In practice, however, the information they extract is too low level, making their results difficult to be interpreted. The conceptual problem is that, in the general case, it is challenging to recover the high-level *semantics* of an app’s behavior solely starting from the list of recorded syscalls. For instance, even a simple operation, such as instantiating an SSL connection to a remote server, which an Android app can perform by invoking a single high-level API, generates a complex sequence of multiple syscalls, most of them seemingly unrelated with the triggering of the high-level functionality. In this specific case, for instance, the complexity is due to the fact that the analyzed app ultimately has to invoke a series of syscalls belonging to different technical areas to complete this task, including inter-process communication with the system service that provides the trusted CA certificates, random-number generation for creating the nonce used to setup the connection, and network-related syscalls to perform the handshake with the remote server. While there are few works that reconstruct parts of this behavior (e.g., CopperDroid [30] focuses on reconstructing the semantics of specific activities, including Binder-related operations), it is not clear whether reconstructing this semantics gap is in fact possible or practical in the *general* case. In fact, even though it may be practical to scan for specific patterns in a sequence of syscalls, traces often contain thousands of syscalls that do not seem to relate to any common pattern.

Goal of this work. To date, we are not aware of any work that actually investigates the feasibility of reconstructing the high-level semantics from generic low-level syscall traces. The goal of this work is to fill this gap: *this paper presents the first systematic exploration of the challenges and feasibility of bridging the gap from trustworthy system calls to semantics-rich, but difficult-to-obtained high-level APIs.*

To this end, we have built a new analysis framework aiming at exploring the complexity of this research problem with a data-driven approach. The first key challenge is the scale: by dynamically analyzing 750 Android apps, we have col-

lected data on over 40 million API invocations, which in turn generated over 13 million syscalls invocation (interestingly, many API invocations do not invoke any syscall). We then process this low-level data to build a knowledge base of so-called “models,” which aim at summarizing the big amount of raw data that we have collected in the previous step, to make it more viable for subsequent analysis. The complexity of the Android framework, the high number of exposed high-level APIs, the API’s non-deterministic behavior, and the overlap generated by these APIs, make the analysis of this dataset far from trivial. To the best of our knowledge, this paper performs the first data-driven exploration of this problem space, and it provides evidence that this is a very difficult problem, significantly more challenging than what previously thought.

In summary, this work brings the following contributions:

- We systematically explored the research problem of *semantically lifting* a generic trace of performed system calls to a trace of invoked high-level APIs, with a focus on Android.
- We built a large-scale, annotated dataset that maps high-level APIs to the various “representations” of low-level syscall traces, and we provide an in-depth discussion of patterns and other interesting aspects.
- We develop and test different approaches attempting to perform the aforementioned semantic lift problem, and we show that this is a much more difficult problem than what previously thought.
- We provide recommendations and lessons learned that future work needs to consider when tackling this problem.

In the spirit of open research, we make our instrumentation framework, the collected dataset, and the analysis results publicly available at: <https://github.com/eurecom-s3/syscall2api> .

2 Background on Dynamic Analysis

Android framework API. Programming languages are commonly divided in two categories, depending on whether they provide a high- or a low-level abstraction over the computing system. High-level programming languages provide to the programmer a closer experience to a natural language and they are designed to perform complex tasks in few lines of code. On the other hand, low-level programming languages allow the programmer to interact with those aspects of computation that high-level programming takes for granted.

High-level programming languages expose to programmers a set of functionalities, called Application Programming Interface (API). In Android, these APIs are implemented in the so-called Android Framework. Some of these APIs are not implemented solely in Java, since their behavior exceeds the expressiveness of this language. They rely instead on the Java Native Interface (JNI), which provides a bridge toward parts of the framework written in C or C++. This is the case for some of the most complex and, arguably, the most security relevant APIs, like those that handle the personal data of the user, interact with the broadband, access the Internet, etc. Indeed, those functionalities require the intervention of the operating system to be accomplished. Being based on the Linux kernel, in the Android operating system a user space application can take advantage of the services exposed by the kernel by means of system calls (syscall from now on).

Even though it is true that any security sensitive operation is performed by means of syscalls, the contrary is not true. In fact, a vast number of syscalls are actually invoked to implement behaviors that are not strictly security-sensitive, such as user interaction, memory management, and thread synchronization.

It is important to note that not only the framework, but also apps can contain pieces of code written in low-level languages. Moreover, both the high- and low-level code run in the same process and with the same privileges and there is no security boundary between the two. This is a common misconception, which led previous works to overlook the role of native code in the realm of Android dynamic analysis [8].

Dynamic analysis. Understanding the behavior of a program is an important step toward determining whether it is malicious or not. Dynamic analysis aims to gather this information from running the program in a controlled environment, recording as much evidence of malicious activities as possible.

Depending on the type of the controlled environment, the collected information can vary. Execution traces are one of the most common types of evidence collected during dynamic analysis and they describe a timeline of what was executed in the context of the program under analysis. Different granularities are possible, including API- and syscall-level traces.

API tracing records all the high-level functions invoked during the execution. Different mechanisms have been proposed to obtain such traces, including framework modification, run-time hooking and Ahead-of-Time (AOT) compilation instrumentation. Unfortunately, all of them can be detected and evaded by native code components.

Framework modifications, for example, can be identified by a malicious application through memory introspection. Moreover these techniques rely on the assumption that the program uses the default run-time provided by the system, but a malicious application could ship its own run-time li-

brary as a native library, avoiding completely the instrumentation. Run-time hooking and AOT compilation instrumentation suffer from similar problems. They both assume that the malicious code is implemented by the app in the high level language. However, the malicious behavior could be perpetrated by the native code, for example by mimicking the same syscalls that the framework would invoke to complete the same task. More fundamentally, the fallacy of API tracing mechanisms resides in that the instrumentation is in the same security context of the program under analysis.

On the contrary, syscall traces can be obtained directly from the kernel, in a transparent way from the program perspective. There are several techniques to acquire syscall traces, the two most prominent being `strace`, a `ptrace`-based mechanism, and SystemTap [12], which inserts probes in kernel space and logs relevant information. The main drawback of syscall tracing is that the information collected are difficult to interpret. Finding evidence of malicious activity from a syscall trace alone can be a hard task.

3 Challenges

Reconstructing the semantic gap from a syscall trace is a task made particularly difficult by several challenges, which this section systematizes. We note that the discussion of these challenges is “conceptual” — a priori, it was not known whether these challenges would or would not actually pose problems when dealt with in practice. To the best of our knowledge, in fact, no previous work has ever explored the actual practicality issues that these challenges create. One of the contributions of this work is to fill this gap: as we will present throughout the paper, our experiments provide the first data-backed evidence that these challenges do cause profound problems.

Multiple possible execution paths. The first challenge is that different invocations of the same API could follow different execution paths. This could be the case for a number of reasons. An API could behave differently depending on the arguments with which it has been invoked. However, its behavior could also differ depending on the execution environment and context. Different execution paths of course imply that the number and type of syscalls that are executed upon API invocation can widely vary. For example, consider an HTTP-related API: from the perspective of syscalls invocations, the recorded trace can widely change depending whether the API’s argument is a valid URL, or whether the device has network connectivity. These aspects can clearly influence whether we would see network-related activity in the syscall traces.

Non-determinism. Another potential problem is non-determinism. With this term, we refer to those cases for which even if an API is invoked with the same arguments and within a “similar” environmental context, the syscall traces

could still differ due to inherent non-determinism of the system or because of very subtle “internal” differences (e.g., the current internal state of the memory allocator). Naturally, one could argue that the lowest-level aspect of the system could be considered as part of the “context” and that this challenge is overlapping with the previous one. This would be, of course, a valid argument. Nonetheless, we opted to make this distinction explicit due to the different nature of the source of potential divergent behaviors. As we will discuss throughout the paper, the different nature greatly influences the *frequency* with which such non-determinism arise and *how* these problems should be tackled in practice.

Multiple layers of APIs. The Android framework is organized as a complex, multi-layer system of APIs: each API, especially the ones “exposed” to third-party apps, are implemented by invoking several others lower-level APIs. Indeed, it is rare that a high-level API directly invokes syscalls. This means that, when dealing with these higher-level APIs, every potential behavioral difference and non-determinism that affect lower-level APIs will be somehow combined—in a potentially combinatorial way. This makes capturing all different behaviors of a non-trivial API very challenging in the general case.

Inherent ambiguity of syscall traces. Different APIs often use the same syscalls to implement their behaviors. In other words, a given sequence of syscalls can (and often does) overlap across the execution of different APIs. From the perspective of analyzing a syscall trace to then understand which APIs have been actually invoked, this poses a significant challenge: it is very complex to “go back” with certainty as there are many different possibilities that may explain a particular sequence of syscalls. We then say that these syscall traces are *ambiguous* as it is often not possible to determine which, across a number of potential candidates, is the real API that has been actually invoked.

No clear boundaries. Given a syscall trace, it is challenging to determine when the syscall sub-trace of a specific API is starting or ending. In fact, there are no clear-cut markers signaling these aspects, and traces of different APIs (or even of the same one) may have different lengths. This aspect, together with the other aspects and the combinatorial nature of how low-level APIs are used to implement higher-level APIs, makes associating a series of syscalls to a given API much more challenging.

Event-based nature of Android apps. Android apps are written following an event-driven paradigm, which implies that apps often make use of callbacks. The classic example is the definition of an `onClick` callback to define what should happen when the user clicks on a specific button.

This pattern often causes several, nested control flow transitions from the Android framework to the Android app, and vice versa. In fact, consider what happens in the scenario

where a user clicks on a button: 1) the control flow transitions from the framework to the `onClick` callback method, implemented in the app; 2) the `onClick` method may invoke several Android APIs, which would cause the control flow to transition back to the Android framework; 3) when the execution of these APIs is over, the control flow goes back to the `onClick` method; 4) when the `onClick` method ends its execution, the control flow goes back to the Android framework once again. These various control flow transitions make our analysis significantly more complicated. We note that these problems do not affect more traditional programs that do not heavily rely on asynchronous callbacks.

APIs cannot be invoked without proper context. With the aim of collecting data about which syscalls are invoked by which API, one possibility would be to consider each API separately and automatically invoke it within an instrumented environment. Unfortunately, this approach would not work in practice. In fact, the vast majority of APIs need to be invoked with the appropriate context, or otherwise they would quickly quit their execution due to errors. Moreover, many of these APIs require a proper “receiving” object to be invoked on, and the automatic creation of such objects is a very challenging task per-se.

4 Approach

This section discusses how we approached the various challenges discussed in the previous section. Our approach is summarized in Figure 1.

The first step of our approach consists in building a dataset containing *which syscalls are invoked by which API*. Conceptually, the idea is to use this dataset as a sort of ground truth, to then use it to perform additional experiments. As mentioned earlier, this task is challenging per-se. In fact, we cannot just create code to execute the various APIs, as we would not know with which arguments we would need to invoke them and from which context. We approached this problem by taking a large number of *benign* Android apps and by executing them in an instrumented environment to produce both API- and syscall-level traces. This step is discussed in Section 5.

This raw data is sparse and contains a remarkable amount of redundancy, and the scale of this data does not make it suitable to be used for additional analysis without a pre-processing step. Thus, in a second step, the raw data is then organized in a data structure (that we refer to as *knowledge base*), which lays the foundation for subsequent analysis. For this step, the idea is to eliminate unneeded redundancy and to somehow obtain a concise representation of what contained in the dataset. The output of this analysis is a set of so-called *API models*. These models offer a “usable” over-approximation of all the behavior collected during the initial analysis phase (see Section 6).

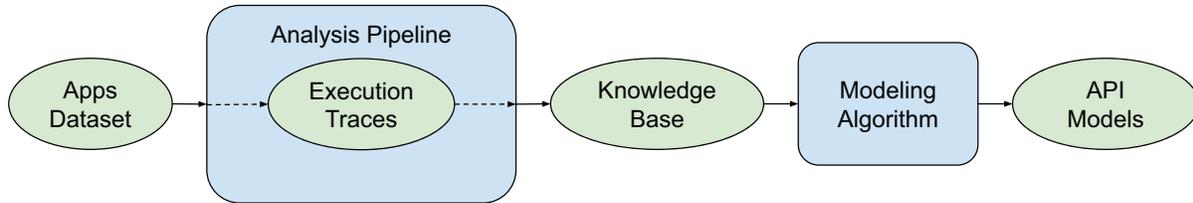


Figure 1: Overview of the approach.

The specifics of these API models have been designed to be useful for two different purposes. First, we perform the first empirical data exploration on this peculiar dataset (see Section 7), and we use it to uncover patterns and high-level metrics that show how challenging the problem of semantics reconstruction actually is. Second, we use these API models to take the first steps toward *mapping a generic sequence of syscalls to their associated APIs*, as discussed in Section 8.

5 Knowledge Base

```

API A: Entering
  Syscall w
  Syscall x
API B: Entering
  Syscall y
API B: Exiting
API C: Entering
  Syscall z
API C: Exiting
API A: Exiting
  
```

Listing 1: Example of an analysis trace.

In this section we present the methodology we followed to create our knowledge base. We started by considering a set of *benign* Android apps, which we then analyzed within our analysis framework, discussed in this section. This analysis framework consists in an instrumented environment capable of logging traces of *both* syscalls and APIs. These raw analysis traces are then parsed and loaded in a more suitable tree-based data structure.

5.1 Analysis Tracing Pipeline

Syscall-level tracing. To log the syscalls invoked by a given app we relied on `strace`, which is a robust, off-the-shelf tool based on the `ptrace` syscall. For each syscall, we traced the timestamp, the calling *thread id*, the syscall name and its arguments. For obvious performance reasons, this behavior can be selectively enabled on the application under analysis only, so to avoid to slow the entire system down with unneeded instrumentation.

We note that, in principle, relying on `strace` has two disadvantages. First, it can be detected by an app. While this is true, this is not a problem at this stage because our goal is to collect the behavior of benign apps, which we assume to not contain anti-debugging techniques. Moreover, `strace` can be completely implemented in kernel space [12, 20], making it more resilient to anti-debugging techniques and suitable for the analysis of malicious programs. Second, `strace` can cause a significant slowdown. However, once again, this is not a significant concern in our scenario as we are analyzing apps to collect “as much behavior as possible,” and we do not necessary need to cover “all” the behavior of an app. In other words, while the slowdown may make us lose some behavior, this aspect does not threaten the validity of our experiments. The aspect that is actually of critical importance is that all the events (both syscalls and APIs) are logged in the appropriate chronological order, which is the case for our system.

API-level tracing. To log the APIs invoked by a given app, we first considered well known instrumentation frameworks, such as Xposed [6] and Frida [2]. In fact, one of the main features of these frameworks is the possibility of tracing specific API methods. Unfortunately, it turns out that when attempting to hook more than a few hundreds APIs, these frameworks make the system unstable, leading to repeated crashes. This is a problem as the Android framework is constituted by tens of thousands of APIs.

To this end, we have developed a new solution, which is based on source code instrumentation. By means of JavaParser [4], we automatically instrumented *all* the public methods in the AOSP framework. In particular, we added a call to `logApi()` — a new static method that we defined in the `java.logging.Logger` class — at the entry point and at all exit points (e.g., return statements, catch blocks of exceptions) of every instrumented method.

The `logApi` method takes a string as its first argument, which our instrumentation pass uses to specify *which* API has been invoked. In particular, this string contains the name of the instrumented method and whether the call originates from an entry point or an exit point (i.e., whether the method has been just invoked or whether its execution is about to end). Under the hood, this `logApi` method simply invokes

a `write` syscall, using as an argument the same argument received by the `logApi` method itself.

This technical solution gives us a setting where both syscalls and APIs logging converge in the same *unified tracing channel*. Since these analysis traces are also thread-aware (by simply logging the thread id of the thread that invoked the API or syscall), all the log entries are already chronologically ordered and consistent, by design.

Example of an analysis trace. The result of this step is a merged analysis trace, which contains both syscalls and APIs, with the corresponding thread id and timestamp. Listing 1 shows an example of an analysis trace. In the listing, it is possible to see how the system can transparently log both *enter* and *exit* events for both syscalls and APIs.

5.2 Building a Knowledge Base

The analysis traces created in the previous step contain *all* the information collected, but they are not easily processable. To this end, we post-process these traces and we organize them in a more suitable data structure. This structure consists in a key-value store in which each key is the fully qualified method name of an API, and each value is a list of entries, each of which represents a specific instance of an API invocation. Each of these entries contains a list of events recorded between the start and the end of that specific instance of the API invocation. The events can either be “syscall invocation” or “API invocation.”

6 API Models

Invocations of the same API usually share common features but they are not always completely identical. For example, the two different branches of an if-else construct in the API code can lead to different sequences of API calls or syscalls (see Section 3 for a more systematic discussion of similar challenges). The knowledge base discussed in the previous section contains all relevant information and it can be already used as a source of interesting data. However, it cannot be used to recover the high-level semantics without some kind of pre-processing. The reason for this is that APIs can potentially have a big number of different invocations (see Section 7) and each invocation can be significantly different from others.

In this section we introduce the concept of *API models*, their design and the rationale behind it. We then present an algorithm that creates API models starting from the invocations of an API. The last part of this section discusses how a sequence of syscalls can be then matched against these API models.

6.1 Anatomy of an API Model

In the context of this paper, an API model is an object that summarizes the common features between different invocations of the same API. A model is constituted by an ordered sequence of symbols representing the various APIs and syscalls found in the invocations. Each symbol in the model can have an optional *modifier* that indicates that it can appear up to an unlimited number of times.

API models represent an over-approximation of all the information stored in the knowledge base. In fact, by assuming that a syscall pattern can be repeated up to an unlimited number of times, an API model can match sequences that have not been observed in the API invocations.

The choice to model repetitions of syscalls in this way is driven by the intuition that repeated patterns generate from loops in the execution. Those loops can be repeated either a fixed or a variable number of times. Our API models make use of the repetition modifier only if the same pattern has been observed repeating itself a different number of times in distinct invocations.

6.2 API Models Creation Algorithm

The model creation phase consists in applying a two-step algorithm to all the APIs in the knowledge base. In the first step we identify all those invocations that are identical according to the following definition: *two invocations are identical if they contain the same API calls and syscalls in the same order*. Note that the syscalls arguments are not taken into account. Duplicate invocations are not taken into account from further processing.

In the second step the algorithm processes each of the remaining invocations to create a list of models for each API. In particular, the algorithm proceeds as follows. It first attempts to find the longest repeated pattern in the invocation under analysis. If a repeating pattern is found (e.g., a single syscall or a sequence of syscalls that keep repeating itself), the algorithm creates a model similar to the original invocation, except for the repeating pattern that is marked with the repetition modifier. This model is then checked against the other invocations. If at least one of them produces a match, the generalization is considered “useful” and the model is added to the list of API models. If not, it means that this over-generalized model was not useful: the algorithm thus discards it, and adds to the list of API models the trivial model matching the “exact” invocation under analysis.

6.3 API Models Matching

Once these API models are computed, the next step is to approach the *mapping problem*. Given a sequence of syscalls, this problem aims at determining which API is the most likely to have generated such a sequence. In a way, the goal

is to *map* a given sequence to the “correct” API. There are of course many different possible algorithms and strategies to implement this.

For this paper, we opted to implement two *extreme* strategies: the longest match and shortest match strategies. In both cases, the algorithm starts from the very beginning of the syscall sequence. It then considers all the API models in our database and it determines which of these API models actually match the given sequence of syscalls. The algorithm then selects the longest (or the shortest) of these matches, and this initial sequence of syscalls is considered as covered. The algorithm then proceeds by applying the same method to the sequence of syscalls that followed the one that is covered by the selected API model.

We note that this constitutes the first step into reconstructing the API trace starting from a sequence of syscalls. We present an evaluation of these two strategies in Section 7. Of course, we acknowledge that there are in fact many other potential strategies. However, we believe that considering these two extreme strategies is a promising first step toward understanding and exploring this relevant research problem.

7 Data Exploration

This section explores our knowledge base (KB) by discussing interesting statistics and insights. We start by giving more precise information about the apps that we analyzed for collecting the analysis traces and the experimental setup. We then present measurements about the information stored in the KB. These measures provide empirical data highlighting the difficulties in reconstructing the high-level semantics from generic low-level syscall traces. Specifically, we will show how the APIs in the KB are very diverse and, because of their nature, how different APIs present different challenges for semantic reconstruction. We will also show how a human analyst can query the KB and how this is important in highlighting the problematic nature of two aspects, namely noise and ambiguity, making semantic reconstruction a task more challenging than what previously thought. We then explore these two aspects in an automated fashion and we discuss the gained insights in Section 7.3 and 7.4, respectively.

7.1 Apps Dataset and Experimental Setup

As mentioned throughout this paper, we opted for a data-driven approach to explore the problem of semantics gap reconstruction. We build our ground truth of analysis traces by recording the execution of a set of 750 apps. We collected these apps from the F-Droid Open Source Android App Repository [1]. In particular, we selected *all* apps from this dataset that used at least one dangerous permission. The rationale behind this choice is that these apps would tend to be more complex than others not requiring any permission,

	Leaf APIs	Non-Leaf APIs
Empty APIs	1730	-
Monoform APIs	29	810
Multiform APIs	573	1488

Table 1: API occurrences in KB. Note: there cannot be Empty APIs that are also Non-Leaf APIs

	Leaf APIs	Non-Leaf APIs
Empty APIs	2850	-
Monoform APIs	59	665
Multiform APIs	94	962

Table 2: API occurrences in KB (after noise reduction)

and would thus have more chances to expose interesting behavior.

Each app was executed for five minutes on a Google Nexus 5X device, which was previously instrumented with our modified Android framework, as described in Section 5.1. We then used the Android Monkey Runner [16] to stimulate the app’s user interface. We fully acknowledge that the Monkey Runner is not sophisticated and may not trigger deep parts of the app’s codebase. However, we note that the rationale behind these experiments is not to fully cover a specific app, but to execute many apps and collect what we can from each of them.

7.2 API Classification and Statistics

Our KB contains invocations for a total number of 4,630 distinct APIs. The total number of API invocations observed is over 40 million, while the total number of syscall that these API invocations invoked is over 13 million. Note that the number of API invocations is larger than the number of syscalls, which means that a significant number of API invocations do not result in any syscall. In average, each API has been observed 8,721 times, while the average number of events in the invocation lists is 0.84 (3.63 without considering empty invocation lists).

We categorize the APIs according to two different aspects. First, we consider how many different entries each API has in its invocation list. That is, for each API we look at how many different syscall sequences we have recorded in our dataset. We distinguish three different cases: 1) *Empty APIs*, those whose invocations are all empty lists; 2) *Monoform APIs*, those for which all the invocations are equals (and non-empty); and 3) *Multiform APIs*, those that have at least two different non-empty invocations. Second, we cluster APIs according to the *type* of events that each of their invocations contains. For this aspect, we distinguish between 1) *Leaf APIs*, those whose invocations contain only syscalls, and 2) *Non-Leaf APIs*, those containing at least one API in their invocation lists.

Table 1 shows the occurrences of each category of APIs in our KB. It is interesting to understand how each category of API plays a different role in the context of semantic reconstruction. *Empty APIs* are those that are completely implemented in user space and do not make any system calls. For this reason, their behavior cannot be identified at all based on syscall information only. *Multiform APIs* make the overall task of semantic reconstruction very challenging because all of their invocations must be taken into account. Our modeling algorithm, for example, could produce more than one model for each API in this class. *Monoform APIs*, on the other hand, are simpler because their only invocation can also be used as model.

Leaf APIs are the closest to syscalls in terms of semantics. Some of them invoke always the same syscall (e.g., *android.net.LocalSocket.setSoTimeout* always executes the syscall *setsockopt*). They are also the easiest to reconstruct, since their behavior can be recognized directly from the executed syscall. To reconstruct a *Non-Leaf API*, instead, one needs first to reconstruct the other APIs that it could invoke.

7.3 Noise Patterns Identification

While inspecting the data offered by our knowledge base, we came across surprising insights. For example, we found some syscalls in the invocation list of a few of those APIs that were expected to be *empty*. One example is the *java.lang.StringBuilder.append* API. Interestingly, while the vast majority of the invocations of these APIs were indeed empty, (very) few of these invocations contained peculiar syscall patterns that, at first glance, we could not explain with the expected behavior of the API. To our surprise, we then found that these patterns were also observed in the models built for *other* APIs.

To investigate this unexpected finding, we developed a post-process analysis pass to automatically identify similar cases. The key idea is to perform anomaly detection. Our system identifies a model of an API as an outlier if the model describes a number of invocations (of that API) that is lower than a certain threshold (for our tests, we used 1/1000 as a threshold). With this tool, we identified three “noise” patterns. The first relates to thread synchronization (e.g., *mutex*, *sched_yield* and *clock_gettime* syscalls). The second relates to memory management (e.g., *madvise* and *mprotect*) or their combinations. Finally, the third pattern we identified relates to the specific malloc implementation in Android’s *bionic* C library: since it is used to obtain memory for the allocation of new objects, it can potentially appear during the invocation of any API that allocates Java objects—and, similarly to the other two cases, this is what causes the noisy pattern.

To better explore the role that these noisy patterns have in our dataset, we opted to eliminate from our models all the syscall patterns that contribute to such noise (i.e., the ones mentioned above), since we believe that these classes of

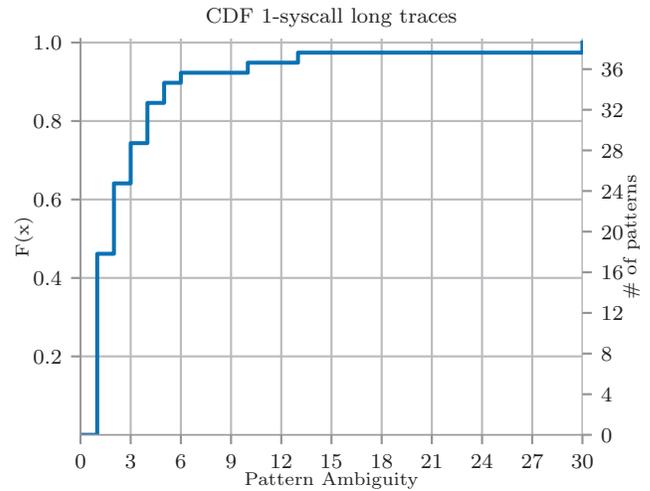


Figure 2: Pattern Ambiguity for 1-syscall long patterns

syscalls do not carry any meaningful information that can be used to reconstruct any API semantic. Table 2 shows statistics for each class of APIs in the KB after removing the noisy patterns. A comparison of these data with those in Table 1 suggests that noise reduction leads to more consistent data. For instance, the number of APIs that have two or more different invocations decreased by almost 50%.

7.4 Ambiguity Measurement

Another aspect we explored relates to the inherent ambiguity of models included in our KB. For example, we noticed that some models overlap or are identical, even though they belong to different APIs. This means that potentially more than one API model can provide a match for the same syscall pattern, leading to ambiguity in the results of any matching algorithm.

With the goal of quantifying the ambiguity of the results in our dataset, we define a new metric, which we call *ambiguity score*. This metric is an integer number that can be computed for each pattern of syscalls. We define this metric as the number of different APIs (in our KB) that match against a given pattern of syscalls. We tackle this problem by considering syscalls patterns of different lengths. Moreover, we consider two different values: *pattern ambiguity score* and *total pattern ambiguity score*, the only difference between the two being that in the latter case we weight a pattern according to how many times it appeared in our traces. Figure 2 and Figure 3 show the cumulative distribution functions (CDF) of the ambiguity score for 1-syscall and 2-syscall long patterns respectively (after having removed the noise). Figure 4 and Figure 5, instead, plot the CDF of the *total* ambiguity score, for the same patterns. These figures show the data before and after removing the noise. For a better visual comparison be-

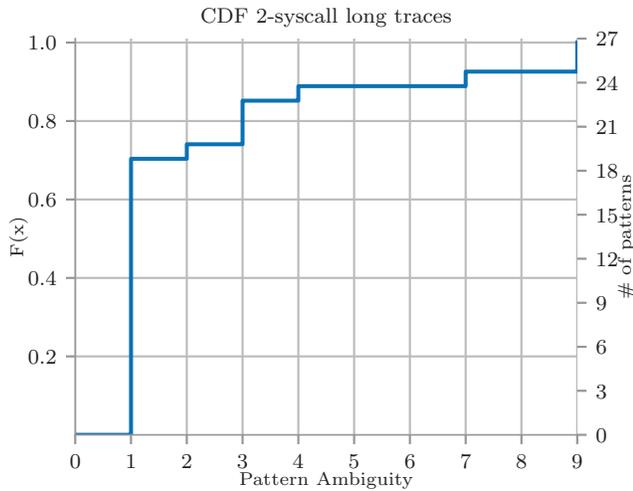


Figure 3: Pattern Ambiguity for 2-syscall long patterns

	Shortest Match		Longest Match	
	w/ noise	w/o noise	w/ noise	w/o noise
Trace coverage	26.4%	45.2%	33.0%	61.5%
Correct matches	30.9%	38.9%	26.4%	46.9%

Table 3: Results of two variants of the matching algorithm.

tween the CDFs, the figures show the data around the point in which the CDF of the noiseless KB reaches 1.0. The CDF of the noisy KB instead reaches 1.0 at much higher abscissa (not shown in the figures), since it raises at a slower pace with respect to those of the noiseless KB. This is also true for the CDFs built for other syscall pattern lengths, meaning that models built without removing the noise are more ambiguous than their noiseless counterpart. For the interested reader, we also report, in the Appendix, the CDFs for patterns of different lengths (from three to up to five).

8 Exploring the Mapping Problem

This section discusses a first attempt to reconstruct the semantics gap of a *generic* sequence of syscalls. The input to this step is a non-annotated syscall trace and we investigate how two strategies would perform in this context. The challenges discussed in Section 3 make this task particularly difficult.

To this end, we define the notion of correct match as follows. A match is *correct* if it spans the same syscalls of an API in the annotated trace and if said API is in the set of those that the algorithm selected as candidates. This means that a match is not considered correct if, for example, it covers more syscalls than the ones actually produced by the API, or if it does not start exactly on its first syscall.

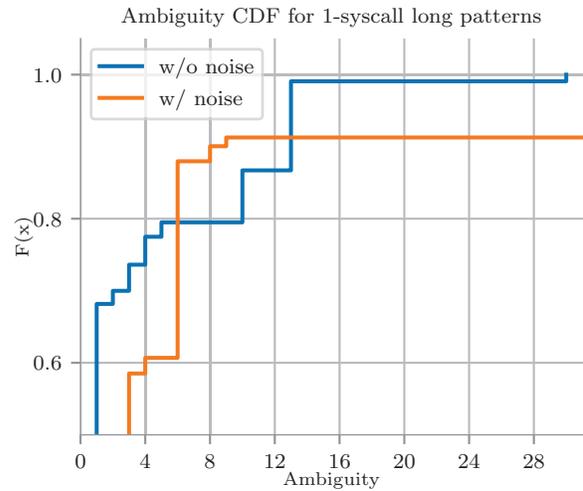


Figure 4: Total Pattern Ambiguity Comparison (1-syscall long patterns)

	Method	Class	Package
Trace Coverage	61.8%	62.0%	63.1%
Correct Matches	46.9%	47.0%	49.3%

Table 4: Accuracy results under relaxed definition of correctness.

We measure the results of the reconstruction process in terms of percentage of APIs correctly identified (i.e., the ratio between the number of APIs correctly identified and the total number of APIs in the annotated trace) and percentage of the traces covered by correct matches (i.e., the ratio between the number of syscalls correctly assigned to a candidate API and the number of syscalls effectively in the trace).

We implemented two variants of a matching algorithm: a *shortest match* and a *longest match* policy. Table 3 reports the results in terms of *trace coverage* (i.e., which percentage of the trace was possible to cover) and *correct matches* (i.e., the ratio of matches that are correct). These results show that, clearly, the “longest match” heuristic results are better, as it produces higher rates of correct matches and trace coverage in each single test. Table 3 also provides a comparison of the results obtained by adopting the models built before and after noise reduction. It is interesting to note that not only noise reduction decreases the ambiguity (as shown in Section 7.3), but it also increases the amount of correct matches.

We note that the results reported thus far use a quite aggressive definition of “correctness.” In a way, we consider a match correct if and only if the algorithm is able to identify the specific, exact API. Since there are cases in which different APIs belonging to the same class (or package) actually have the same semantics, we decided to explore how the accuracy would change under a more relaxed definition of “correct match.” Table 4 shows the percentage of trace coverage

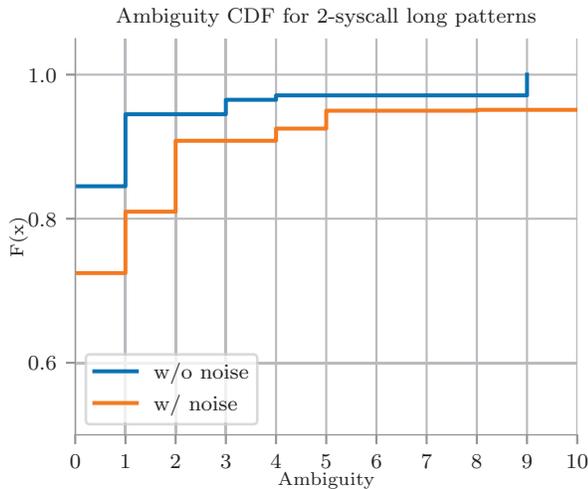


Figure 5: Total Pattern Ambiguity Comparison (2-syscall long patterns)

and match correctness when a match is considered as correct in three different scenarios (using the “longest match” heuristics and after noise removal): we consider a match correct if the method name, the class name, or the package name of the API matched (instead of its fully qualified name, which also contains the types of its arguments). The table shows that the numbers do improve, but that they are still far from ideal. We believe that the reason for the low accuracy of the results of our algorithm resides in the fact that it fails in recovering from an incorrect match caused by a length mismatch. In this situation, the algorithm is out of synchronization as it tries to match the next API from the very next syscall in the trace.

Another source of desynchronization is given by those sections of the trace in which no APIs are recorded (i.e., when the execution of the application returns to the framework, see Section 3 for more details). This issue cannot be solved applying the same approach used for APIs, but they do not share enough features to build meaningful models. Moreover, these areas contain similar patterns to those observed in API models, making it difficult to distinguish between them.

Discussion. These results show that simple strategies are far from enough to properly map sequences of low-level syscalls to high-level APIs. The mapping task appears even more challenging when considering that our experimental setup made the analysis, in theory, much simpler than what it would be in a real scenario. In fact, our experiments attempt to map sequences of syscalls to APIs that have been extracted from the same knowledge base. In a real scenario, instead, the algorithm would not have any guarantee that the potential target API is one API already in the knowledge base (as an unknown app may make use of APIs never seen in the training set). Moreover, we even simplified the problem by assuming that the starting point of the first API in the trace

is known. Last, our “correctness” definition is quite generous, as it considers an API match as correct if the correct API is among one of the selected candidates: ideally, the perfect matching system should indicate only one API for each match. We believe these observations strength our key hypothesis: *that, even under these very favorable conditions, the mapping problem presents inherent difficulties that are very challenging to overcome.*

Future directions. We believe that reducing the noise in the models is a key component for a successful approach to the mapping problem, since we identified that the noisy patterns produced by the framework and the ART runtime a strong source of ambiguity. To this aim, we believe that future work should investigate a more aggressive noise reduction strategy to improve the results. Another direction for improvements could be to leverage the fact that some APIs are often used in conjunction with others, providing a heuristic for choosing between multiple candidate APIs.

At the moment, our system collected information about the APIs exposed via Java public method. A different approach would be to monitor the invocation of every single Java method (including private ones) in the framework. On the one hand, this approach could shed light on those areas of the traces that seemingly do not contain any API. On the other hand, private methods are more difficult to interpret for an analyst since they are undocumented and require knowledge of the internals of the Android framework to be fully understood.

Another step of our pipeline that can be enhanced is the model creation algorithm. In general, we believe that future works should focus on creating API models that describe as many features of the API invocations as possible. In this work we show how to model repeating patterns, but there are other features that are worth modeling. For example, one possible improvement can be to summarize in the same model all those invocations that differ for a small number of syscalls and/or API calls only.

In an attempt to model this type of scenarios, a first version of our prototype relied on the Needleman-Wunsch sequence alignment algorithm [26]. The rationale was that by aligning two invocations is possible to find those syscalls and API calls that appear in only one of them. We leveraged the aligned sequences to create models in which the symbols corresponding to these API calls and syscalls were marked with an additional modifier. This modifier indicates that the symbol to which it is applied is “optional,” meaning that the model matches a sequence regardless of its presence. This approach, however, led to unacceptably high computational complexity of the pattern matching phase, to the point of making it unfeasible in practice for models with many entries. Still, we believe there could be some value in adopting this technique for only a subset of the APIs, especially those whose invocations contain only a small number of syscalls and API calls.

Lastly, we believe our approach would clearly benefit from integrating the results from existing systems like CopperDroid [30], which already perform several steps to recover the semantics of the *ioctl* syscalls. This particular class of syscalls is among the most frequent and ambiguous ones in our knowledge base. This is due to its fundamental role in the low-level implementation of the Binder subsystem, which relies on the *ioctl* syscall to exchange “parcels” of data between user apps and system services. Integrating CopperDroid with our system would add the corresponding Binder semantics to each *ioctl* syscall, which would eventually reduce the ambiguity for this class of syscalls significantly.

9 Related Work

The Android security community has published a vast number of works related to program analysis of unknown apps. This section places our work in the context of two main related areas, namely static and dynamic program analysis.

Several static analysis approaches have been proposed to analyze Android apps, and malware in particular. Some of the early works in this area include RiskRanker [18] and DroidRanger [34], which rely on symbolic execution and a set of heuristics to detect unknown malicious applications. Another work is Apposcopy, which uses a signature-based approach to detect known malware samples [14]. Other works do not only focus on malware detection, but are more generic and attempt to identify suspicious data flows via taint analysis. Two relevant works in this area are FlowDroid [10] and DroidSafe [17].

Another important trend of works attempts to perform malware classification by using machine learning techniques. Some of the early works include Drebin [9] and DroidAPIMiner [7], which both extract several features from Android applications (e.g., requested permissions, invoked framework APIs) and then apply machine learning techniques to perform classification. A different system is AppContext [32], which uses machine learning techniques to identify malware by using the “context” of each behavior as a feature. More recently, Mariconti et al. proposed MaMaDroid [23], a tool that uses Hidden Markov Model chains and, once again, starts from the API function calls to build behavioral models. Another recent work in a similar direction is SLAP [22], which also uses machine learning with features based on API-related information, with the difference that it attempts to be more resilient to adversarial samples.

There has also been extensive research on program analysis of Android apps through dynamic analysis. Enck et al. [13] present TaintDroid, a dynamic taint analysis that performs whole-system data flow tracking through modifications to the underlying Android framework and native libraries. Other efforts, such as Mobile Sandbox [29] and Andrubis [21], developed tools and techniques to dynamically analyze unknown Android applications. Another trend

of works has proposed approaches based on dynamic analysis to perform multipath execution and dynamic symbolic execution on Java and Android applications [5, 15, 25, 27, 31]. These approaches achieve higher code coverage than simpler dynamic analysis tools.

We note that *all these existing works use API-level information as the main building block for their analysis*. Their main rationale is that API-level information provides semantics-rich data, which in many cases is enough to discern benign apps from the malicious ones. This common trait of all these recent works underline the importance that API-related, semantics-rich data can play within the Android security research community. However, as often mentioned in this paper, all these approaches can be detected and evaded [8].

One of the very few works that fully acknowledge this limitation and performs a step forward is CopperDroid [28, 30]. In this work, the authors show how it is possible to reconstruct two categories of high-level behaviors. The first one consists in those implemented through Android Services, which CopperDroid identifies by unmarshaling objects used in Binder transactions (e.g., access to geolocalisation). The second includes those behaviors that result in a sequence of syscalls with a clear data dependency, which CopperDroid reconstructs by means of a value-based data flow analysis technique (e.g., opening a file and performing operations on it).

However, the authors of CopperDroid also note that API-level information, while useful in reconstructing high-level behaviors in some cases, is not fully trustworthy when dealing with some complex scenarios. Take, as an example, the behavior of the *createSocket* method of the *SSLSocketFactory* Java class. This API creates an SSL tunnel over an already opened TCP socket. Exploring our dataset we noticed that to perform this task the framework invokes various syscalls, for example, “getrandom” to generate the nonce used for the encryption or “read/write” to perform the handshake. By simply inferring the data dependency between syscalls, CopperDroid would be able to recognize its network-related aspect, but it would fail in understanding that the syscall pattern is actually implementing a tunnelling mechanism that, according to the API documentation [3], enables to instantiate an SSL connection over a proxy.

Our work thus differs from CopperDroid by exploring the behavior reconstruction problem in a more generic way: given a list of syscalls, is it possible to build a pattern identification and “go back” from syscall to API in the general case? In other words, in this paper we are interested in answering a more generic question, and we do not rely on specific patterns or on data-dependency among syscalls to address the mapping problem. The main difference with previous works is that we focused on using a data-driven approach to explore the more-generic problem of performing semantics reconstruction of a generic sequence of syscalls. In the process,

we have also built the first dataset of API-syscall relationship — to the best of our knowledge, the first of its kind.

The problem of reconstructing high-level behaviors from low-level features is not exclusive to the Android security research field. Martignoni et al. [24], for example, modeled a set of malicious behaviors found in different malware families for the Windows operating system. To this aim the authors manually analyzed the executions traces of malware and benign programs to express high-level behaviors in terms of lower-level syscall-like events.

The main drawback of their approach is that the modeling phase cannot be automated because it requires human understanding of each high-level behavior. In our work, instead, the modeling phase is completely automated. Moreover, our approach is more generic since it models more than just a restricted set of behaviors and is not bound to malicious behaviors only.

10 Conclusion

Dynamic analysis has proven to be fundamental in the field of program analysis, especially when it provides high-level and human-friendly information about a program's behavior, like in the case of API traces. As discussed, a vast number of research works rely on these semantics-rich API traces. Unfortunately, all current instrumentation frameworks can be detected and evaded, making these techniques not suitable to be used in an adversarial context, like malware analysis.

This paper provides the first systematic exploration on the challenges of automatically reconstructing API traces semantics from low-level syscall traces. Ideally, a system that can complete this task in an accurate manner would provide all the advantages of API semantic analysis, while remaining undetectable by malicious programs.

The evaluation of this work shows that this task is challenging — arguably much more challenging than what previously thought. While previous works focus on recognizing specific patterns, we adopt a generic data-driven approach and we collected and analyzed data on several millions API and syscall invocations. As one of the core contributions, we built a new dataset and an analysis pipeline, which we publicly release to encourage future work in this important area.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. We are also grateful to our shepherd Lorenzo Cavallaro for his suggestions, and Andrea Possemato for helping with some experiments. There are several others we are also in debt with: Carlo, Harvey, Ivan, Luca, Lorenzo, and, it goes without saying, Betty Sebright. Thanks to all of you.

This material is based upon work supported by the NSF under Award number CNS-1849803. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

- [1] F-droid — free and open source software applications for the android platform. <https://f-droid.org/>.
- [2] Frida analysis framework. <https://www.frida.re>.
- [3] Java documentation for `javax.net.ssl.SSLSocketFactory.createSocket`. <https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLSocketFactory.html#createSocket-java.net.Socket-java.lang.String-int-boolean->. Accessed: 2019-06-27.
- [4] Java Parser and Abstract Syntax Tree for Java. <https://github.com/javaparser/javaparser>.
- [5] JPF-symbc: Symbolic PathFinder. <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc>.
- [6] Xposed installer (framework). <http://repo.xposed.info/module/de.robv.android.xposed.installer>, 2017.
- [7] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Proceedings of the Security and Privacy in Communication Networks (SecureComm)*, 2013.
- [8] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupe, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium*, pages 1–15, 2016.
- [9] Daniel Arp, Michael Spreitzenbarth, Hubner Malte, Hugo Gascon, and Konrad Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.
- [10] 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 *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2014.

- [11] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8. ACM, 2016.
- [12] Frank Ch Eigler and Red Hat. Problem solving with systemtap. In *Proceedings of the Ottawa Linux Symposium*, volume 2006, 2006.
- [13] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2010.
- [14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-Based Detection of Android Malware Through Static Analysis. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2014.
- [15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM Sigplan Notices*. ACM, 2005.
- [16] Google. UI/Application Exerciser Monkey | Android Developers. <http://developer.android.com/tools/help/monkey.html>.
- [17] Michael Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.
- [18] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.
- [19] Roei Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128. ACM, 2015.
- [20] Tobias Holl, Philipp Klocke, Fabian Franzen, and Julian Kirsch. Kernel-assisted debugging of linux applications. In *2nd Reversing and Offensive-oriented Trends Symposium 2018 (ROOTS)*, November 2018.
- [21] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 3–17. IEEE, 2014.
- [22] Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Using Loops For Malware Classification Resilient to Feature-unaware Perturbations. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.
- [23] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*, 2016.
- [24] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C Mitchell. A layered architecture for detecting malicious behaviors. In *International Workshop on Recent Advances in Intrusion Detection*, pages 78–97. Springer, 2008.
- [25] Nariman Mirzaei, Sam Malek, Corina S. Pasreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android Apps Through Symbolic Execution. In *ACM SIGSOFT Software Engineering Notes*, 2012.
- [26] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.
- [27] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting Runtime Values in Android Applications that Feature Anti-Analysis Techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.
- [28] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.
- [29] Michael Spreitzenbarth, Felix Freiling, Florian Echter, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2013.
- [30] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

- [31] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.
- [32] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *Proceedings of the International International Conference on Software Engineering (ICSE)*, 2015.
- [33] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. Sandprint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 165–187. Springer, 2016.
- [34] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

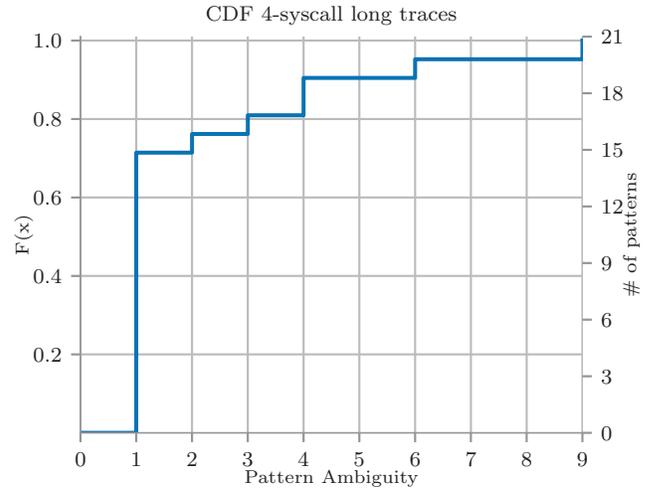


Figure 7: Pattern Ambiguity for 4-syscall long patterns

11 Appendix

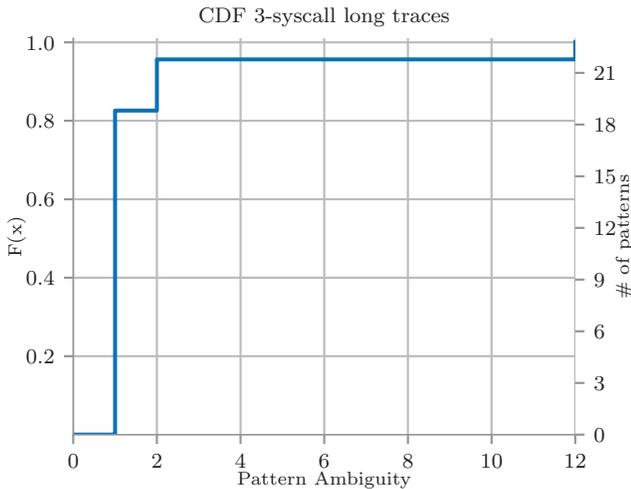


Figure 6: Pattern Ambiguity for 3-syscall long patterns

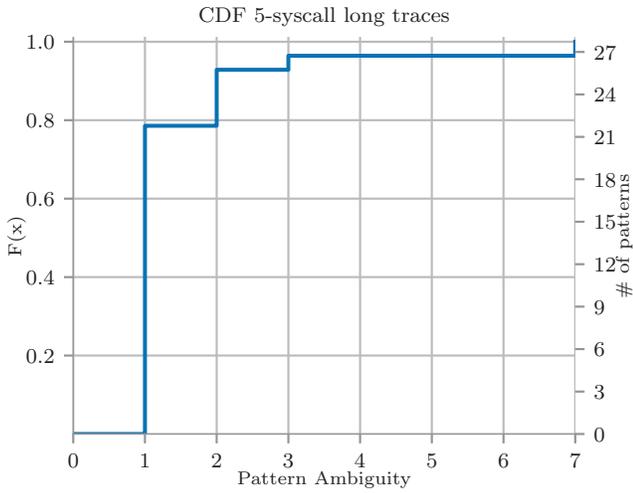


Figure 8: Pattern Ambiguity for 5-syscall long patterns

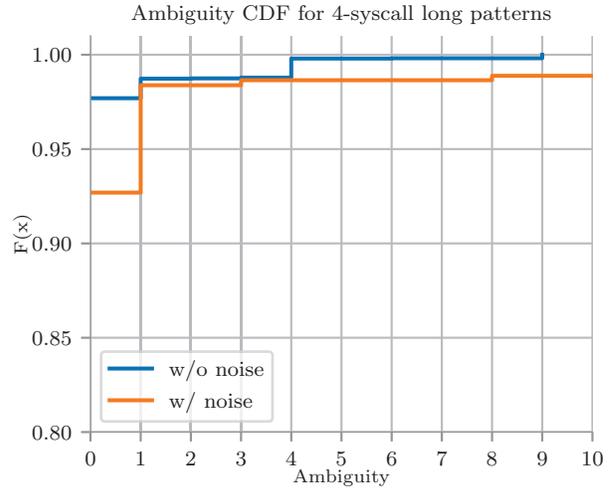


Figure 10: Total Pattern Ambiguity Comparison for 4-syscall long patterns

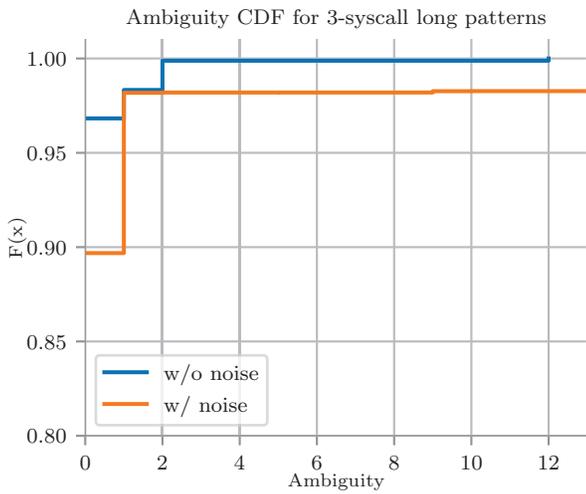


Figure 9: Total Pattern Ambiguity Comparison for 3-syscall long patterns

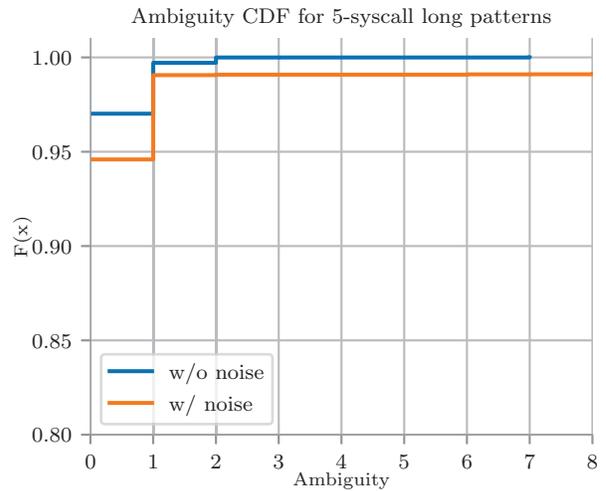


Figure 11: Total Pattern Ambiguity Comparison for 5-syscall long patterns

Towards Large-Scale Hunting for Android Negative-Day Malware

Lun-Pin Yuan
Penn State University
lunpin@psu.edu

Wenjun Hu
Palo Alto Networks Inc.
whu@paloaltonetworks.com

Ting Yu
Qatar Computing Research Institute
tyu@hbku.edu.qa

Peng Liu
Penn State University
pliu@ist.psu.edu

Sencun Zhu
Penn State University
szhu@cse.psu.edu

Abstract

Android malware writers often utilize online malware scanners to check how well their malware can evade detection, and indeed we can find malware scan reports that were generated before the major outbreaks of such malware. If we could identify in-development malware before malware deployment, we would have developed effective defense mechanisms to prevent malware from causing devastating consequences. To this end, we propose Lshand to discover undiscovered malware before day zero, which we refer to as *negative-day* malware. The challenge includes scalability and the fact that malware writers would apply detection evasion techniques and submission anonymization techniques. Our approach is based on the observation that malware development is a continuous process and thus malware variants inevitably will share certain characteristics throughout its development process. Accordingly, Lshand clusters scan reports based on selective features and then performs further analysis on those seemingly benign apps that share similarity with malware variants. We implemented and evaluated Lshand with submissions to VirusTotal. Our results show that Lshand is capable of hunting down undiscovered malware in a large scale, and our manual analysis and a third-party scanner have confirmed our negative-day malware findings to be malware or grayware.

1 Introduction

Imagine, if you were an Android malware writer developing a zero-day malware, what would you do to check how well your malware can evade malware detection? One very convenient and convincing means is to anonymously submit the malware to online malware scanners and check the scan reports. In fact, it has been reported that some in-development malware were found on VirusTotal before their major outbreaks. For example, the very first known LeakerLocker sample could date back to November 2016 when it was submitted to VirusTotal [38], but not until July 2017 did security experts find it widespread. In addition, not just one malware sample but a

trace of evolving malware samples did malware writers leave on online scanners during the *continuous submit-and-revise process*. However, no signature was available on *day zero* (the time that such a threat is known to public) because the development case was not identified even though the malware was given malware labels at its early stage. Belated signatures could cause devastating consequences [33], and unfortunately it usually takes more than six months to generate well-crafted signatures [6]. If the ongoing development of malware had been identified and studied earlier, its breakout would not be as devastating as it had become. Therefore, our goal is to find Android malware before day zero, which we refer to as *Android Negative-Day Malware*.

We propose *Lshand* (abbreviated from *Large Scale Hunting for Android Negative-Days*) to discover potential negative-day Android malware development cases (Neg-Day cases). Lshand faces the following challenges. First, with regard to binary, there is no obvious indication of malware development or kin relations among Neg-Day malware samples when detection-evasive techniques (e.g., dynamic-loading and obfuscation techniques such as [9, 14, 16, 19, 30, 41, 43, 44]) were applied to malware samples. Second, with regard to submitters, anonymization techniques may prevent us from inferring or linking the identities of submitters. We have observed many cases where a careful malware writer may submit malware samples from different anonymous identities (e.g., submitting samples from different sockpuppet accounts or with no account via free proxies or tor-network) and sign different malware samples with different keys. Third, with regard to scalability, online scanners constantly receive an enormous number of submissions (e.g., during January 2016, VirusTotal received 1.34M Android-related brand-new submissions that were never seen before). It is not practical to perform detailed analysis on each malware sample every time we observe a new one.

Lshand is designed carefully to overcome the above challenges. The key observation is that malware development is a continuous process, and malware variants will inevitably share certain characteristics throughout the development process.

Accordingly, our *first principle* is to process malware submissions based on similarity. However, as nowadays malware writers can easily leverage existing automated obfuscation tools, our *second principle* is to select features that are not likely to be obfuscated (e.g., set of permissions, contacted hosts, numbers of components). Finally, to be a practical solution, our third principle is that Lshand must be efficient, and is capable of processing a large number of submissions within a reasonable time. Lshand, in a nutshell, consists of a *Data digester*, a *Report clusterer*, an *AMDT* (Android Malware Development Trace) *extractor*, and a *Neg-Day alerter* (Fig. 1). To be brief, Lshand clusters malware reports (structural text format), and only if necessary it classifies malware samples (in binary) based on similarity and maliciousness.

We implemented and evaluated Lshand upon a snapshot of submission-stream towards VirusTotal during January 2016, which contains 1.3 million Android-related first-seen submissions from 3,852 different submitter identities (SIDs). Lshand, ran by a single thread on a desktop, took only an hour before giving us 10 Neg-Day cases. These 10 Neg-Day cases include 48 malware samples that were given no malware labels by any of the 62 scanner engines on VirusTotal by January 2016. In the end, according to submission rescans, our manual analysis, and scan results from Palo Alto Networks WildFire [29], we have confirmed that 100% of these 48 Neg-Day malware are actually malware. We also deployed Lshand over a more recent dataset dumped during May 2018, and Lshand hunted down 15 Neg-Day cases that include malware samples with zero malware labels. We have manually confirmed that 80% of these 15 Neg-Day cases are actually malware or grayware.

With regard to finding Android malware at their development stage, AMDHunter [17] is a strongly related work. However, unlike AMDHunter, which relies on the strong assumption that variants of the same malware development are submitted from the same SID, Lshand focuses on finding Neg-Day cases that were knowingly submitted from different accounts (e.g., sockpuppet accounts or no account), from different IPs (e.g., via proxies or via tor-network), or from decorated SIDs (e.g., knowingly submit benign apps in hope of confusing AMDHunter). Section 7 includes more details about the differences between Lshand, AMDHunter, and other related work. We make the following contributions.

- To the best of our knowledge, Lshand is the first malware hunting system that is capable of hunting down Android Neg-Day malware from multiple anonymous submitter identities through the analysis of submission records available on online scanners.
- We designed and implemented Lshand to overcome the following three challenges: lack of malware relations with regard to binaries, lack of development evidence with regard to identities, and scalability.
- We evaluated Lshand with two datasets. Lshand hunted

down 10 Neg-Day cases from the submission records of VirusTotal during Jan. 2016 and 15 Neg-Day cases from records during May 2018. Our results show that Lshand is efficient and accurate.

2 Background and Motivating Example

Android malware detection has been heavily studied in recent years (e.g., [2, 3, 7, 10, 15, 20, 21, 23, 24, 27, 36, 37]). Although previous work has established significant detection capability, there are limitations when facing new challenges. Once a novel detection method is published, malware writers can knowingly evade the detection logic by leveraging, for example, code-loading (e.g., [30, 43]) and sandbox detection (e.g., [26]). Furthermore, malware writers can use malware-development tools (e.g., [19]) to embed detection-evasive modules. Wei et al. [41] and Suarez-Tangil et al. [35] detailed the trend of malware evolution and evasive techniques. Other related work regarding malware development includes [9, 14, 16, 40, 44].

We consider LeakerLocker [31] as a motivating example. Only after it sleeps for a while and checks non-sandbox artifacts, it loads dynamic code. Hence, its first VirusTotal report [38] only had one malware label from one scan engine. Fig. 2 illustrates an example of what information about LeakerLocker one can obtain from a VirusTotal scan report. The information in Fig. 2 is open to public except the submitter identity, which is only available to VirusTotal premium users.

While researchers use online malware scanners as partial ground truth (e.g., [34, 41]), malware writers also use them as convenient and convincing oracles to test malware variants and evasive techniques (e.g., [19]). Via public-API or web interface, a malware writer can submit up to four malware samples per minute to VirusTotal (exceeding this rate will cause empty JSON or reCAPTCHA depends on channel). It has been reported that a set of in-development malware were found on VirusTotal; for example, the very first LeakerLocker sample dated back to November 2016 [38], but its outbreak did not happen until August 2017.

In practice, Android malware writers often repackage a popular app or a handy utility app in order to lure innocent users to install or distribute the malware. In LeakerLocker's example, some variants impersonate Call Recorder, and others impersonate WallPapers HD Blur. Consequently, each group demonstrates a set of shared characteristics (e.g., Call Recorder variants share 16 permissions, 13 activities, and 3 services, and Wall Papers HD Blur variants share 18 permissions, 8 activities, and 3 services). Based on this observation, Lshand hunts malware development traces by examining shared characteristics.

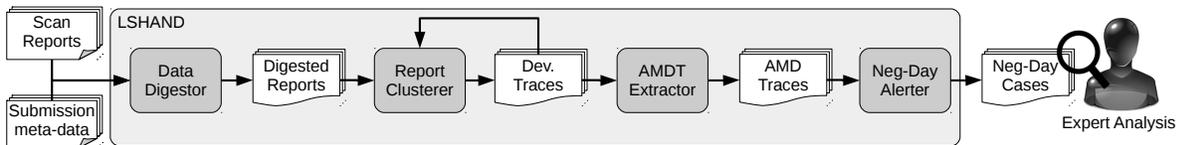


Figure 1: LSHAND Workflow Overview

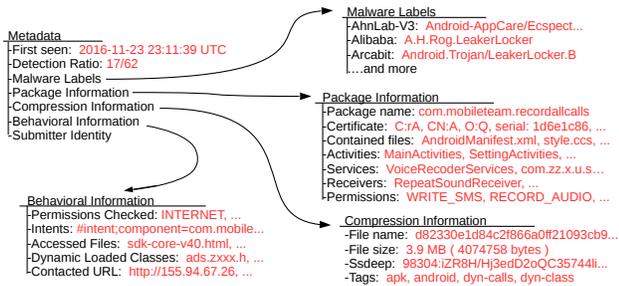


Figure 2: A Digested Report for a LeakerLocker variant

3 Problem Statement

We are specifically interested in answering the following research question: how can we find potential Android Neg-Day malware in a large scale from online malware scanners? The challenges include the followings. First, with regard to malware binaries, there is no obvious indication of malware development or kin relations among or between Neg-Day malware samples. Second, with regard to submitters, there may not be evidence such as submitter identity that can help us with inferring or linking malware writers. Third, it is not realistic to perform detail analysis on every newly received malware samples submitted to an online scanner.

3.1 Attack Model

Our attack model includes a careful malware writer who does not want to leave obvious traces but does need online malware scanners for up-to-date analysis of a newly developed malware. We assume that a careful writer has two objectives *detection evasion* and *submission anonymization*, and we assume that the malware writer will keep continuing the development process in a *submit-and-revise* fashion.

Regarding detection evasion, we assume that a malware writer would apply techniques such as package name obfuscation, class name obfuscation, method and variable name obfuscation, inter-component interaction injection, dataflow-analysis evasive code injection, native code and bytecode injection, dummy and benign methods injection, and dropper payload loading. However, trying to be less suspicious just

like a regular app, evasion techniques such as *incrementally* adding *large number* of dummy Activities, dummy Services, and irrelevant files on the malware are not considered (in other words, numbers of activities, services, and files stay similar).

Regarding submission anonymization, we assume that a malware writer would submit malware samples anonymously from different free accounts (or no account) via different free proxies (or tor-network) throughout the continuous development process. In addition, a malware writer can submit many irrelevant apps in hope of decorating his profile as if the submitter was benign. However, trying to keep a low profile, we assume that a malware writer will not submit more than on-average 100 newly developed samples per-day, per-account, and per-proxy. We will justify in Section 5 that this assumption is realistic by submission statistics.

3.2 Definitions

Since Neg-Day hunting is a relatively new research topic, we would like to define terms to avoid confusion.

Digested Report: The outcome of preprocessing a scan report from online malware scanner is a digested report, and it includes only necessary fields, including submission timestamps, submitter identity, malware labels, package information, compression information, and behavioral information (malware sample binary is not included). Fig. 2 shows an digested report example.

Submitter Identity (SID): An SID is a unique identifier that specifies a submitting individual or a submitting organization; however, an individual or organization may correspond to one or more SIDs. SID on VirusTotal is a hash value calculated based on account profile, or based on IP-port profile if no account is presented. In most online malware scanners, SID information is available only to premium users.

Malware Label: In a malware scan report, each scan engine may list multiple informative labels, including malware family (e.g., leakerlocker, svpeng, slocker), malicious functionality (e.g., ransom, trojan, dropper), suspiciousness (e.g., dangerous, high confidence malware, potentially unsafe), non-malicious characteristics (e.g., adware, riskware, potentially unwanted), and some other information (e.g., downloader, obfuscated, xor-crypt).

Development Trace (DT): A DT implies a development of an app. A DT is represented by a chronological sequence of digested reports that share certain features. A DT may

include digested reports that are sourced from different SIDs. We say two samples are “*kins*” to each other if they were put into the same DT. Unlike variants of a malware family (e.g., LeakerLocker), if there are two different fake apps (e.g., Call Recorder and Wallpaper HD Blur), we consider them as two different DTs.

Android Malware DT (AMDT): An AMDT implies a development of an Android malware. Essentially, an AMDT is a DT; however, unlike DTs, an AMDT includes at least one *malware indicator*, a digested report that has a sufficient number of malware labels. Based on needs, one can define different sufficiency metrics; in this paper, we use detection ratio r (i.e., the ratio of engines that reported malware labels).

Negative-Day AMDT case (Neg-Day case): A Neg-Day case implies a potential development case of a Neg-Day malware. Essentially, a Neg-Day case is an AMDT; however, unlike AMDTs, a Neg-Day case includes at least one recent submission that has no malware labels (i.e., $r = 0$). Since this seemingly benign sample has evaded detection from all scanning engines, terrible harm could be done once it is distributed. Hence, detecting such a Neg-Day case with benign samples is our goal.

4 LSHAND Design

4.1 Design Overview

Lshand’s design is based on our key observation that malware writers tend to test their in-development malware through online malware scanners. We assume that a malware development is a continuous process; hence, malware writers inevitably will leave development traces online, and malware variation will demonstrate certain shared or similar characteristics. However, finding AMDT is challenging because malware writers could always apply evasion and anonymization techniques. Hence, our design follows the following principles: first, we process malware submission based on similarity; second, we select features that are not likely to be obfuscated. For scalability reason, our third principle is that we analyze scan reports (specifically, digested reports), and only if necessary we analyze malware binaries.

Lshand, in a nutshell, consists of a *Data digester*, a *Report clusterer*, an *AMDT extractor*, and *Neg-Day alerter* (Fig. 1). Lshand’s workflow is as follows. Lshand’s Data digester looks for Android-related first-seen submissions, and then produces digested reports. For each digested report, Lshand’s Report clusterer deduces its DTs. From DTs, Lshand’s AMDT extractor extracts AMDTs. From AMDTs, Lshand’s Neg-Day alerter verifies potential Neg-Day cases and raises alarms. In our design, report clusterer is the module that identifies *kin* relation among malware samples, and Neg-Day alerter is the module that identifies the development evidence.

4.2 Data Digester

Data digester finds Android-related first-seen submissions and generates corresponding digested reports. By first-seen submissions, we mean those submissions whose samples were never processed by the selected online malware scanner. We only focus on first-seen submissions, because re-submissions cannot imply malware development.

Data digester learns whether or not a submission is Android-related by checking malware labels (if there is any “android” or android-related label), package information (if there is any Android application components), and compression information (if there is any “apk”, “dex”, or “android” tag). Data digester learns whether or not a submission is first-seen by checking “last-seen” or “last-scan” timestamps in the submission metadata (both should be slightly greater or equal to the submission timestamps), or by checking the existence of previous scan reports (reports with the same file hash should not exist). Upon getting an Android-related first-seen submission, Data digester parses the output information and produces a digested report (as shown in Fig. 2).

4.3 Report Clusterer

For each digested report, Report clusterer finds the corresponding DTs by clustering digested reports along with existing DTs. We say two samples are *kins* if two samples belong to the same DT. This approach is based on our observation that kin malware variants often bear some similarity or characteristics even though some contents could be obfuscated. Here we detail our design choices, including feature extraction and clustering scheme.

4.3.1 Feature Extraction

We cannot directly use the informative knowledge in the digested reports, because malware writers may have applied obfuscation. Rather, Lshand extracts features that are less likely to be obfuscated or manipulated. We split our features into the following categories: submission timestamps, package information, compression information, and behavioral information. Note that Lshand does not take malware labels as part of the features, because Neg-Day submissions may receive no labels whatsoever.

Submission Timestamps: We consider malware development a continuous submit-and-revise process, hence the chronological gap between two kin submissions will not be large. If two similar submissions are faraway apart in time, then they are less likely to be developed by the same malware writer, as it is more likely to be a new development case of different malware writers who are trying to gain benefits from a previously known malware. Submission timestamps are issued by online malware scanner, so malware writers cannot manipulate them (note that file and certificate timestamps can be manipulated to such as 1980-00-00 and 2107-15-19).

Package Information: Lshand examines structural information and permission information of a package. For structural information, although class names and file names can be obfuscated, we can still count the numbers of Activities, Services, and other files by types (we consider 26 different file types, including mp3, png, and dex). Hence, Lshand has $1 + 1 + 26 = 28$ numerical features. Lshand will not be affected by name obfuscation because names are not used as features. As for permissions, besides predefined permissions in Android framework, developers can define app-specific or device-specific permissions. Upon 399.9K digested reports, we split 22K known permissions into 56 categories (e.g., network, storage, camera, c2dm). For each category, Lshand counts the number of related permissions listed in the digested reports. In summary, Lshand has $28 + 56 = 84$ package-related features in total.

Compression Information: Lshand examines file tags and uncompressed file size of a submitted sample. Most online malware scanners provide tags for file types (e.g., “apk”, “jar”, “android”) and tags for characteristics (e.g., “dyn-class”, “via-tor”, “check-gps”). Lshand considers 21 tags as features, where each feature is represented by either true or false whether or not the tag appears in the digested report. As for uncompressed file size, Lshand treats it as one feature (hence $21 + 1$ features in total). Uncompressed file size is essential because simply using package-related features and tags is not enough to accurately deduce DTs. Our experiments showed that, without uncompressed file size, apps with similar structural information and similar file tags will be wrongly put together. With uncompressed file size, in contrast, we have had better clustering results. Since uncompressed size does not have impact on malware scanners’ judgement, it is often not malware writers’ concern to obfuscate. Therefore, uncompressed size often stays similar from one malware kin to another malware during a development.

Behavioral Information: Contacted URLs, accessed files, sent SMS, and inter-process communication could be available in the digested reports if the selected online malware scanner(s) provides sandbox analysis. While most strings can be obfuscated in the package, certain strings cannot be obfuscated at runtime. For example, contacted hosts (domain and IPs) and inter-app intents (actions and components) cannot be obfuscated (but parameters and contents can still be obfuscated). Hence, Lshand extracts contacted hostnames and IP from HTTP(s) communication and intended actions and components from intent communication. There are many ways to represent strings in the feature space. Although it is an option to apply complex string comparison methods which are accurate but slow, Lshand simply counts alphanumeric (i.e., a-z and 0-9) n -grams, where $n = 1$ in our current implementation; for example, “google.com” has numeric feature “3” for the character “o”. Lshand has 36 features for contacted hosts and for inter-app intents; therefore $36 + 36 = 72$ behavioral fea-

tures in total. Note that it is also optional to use $(n > 1)$ -grams for a more fine-grained feature set.

4.3.2 Clustering Scheme

In order to produce accurate results, we carefully studied clustering models, weighted features, and distance thresholds. However, note that since our ultimate goal is to hunt down Neg-Day cases, we tune Report clusterer with only AMDTs rather than DTs in general; whether or not a benign DT is accurate is not a major concern.

Clustering Model: We consider malware development a continuous process, so AMDTs will demonstrate progressive evolution of a malware. As such, Lshand leverages incremental density-based clustering model. Under this clustering model, clusters are formed by merging observations (feature-based representation of digested reports), and the linkage distance between two clusters is defined by the euclidean distance between the two nearest observations (single-linkage). If the linkage distance between two clusters is smaller than a given quality threshold τ , then the two clusters will be merged into one. We do not consider the other linkage approaches (e.g., complete linkage and average linkage) because they are not suitable for progressive evolution (where a malware variant demonstrates similarity with a few previous variants rather than with the entire development), and we do not consider k -clustering model because we do not know how many DTs are out there. There are a few density-based clustering models that are applicable to our research problem, but for scalability reason we implemented an incremental density-based clustering model, whose time complexity is $O(n \times (m + n))$, where n is the number of new observations, and m is the number of old observations (hence $O(n^2)$ from scratch or $O(m)$ for an incremental step). Discussion on optimizing clustering algorithm for malware analysis can be found in [4, 8, 32].

Weighted Features: We assign different weights to features because different features could vary in different scale throughout its development. For example, features in package information (e.g., number of Activities and permissions) shall be very similar if not the same, whereas features in submission timestamps and features in compression information (e.g., uncompressed size and tags) can be different every time. The resulting DTs will be problematic if Lshand treats these features equally. Hence, Lshand puts less weight on features in timestamps and in compression information, but more weight on features in package information (Table 1). Another factor of assigning weights to features is how thorough the analysis can be provided by the selected online scanners (for example, we put light weights on behavioral features because VirusTotal does not always apply behavioral analysis to Android submissions). To determine weights, we collected 50 confirmed AMDTs and conducted an empirical study. We repeatedly tuned weights and ran experiments until we reached a state

Table 1: Weighted Features

Category	Feature	#	Weight
Timestamps	submission timestamps	1	medium
Package	# of Activities	1	heavy
Package	# of Services	1	heavy
Package	# of file by types	26	medium
Package	# of Permissions by categories	56	heavy
Compression	file tags and labels	21	medium
Compression	uncompressed size	1	medium
Behavioral	contacted URLs	36	light
Behavioral	inter-app communication	36	light

where AMDTs were formed only by true kin observations and as many observations in each AMDT as possible.

Clustering Threshold: Quality threshold τ is an important parameter in density-based clustering models (some may use the term density distance ϵ). If τ is too large, a cluster would wrongly include outsider observations that otherwise should not be included, and if τ is too small, a cluster would not include kin observation that otherwise should be included (or equivalently, split one AMDT into smaller AMDTs). In this research, we consider having outsider observations much more destructive than having one AMDT split into smaller AMDTs. The reason is that our ultimate goal is to provide malware analysts with Neg-Day cases for comprehensive analysis so that they can craft signatures that can detect undiscovered malware; however, contrary to our goal, outsider observations could cost malware analysts more time to conduct inaccurate analysis. To determine τ , we conducted an empirical study upon 50 confirmed AMDTs. We first set a large τ and then tuned τ down until we reached a state where we have no outsider observations.

4.4 AMDT Extractor

Lshand extracts AMDTs from DTs. However, AMDT extractor is not just a noise filter. Besides extracting AMDTs, it also updates digested reports in seemingly benign DTs, and it also removes digested reports that are no longer useful in AMDTs.

Extract AMDTs from DTs: An AMDT essentially is a DT with at least one *malware indicator*, a digested report that has sufficient number of malware labels. Depending on label availability from the selected online malware scanner, one may opt to use different metrics for determining malware indicators. In our implementation, Lshand simply checks detection ratio r (e.g., 30 out of 62 engines) rather than analyze label semantics (e.g., differentiate “malware” and “riskware”). Checking detection ratio r has an advantage that it does not require prior knowledge of existing labels, hence it is generic to engine-specific labels and flexible to newly-defined labels. It is a common practice to consider a submission suspicious when there are many engines reported it with malicious or suspicious labels regardless of label semantics (note that the

LeakerLocker example [38] was given only suspicious labels and no malicious labels).

Update digested reports in benign DTs: Previously seemingly benign digested reports could become malware indicators once the scan engines are patched, and once a digested report becomes a malware indicator, a DT becomes an AMDT. Hence, for each seemingly benign DT, Lshand needs to submit rescan requests for those seemingly benign samples that do not have recent digested reports. Note that updating malware labels will not merge or split DTs, because Lshand does not consider malware labels as clustering features.

Remove digested reports from AMDTs: The growth in the number of digested reports is enormous, and it will not be scalable if Lshand keeps all the digested reports and process all of them repeatedly. However, Lshand cannot simply discard *ancient* digested reports. By ancient, we mean that the submission timestamp of a digested report is so far away from current timestamp that the distance between two weighted timestamp features has exceeded the quality threshold τ (in other words, ancient digested report will no longer affect clustering results). We cannot simply discard ancient digested reports because ancient digested reports could become malware indicators someday, and discarding malware indicators may cause AMDTs to become DTs. Rather, Lshand discards two kinds of digested reports: first, ancient digested reports that are prior to any malware indicator in the same AMDT (hence no AMDT will become DT); second, digested reports in an ancient DT that will not be clustered with any new digested reports.

4.5 Neg-Day Alerter

Neg-Day alerter verifies Neg-Day cases by examine *maliciousness* and *similarity*. On one hand, Lshand skips those AMDTs that are already known to be malicious, because we are particularly interested in Neg-Day cases that nobody knows they are malicious. On the other hand, Lshand looks for package-level similarity as evidence of development (even though some correlation can be derived from digested reports, certificate and binary and GUI correlation are not yet assured). In the end, if a sample from an AMDT passes both maliciousness test and similarity test (that is, not malicious but similar), Lshand will raise a Neg-Day alert to malware analysts with a Neg-Day case. Note that Neg-Day alerter only checks binary only when necessary.

Maliciousness Test: Since we are interested in Neg-Day samples that nobody knows they are malicious, only when a sample is not obviously malicious will Lshand proceed to the next step. To examine the maliciousness of a new sample in an AMDT, Lshand checks reports, signatures, and binaries: first, Lshand examines the digested report and will proceed if there are very few malware labels (e.g., $r = 0$); second, Lshand examines its certificate and will proceed if the certificate is not publicly known to be benign (white-list certificates);

third, Lshand checks its binaries by using third-party binary-level Android malware classifiers (e.g., MaMaDroid [24] and Drebin [3]), and will proceed if the results are benign (specifically, Lshand trains classifiers with publicly known malware and publicly known benign apps). If a sample passes all the above tests, Lshand will then test whether the sample is similar to its AMDT.

Similarity Test: Different from Report clusterer which compares feature-space similarity, here we compare signature and binary similarity among malware samples within an AMDT. If a new sample does not show similarity at this stage (hence it is not a new variant but an outsider who coincidentally has similar features), then Lshand needs to label it as an outsider and remove it from the AMDT. We say a new sample has passed the similarity test if *any* of the following comparisons show similarity: in certificate comparison Lshand checks whether or not the information in the signing certificate (e.g., public key and serial number) is the same as that in the other samples in the AMDT, and in binary comparison Lshand checks whether or not the control call graph or GUI-callback methods has similar characteristics to the other samples in the AMDT by using repackaging classifiers (e.g., ViewDroid [45]) and malware classifiers (e.g., MaMaDroid [24] and Drebin [3]). Specifically, for each AMDT Lshand trains the classifier with *malware indicators of the AMDT* and publicly known benign apps, and Lshand considers a sample to be similar to its AMDT if the classifier reports the sample “malicious”, which is the case that the sample is closer to the malware indicators than to the other irrelevant apps in the feature space (different from the maliciousness test where we compare binary features with known malware).

5 Accuracy Evaluation

5.1 Dataset and DT Statistics

Our evaluation was conducted upon submissions to VirusTotal. Since neither do we nor VirusTotal have a solid ground truth for recent submissions, we test Lshand upon old submission-stream captured during January 2016 (detection ratio is denoted as r_u), but then we evaluate Lshand with the re-scan results that were produced during May 2018 (detection ratio is denoted as r'_u). Even after two years, we consider these re-scan results as *partial* ground truth rather than complete ground truth because it is possible that some malicious samples are still not discovered. During January 2016, VirusTotal received 1,345K Android-related first-seen submissions from 3,852 different submitter identities (SIDs). The HTML-based reports totally used 154 GB storage space, whereas the digested reports took only 8.3 GB.

Subset Selection: We do not need to consider every single submission, as we can exclude premium users that are not anonymous. Table 2 and 3 show the statistics of SID-to-peak-rate and SID-to-monthly-rate. If we consider terminal-based

Table 2: SIDs and Peak Rate

Peak Rate	≤ 4/min	≤ 60/min	≤ 120/min	Unlimited
# of SIDs	3,364	3,750	3,788	3,852
# of subs	5,318	14,309	19,837	1,345,696

Table 3: Clustering Statistics

Monthly Rate (ρ)	≤ 3100	≤ 12400	≤ 24800	Unlimited
# of SIDs	3,837	3,841	3,843	3,852
# of subs	58,765	92,603	125,267	1,345,256
# of observ.	21,800	49,205	62,773	371,187
# of clusters	14,266	32,986	38,236	85,139
Time Used	01:01:36	4:59:58	07:51:07	291:28:12

Note that the difference between the number of submissions and the number of observations comes from the fact that our Lshand evaluation does not consider submissions that have missing information and submissions that have a small number of application components (i.e., < 5 activities and services counted together).

non-premium submissions (i.e., peak rate $\leq 4/\text{min}$), then we only need to worry about 5,318 submissions from 3,364 SIDs. Nevertheless, to test the scalability of our approach, we relax this constraint to submission peak rate under 120 submissions per minute (19,837 submissions from 3,788 SIDs), or alternatively 3,100 submissions per month (58,765 submissions from 3,837 SIDs) assuming malware writers will keep their SIDs low profile. In our dataset, the latter subset is a superset of the former subset, and hence we consider the submission-subset of SIDs that have no greater than 3,100 submission per month (denoted as $\rho \leq 3100$) a sufficiently large hunting ground. This statistics justifies our assumption in Section 3 that a malware writer will not submit more than on-average 100 new malware samples per-day, per-account, and per-proxy.

DT Statistics: Table 3 also shows the time usage from scratch (there was no pre-built cluster, and clusters were built up incrementally on a single thread). For the recommended dataset-subset of $\rho \leq 3,100$, Lshand spent an hour in clustering DTs on a single thread. Although we state that the submission-subset of $\rho \leq 3,100$ is sufficient, our approach is certainly computational feasible to higher rates (a more scalable alternative approach is discussed in Section 6). Table 4 shows the resulting DT statistics. We can see that 1,005 DTs out of 14,226 DTs were submitted from multiple SIDs.

Table 4: Development Trace Statistics

Intervals	# of DTs to [a,b) of SIDs	# of DTs to [a,b) of reports
1	13,261	12,650
[2, 10)	983	1,499
[10, 10 ²)	19	108
[10 ² , 10 ³)	3	9
MAX	194	562

5.2 AMDT Accuracy Analysis

Lshand extracted AMDTs from the recommended submission-subset of $\rho \leq 3,100$. Each AMDT has at least five digested reports and at least one malware-indicating digeseted report u that has detection ratio $r'_u \geq 30/62$ by May 2018. The reason why we did not evaluate low detection-ratio DTs is twofold: first, we are focusing on checking AMDT rather than clusters in general; second, we have better partial ground truth when r'_u is higher.

Baseline Selection: We compare Lshand AMDTs with AMDTs from Ssdeep-based clustering method SSDC [39], because SSDC is similar to our work in the way that it requires only the metadata to produce AMDTs. Ssdeep has been leveraged in several work that aims to cluster malware (e.g., SSDC and Graziano et. al [13]). These work requires only the Ssdeep values available in VirusTotal scan reports: two samples that have homologies will have common byte sequences in their Ssdeep values, and two Ssdeep values can imply how homologous the two corresponding samples are.

Oracle Setup: We built an oracle for comparing AMDTs of different approaches in binary-level. The oracle consists of Android malware classifiers MaMaDroid [24] and Drebin reimplementaion [28], and it considers a sample to be *affiliated* to its AMDT if and only if (1) the sample is similar to the other samples in the same AMDT, and (2) the sample is verified malicious by recent partial ground truth we gathered by May 2018 (unlike Neg-Day alerter which looks for benign samples using partial ground truth by Jan. 2016). For each AMDT, the oracle gives us a correlation score S_i calculated with Eq. 1. The training details is stated as follows. We carefully verified an *outsider malware set* and an *outsider benign set*, where malware set includes 362 malware ($r'_u \geq 30/62$) that each has few kins ($|kinds| \leq 2$), and benign set includes 631 apps ($r'_u = 0/62$) that each has few kins ($|kinds| \leq 2$). For similarity test upon an AMDT, we split AMDT samples into two halves based on the detection ratio r'_u , and we trained the classifiers with the high-ratio half and *outsider benign set* to verify the low-ratio half. For maliciousness test upon an AMDT, we trained the classifiers with *outsider malware set* and *outsider benign set* to verify the samples. Note that, we tried to include some other tools, but each of them failed to dissect a large portion of our dataset, either because its design was not for the latter Android app framework (e.g., ViewDroid [45]), or because it took too much time and eventually timed out (e.g., Asteroid [11]).

$$\text{Correlation score } S_i = \frac{\# \text{ of affiliated kin samples}}{\# \text{ of samples}} \quad (1)$$

$$\text{cell}_{xy} = \left[\frac{\frac{\# \text{ of samples in the correlation level } y}{\# \text{ of all samples in column } x}}{\frac{\# \text{ of AMDTs in the correlation level } y}{\# \text{ of all AMDTs in column } x}} \right] \quad (2)$$

Table 5: Accuracy of different AMDT sets

$\rho = 3,100$	SSDC AMDT	Lshand AMDT	Lshand Neg-Day
# of samples	1,532	1,698	253
# of AMDTs	68	56	10
Perfect (100%)	85.51%	99.16%	100.00%
Excellent (90-100%)	79.41%	96.43%	100.00%
Good (80-90%)	3.59%	0.00%	0.00%
Fair (70-80%)	2.94%	0.00%	0.00%
Problematic (60-70%)	2.81%	0.00%	0.00%
Bad (0-60%)	4.41%	0.00%	0.00%
	0.00%	0.00%	0.00%
	0.00%	0.48%	0.00%
	0.00%	1.79%	0.00%
	8.09%	0.36%*	0.00%
	13.24%	1.79%*	0.00%
hours per revision	168.84	207.81	16.26

* False negatives were mistakenly introduced by the oracle; we have verified this cluster a perfect cluster. Hour-per-revision is calculated based on the highest file-timestamp in an apk.

Table 6: SID Statistics of Neg-Day Cases

Neg-Days	Jan 2016	May 2018
# of samples	253	256
# of DTs	10	15
# of DTs with 1 SID	1	3
# of DTs with [2, 10) SIDs	8	5
# of DTs with [10, ∞) SIDs	1	7
Maximum # of SIDs	25	41

AMDT Results: Table 5 shows our accuracy evaluation. For presentation purpose, we split the correlation scores into six levels (i.e., perfect, excellent, good, fair, problematic, and bad), and each table cell is calculated with Eq. 2 (i.e., upper scores represent the percentage among all samples, whereas lower scores represent the percentage among all AMDTs). For example, an AMDT with five affiliated kins and three unaffiliated samples has correlation score $5/8 = 62.5\%$, and hence this AMDT is *Problematic*. This one AMDT (out of 56) with eight samples (out of 1,698) contributes to the *Problematic* cell 0.48%-1.79% in Table 5. We can see that Lshand outperforms SSDC in terms of providing accurate AMDTs: not only did Lshand provide more accurate AMDTs (99.16% vs 85.51%), but also it did provide more samples in total (1,698 vs 1,532). In addition, Lshand produced less *Bad* AMDTs. Note that our AMDT results were given by AMDT extractor before attested by Neg-Day alerter.

5.3 Hunting Negative-Day Malware

Our prey is Neg-Day cases, which are AMDTs with seemingly benign submissions. We evaluate Lshand with an old submission-stream snapshots, but we also deploy Lshand for a more recent submission-stream snapshot.

5.3.1 Submission-stream of January 2016

Lshand hunted down 10 Neg-Day cases from the submission-subset where $\rho \leq 3,100$ subs/month. Each Neg-Day case includes at least one seemingly benign digested report u that has detection ratio $r_u = 0/62$, at least one malware-indicating digested report v that has $r_v \geq 3/62$, and at least five samples in total. Table 6 shows that nine (out of 10) Neg-Day cases were submitted from multiple SIDs. Among these 10 Neg-Day cases, Lshand identified 49 potential Neg-Day Android malware ($r_u = 0/62$ by Jan 2016). In addition, 96 (out of 253) samples have the tag "xorcrypt" (sample is xor-encrypted), and 55 samples have obfuscated class name.

Table 5 shows our Neg-Day accuracy evaluation. A score of 100% is attested by the same oracle that was trained by the selective outsider datasets by May 2018. To be more convincing, we also manually verified these Neg-Day cases. We submitted rescan requests to VirusTotal, studied the rescan reports, and verified potential AMDT evidences (e.g., malware labels, Ssdeep values, certificates, permissions, components, and names if not obfuscated). Our manual verification aligns with our oracle's evaluation: every Neg-Day cases are indeed homologous and malicious. All 48 potential Neg-Day malware ($r_u = 0/62$ by Jan. 2016) have become malicious ($r'_u > 0/62$ by May 2018). The average detection ratio for these 253 submissions was $\bar{r}_u = 4.31/62$ by Jan. 2016 and has become $\bar{r}'_u = 32.06/62$ by May 2018. Malware labels for these 10 Neg-Day cases include Dnotua, Dowgin, Ewind, Huer, Jiagu, Rootnik, SmsPay, SmsReg, and Triada. Based on this evaluation, we conclude that Lshand is capable of hunting down Neg-Day Android malware. Yet, note that since we do not have ground truth for false negatives, we think that there still could be some undiscovered Neg-Day cases in DTs.

5.3.2 Submission-stream of May 2018

We also deployed Lshand for more recent data which was captured during May 2018. We fed Lshand's report clusterer the same features, weights, and thresholds; however, we fed Lshand's AMDT extractor a different set of parameters in order to narrow down possible Neg-Day cases due to the limited computational and personnel resource we have.

During May 2018, Lshand hunted down 15 Neg-Day cases with totally 256 samples (114 samples have $r'_u = 0/62$ by May 2018). Table 6 shows that 12 out of 15 Neg-Day cases were submitted from multiple SIDs. We asked the same oracle for automatic judgment and got 96.77% similarity score but 31.25% maliciousness score. Without recent data in the training set, the evaluation oracle failed to judge maliciousness due to concept drift [1, 18]. Hence, we manually analyzed the latest sample in each Neg-Day cases that were reported benign ($r'_u = 0/62$).

In our manual analysis, we submitted 15 samples to Palo Alto Networks WildFire [29] for behavior scanning, and we checked samples with JEB decompiler (it is possible that we

missed the malicious part in some cases). By the time we wrote this paper, we have confirmed that 12 out of 15 Neg-Day cases (80%) are malware, but VirusTotal and 62 engines failed to identify these variants. Their malware labels include SLocker, Triada, Trojan, riskware, downloader, and potentially unwanted. One may argue some listed labels are not truly malicious, but please recall that the maliciousness of a Neg-Day malware is often underestimated (e.g., LeakerLocker had only one suspicious label [38]), and it is not Lshand's job but malware analysts' job to tell its true maliciousness.

5.3.3 False-Positive Case Study

Lshand relies on the assumptions that malware development is a continuous submit-and-revise process. However, it is not always true. In the results of May 2018, we got two (out of 15 Neg-Day cases) inevitable false positive cases that violate our assumptions, and both cases are apps (54 apps in total) created by Appbyme [5], an online DIY app-creation platform for non-programmer users with a variety of templates and resources. We consider them false positive cases because their latest kins are not actually malicious. These two cases show that Lshand will be inaccurate in judging apps from such a platform.

The reason that Lshand reported Appbyme is threefold: first, the after-creation apps were structurally similar (they share the same set of 81 activities, 9 services, 28 permissions, and they all contains more than 1700 files) because of similar selections on templates and resources even though these apps were from different non-programmer users; second, one of their malware-indicating kin is malicious (labels include Adware, Malware-HighConfidence, PUA, and TencentProtect, an anti-cheat system monitor); third, the latest kin seems not malicious. For confirmation, we submitted these malware indicators to Palo Alto Networks WildFire, and the results are that the malware indicators are indeed malicious, but the latest benign apps are indeed benign. In Lshand's point of view, the latest benign Appbyme apps do look like variants of the malicious Appbyme apps. After all, these apps are developed by the same proxy developer using the same set of templates and resources. Since the development was not a submit-and-revise process, it becomes Lshand's limitation.

6 Limitations and Discussions

6.1 What are the limitations?

Our goal is to hunt down potential Neg-Day cases based on a few assumptions (Section 3), and the limitations are the following cases that violate our assumptions. First, we assume malware development is a continuous submit-and-revise process, and hence Lshand cannot accurately judge an app development if the development case involves no submission for revision (thus no scan reports) or if the development is done on a proxy app-creation platform (see the above false-positive case

study). Second, we assume malware writers never incrementally add a large number of dummy components or dummy files, and hence Lshand cannot accurately judge a malware once adding dummy components become automated (see the below discussion). In addition to the above limitations, there are two requirements: first, the scan reports from the selected online malware scanner must be comprehensive; second, feature weights and distance thresholds must be studied because information availability may differ across different selected online malware scanners.

6.2 Can we learn weight and threshold?

In our current implementation, we tune the weights and thresholds based on our confirmed AMDTs. Nevertheless, this process can be robust and automatic by applying machine learning techniques (e.g., [25, 42]). Specifically, we can learn weights and thresholds from *unobfuscated same-certificate AMDTs*, which can be derived by collecting malware samples that have high detection ratio ($r \geq 30$), overlapping components (> 5), and the same certificate. However, we did not incorporate the automation of weights and thresholds into our current implementation because we do not have an AMDT dataset that is large and convincing enough to learn from, as we only have two month-long submission snapshots. If we focus on *unobfuscated same-certificate AMDTs* that have at least five kin samples and were compiled within seven days from submission (based on the most recent file-timestamp), then we have 24 AMDTs from January 2016 based on the rescans submitted in 2018 (on average 21.99 hours per revision), and 11 AMDTs from May 2018 (7.87 hr/revision). During this research, we also have discovered 16 *unobfuscated multi-certificate AMDTs*, that each has at least five kin samples but no more two kin samples sharing the same certificate, compiled in January 2016 (34.32 hr/revision). We plan to incorporate the automation in the future when we acquire more data.

6.3 Is there a way to evade Lshand?

Lshand bases its hunting skills mainly on clustering techniques, and thus Lshand is vulnerable to manipulation upon features. We came up with two evasion approaches. First, a malware writer can knowingly incrementally adds a large number of dummy components into a malware, so that the kin malware samples could end up in different DTs because of the enlarged distance in feature space. Second, a malware writer can develop a malicious module with only necessary components and then embed this module into a fake app that is far away in feature space (hence the product malware could be in a different DT).

Although Lshand is not fully evasion-resilient, the outcome of the above evasion techniques can be limited because of the following reasons. First, malware writers cannot easily

predict our Neg-Day results unless they deploy the same algorithms with the same variables and collect the same set of samples. Second, either above evasion techniques has drawbacks making them less compelling to malware writers: on one hand, incrementally adding a large number of dummy components will make the latter variants more suspicious to malware analysts and malware classifiers; on the other hand, the development of a small malicious module can still be detected and it is much easier to craft effective malware signatures from a small codebase.

6.4 Can Lshand be more scalable?

To be more scalable, Lshand is equipped with a multi-phase clustering option, and the idea is based on the map-reduce concept. To be brief, digested reports are split based on selective information (e.g., SID or timestamps), then digested reports are clustered into development subtraces, and then development subtraces are clustered into DTs. In our implementation, features for subtraces are aggregated from the digested reports by taking the mean value of features from the latest few reports. Compared to the single-phase clustering approach that took 291 hours in clustering 371,187 observations ($\rho = \text{unlimited}$), our two-phase clustering implementation took 93 hours for the same set of observation (note that clusterers were executed sequentially without parallelization).

However, since features are aggregated, multi-phase clustering approach may incorrectly filter out more AMDT samples (false negatives) than single-phase clustering approach. Indeed, two-phase approach reported 56 AMDTs with 1,622 samples (76 less samples than its single-phase approach), and the portion of perfect clusters is 99.14%-96.43%. A simple workaround to reduce false negatives is to set different threshold τ_i at different phase i . For example, setting $\tau_2 > \tau_1$ could reduce false negatives but it could also incorrectly report benign samples (false positives). Indeed, two-phase two-threshold approach reports 58 AMDTs with 1,799 samples, and the portion of perfect cluster is 86.83%-89.66%.

7 Related Work

Many research on Android malware countermeasures (e.g., [9, 14, 16, 19, 26, 30, 35, 40, 41, 43, 44]) against Android malware detection (e.g., [2, 3, 7, 10, 15, 20, 21, 23, 24, 27, 36, 37]) has been studied, but only a few focus on detecting malware development (e.g., [17]) or detecting a variant of a malware family (e.g., [12, 13]).

AMDHunter [17] aims to find Android malware development; however, AMDHunter has some critical flaws. First, AMDHunter tracks malware development cases by profiling SIDs (assuming an SID is more malicious if its submissions demonstrate little diversity and little repetition), and hence a malware writer can easily evade AMDHunter by submitting

samples from multiple SIDs or decorated SIDs (e.g., repeatedly submit a variety of apps). Second, AMDHunter highly depends on the trends of detection ratio (assuming malware development will demonstrate decreasing trends which is not true for LeakerLocker), and hence a malware writer can easily evade AMDHunter by submitting detectable variants of known malware. In contrast, Lshand leverages neither SID profiles nor trends of detection ratio, and hence not only is Lshand capable of hunting down Neg-Day cases from multiple SIDs (Table 6), but also did Lshand find more Neg-Day samples in the five common Neg-Day cases in the overlapped dataset (January 2016).

RevealDroid [12] and Graziano et. al [13] each solves a similar research problem. RevealDroid [12] aims to efficiently and accurately detect-and-identify malware family, but RevealDroid is not as scalable as Lshand, because it requires all malware samples to be dissected. The author stated that RevealDroid spent 3.5 days in analyzing 9,731 apps, and most of the execution time was spent on extracting features from samples. Graziano et. al [13] aims to mine malware intelligence from public dynamic analysis sandboxes; however, their work focuses on unobfuscated and unpacked Windows PEs, and thus some of their “most effective” features (e.g., filename edit distance) could be easily obfuscated.

Graziano et. al [13] and SSDC [39] also aim to detect-and-identify malware family based on Ssdeep rather than malware dissection. Ssdeep-based solutions require only the Ssdeep values: two samples that have homologies will have common byte sequences in their Ssdeep values, and two Ssdeep values can imply how homologous the two corresponding samples are. Although Ssdeep seems very convenient, there are a few reasons why we avoid using Ssdeep in Lshand: first, obfuscation and compression may mess up Ssdeep triggers, so two kin obfuscated APKs may have mismatched Ssdeep values; second, even when decompressed, repackaged fake apps will still demonstrate matched Ssdeep values with benign apps and cause false classification; third, fuzzyhash-based clustering for malware has already been studied and considered problematic [22]. The downside of Lshand compared to Ssdeep-based solutions is that Lshand focuses only on Android malware; nevertheless, the idea in Lshand can be extended to different malware types by importing different features and classifiers.

8 Conclusion

We propose Lshand that is capable of hunting down 10 Neg-Day cases (with 253 samples) from January 2016 and 12 Neg-Day cases (with 256 samples) from May 2018. Our results show that Lshand is efficient and accurate in hunting down Neg-Day malware samples that were given no malware label on VirusTotal.

Acknowledgments

The work of Zhu was partially supported through NSF CNS-1618684 and ARO W911NF-17-S-0002. Peng Liu was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1814679, and ARO W911NF-15-1-0576.

References

- [1] TESSERACT: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019. USENIX Association.
- [2] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *Computers and Security*, 65:230 – 246, 2017.
- [3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*. The Internet Society, 2014.
- [4] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.
- [5] Beijing Infinite Effect Media Information Technology Co., Ltd. Appbyme, an online creation platform of diy mobile applications., 2012. <http://appbyme.com/>.
- [6] David Braue. Security tools taking too long to detect new malware, analysis warns, 2015. <https://www.cso.com.au/article/566738/security-tools-taking-too-long-detect-new-malware-analysis-warns/>.
- [7] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. An hmm and structural entropy based detector for android malware. *Comput. Secur.*, 61(C):1–18, August 2016.
- [8] Sanjay Chakraborty and N. K. Nagwani. Analysis and study of incremental DBSCAN clustering algorithm. *CoRR*, abs/1406.4754, 2014.
- [9] Melissa Chua and Vivek Balachandran. Effectiveness of android obfuscation on evading anti-malware. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY '18*, pages 143–145, New York, NY, USA, 2018. ACM.

- [10] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell. Andro-dialysis: Analysis of android intent effectiveness in malware detection. *Computers and Security*, 65:121–134, 2017.
- [11] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *NDSS Symposium 2017*, San Diego, CA, 2017.
- [12] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, pages 497–497, New York, NY, USA, 2018. ACM.
- [13] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 1057–1072, Washington, D.C., 2015. USENIX Association.
- [14] 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, ICSE '18*, pages 421–431, New York, NY, USA, 2018. ACM.
- [15] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, pages 1–7, Aug 2014.
- [16] Wenjun Hu, Xiaobo Ma, and Xiapu Luo. Protecting android apps against reverse engineering. *Protecting Mobile Networks and Devices: Challenges and Solutions*, page 155, 2016.
- [17] Heqing Huang, Cong Zheng, Junyuan Zeng, Wu Zhou, Sencun Zhu, Peng Liu, Suresh Chari, and Ce Zhang. Android malware development on public malware scanning platforms: A large-scale data-driven study. In *Proceedings of the 2016 IEEE International Conference on Big Data (Big Data)*, BIG DATA '16, Washington, DC, USA, 2016. IEEE Computer Society.
- [18] Roberto Jordaney, Kumar Sharad, Santanu K. Dash, Zhi Wang, Davide Papini, Iliia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 625–642, Vancouver, BC, 2017. USENIX Association.
- [19] Jinho Jung, Chanil Jeon, Max Wolotsky, Insu Yun, and Taesoo Kim. AVPASS: Leaking and Bypassing Antivirus Detection Model Automatically. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, July 2017.
- [20] Dongwoo Kim, Jin Kwak, and Jaecheol Ryou. Dwroid-dump: Executable code extraction from android applications for malware analysis. *International Journal of Distributed Sensor Networks*, 11(9):379682, 2015.
- [21] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 287–301, San Diego, CA, 2014. USENIX Association.
- [22] Yuping Li, Sathya Chandran Sundaramurthy, Alexandru G. Bardas, Xinming Ou, Doina Caragea, Xin Hu, and Jiyong Jang. Experimental study of fuzzy hashing in malware clustering analysis. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*, Washington, D.C., 2015. USENIX Association.
- [23] Arvind Mahindru and Paramvir Singh. Dynamic permissions based android malware detection using machine learning techniques. In *Proceedings of the 10th Innovations in Software Engineering Conference, ISEC '17*, pages 202–210, New York, NY, USA, 2017. ACM.
- [24] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *NDSS Symposium 2017*, San Diego, CA, 2017.
- [25] M. Marszaek and C. Schmid. Spatial weighting for bag-of-features. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2118–2125, June 2006.
- [26] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1009–1024, May 2017.
- [27] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu. Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):157–175, June 2017.
- [28] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):157–175, 2017.

- [29] Palo Alto Networks. Wildfire malware analysis. <https://www.paloaltonetworks.com/products/secure-the-network/wildfire>.
- [30] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS Symposium 2014*, San Diego, CA, 2014.
- [31] Ford Qin. Leakerlocker mobile ransomware threatens to expose user information, 2017. <http://blog.trendmicro.com/trendlabs-security-intelligence/leakerlocker-mobile-ransomware-threatens-expose-user-information/>.
- [32] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19:639–668, 2011.
- [33] James Scott. Signature based malware detection is dead. *The Cyber Security Think Tank, Institute for Critical Infrastructure Technology*, 2017.
- [34] Linhai Song, Heqing Huang, Wu Zhou, Wenfei Wu, and Yiyang Zhang. Learning from big malwares. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 12:1–12:8, New York, NY, USA, 2016. ACM.
- [35] Guillermo Suarez-Tangil and Gianluca Stringhini. Eight years of rider measurement in the android malware ecosystem: Evolution and lessons learned. *arXiv preprint arXiv:1801.08115*, 2018.
- [36] L. Sun, Z. Li, Q. Yan, W. Srisa-an, and Y. Pan. Sigpid: significant permission identification for android malware detection. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–8, Oct 2016.
- [37] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang. Monet: A user-oriented behavior-based malware variants detection system for android. *IEEE Transactions on Information Forensics and Security*, 12(5):1103–1112, May 2017.
- [38] VirusTotal. A leakerlocker sample found on virustotal (first seen: 2016-11-23), 2016. <https://www.virustotal.com/en/file/d82330e1d84c2f866a0ff21093cb9669aaef2b07bf430541ab6182f98f6fdf82/analysis/1479960699>.
- [39] Brian Wallace. Optimizing ssdeep for use at scale and ssdeep cluster. 2015. <https://www.virusbulletin.com/virusbulletin/2015/11/optimizing-ssdeep-use-scale> and <https://github.com/bwall/ssdc>.
- [40] X. Wang, Y. Yang, and S. Zhu. Automated hybrid analysis of android malware through augmenting fuzzing with forced execution. *IEEE Transactions on Mobile Computing*, pages 1–1, 2019.
- [41] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. *Deep Ground Truth Analysis of Current Android Malware*, pages 252–276. Springer International Publishing, Cham, 2017.
- [42] Dietrich Wettschereck and David W. Aha. Weighting features. In Manuela Veloso and Agnar Aamodt, editors, *Case-Based Reasoning Research and Development*, pages 347–358, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [43] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security*, 12(7):1529–1544, July 2017.
- [44] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appsppear: Bytecode decrypting and dex reassembling for packed android malware. In Herbert Bos, Fabian Monrose, and Gregory Blanc, editors, *Research in Attacks, Intrusions, and Defenses*, pages 359–381, Cham, 2015. Springer International Publishing.
- [45] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless Mobile Networks*, WiSec '14. ACM, 2014.

***DroidScraper*: A Tool for Android In-Memory Object Recovery and Reconstruction**

Aisha Ali-Gombe
Towson University
aaligombe@towson.edu

Sneha Sudhakaran
Louisiana State University
ssudha1@lsu.edu

Andrew Case
Volatility Foundation
andrew@dfir.org

Golden G. Richard III
Louisiana State University
golden@cct.lsu.edu

Abstract

There is a growing need for post-mortem analysis in forensics investigations involving mobile devices, particularly when application-specific behaviors must be analyzed. This is especially true for architectures such as Android, where traditional kernel-level memory analysis frameworks such as Volatility [9] face serious challenges recovering and providing context for user-space artifacts. In this research work, we developed an app-agnostic userland memory analysis technique that targets the new Android Runtime (ART). Leveraging its latest memory allocation algorithms, called region-based memory management, we develop a system called *DroidScraper* that recovers vital runtime data structures for applications by enumerating and reconstructing allocated objects from a process memory image. The result of our evaluation shows *DroidScraper* can recover and decode nearly 90% of all live objects in all allocated memory regions.

1 Introduction

In recent years, there has been a significant increase in the adoption and reliance on memory forensics for incident response and malware analysis. While traditionally memory analysis was used to supplement disk forensics in the recovery of critical data such as deleted messages from volatile storage, memory forensics has evolved into an advanced methodology for investigating and identifying memory-resident, kernel-level attacks and malicious behaviors that do not leave an identifiable footprint on the disk [16].

In mobile devices, the advancement and sophistication in application development and the reliance on these devices by many users make them a critical source of evidence for digital investigations. The ability to recover and reconstruct pieces of application data which otherwise may be difficult to identify using traditional static and dynamic analysis can provide investigators with substantial evidence to leverage in cybercrime and malware analysis. However, due to the layers of abstraction between the kernel and the application

in the mobile architecture, recovering in-memory application data is not feasible from the residual in-memory kernel data structures. On the other hand, userland memory artifacts can provide sufficient information for investigators to recover application functionality and outline the actions, strategy, and attack evidence without the need for prior knowledge of application logic.

In this research, we target userland memory analysis on Android. The objective is to develop an app-agnostic, per-process technique that can perform post-mortem analysis on Android userland memory dumps to recover and reconstruct vital in-memory evidence for cybercrime investigations and malware and vulnerability analysis. Our effort leverages the new Android Runtime (ART) to trace all objects allocated within a process' address space and then make a best effort to rebuild those objects. This approach is built upon ART's newest garbage collection algorithm, Concurrent Copying collector, which uses region-based memory management for object allocations. In this algorithm, objects are allocated in program specific memory regions, and during garbage collection, an entire region is collected if its number of live objects is less than a certain threshold. With this algorithm, objects tend to live longer in memory when allocated in a region with high live threshold even if the specified object is marked for deallocation. The design of our approach involves identifying crucial ART structures, and subsequently, all objects allocated at runtime in a target process' address space.

In comparison with traditional static app analysis systems, which are often affected by obfuscation such as class and data encryption, dynamic class and Java reflection, etc., this approach has the advantage of recovering de-obfuscated runtime artifacts. Furthermore, unlike dynamic analysis methodologies, our technique relies only on having a process memory dump and is therefore less likely to be affected by anti-analysis techniques.

The contributions of this research work are: (1) We propose a new memory forensics technique, *DroidScraper*, that relies on the design of Android's ART region-based memory allocation to recover and reconstruct in-memory runtime artifacts.

(2) The proposed *DroidScraper* can extract running threads, enumerate objects allocated in the heap region, and then decode objects based on their class definitions. (3) Our evaluation of *DroidScraper* shows that it can recover in-memory objects with an almost 90% success rate and can be applied to behavioral post-mortem monitoring of Android applications.

2 Background

Android applications are typically written in Java and compiled into bytecode which is then executed on a Dalvik Virtual Machine (for older devices) or Android Runtime (ART) (from Android 5.0 and beyond). These Java applications can also be integrated with native code written in C/C++, with the help of Java Native Interface (JNI). ART provides a number of new features, most notably enhanced garbage collection (GC) algorithms, which affect object allocations in the new runtime.

The new Android runtime - ART has two groups of garbage collectors based on the ActivityManager process state [11]. The *ForegroundCollector* handles GC when an app is in the foreground, while the *BackgroundCollector* handles GC when an app is running in the background. A variable for the *ForegroundCollector* comes preconfigured as part of the default runtime options on stock Android as shown in Listing 1. At heap initialization, the system reads the runtime options to determine which collectors to use. Also, this variable defines the memory allocation scheme (*AllocatorType*) to be employed, and subsequently, the memory space (*Space*) to be created and how objects are organized in these spaces. Listing 2 shows all the available allocators on ART.

```
bool Runtime::Init(RuntimeArgumentMap&&
→ runtime_options_in) {
    .....
    XGcOption xgc_option =
→ runtime_options.GetOrDefault(Opt::GcOption);
    heap_ = new gc::Heap
→ (runtime_options.GetOrDefault
→ (Opt::MemoryInitialSize),
    .....
    runtime_options.GetOrDefault
→ (Opt::ImageInstructionSet),
    // Override the collector type to CC
→ if the read barrier config.
    kUseReadBarrier ? gc::kCollectorTypeCC :
→ xgc_option.collector_type_,
    kUseReadBarrier ? BackgroundGcOption
→ (gc::kCollectorTypeCCBackground)
    runtime_options.GetOrDefault
→ (Opt::BackgroundGc),
    .....);
}
```

Listing 1: Runtime Initialization in runtime.cc

```
enum AllocatorType {
    kAllocatorTypeBumpPointer,
    kAllocatorTypeTLAB,
    kAllocatorTypeRosAlloc,
    kAllocatorTypeDlMalloc,
    kAllocatorTypeNonMoving,
    kAllocatorTypeLOS,
    kAllocatorTypeRegion,
    kAllocatorTypeRegionTLAB,
};
inline constexpr bool IsTLABAllocator
→ (AllocatorType allocator) {
    return allocator == kAllocatorTypeTLAB ||
→ allocator == kAllocatorTypeRegionTLAB;
}
```

Listing 2: Available memory allocator types for ART

ART is designed with four major garbage collection algorithms with each one utilizing one or more of the *AllocatorTypes* in Listing 2. The complexity of choosing collectors and mapping them to one or more allocator types makes the memory allocation and GC on the new Android runtime a very complicated process. As mentioned above, the chosen collector at system startup determines the garbage collection algorithm, which in turn determine the specific *AllocatorType* to be used by the runtime environment. Below is a detailed description of the GC algorithms in ART.

- **Semi-Space - SS** - this algorithm uses the semi-space garbage collection to copy movable objects between two Bump Pointer spaces. In the SS algorithm, objects are allocated in free memory space using *BumpPointer* and *DlMalloc* for mutable and non-mutable objects, respectively. When the use of Thread Local Allocation Buffers is enabled, this algorithm defaults to using the *TLAB* allocator. On newer Android versions, the *BumpPointer* spaces are currently only used for *ZygoteSpace* construction.
- **Generational Semi-Space - GSS** - This algorithm leverages heap organization to optimize the simple Semi-Space GC above. The generational hypothesis states that most objects die young [28] and as such where long-lived reachable objects exist, they are relocated to a large *RosAlloc* space. The default object allocator for GSS is the *BumpPointer* for mutable objects.
- **Concurrent Mark Sweep - CMS** - this collector uses the concurrent mark-sweep algorithm to collect allocated objects unreachable from their roots from only the region of memory modified since the last GC operation [4]. The default memory *AllocatorType* for CMS is the *RosAlloc* - which is used to allocate mutable objects in runs-of-slots of the same sizes. It also uses the default C malloc *DlMalloc* for non-mutable objects. CMS was introduced

in Android 5 as the default CG algorithm for the Android runtime environment. It is designed to improve app performance through Concurrent collection. While this algorithm has significantly improved GC effort, however, it causes two long pauses during each collection cycle that often adversely affect UI responsiveness [17].

- **Concurrent Copying** - This technique utilizes an efficient concurrent and moving garbage collection algorithm. Utilizing region-based memory allocation, allocated objects are evacuated from a region and subsequently destroyed if and only if the region has live objects whose count is less than some percentage threshold. Furthermore, this algorithm creates a compacting heap and introduces very short pauses during collection [1]. It also utilizes a read barrier configuration that ensures mutators never see old versions of objects [3]. This configuration allows threads to efficiently and concurrently access heap objects during collection. The CC algorithm uses the *RegionSpace* allocator and if the use of TLAB is enabled, the system uses the *RegionSpaceTlab* allocator for movable objects. On newer Android versions, *RegionSpaceTlab* is the default for most small object allocations [20].

In the earlier version of *libart* (5,6,7), the CMS collector was favored among the other collectors, thus defaulting to the use of RosAlloc for moving objects. However, in newer Android versions (8,9,10), the development and subsequent enforcement of the read barriers in the runtime options as shown in Listing 1 which favors concurrent access to the heap during GC overrides the default CMS collector type to the CC [12, 17]. Furthermore, the introduction of *RegionSpaceTLAB* for per-thread objects makes it an ideal allocation mechanism for small objects.

In this research, our focus is on recovering objects allocated using the *RegionSpace* and *RegionSpaceTlab* allocators, which are based on the Concurrent Copying Collection algorithm. To the best of our knowledge, this is the first work that explores in-memory data recovery from *RegionSpace* memory maps for the new Android Runtime.

3 System Design

DroidScrapper is an Android in-memory object recovery and decoding system that analyzes process address spaces (in the form of per-process memory dumps) for remnants of runtime objects. This system leverages low-level data structure definitions as well as generic class and references constructs provided by the Android runtime library (*libart.so*) to recover and reconstruct objects within a target process' address space. The design of *DroidScrapper* provides investigators and malware analysts access to well-structured and forensically interesting data across threads and various process components such as activities and services.

As shown in Figure 1, *DroidScrapper*'s workflow begins with process memory dump acquisition. This element of the workflow utilizes any available open-source tools such as Memfetch [6] to generate per process Android memory map. In cases where a complete memory image is acquired using tools like AMD [39] and Lime [36], *DroidScrapper* can utilize the output of Volatility's *memdump* [10] plugin to access a process' address space. The memory acquisition process is then followed by the Runtime Data Structure Recovery (RDS) and the Object Recovery and Reconstruction (ORR) modules. These two elements of the workflow constitute the main contribution of our system. The RDS module leverages the per-process memory dump to identify and recreate major runtime structures that are essential for object allocations and deallocation such as Runtime, Heap, Heap Regions, and Threads. The ORR module then utilizes the metadata definitions in the recovered data structures to enumerate and decode reachable live objects within the process memory region.

3.1 Runtime Data Structure Recovery - RDS

DroidScrapper's RDS is built upon the low-level data structures defined in *libart.so*. These structures are essential for building and maintaining the runtime environment, object allocation and accounting, as well as garbage collection. The RDS module begins by identifying the main runtime object of a target process, followed by the identification and extraction of all Linux threads belonging to the target process. It also consists of other sub-modules for heap structure recovery, the identification of the Region Space structure, and the allocated heap regions.

3.1.1 Identification of the Runtime Object

The *Runtime* object is the most crucial data structure required for Android process execution. Its members constitute essential components needed by the process for memory allocation, thread creation, JNI calls, and it serves as a link between the running process and outside environment. On Android, every process executes in its own runtime environment, which itself is forked from the zygote process. The zygote process is started by the init process at system boot together with the default Android runtime. The zygote then listens for a connection on its socket for a request to start up a new application. Upon receipt of such a request, it will fork a new Linux process, which forces the creation of a *Runtime* object that establishes the runtime environment. The zygote process then maps a copy of its shared library (*libart.so*) into the new process' address space.

```
_ZN3art7Runtime9instance_E offset = 0070a980  
Runtime Base Address = 0xf233aa80
```

Listing 3: Instance address offset and the base address of the Runtime object recovered

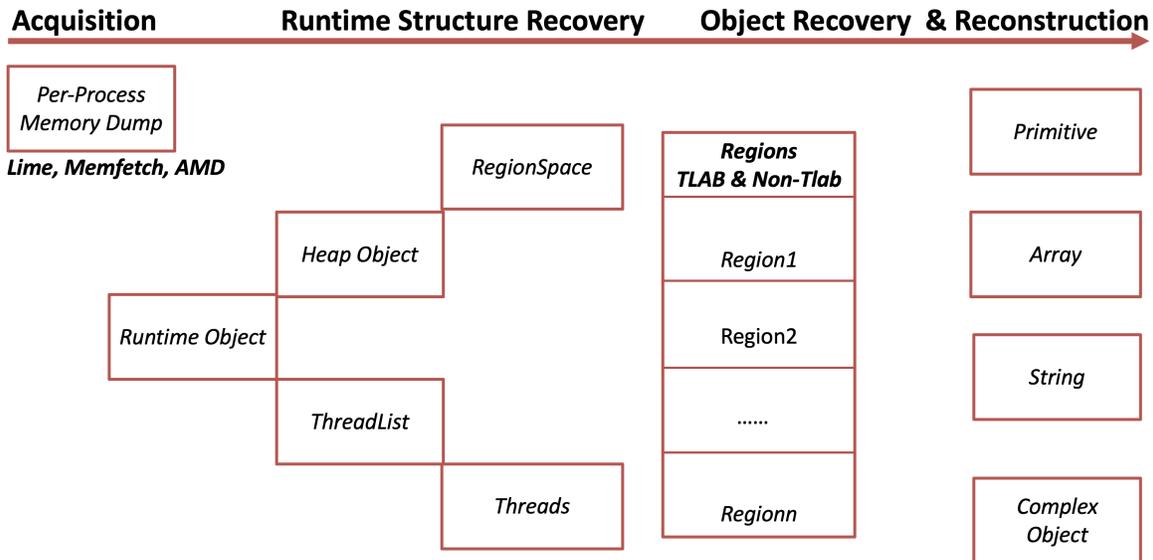


Figure 1: *DroidScraper* Workflow - Showing Acquisition, Runtime Data Structure Recovery and Object Recovery and Reconstruction modules.

In this task, our objective is to identify the location the process's *Runtime* object in the memory dump and then reconstruct the structure based on its class template definition, as given in the *libart.so*.

```
'Runtime' : [ 0x340, {
  'callee_save_methods_': [0],
  'pre_allocated_OutOfMemoryError_': [32],
  'pre_allocated_NoClassDefFoundError_': [36],
  'resolution_method_':[40],
  ....
  'heap_':[244],
  'jit_arena_pool_':[248],
  'arena_pool_':[252],
  'low_4gb_arena_pool_':[256],
  'linear_alloc_':[260],
  ....
  'monitor_list_':[268],
  'monitor_pool_':[ 272],
  'thread_list_':[ 276],
  .....}]
```

Listing 4: The Runtime object for Android 8, represented as a C structure.

Utilizing a static exploration of *libart.so* with Linux *nm* command, we identify an address offset `_ZN3art7Runtime9instance_E` in the list of symbols. This offset held the address of the *Runtime* instance when the zygote forked a new process. Because this step is critical to the entire recovery process, we verified on two different devices (Unrooted Samsung S9+ and Samsung S8 AVD) that the object file - *libart.so* has some basic symbols compiled

in it by default and that includes the Runtime instance address. Located in the uninitialized data section of the memory where *libart.so* is mapped in process space, we identify and dereference the address in this offset to locate the beginning of the active *Runtime* instance as shown in Listing 3. We represent the Runtime class as defined in the Android documentation [12] using a C structure, with the class's fields represented as members of the structure. Listing 4 illustrate a partial Runtime object with size and member offsets for Android 8.0.

3.1.2 Enumerating Threads in User Processes

When an Android process begins executing, it spawns its first thread, called the main thread. Depending on how the program is designed, other components of the application such as activities, receivers, providers, and services may create other threads. The **ThreadListing** sub-module is tasked with enumerating all the living threads owned by the user process. Identifying and enumerating running threads is essential for virtually every memory forensics effort. Specifically for *DroidScraper*'s in-memory data recovery and reconstruction, identifying running threads is essential because the system often uses thread local allocation buffer *TLAB* for faster and more efficient object allocations. *TLAB* is a memory region assigned to a single thread for its own objects and can be used without the need to acquire and/or release locks. When the *use_tlab* option is enabled during heap creation, objects are allocated on a per thread basis and thus recovering those objects will require obtaining and analyzing the thread metadata structure.

Threads	TID	Name
0xe3e66e00	20392	hwuiTask1
0xe9137c00	20391	Binder:20370_3
0xe9134000	20388	RenderThread
0xe67fb000	20386	Thread-3
0xe663e400	20384	Profile Saver
0xe911a800	20383	Binder:20370_2
0xe407e400	20382	Binder:20370_1
0xe663d200	20381	HeapTaskDaemon
0xe8f7de00	20380	FinalizerWatchdogDaemon
0xf674be00	20378	ReferenceQueueDaemon
0xe8f7d800	20379	FinalizerDaemon
0xe910bc00	20377	JDWP
0xf674ac00	20376	Signal Catcher
0xe9108000	20375	Jit thread pool worker thread 0
0xf674a000	20370	main

Figure 2: *DroidScaper* Runtime Data Structure Recovery - ThreadListing.

From the process *Runtime* object, the **ThreadListing** finds the pointer to the beginning of the *ThreadList* object, which is at offset 0x276 on Android 8.0. The threads in the *ThreadList* structure are organized in a cyclical double-linked list. The list header contains the pointers to the first and last *Thread* objects in the list as well as the size of the list. After the identification and subsequent dereferencing of each individual thread pointer to recover all the *Threads* objects, the **ThreadListing** uses the definition of the *Threads* object in the *libart.so* documentation to retrieve each thread ID (tid) and thread name for all the living threads in the user process. Figure 2 illustrates the output of the **ThreadListing** sub-module of the *DroidScaper*'s *RDS*.

3.1.3 Recovering the Heap Structure

As the *Runtime* object initializes, it creates the *Heap* object, which is tasked with managing memory space creation, object allocation mechanisms, and accounting. At initialization, most of the arguments passed to the *Heap* are manufacturer's or design standard's runtime_options such as instruction sets, heap limit, etc. But two vital runtime_options that are crucial to *DroidScaper*'s object recovery are the *UseTlab* and *kUseReadBarrier* option. The *use_tlab* option, if present, causes threads to request and utilize the Thread local allocation buffer (Tlab) for small object allocations. As referenced in Listing 1, the *kUseReadBarrier* forces the system to use the Concurrent Copying (CC) collector which then causes the heap to allocate a different *NonMovingSpace* for non-mutable objects and a *RegionSpace* for mutable objects. On newer versions of Android Open Source Project AOSP (8,9 and 10), *kUseReadBarrier* is enabled by default thus making the Concurrent Copying the default garbage collection technique, and the memory allocations use the region-based memory management, which is the target for *DroidScaper*.

In ART's Concurrent Copying garbage collection, the heap creates a *RegionSpace* structure, which is a continuous space memory map used for the creation of equal-sized memory regions and storing their metadata. As shown in Listing 5, the *RegionSpace* holds the tally of the number of regions created by the *Heap*, the number of non-free regions, and the pointer to the array of all available *Regions*.

```
'RegionSpace' : [ 0xa8, {
    'ContinuousMemMapAllocSpace' : [0],
    'region_lock_': [56],
    'time_': [96],
    'num_regions_': [100],
    'num_non_free_regions_': [104],
    'regions_': [108],
    'non_free_region_index_limit_': [112],
    'current_region_': [116],
    'evac_region_': [120],
    'full_region_': [124],
    'mark_bitmap_': [164],
}]
```

Listing 5: The *RegionSpace* object for Android 8, represented as a C structure.

Each available *Region* structure, as shown in Listing 6, holds pointers to the beginning and end of a region, as well as the top of the region (the address of the last object allocated). It also holds the total number of objects allocated in the region and a boolean value that shows whether this region is a TLAB region or not. If a region is TLAB, then the pointer to the thread offset will be non-zero.

```
'Region' : [ 0x28, {
    'idx_': [0],
    'begin_': [4],
    'top_': [8],
    'end_': [12],
    'state_': [16],
    'type_': [17],
    'objects_allocated_': [20],
    'alloc_time_': [24],
    'live_bytes_': [28],
    'is_newly_allocated_': [32],
    'is_a_tlab_': [33],
    'thread_': [36],
}]
```

Listing 6: The *Region* object for Android 8, represented as a C structure.

For this task, we developed a sub-module called the **Heap** that identifies the beginning of the process heap in the *Runtime* object. Dereferencing the heap address, we walk the structure to determine the offsets for the *RegionSpace* and then the pointer to the array of regions. For each recovered *Region* structure,

```

Heap Offset 0xf2297400
RegionSpace Offset 0xf6730300
Number of Regions 2048
Number of Non Free Regions 13
Region Array Offset 0xf22c06c0
TLAB Regions
Index  Begin      Top      End      Thread      Objects      Tid Thread_Name
0  0x12c00000 0x12c40000 0x12c40000 0xe8f7d800 3      20379 FinalizerDaemon
2  0x12c80000 0x12cc0000 0x12cc0000 0xe9108000 7      20375 Jit thread pool worker thread 0
3  0x12cc0000 0x12d00000 0x12d00000 0xe9134000 5      20388 RenderThread
4  0x12d00000 0x12d40000 0x12d40000 0xf674a000 630    20370 main
5  0x12d40000 0x12d80000 0x12d80000 0xe911a800 11     20383 Binder:20370_2
6  0x12d80000 0x12dc0000 0x12dc0000 0xe407e400 10     20382 Binder:20370_1
7  0x12dc0000 0x12e00000 0x12e00000 0xe9137c00 5      20391 Binder:20370_3
8  0x12e00000 0x12e40000 0x12e40000 0xe3e66e00 5      20392 hwuiTask1
9  0x12e40000 0x12e80000 0x12e80000 0xe910bc00 416    20377 JDWP
Non-TLAB Regions
Index  Begin      Top      End      Objects
1  0x12c40000 0x12c80000 0x12c80000 2472
70 0x13d80000 0x13da1ec8 0x13dc0000 1508
71 0x13dc0000 0x13e00000 0x13e00000 650
118 0x14980000 0x14982828 0x149c0000 48

```

Figure 3: *DroidScrapper* Runtime Data Structure Recovery - Heap plugin output showing non-free regions in *RegionSpace*.

we walk its metadata to retrieve its index, begin address, end, top, and the number of objects allocated. Where a region is a Tlab region, the **Heap** recovers the thread offset and then utilizes the **ThreadListing** to identify the thread name and ID. The output of the **Heap** sub-module as illustrated in Figure 3 shows the offsets of the Heap and RegionSpace. The output also shows the process has a total of 2048 regions with only thirteen in use. The output further indicates the distribution of the thirteen occupied regions as having nine Tlab regions and four Non-Tlab regions.

3.2 Object Recovery and Reconstruction - ORR

The objective of ORR module is to perform the actual recovery of the dynamically allocated objects. This module leverages the extracted *Runtime* data structures above to identify reachable and live runtime objects and then makes the best effort to reconstruct each object.

3.2.1 HeapDump - Object Recovery

Our next sub-module is called **HeapDump**. This component is tasked with the identification and subsequent recovery of all reachable and live objects in the non-free heap regions. At creation, every object is associated with an object tree which has one or more root objects. This new object is marked reachable if it can be referenced from another reachable object. Android uses direct references to manage Java objects and indirect references for JNI code. Every instance of an object allocated is derived from the *Object* class. The object class contains two members - the class member which is defined as a *HeapReference* to the class description of the object and four bytes of monitor/hash information. The class definition

determines the type of the object allocated, which in turn specifies its size. Using the algorithm defined in Algorithm 1, the **HeapDump** traces each object by decoding its object class reference to get its name and class flag which is then used to determine the object type. The sub-module then computes the size of the object based on the defined type. The location of the next object is calculated based on the size of the previous object. The output in Figure 4 prints an object offset, the class name for the object, and the object size.

Algorithm 1: HeapDump Algorithm

```

1 pos = regionBegin
2 while pos < regionTop do
3   if clazz -> ResolveClass(pos) != nullptr then
4     obj = GetObject(pos, clazz)
5     size = GetSize(obj)
6     pos = pos + size + kAlignment
7   else
8     pos = pos + kAlignment
9   end
10 end

```

3.2.2 Object Decoding

When a process or thread allocates an object, the class and the size of the object are provided as parameters to the allocation function. At a higher level, the classes can be broadly classified into four types - Primitives, Arrays, Strings, and Complex classes. Objects in each of these classes are allocated in a unique way and as such our decoding algorithm handles each one of them differently, as shown in Algorithm 2.

```

Address 0x12c738c0 java.lang.Class - [C 18
Address 0x12c738d8 java.lang.Class - java.nio.charset.CharsetDecoderICU 72
Address 0x12c73920 java.lang.Class - [I 24
Address 0x12c73938 java.lang.Class - libcore.util.NativeAllocationRegistry$CleanerThunk 24
Address 0x12c73950 java.lang.Class - sun.misc.Cleaner 36
Address 0x12c73978 java.lang.Class - libcore.util.NativeAllocationRegistry$CleanerRunner 12
Address 0x12c73988 java.lang.Class - java.nio.HeapByteBuffer 47
Address 0x12c739b8 java.lang.Class - [B 15
Address 0x12c739c8 java.lang.Class - java.lang.String 19
Address 0x12c739e0 java.lang.Class - android.app.ActivityThread$ProviderKey 16
Address 0x12c739f0 java.lang.Class - android.app.ContentProviderHolder 21
Address 0x12c73a08 java.lang.Class - android.content.pm.ProviderInfo 95
Address 0x12c73a68 java.lang.Class - java.lang.String 59
Address 0x12c73aa8 java.lang.Class - java.lang.String 47
Address 0x12c73ad8 java.lang.Class - android.content.pm.ApplicationInfo 234
Address 0x12c73bc8 java.lang.Class - java.lang.String 47
Address 0x12c73bf8 java.lang.Class - java.lang.String 47
Address 0x12c73c28 java.lang.Class - java.lang.String 33
Address 0x12c73c50 java.lang.Class - java.util.UUID 24

```

Figure 4: *DroidScraper* Runtime Data Structure Recovery - HeapDump.

[String Object]

```

Aishas-MBP-2:ART aishacct$ python artProj.py mal8/ Heap decodeObject 0x13014130
@ Address 0x13014130
0x6f2ca108 is a Global Reference
+++++
Reference Class is a Class Instance
Number of Reference Instance Fields = 0
The data for java.lang.String is plbslog.umeng.com

```

[Complex Object]

```

Aishas-MBP-2:ART aishacct$ python artProj.py mal8/ Heap decodeObject 0x12fa23a0
@ Address 0x12fa23a0
No reference for 0x6f6dc2e0
+++++
Reference Class is a Class Instance
Number of Reference Instance Fields = 2
android.app.ContextImpl$ApplicationContentResolver kClassFlagNormal
Super Class Offset android.content.ContentResolver
Object Size 32
FieldName - mContext - Landroid/content/Context; offset 8
Data --- 0x12fa22d8
FieldName - mPackageName - Ljava/lang/String; offset 12
Data --- 0x12f82010
FieldName - mRandom - Ljava/util/Random; offset 16
Data --- 0x12fa3388
FieldName - mTargetSdkVersion - I offset 20
Data --- 26
FieldName - mMainThread - Landroid/app/ActivityThread; offset 24
Data --- 0x12f809a8
FieldName - mUser - Landroid/os/UserHandle; offset 28
Data --- 0x6f61da20

```

Figure 5: *DroidScraper*'s DecodeObject sub-module - Recovering a String object and an instance of java.net.URL.

1. Primitive - A primitive class stores basic data types. Objects in this category are allocated according to the data type they represent. The component size for primitive ranges from 0 for Void, 1 for Boolean and Byte, 2 for Short and Char, 4 for Integer and Float, and 8 for Long and Double. If an object class is of type primitive, the data is read based on its component size.
2. Array - An object of type array holds a group of data items of the same size beginning at a contiguous memory location. When an object class resolved to an array, the 4 bytes after the *Object*'s inheritance structure is the length of the array followed by the data offset. The total size of an array is the sum of its metadata and the product of its component size and length. For instance, an integer array

Algorithm 2: ObjectDecode Algorithm - GetObjectData(obj)

```

1 if clazz==Primitive then
2 | len = obj->getType()->getComponentSize()
3 else if clazz==String then
4 | len = obj->len
5 else if clazz==Array then
6 | type = obj->getType()
7 | len = obj->getComponentSize(type)
8 else
9 | clazzName = obj->getName()
10 | fields[] = obj->getFields()
11 | len = fields->getLength()
12 end
13 data = read(obj, len)

```

of size four will have a total size of 32 bytes including alignment.

3. String - An object of type string is allocated by providing the length of the string encoded in a 32-bit integer variable called count. The count is written right after the *Object*'s inheritance structure, followed by the hashcode of the string and then the beginning of the data.
4. Complex Class - This category hold instances of any class that is not primitive, array, or string. Objects in this category have a complex structure called *Class* that holds the name of the object, its size, its class size, superclass, fields, and methods, among other members. To decode an object in this category, we first find the pointer to the *Art_Field* structure. This structure helps us identify and decode all the members of this instance, their names, and offsets.

As shown in Figure 5, we developed a sub-module called the **DecodeObject** that decode and reconstruct live and reachable objects from the process memory dump.

4 Evaluation

DroidScrapper is developed as a standalone memory forensics tool that analyzes per-process Android application memory maps for remnants of runtime artifacts. With its generic implementation, *DroidScrapper* can be utilized to examine process memory directly extracted from Android devices using tools such as memfetch [6] or indirectly generated by plugins such as Volatility's memdump [10], without any background knowledge of the target application's data structures. The current version of *DroidScrapper* is written in Python and works on process memory images obtained from any Android device running versions 8.0 and 8.1. The free and open source version of this tool will be released with the publication of this article.

4.1 Accuracy of Objects Recovery and Reconstruction

To assess the effectiveness of our approach for object recovery, we performed a series of experiments on memory images generated across a wide variety of applications. These include six malware samples from the CICAndMal2017 dataset [22] and VirusShare [37] and six benign applications - *Signal*, *EvolveSMS*, *Keeper*, *Calculator Vault*, *Clock Vault*, and *Google Chrome* running the *Facebook* web application, all downloaded from Google play [19].

4.1.1 Experimental Setup

Our evaluation used the Genymotion Desktop Android emulator as the execution environment. With over 3000 virtual Android device configurations and full sets of hardware sensors, Genymotion can emulate real Android devices with a high degree of accuracy [5]. We created AVDs for Google Pixel running Android 8.1-API 27 and Google Nexus 6 and Samsung S8 running Android 8.0-API 26. All the emulators have 4GB memory and are equipped with one Gmail account and a couple of fake SMS and contacts to simulate real devices. Each application was then installed on a selected device and interacted with manually by the authors to generate sufficient activity. We then captured the process memory image using Memfetch.

4.1.2 Object Recovery

To analyze the memory dumps using *DroidScrapper*, we used a MacBook Pro as the host system. The captured memory images on the AVDs are pushed to the Mac using the Android *adb* [2] utility. We first execute the **ThreadListing** to enumerate all living threads and then run the **Heap** module to itemize the allocated regions (Tlab and non-Tlab) and obtain an object count per region. Finally, we run **HeapDump** to recover the reachable live objects. As shown in Table 4.1, the Total

Objects column represents the cumulative total objects count from each region. This value tallies the *objects_allocated_* field of the *region* structure for all the non-Tlab regions and the *thread_local_objects* field of each thread in the Tlab regions. As mentioned in Section 3, the **HeapDump** module makes the best effort to decode the type of object and its size in an allocated memory region. However, not all objects can be recovered as some may have been deallocated even though the region is not collected. Nevertheless, in Android's Region-based memory management, a region will be completely collected only if the live threshold is below 75%. Thus, for each allocated region, the percentage recovery will be higher than 75 which in turn means the total recovery percentage for each of the test apps must be higher than 75.

For better analysis, our recovered objects are grouped into Primitives, Arrays, Strings, and Objects. The total of all the recovered objects is given in the Total Recovered column. For the measure of performance of *DroidScrapper*, the recovery percentage is computed as a fraction of the total recovered objects to the cumulative tally of the objects in all the allocated regions. For each test app, the recovery percentage is given in the % column. The average recovery percentage across all the test applications is approximately 89.6%. This means that *DroidScrapper* can recover nearly 90% of all objects allocated within the process memory space in well-structured formats. This percentage likelihood further buttresses *DroidScrapper*'s ability to retrieve crucial and forensically interesting runtime artifacts created by an Android process.

4.1.3 Object Reconstruction

From the results obtained in the object recovery above, we performed an in-depth analysis of two samples. As a representative of the application dataset, we selected one malware and one benign application and examined them for the presence of any forensically interesting data.

RansomBO (com.yandex226.yandex967) - In the analysis of this malware sample we found evidence that it utilizes an instance of Chromium *org.chromium.content.browser.PopupZoomer* as its main activity View. This View is designed to automatically terminate itself and delete the application icon after a few seconds, changing the *Animating* and *Showing* members of the *PopupZoomer* View to false. We also found evidence of file activities in the *android.app.SharedPreferencesImpl* object, where the malware opened and read data from a shared preferences XML file called *maxiettings.xml*. This file contains the server URL *http://212.56.214.233/task.php* and other connection information. We also recovered and decoded the *com.android.okhttp.internal.huc.HttpURLConnectionImpl* object, which showed that the malware made a POST request to the server using an instance of *com.android.okhttp.Request* with the following header values:

Applications	Threads	Regions	Total Objects	Primitives	Arrays	Strings	Objects	Total Recovered	%
com.baidu.mbaby	16	3	2780	6	526	671	1235	2438	87.7
com.losg.netpack.BaApp	28	10	12493	77	2555	1929	6436	10997	88.1
com.caf.fmradio	15	7	7707	43	3016	1126	2878	7063	91.6
com.yandex226.yandex967	30	12	10346	161	2126	1547	5895	9729	94
cn.myhug.baobao	30	17	50529	133	6410	8571	12025	44297	87.7
com.easyhin.usereasyhin	31	15	29654	2847	6856	5391	29203	27139	91.5
Keeper	44	102	264237	2623	71823	39507	107348	221301	83.8
CalculatorVault	53	22	44757	271	8464	7320	23225	39280	87.8
ClockVault	87	31	99641	2428	15664	10287	60309	88688	89
EvolveSMS	24	36	33234	169	6565	6195	18530	31459	94.7
Signal	36	48	287650	129599	22671	24369	79528	256167	89.1
Chrome Browser	31	11	20628	208	3563	4566	10315	18652	90.4

```
'Content-Type', 'multipart/form-data; boundary="====1552903509936====", 'Accept', 'application/json', 'http.agent', 'User-Agent', 'Dalvik/2.1.0 (Linux; U; Android 8.0.0; Samsung Build/OPR6.170623.017)'
```

Checking for a response to this connection, we found that the connection did not go through. The DetailedMessage member of the *java.net.SocketTimeoutException* has the following exception message:

```
failed to connect to /212.56.214.233 (port 80) from /10.0.3.15 (port 47488) after 5000ms.
```

Keeper - In the analysis of this Vault app, we found that the application creates a database file using the email address provided by the user and then saves the hash of the user-supplied password in the file. As contents are added to the vault, the application queries the database for the hash, then encrypts the data using the password hash, an initialization parameter, and an encryption key. We found evidence of 115 instances of *javax.crypto.spec.IvParameterSpec* and 152 instances of the *javax.crypto.Cipher\$InitParams* objects in the memory dump. The reconstruction of each initialization vector object (IV) reveals the 16 byte IV values in clear text. The IV value is random and unique across all the 115 instances. For the Initialization parameter, the *javax.crypto.spec.SecretKeySpec* members show the data is encrypted using an AES algorithm and the 32-byte key is shared across all the 152 instances. The complete transformation of any content uses a combination of AES/CBC/PKCS7Padding for crypto, feedback mode, and padding respectively. But the most interesting part of this analysis is at the beginning of the first instance of *IvParameterSpec*, we found the database query and hash of the password. This is then followed by the new instance of *IvParameterSpec*, *Cipher\$InitParams*, and *Cipher\$Transform* with all the bytes in the block stored consecutively in a row. Thus it is easy to figure out and reconstruct the original block, the IV, the key and the transformation algorithm from the recovered objects.

4.2 Case Study

In this part of the evaluation, we will use a case study to illustrate the application of *DroidScrapper*'s data recovery and reconstruction for program analysis, specifically, in regenerating program control flow to prove database access based on remnants of runtime allocations. We created a small piece of code that queries three columns (*_id*, *address*, *body*) from the SMS database as shown in Figure 6 - (1) below. The result of this query is populated in a *Cursor*, and then the data is read into a *StringBuilder* by moving the *Cursor* from the first position to the last. As mentioned in Section 2, with region-based memory management, objects are added at the top of a region and the new top is calculated by adding the object byte size to the old top. Thus, with such allocation, it is possible to trace precise control flow of the code by examining the objects allocations as shown in Figure 6.

Mapping the code segment (1) one-to-one with the partial output of the **HeapDump** plugin in (2), it is noticeable that right after the *CursorWrapper* object is allocated, the *StringBuilder* and the subsequent strings that are used to read the data from the *Cursor* were also created in the same region. Thus as shown in (9), the return string of the *StringBuilder.toString()* function, which contains all the messages queried from the SMS database, is allocated at 0x12c748a0. However, to give context to the read messages, we need to understand how Android performs database accesses.

In order to access any database content on an Android device, the requesting application uses its *Resolver* object to perform a CRUD operation. Each of the available CRUD functions requires at least a *URI* and in some cases, valid column names and conditions. Based on the *URI*, the *Resolver* sends the request to its corresponding *Provider*. The task of the *Provider* is two-fold - it validates the *URI* by matching its authority to the *URI*'s authority, and then performs a permission check on the requesting app, after which it creates and sends a valid *SQLiteStatement* to the native *SQLite* engine for processing. Thus, based on this code segment, the runtime creates a *URI* object as shown in (3), which authority is "sms" and uriString is "content://sms/inbox". The corre-

sponding *Provider* object whose authority matches that of the SMS URI is shown in (4). Upon approval of the *Resolver*'s database request, the runtime creates a *CursorWrapper* object, which implements the *Cursor object*. The instance of this object is allocated at address 0x12c743d0 as shown in (5). The instance of the *CursorWrapper* contains pointers to the a) *Cursor* object, as shown in (6), which holds the column names, count and the data, b) *Provider*, as shown in (4), points to the database currently being accessed and c) *Resolver* in (7) which points to the client currently accessing the content - in this case, the package of the test application. Thus, by identifying data of interest, we can utilize *DroidScraper* to regenerate program control flow, give context to the data within the flow, and then reconstruct its members and inner members.

4.3 Challenges and Limitations

The primary limitation of any memory forensics techniques on mobile devices is the ability to acquire a memory image. The security design of smartphones limits user access to low-level components and as such almost all the available techniques currently used in practice either require rooting the device to run sudo-based utilities such as Memfetch or creating a custom ROM and/or custom kernel module like Lime to acquire the entire RAM image [36]. Other hardware-based solutions are also available using JTAG [21] and recently the work of [39] explored the use of recovery partitions for acquiring device memory images.

Other limitations of *DroidScraper* include dead Objects that have been deallocated but remain in an allocated region. Although *DroidScraper* has an almost 90% recovery rate, the remaining 10% of unrecovered objects can still pose some limitations in identifying or giving context to data of interest. Also, when garbage collection occurs, regions with less than 75% live percentage are collected, and every object in this region is deallocated. While from our observations the presence of a large corpus of objects needed for rendering GUI and other vital user data often prevents a regions' live data from inching below 75%, there is no obvious way to prevent GC from happening.

5 Related Literature

Volatile memory or RAM is a core component of modern computing devices. Every application requires some chunk of available memory to layout its code, data, and other resources at runtime. As instructions are executed, the runtime environment will consistently create new allocations, update existing ones, and deallocate unused ones, as needed. This process makes the RAM a hive of potentially interesting forensic evidence. Furthermore, mobile phones are highly dynamic execution environments, often driven by external events and user interactions, resulting in many sensitive forensic artifacts appearing *only* in volatile memory. As such, in this research

work, we leverage the idea of memory forensics to develop a system that recovers and reconstructs remnants of in-memory application data from Android userland address spaces.

Traditional memory forensics has focused largely on analyzing content in kernel space [8, 9, 16]. These techniques, while effective in recovering important artifacts such as running processes, network sockets, etc., fall short in many ways. Because of the architecture of Android and other object-oriented design models, focusing solely on kernel-based memory analysis fails to target key process components such as *ActivityInfo*, *ActivityThread*, *PackageManager* and other objects like decrypted *HttpRequests*, *Cursor*, *StringBuilder*, and *Arrays*.

Thus more recent memory forensics research has explicitly targeted managed runtime recovery efforts as presented in the work of [15, 25, 28, 35]. These techniques are designed to recover and reconstruct data allocated on a per-process basis. In 2011, Case presented the first approach for examining the contents of the Android Dalvik Virtual Machine (DVM) [15]. This work was extended to cover a newer version of Android DVM in [25]. Soares developed a technique for extraction and analysis of contents in the Android ART runtime allocated using the new RosAlloc memory management scheme [35]. Much like [15, 25, 35], *DroidScraper* is also an Android runtime-based recovery technique that specifically targets the recovery and reconstruction of objects allocated using the Region-based memory management. The Region-based memory allocation is the default memory management scheme included in the latest revision of libart beginning 2017 [17]. A prior object recovery effort targeting Region-based memory management called RecOOP was presented in the work of Pridgen et al. [28, 29]. Unlike *DroidScraper* which is designed for the Android ART, RecOOP was specifically designed for HotSpot Virtual Machine (JVM). In addition to the obvious architectural differences, the memory allocation scheme and its corresponding garbage collection mechanisms, which determine the base runtime data structures and the layout of objects, are entirely different.

Other widely adopted memory forensics techniques in practice include generic memory scanning with utilities such as *strings* and *jbgrep*. These methods are often employed for identifying textual data in allocated or insecurely deallocated memory spaces [26, 40]. However, due to various program obfuscation techniques and the general complexity of programs, the data of interest may not be laid out sequentially. In addition, considering that these techniques are oblivious of the memory allocation scheme, the recovered data often lacks context. With *DroidScraper*, the recovery and reconstruction effort is designed based on a specific Android memory allocation scheme. This system identifies and decodes crucial runtime data structures that hold the definitions, metadata, and accounting information of allocated objects. Other specialized memory scanning techniques have targeted more specific data structure other than strings [7, 14, 18, 23, 24, 27, 34, 38]. These

plication, these objects can range from as little as 2000 to as many as 250,000 objects. With such large sets of objects, looking for forensically interesting data can be a tedious process. Although the layout of the region-based allocation helps tremendously in tracing and predicting control flow, a more automated approach to program reconstruction is needed. Thus as part of our future work, we will develop an automated system that can reconstruct an application's components by mapping the allocated objects to the code section of the application. This automated process will help us generate a proper execution path, reconstruct the GUI, and trace other program segments, such as background services. Furthermore, we are currently working to extend *DroidScraper* to cover Android 9 and 10.

7 Conclusions

As mobile devices continue to evolve, program analysis remains crucial for forensics investigations. From cybercrime to malware and vulnerability analysis, userland memory forensics can provide a better alternative to traditional techniques especially in multi-stack architecture. In this paper, we presented *DroidScraper* - a userland in-memory object recovery and reconstruction system that recovers the remnant of runtime artifacts from Android process memory space. The evaluation of *DroidScraper* has shown that it can recover in-memory data allocated using Android's region-based memory allocation with a recovery percentage of almost 90%. In addition, *DroidScraper* can reconstruct and give context to the extracted objects, which in practice can be utilized for detecting evidence of file and network activities, database accesses as well as recovery of cryptographic keys.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant #1703683 and #1850054.

References

- [1] Android 8.0 ART Improvements. <https://source.android.com/devices/tech/dalvik/improvements>. [Online; accessed 03-January 2019].
- [2] Android Debug Bridge (adb). <https://developer.android.com/studio/command-line/adb>. [Online; accessed 16-March 2019].
- [3] Concurrent and Parallel Garbage Collection. <http://www.cs.umd.edu/class/fall2013/cmsc433/lectures/concGC.pdf>. [Online; accessed 25-February 2019].
- [4] Concurrent Mark Sweep (CMS) Collector. <https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html>. [Online; accessed 11-December 2018].
- [5] Genymotion Desktop. <https://www.genymotion.com>. [Online; accessed 10-January 2018].
- [6] Memfetch. <https://github.com/citypw/lcamtuf-memfetch>. [Online; accessed 17-March 2018].
- [7] Memparser Analysis Tool by Chris Betz. <http://old.dfrws.org/2005/challenge/memparser.shtml>. [Online; accessed 15-March 2019].
- [8] Rekall Forensics. <http://www.rekall-forensic.com>. [Online; accessed 03-March 2019].
- [9] The Volatility Framework. <https://www.volatilityfoundation.org>. [Online; accessed 15-January 2019].
- [10] Volatility Command Reference. <https://github.com/volatilityfoundation/volatility/wiki/Command-Reference#memdump>. [Online; accessed 21-March 2018].
- [11] Android. Debugging ART Garbage Collection. https://source.android.com/devices/tech/dalvik/gc-debug#invalid_root_example, 2015.
- [12] AndroidXRef. Androidxref oreo 8.0.0_r4. http://androidxref.com/8.0.0_r4/xref/art, 2018.
- [13] Rohit Bhatia, Brendan Saltaformaggio, Seung Jei Yang, Aisha Ali-Gombe, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. Tipped Off by Your Memory Allocator: Device-Wide User Activity Sequencing from Android Memory Images. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'18)*, San Diego, 2018.
- [14] Chris Bugcheck. Grepexec: Grepping Executive Objects From Pool Memory. *Annual Digital Forensic Research Workshop (DFRWS)*, 2006.
- [15] Andrew Case. Memory Analysis of the Dalvik (Android) Virtual Machine. *Source Seattle*, 2011.
- [16] Andrew Case and Golden G Richard III. Memory Forensics: The Path Forward. *Digital Investigation*, 20:23–33, 2017.
- [17] Matheiu Chartier. Performance and Memory Improvements in Android Run Time (ART)(Google I/O '17), 2017.
- [18] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust Signatures for Kernel Data Structures. In *Proceedings of the 16th*

- [19] Google. Google Play. <https://play.google.com/store?hl=en>, year=2019.
- [20] GoogleGit. Git Repositories on Android. https://android.googlesource.com/platform/art/+master/runtime/gc/allocator_type.h, 2018.
- [21] Keonwoo Kim, Dowon Hong, Kyoil Chung, and Jae-Cheol Ryou. Data Acquisition from Cell Phones Using a Logical Approach. In *Proceedings of the World Academy of Science, Engineering and Technology*, volume 26, 2007.
- [22] Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani. Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–7. IEEE, 2018.
- [23] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. Dimsum: Discovering Semantic Data of Interest from Un-mappable with Confidence. In *in: Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. Citeseer, 2012.
- [24] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *NDSS*, 2011.
- [25] Holger Macht. Live Memory Forensics on Android with Volatility. *Friedrich-Alexander University Erlangen-Nuremberg*, 2013.
- [26] Tilo Müller and Michael Spreitzenbarth. Frost. In *International Conference on Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.
- [27] Nick L Petroni Jr, Aaron Walters, Timothy Fraser, and William A Arbaugh. FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory. *Digital Investigation*, 3(4):197–210, 2006.
- [28] Adam Pridgen, Simson Garfinkel, and Dan S Wallach. Picking up the Trash: Exploiting Generational GC for Memory Analysis. *Digital Investigation*, 20:S20–S28, 2017.
- [29] Adam Pridgen, Simson L Garfinkel, and Dan S Wallach. Present but Unreachable: Reducing Persistent Latent Secrets in the HotSpot JVM. In *Proceedings of the 50th Hawaii International Conference on System Sciences*. University of Hawai'i at Manoa, 2017.
- [30] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. GUITAR: Piecing Together Android App GUIs from Memory Images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 120–132. ACM, 2015.
- [31] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. VCR: App-agnostic Recovery of Photographic Evidence from Android Device Memory Images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 146–157. ACM, 2015.
- [32] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. Screen after Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images. In *USENIX Security Symposium*, pages 1137–1151, 2016.
- [33] Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse. In *23rd USENIX Security Symposium (USENIX Security)*, pages 255–269, 2014.
- [34] Andreas Schuster. Searching for Processes and Threads in Microsoft Windows Memory Dumps. *Digital Investigation*, 3:10–16, 2006.
- [35] Alberto Magno Muniz Soares and Rafael Timóteo de Sousa Jr. A Technique for Extraction and Analysis of Application Heap Objects within Android Runtime (ART). In *ICISSP*, pages 147–156, 2017.
- [36] Joe Sylve. Lime-Linux Memory Extractor. In *Proceedings of the 7th ShmooCon Conference*, 2012.
- [37] VirusShare. VirusShare.com - Because Sharing is Caring, 2017.
- [38] Robert J Walls, Erik G Learned-Miller, and Brian Neil Levine. Forensic Triage for Mobile Phones with DECODE. In *USENIX Security Symposium*, 2011.
- [39] Seung Jei Yang, Jung Ho Choi, Ki Bom Kim, Rohit Bhatia, Brendan Saltaformaggio, and Dongyan Xu. Live Acquisition of Main Memory Data from Android Smartphones and Smartwatches. *Digital Investigation*, 23:50–62, 2017.
- [40] Li Ying. Where in your RAM is "python san_diego.py"? - PyCon 2015. <https://www.youtube.com/watch?v=tMKXcc2-x08>, 2015.

