

**HARDWARE-ASSISTED PROCESSOR TRACING FOR AUTOMATED BUG
FINDING AND EXPLOIT PREVENTION**

A Dissertation
Presented to
The Academic Faculty

By

Carter Yagemann

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Cybersecurity and Privacy
College of Computing

Georgia Institute of Technology

August 2022

© Carter Yagemann 2022

HARDWARE-ASSISTED PROCESSOR TRACING FOR AUTOMATED BUG FINDING AND EXPLOIT PREVENTION

Thesis committee:

Dr. Wenke Lee, Advisor
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Brendan Saltaformaggio
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Weidong Cui
Partner Research Manager
Microsoft Research

Dr. Mustaque Ahamad
School of Cybersecurity and Privacy
Georgia Institute of Technology

Date approved: May 5, 2022

ACKNOWLEDGMENTS

I wish to thank my advisor Prof. Wenke Lee and my co-advisor Prof. Brendan Saltaformaggio for guiding and supporting me over the course of my work. I am grateful to them for the opportunities they have provided me.

I also want to thank Adam Bates for collaborating with me on my research. His domain expertise and guidance has proven invaluable.

I would like to acknowledge my peer Simon Chung for being a close collaborator and friend over the years. The outcomes of our countless discussions are reflected in the words of this dissertation.

I thank my colleagues Evan Downing, Yisroel Mirsky, Matthew Pruett, Mohammad Nouredine, Hong Hu, Yanick Fratantonio, and many others for their insights. No research is conducted in a vacuum, and their feedback helped shape the direction of my work.

Finally, I wish to acknowledge my friends and family for their encouragement and emotional support. Research is a trying journey riddle with uncertainty and doubt, and without them I would have given up long ago.

TABLE OF CONTENTS

Acknowledgments	iii
List of Tables	ix
List of Figures	x
List of Acronyms	xiii
Summary	xv
Chapter 1: Introduction	1
Chapter 2: Related Work	7
2.1 Control Flow Bending Attacks	7
2.2 Program Security Analysis	8
2.2.1 Symbolic Analysis	8
2.2.2 Fuzz Testing	10
2.2.3 Root Cause Analysis	11
2.3 Attack Reconstruction	12
2.3.1 Data Provenance Log Reduction	13
Chapter 3: ARCUS: Symbolic Root Cause Analysis of Exploits	14

3.1	Introduction	14
3.2	Overview	16
3.2.1	Real-World Example	18
3.2.2	Threat Model	19
3.3	Design	19
3.3.1	Symbolic Execution Along Traced Paths	20
3.3.2	“What If” Questions	21
3.3.3	Analysis Modules	22
3.3.4	Capturing the Executed Path	27
3.3.5	Snapshots & Memory Consistency	30
3.3.6	Performance Constraints	31
3.3.7	Vex intermediate representation (IR) Tainting	31
3.4	Evaluation	32
3.4.1	Accuracy on Micro-Benchmarks	33
3.4.2	Locating Real-World Exploits	36
3.4.3	Consistency to Advisories & Patches	39
3.4.4	Runtime & Storage Overheads	41
3.4.5	Case Studies	42
3.5	Discussion & Limitations	43
3.6	Conclusion	45
	Chapter 4: MARSARA: Preventing Execution Repartitioning Attacks	46
4.1	Introduction	46

4.2	Background & Motivation	50
4.2.1	Existing Defenses & Limitations	51
4.2.2	Insights & Lessons Learned	52
4.3	Execution Repartitioning Attacks	53
4.3.1	Spoofing Attacks	54
4.3.2	Delay Attacks	55
4.3.3	Crafting Real-World Exploits	56
4.4	Threat Model & Assumptions	57
4.4.1	Quantifying execution unit partitioning (EUP) Attack Surface	58
4.5	Design & Implementation	59
4.5.1	Intel Processor Trace	61
4.5.2	Offline Profiling	62
4.5.3	Online Auditing	64
4.5.4	Signature Match Validation	65
4.5.5	Execution Partitioning	68
4.6	Evaluation	69
4.6.1	Partition Validation Accuracy	71
4.6.2	Partitioning Attack Surface Reduction	73
4.6.3	Attack Investigation	74
4.6.4	Runtime & Space Overhead	75
4.7	Discussion	77
4.8	Conclusion	79

Chapter 5: Bunkerbuster: Proactive Bug Hunting and Reporting	80
5.1 Introduction	80
5.2 Overview	83
5.2.1 Real-World Example	83
5.2.2 Goals & Assumptions	85
5.2.3 Bug Class Definitions	86
5.3 Design & Implementation	87
5.3.1 Capturing & Filtering Traces	87
5.3.2 Symbolizing Memory Snapshots	89
5.3.3 Symbolic State Reconstruction	91
5.3.4 Use-After-Free & Double Free Bugs	93
5.3.5 Overflow & Format String Bugs	95
5.4 Evaluation	97
5.4.1 Bug Hunting in Real-World Programs	100
5.4.2 Comparing Prior Exploration Techniques	102
5.4.3 Comparing Prior Root Cause Techniques	104
5.4.4 Effectiveness of Exploration Techniques	106
5.4.5 Performance & Storage	107
5.4.6 Verifying the Root Cause Analysis	109
5.5 Limitations & Threats to Validity	109
5.6 Privacy & Legal Considerations	111
5.7 Conclusion	112

Chapter 6: Conclusion	117
Appendices	120
Appendix A: Bunkerbuster Algorithms & Additional Experiments	121
References	125
Vita	147

LIST OF TABLES

3.1	ARCUS Modules Summary	23
3.2	Symbolically Executing CISC Repeat Instructions	29
3.3	RIPE and Juliet Test Cases	34
3.4	System Evaluation for Real-World Vulnerabilities	37
4.1	Performance, accuracy, and storage overhead of MARSARA. Time captures the seconds to analyze and validate events. Baseline storage corresponds to running the Linux Audit framework and application log tracking without MARSARA. The low warnings are categorized by the model edge type for additional granularity.	70
4.2	Partitioning Attack Surface (PAS) for several real-world programs and defenses.	72
5.1	System Evaluation for Real-World Programs	98
5.2	Bunkebuster Vs. AFL & QSYM	103
5.3	Bunkebuster Vs. AddressSanitizer	104
5.4	Symbolic Root Cause Verification	108
A.1	Manually Verified APIs for Binary-Only Recovery	122

LIST OF FIGURES

3.1	CVE-2018-12327 in <code>ntpq</code> . A stack overflow occurs if there is no ‘]’ within the first 257 characters of <code>hname</code>	17
3.2	ARCUS architecture. The user program executes in the end-host while the ARCUS kernel module snapshots and traces it using Intel processor tracing (PT). When a runtime monitor flags a violation or anomaly, the data is sent to the analysis environment where symbolic states are reconstructed, over which the modules detect, localize, and report vulnerabilities.	17
3.3	Revisiting CVE-2018-12327 in more detail. Part of the snapshot and constraints tracked by ARCUS are shown on the right with registers and addresses substituted with variable names for clarity. PT is on the left.	20
3.4	CVE-2006-2025. Attacker controls the TIFF image and thus <code>tdir_count</code> , which can be used to overflow <code>cc</code> . ARCUS automatically finds a new constraint to prevent it.	24
3.5	Using the trace (left), with snapshot and PT packets, to recover the executed sequence of instructions (right).	28
3.6	Performance overhead and storage size of tracing the SPEC CPU benchmark. The average overhead is 7.21% and the geometric mean is 3.81%. The average trace size is 110 MB and the geometric mean is 38.2 MB.	40
3.7	Performance overhead and storage required to trace Nginx. The performance overhead is under 2% and the maximum storage is 1.6 MB per request.	40
4.1	Motivating example. The attacker sends a request ① that produces a seemingly normal response ②. However, it has actually employed a delay to trigger the payload ⑤ during a benign request ③ to exfiltrate a sensitive file ⑥, which is further obfuscated using spoofed log messages.	50

4.2	High level example of augmenting an exploit with spoofing to thwart data provenance. By adding a close socket system call, the call to execute Bash is partitioned into a different unit, isolating it from the attacker's exploit. . .	53
4.3	High level example of augmenting an exploit with delaying to thwart data provenance. By corrupting a code pointer, rather than directly executing the payload, a different unit can be exploited into triggering the next stage. . .	55
4.4	MARSARA architecture overview. An offline profiling phase yields a model of expected program behavior, which is used alongside execution traces and audit logs collected during online auditing to perform verified partitioning in post-forensic analysis.	60
4.5	Performance overhead for the real-world programs. The average is 8.7%. . .	75
4.6	Performance overhead for the SPEC CPU 2006 benchmark. The average is 7.21% and the geometric mean is 3.81%.	76
5.1	Source code pertaining to CVE-2017-11403 in GraphicsMagick, summarized. <code>ReadMNGImage</code> calls <code>ReadOneJNGImage</code> without realizing that it may free <code>image</code> , making Line 5,143 a double free bug for some paths. . .	83
5.2	Bunkerbuster architecture. End-hosts with PT-enabled kernel drivers collect and filter traces of the target program, forwarding novel segments to the analysis environment. Symbolic states are reconstructed and then expanded by exploration plugins. When a bug is detected, symbolic root cause analysis pinpoints the source and produces a report.	84
5.3	Binary-only scenario, with color added for clarity. The boxes show the usage of non-clobbered values. The first snippet reveals <code>foobar</code> has 3 arguments, the next reveals that the <code>RDI</code> argument is a <code>char</code> pointer (denoted <code>[s8]</code>), and the last reveals <code>RDX</code> is a code pointer (<code>[c]</code>).	90
5.4	control flow graph (CFG) created by the use-after-free (UAF) module for a real-world case (subgraph shown for brevity). Black edges are the path traced by PT and blue nodes are states the module discovered. The blue edges show a discovered path leading to a free, followed by the red path leading to a UAF bug (red node).	113

5.5	Counting loop example. Here the number of iterations of Line 12 depends on <code>length</code> , set by the loop starting at Line 5. When <code>foobar</code> passes <code>my_strcpy</code> a 4097 byte string, the register holding <code>length</code> (RCX) would normally become 4096 by Line 9. Our module overwrites RCX with a symbolic variable, allowing Line 11 to exit sooner, and then verifies the control hijack via a corrupted return pointer at Line 21.	114
5.6	Basic block coverage for traces forwarded to the analysis, cumulatively. . .	115
5.7	Percentage of unique basic blocks discovered using breadth-first search, depth-first search, and our proposed exploration techniques. Our techniques outperform the baselines across our entire dataset of 15 real-world programs.	115
5.8	Performance and storage for tracing the SPEC CPU 2006 benchmark. The average overhead is 7.21% and the geometric mean is 3.83%. The average trace size is 1,348 MB/min and the geometric mean is 602 MB/min. . . .	116
5.9	Overheads for tracing Nginx. The performance overhead is under 2% and the maximum storage is 1.6 MB per request.	116
A.1	Example root cause report for CVE-2018-12326.	122

LIST OF ACRONYMS

ASLR	address space layout randomization
CDG	control dependency graph
CFG	control flow graph
CFI	control flow integrity
COTS	commercial off-the-shelf
CVE	Common Vulnerability Enumeration
DF	double free
EDB	Exploit Database
EUP	execution unit partitioning
FN	false negative
FP	false positive
FPR	false positive rate
FS	format string
ICT	indirect control transfer
IDS	intrusion detection system
IP	instruction pointer
IPC	inter-process communication
IR	intermediate representation
JIT	just-in-time
JVM	Java virtual machine
MSR	model-specific register
NIST	National Institute of Standards and Technology

NMI non-maskable interrupt
NVD National Vulnerability Database
OS operating system
PAS Partitioning Attack Surface
PoC proof of compromise
PT processor tracing
ROP return-oriented programming
RVA relative virtual address
TCB trusted computing base
TIP target instruction pointer
TN true negative
TNT taken-not-taken
TP true positive
UAF use-after-free

SUMMARY

The proliferation of binary-only program analysis techniques like fuzz testing and symbolic analysis have led to an acceleration in the number of publicly disclosed vulnerabilities. Unfortunately, while bug *finding* has benefited from recent advances in automation and a decreasing barrier to entry, bug *remediation* has received less attention. Consequently, analysts are publicly disclosing bugs faster than developers and system administrators can mitigate them. Hardware-supported processor tracing within commodity processors opens new doors to observing low-level behaviors with efficiency, transparency, and integrity that can close this automation gap. Unfortunately, several trade-offs in its design raise serious technical challenges that have limited widespread adoption. Specifically, modern processor traces only capture control flow behavior, yield high volumes of data that can incur overhead to sift through, and generally introduce a semantic gap between low-level behavior and security relevant events.

To solve the above challenges, I propose *control-oriented record and replay*, which combines concrete traces with symbolic analysis to uncover vulnerabilities and exploits. To demonstrate the efficacy and versatility of my approach, I first present a system called ARCUS, which is capable of analyzing processor traces flagged by host-based monitors to detect, localize, and provide preliminary patches to developers for memory corruption vulnerabilities. ARCUS has detected 27 previously known vulnerabilities alongside 4 novel cases, leading to the issuance of several advisories and official developer patches. Next, I present MARSARA, a system that protects the integrity of execution unit partitioning in data provenance-based forensic analysis. MARSARA prevents several expertly crafted exploits from corrupting partitioned provenance graphs while incurring little overhead compared to prior work. Finally, I present Bunkerbuster, which extends the ideas from ARCUS and MARSARA into a system capable of *proactively* hunting for bugs across multiple end-hosts simultaneously, resulting in the discovery and patching of 4 more novel bugs.

CHAPTER 1

INTRODUCTION

Despite persistent efforts by the cybersecurity community to improve the security of computer software with better vetting (e.g., fuzz testing [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) and prevention (e.g., control flow integrity [11, 12, 13, 14, 15, 16, 17]), the rate at which software vulnerabilities are being discovered and exploited is accelerating. According to the National Vulnerability Database (NVD) maintained by the National Institute of Standards and Technology (NIST), the number of vulnerabilities publicly disclosed in 2020 was 6.1% larger than that of the prior year and 396% larger than 2010. Recently discovered vulnerabilities span a broad range of categories, from the classical memory corruption bugs that have plagued software engineering for decades (e.g., buffer overflow [18]) to more novel attacks exploiting newly introduced side-channels (e.g., Meltdown [19]).

One of the fundamental shifts driving this acceleration is the proliferation of *binary-only* program analysis techniques like binary fuzz testing (a.k.a. “fuzzing”) and binary symbolic analysis. For example, just within the past decade, researchers have released into the public domain well documented open source tools like American Fuzzy Lop [20] and angr [21], which have lowered the barrier to entry for applying advanced program analysis techniques to real-world software. Since these tools can operate directly on program binaries — without requiring source code or other intermediate developer artifacts — motivated analysts can search for bugs without requiring the software developer’s consent or cooperation. The result has been a cybersecurity gold rush, whereby actors ranging from students, to researchers, to third-party enterprises and startups, hunt for and disclose software bugs for fame and fortune.

In general, these events can be seen as a positive force that benefits society by raising public awareness and applying pressure on negligent developers to prioritize the security of

their software and remediate outstanding issues. However, this interpretation is only valid under the assumption that pressured developers and alert system administrators actually go on to fix and mitigate the bugs that are being publicly disclosed. This vital last step cannot be performed by the analyst reporting the bug because they are not as familiar with the buggy program (or the systems that depend upon it) and may not have access to its source code or distribution channels.

Unfortunately, the reality of the situation is not as clear-cut as researchers would like to believe. In fact, I observe that there is an automation gap in the steps between bug disclosure and bug remediation that is worsening, driving society towards a state where serious vulnerabilities are publicly known — but are not being fixed in a timely manner — presenting a window of opportunity to adversaries. As an example, of the 4,719 Linux kernel bugs automatically reported by the open source fuzzing service Syzbot, 964 have not been remediated (20.4%). The oldest open issue is almost 3 years old. One Linux developer even wrote¹ in a public thread, “We are drowning in [fuzzer] reports and just throwing them at us doesn’t really help anyone here anymore.” In short, even in sophisticated software projects and production environments, bugs are being reported faster than developers and system administrators can fix them.

Based on these observations, I propose the need to close this gap by creating new systems and techniques for not just finding software bugs, but also automatically localizing and preventing them. The new solutions that I am proposing must be able to keep pace with automated discovery techniques like fuzzing so that developers and system administrators can stay ahead of the curve and in control of their software security posture. This means that traditional analysis techniques like address sanitation [22], built on heavyweight technologies like dynamic binary instrumentation [23, 24] that cannot be reasonably deployed on production or end-user systems, are not sufficient.

Instead, the solutions that I propose make use of the advanced features available in

¹<https://lore.kernel.org/dri-devel/20200710103910.GD1203263@kroah.com/>

commodity processors to capture the telemetry from novel exploits and benign user activity necessary to efficiently find, explain, and prevent future attacks. In particular, I take advantage of the general availability of commodity hardware processor tracing (PT), which makes it possible to observe low-level execution events in software with superior efficiency, transparency, and integrity to previous instrumentation-based solutions like PIN and DynamoRIO. Using PT, tracing can directly bypass processor caches to write directly to physical memory accessible only to the operating system (OS) kernel (or hypervisor, if present), where it can then be securely forwarded for further analysis.

However, in order to make these solutions viable, several technical challenges arising from the trade-offs in PT's design have to be overcome. First, PT achieves its efficiency by limiting recording to only low-level control events, such as whether an instruction branched or not. Without additional information, such facts are insufficient to formulate informed security decisions, such as whether the recorded trace reflects a memory corrupting buffer overflow. Second, tracing captures the lowest possible level of program behavior, creating a semantic gap that must be overcome to model, detect, and analyze prevalent software vulnerabilities and exploit techniques. For example, analyzing UAF vulnerabilities requires knowledge of the program's heap memory layout and how it changes over time. It is not immediately obvious how to recover such information from a trace of instruction level control flow events. Third, the sheer volume of recorded data requires careful management and filtering in order to preserve the low overhead of the hardware, which is necessary to construct deployable systems. Specifically, while the underlying hardware can capture traces with less than 2% runtime overhead, the subsequent buffering and decoding of that data can raise the overhead to over 7% [25].

In this dissertation, I propose solutions to these technical challenges in the context of finding software vulnerabilities and preventing exploitation of deployed software. Specifically, my solutions center around the novel concept of *control-oriented record and replay*, whereby processor traces are contextualized with additional information (collected *a priori*

or alongside hardware tracing) to recover enough facts about the intermediate states in the program’s execution to detect, localize, and remediate software vulnerabilities and attempts to exploit them. This is in contrast to traditional *data-oriented* record and replay [26, 27] that records inputs and outputs between components (e.g., system calls between the user program and the kernel) to answer postmortem forensic questions.²

To demonstrate the efficacy and versatility of my approach, I first present a system called ARCUS, which is designed to assist system administrators in understanding intrusion detection system (IDS) alerts by automatically detecting and localizing bugs contained within recorded traces, yielding reports that can then be forwarded to developers for patching. ARCUS consists of an end-host kernel driver that snapshots a target process’ initial program state and then records its execution using PT. Upon the process being flagged by an IDS, for any reason, the snapshot and trace are forwarded to ARCUS’ analysis component, which may reside in the same system or a different one, where intermediate program states are reconstructed using symbolic execution. This linear sequence of symbolic states encode all possible input values to the program that can lead down the path recorded by PT. With these states in hand, ARCUS then uses an arsenal of bug-class-specific plugins to check the states for symbolic indicators of memory corruption bugs — specifically, overflow, UAF, double free, and format string bugs — and upon detecting one, uses a series of “what if” symbolic tests to localize the root cause, which can then be forwarded to developers as a human-readable report. My collaborators and I implemented ARCUS for Linux and evaluated it on 20 user programs, where it detected and localized 27 previously known vulnerabilities within our target bug classes. Surprisingly, ARCUS also discovered 4 novel vulnerabilities, leading to the issuance of Common Vulnerability Enumeration (CVE) advisors and official developer patches. ARCUS achieves this with a 7.21% recording overhead on the SPEC CPU 2006 standard benchmark.

Next, to expand the scope of my approach to beyond traditional memory safety bugs,

²Example: “Which email infected the user’s system?”

I present a system called MARSARA, which aims to protect EUP (used for system data provenance in post-mortem forensics) from a novel class of attacks that I propose, called execution repartitioning attacks. Data provenance is designed to analyze system audit logs to determine how data flows between objects (e.g., files, sockets) and subjects (i.e., processes), yielding a provenance graph for human analysts or downstream analysis systems. However, long-running processes (e.g., HTTP servers) can accumulate many data dependencies over their lifetime, which can cascade into what is referred to as the dependency explosion problem [28], whereby an event is wrongly associated with many prior events over which no data flow actually occurred. EUP addresses this by partitioning process events into autonomous units of work, thereby partitioning data provenance graphs into manageable subgraphs that more accurately reflect the true provenance. Unfortunately, I discovered that due to implicit assumptions made by existing EUP designs, a knowledgeable attacker can tweak their exploits to weaponize the EUP algorithm, arbitrarily controlling the partitioning of events to yield graphs designed to mislead and frustrate analysts. However, EUP is critical in making data provenance practical for real-world systems, and so I designed MARSARA, which uses additional data collected from PT and a novel program model representation to validate the underlying assumptions of EUP, discarding partitions that cannot be validated against low-level execution behavior, to ensure the resulting partitioned provenance graphs will capture the complete attack progression. MARSARA achieves this with 8.7% overhead over traditional auditing frameworks and a 2.82% increase in partitioned graph sizes, in the worst observed case.

Finally, I present a bug *hunting* and reporting framework called Bunkerbuster, which extends the ideas from ARCUS and MARSARA in order to *proactively* find and localize the root cause of software bugs — potentially before an attacker even has the opportunity to attempt exploitation in the wild. Bunkerbuster represents a significant step forward in generalizing my approach by preserving the concepts originally proposed in ARCUS while adding the capability to operate on *benign* traces collected from multiple end-hosts.

Finding bugs using only benign traces is significantly more challenging than analyzing traces that have already been flagged by an IDS, as performed by ARCUS. Bunkerbuster accomplishes this task by distributively filtering redundant information at each end-host, on-the-fly, while also incorporating new techniques for exploring nearby paths not directly executed in the traces. Bunkerbuster manages path explosion by conservatively pruning paths that cannot give rise to the target bug classes. This enables Bunkerbuster to proactively search for and localize overflow, UAF, double free, and format string bugs over *multiple* paths, whereas ARCUS is limited to *reactively* responding to individual IDS alerts flagging single paths. Compared to ARCUS on the same dataset, Bunkerbuster finds 4 additional novel vulnerabilities, which have been confirmed and patched by developers using Bunkerbuster's reports. Beyond the comparison to ARCUS, Bunkerbuster has continued to analyze traces of popular programs available on Debian, leading to the detection and remediation of 18 unique 0-day vulnerabilities to date.

CHAPTER 2

RELATED WORK

This chapter presents the related work to the proposed solutions, techniques, and target applications. The first section summarizes the related work in control flow bending attacks, which is the dominate methodology currently used by attackers to exploit software vulnerabilities. Due to its prevalence, control flow bending is an underlying motivator for ARCUS, MARSARA, and Bunkerbuster. The next section summarizes related work in program analysis in the context of software security. Specifically, this section covers symbolic analysis, which is utilized in all three systems, fuzz testing, which is another prevalent technique and common point of comparison for ARCUS and Bunkerbuster, and root cause analysis, which is an underlying theme across all three systems and an essential step in closing the automation gap between bug finding and bug remediation. The last section covers attack reconstruction in the context of data provenance, which is vital to understanding the problem scope for MARSARA.

2.1 Control Flow Bending Attacks

Control flow bending is the most prevalent way attackers exploit memory corruption vulnerabilities. From the attack perspective, we have seen a rise in sophistication from code injection, to code reuse (e.g., `ret2libc` [29]), to what is now the predominate exploitation technique: return-oriented programming (ROP) [30, 31, 32, 33, 34, 35]. For defenses, we have seen proposals based on randomization, including address space layout randomization (ASLR) [36], which have been successfully deployed in common systems. Unfortunately, there is still an ongoing battle between circumvention [37] techniques and better defenses [38, 39], motivating the need for systems like ARCUS, MARSARA, and Bunkerbuster that can help find, patch, and mitigate software vulnerabilities.

One commonly discussed technique for preventing control flow bending is control flow integrity (CFI) [40], which aims to ensure that the program adheres to a predetermined control flow model, thereby reducing the attacker’s ability to exploit paths unintended by the developer. Unfortunately, CFI has only seen limited adoption due to conflicts between performance and security. Coarse-grained solutions [41, 42] are fast and compatible with existing programs, but can be bypassed with careful bending [43]. Fine-grained approaches reduce the attack surface [44, 45, 46], but can still be bypassed, require source code, or rely on special hardware for performance [47]. In short, there is no ideal CFI solution to date [48], which motivates the need for alternative approaches and solutions.

In the context of MARSARA, control flow bending is one means by which attackers can conduct EUP attacks, but they can also utilize format string vulnerabilities and other orthogonal classes of bugs. I am the first to propose that online exploitation can explicitly target EUP to hinder forensic investigation. Prior work on bending may evade CFI, but leave the provenance chain intact, posing no hindrance on the attack investigation. Even when CFI is already deployed, MARSARA demonstrates an empirical benefit to defenders.

2.2 Program Security Analysis

There are many techniques for analyzing programs for security properties, depending on the available artifacts, user’s specific security goals, and other factors. In the context of binary-only program analysis, the dominate techniques for finding software bugs are symbolic analysis and fuzzing. Complimentary to these techniques is root cause analysis, which is a vital step to explaining discovered bugs and thwarted attacks to developers and system administrators so future repeat attacks can be prevented.

2.2.1 Symbolic Analysis

The earliest work in symbolic analysis demonstrated how executing programs with symbolic (as opposed to concrete) variables can aid in testing and debugging code [49]. As

solvers became more efficient, literature emerged for how to use symbolic execution to replay protocols [50] and detect vulnerabilities [51, 52, 53, 54]. Symbolic execution was also applied to side-channel research [55], firmware analysis [56], correctness of cryptography software [57], emulator testing [4] and automatic binary patching [58].

Much of this work focused on a subset of symbolic analysis called *concolic execution*. Specifically, rather than performing pure static analysis, which can get stuck on loops and string parsing, concolic systems leverage real executions for guidance [59, 60, 21], exploring outwards from the concrete executions to examine as many paths as possible [61, 62]. However, this can still lead to path explosion, especially as the analysis deviates further from the concrete execution. This motivated proposals for hybrid approaches [63, 64], which alternate between fuzzing and symbolic exploration to manage path explosion.

Although Bunkerbuster also explores nearby paths with guidance from concrete data to discover vulnerabilities [61], Bunkerbuster takes a unique approach to avoiding path explosion. Namely, rather than turning to hybrid techniques that incorporate fuzzing [63, 64, 65, 66], source code [62], or prior crashes [67] to find more *inputs* (that can still lead to path explosion during symbolic analysis), Bunkerbuster leverages execution *traces*. Bunkerbuster’s symbolic states enable it to detect a wide range of vulnerabilities (overflows, UAF, double free (DF), format string (FS)) whereas prior approaches are limited to a specific class, such as heap overflow [68]. Also, whereas many prior concolic systems have to operate in lockstep with the concrete environment [69, 68], Bunkerbuster’s tracing is completely decoupled from analysis, granting low overhead and a solution that can be deployed on production systems.

A less explored direction is single path concolic execution, which has proven useful in automatically generating exploits [70, 61, 71] and reverse engineering. The advantage of the single path approach is it sidesteps the issue of path explosion. Conversely, since only one path is analyzed at a time, it also relies heavily on receiving concrete executions that cover interesting program behaviors. ARCUS makes use of single path analysis, but

distinguishes itself by the way it leverages executed instructions and in its ability to derive root cause reports for developers and system administrators.

There is also work on specialized techniques for symbolic analysis, such as loop-extended symbolic execution (LESE) [52], that aim to achieve better coverage of the target program. Bunkerbuster shares the same technical challenge of improving code coverage by overcoming path explosion, but does so by making novel use of concrete data rather than relying on a novel type of symbolic lattice or grammar. Consequently, whereas approaches like LESE have only been evaluated on small CLI programs like Sendmail to uncover overflows, Bunkerbuster can handle large plugin-based GUI tools like GIMP and also finds instances of UAF, DF, and FS, which are outside the scope of related work. LESE cannot be extended to discover these classes because efficiently exploring loops is an orthogonal problem to detecting UAF, DF, and FS.

2.2.2 Fuzz Testing

An alternative approach to program analysis is fuzzing [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], which instead enumerates possible inputs to a program or API and checks for crashes as an indicator of buggy behavior. Some of the challenges with fuzzing are acquiring good seed inputs, reaching deep APIs, and identifying the nature of the bug when a crash occurs. The latter typically requires using additional tools like address sanitizers, which are slow and can only be applied in an offline context. Although Bunkerbuster does not rely on fuzzing, it addresses the same usability challenges. As a side note, although a key novelty of AR-CUS, MARSARA, and Bunkerbuster is their ability to record traces with low overhead on production systems, there is no technical barrier to having them also analyze inputs derived via fuzzing, should the user not have access to real end-hosts.

In recent years, researchers have recognized the inability of fuzzers to handle large complex programs that are slow to initialize or require GUI interaction. Several proposals have emerged to automatically generate fuzzer harnesses using source code [72, 73]. Un-

fortunately, these solutions still leave commercial off-the-shelf (COTS) and legacy binaries unaddressed. In response, a system called Winnie [74] was proposed, which uses execution traces (rather than source code) to automatically generate harnesses for Windows binaries. While this relates to how Bunkerbuster uses snapshots to decompose large programs into manageable components, 5% of Winnie’s harnesses had to be manually fixed to account for complex structures and callbacks while a large portion of the remaining 95% required minor manual tweaks. Bunkerbuster does not incur these shortcomings.

2.2.3 Root Cause Analysis

One of the earliest techniques for root cause analysis, delta debugging [75, 76], compares program states between successful and failing inputs to narrow down the set of relevant variables. Another popular approach is to use program slicing to extract only the code that contributes to the failure condition [77]. Delta debugging struggles to generate enough inputs in both classes to be effective while slicing requires tainting or lightweight replay to keep slices small and concise.

There is also failure sketching, which can handle security bugs like overflows [78], however most proposals based on this technique focus on race conditions because they are harder to reproduce [79]. Although races have serious security implications, they are not the focus of my proposed solutions, nor are they the only class hindering modern programs. There is also work on application layer root cause, including analysis of browser warnings and websites, trace-based pinpointing of insecure keys, and bug finding using written reports, which is orthogonal to ARCUS, MARSARA, and Bunkerbuster.

It is also possible to produce root cause explanations by triaging the many crashes produced by tools like fuzzers into buckets of related cases. Bucketing can be done symbolically [80], semantically with program transformations [81], or statistically [82]. These lines of research are spiritual successors to delta debugging and carry similar limitations. Namely, they can only analyze bugs that result in a crash and require multiple crashing

and non-crashing inputs to yield good explanations. The solutions I propose avoid this shortcoming.

Another direction is crash dump analysis [83], which aims to locate the cause of software crashes. However, while the motivations overlap with ARCUS and Bunkerbuster, the assumptions and scope do not. Crash dump analysis assumes bugs will manifest into crashes, but ARCUS and Bunkerbuster can detect non-crashing bugs. Crash dumps yield partial stack and memory information whereas my solutions use PT traces and snapshots. Data in crash dumps can be corrupt whereas the integrity of PT is protected by the kernel and hardware. These factors make the solutions I propose unique to those proposed in the context of analyzing crash dumps.

2.3 Attack Reconstruction

MARSARA is the first system to analyze binary events during system-level provenance collection and solve the challenges associated with protecting the integrity of EUP signature matches. A lot of work has been done to leverage provenance for forensic analysis [84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95], network debugging, auditing and troubleshooting [96, 97, 98, 99, 100], alert triage [101, 92], and intrusion detection and access control [102, 103, 104]. MARSARA complements all these systems by offering more secure EUP. MARSARA also complements the existing EUP systems such as BEEP [85], MPI [88], and MCI [84], which improve postmortem analysis by solving the problem of dependency explosion.

A large amount of research effort has focused on the generation and use of system call logs in forensic analysis, investigation, and recovery [105, 106, 107, 108, 109, 110]. However, none of the existing work focuses on defending postmortem analysis against execution repartitioning attacks. Provenance visualization techniques [111, 112] are also proposed to facilitate causality analysis. MARSARA can leverage these techniques to provide provenance graph summaries to system administrators, accelerating threat investigations.

Several systems [113, 114] have been proposed to detect the tampering of audit logs. Both Custos and SGX-Log use protocols that leverage Intel SGX and cryptographic data structures to protect audit log integrity. Several formats have also been proposed in the literature for storing data in a tamper-evident fashion, such as history trees [115, 116] and hash treaps [116]. These tamper-evident systems only detect if certain entries in the audit log are modified *after being committed*, which is orthogonal to the online threat addressed by MARSARA.

2.3.1 Data Provenance Log Reduction

The solution presented in MARSARA is orthogonal to provenance graph compression and deduplication techniques [117, 118, 119], since they compress the provenance graph instead of defending against EUP attacks. Many approaches [120, 117, 121, 86, 102, 93, 119, 122, 123, 87, 124, 125] are proposed to reduce the size of the audit log for long-term storage and to speed up after-the-fact forensic analysis. MARSARA can leverage those techniques to reduce its storage overhead.

LogGC [126] provides offline techniques to garbage collect redundant events that have no forensic value. Similarly, Winnower [124] and Process-centric Causality Approximation [127] both reduce log size by over-approximating causal relations. These techniques can be applied alongside MARSARA to decrease storage overhead. MARSARA can also leverage these approaches to speed up its analysis, which is left to future work.

CHAPTER 3

ARCUS: SYMBOLIC ROOT CAUSE ANALYSIS OF EXPLOITS

In this chapter, my collaborators and I present ARCUS, a system designed to detect and localize memory corruption bugs occurring in executions flagged by end-host IDS monitors, yielding a human-readable report that can then be forwarded to developers for remediation.

3.1 Introduction

End-host runtime monitors are designed to enforce security properties like CFI [47, 25, 11, 12, 13, 14, 128, 15, 16, 17] or detect anomalous events (system calls [129], segmentation faults [130, 131, 132, 133]). They can effectively halt attacks that rely on binary exploits and are seeing real-world deployment [134, 135]. However, these systems are designed to react to the *symptoms* of an attack, not the *root cause*. A CFI monitor responds to an invalid control flow transfer, not the buggy code that allowed the code pointer to become corrupted in the first place. A host-based IDS responds to an unusual sequence of system calls, without concern for how the program was able to deviate from the expected behavior model.

Traditionally, symptoms of an attack are easier to detect than root causes. Namely, it is easier to detect that the current state *has* violated a property than to diagnose what led to that violation. Unfortunately, this has led security professionals to adopt brittle stopgaps (e.g., input filters [136, 137, 138, 139] or selective function hardening [140]), which can be incomplete or incur side effects (e.g., heavyweight instrumentation [23]). Ideally, the developers that maintain the vulnerable program must fix the code and release a patch, but this creates a conundrum: *where is the bug that led to the detected attack?*

Unfortunately, the journey from a detected attack to a patch is rarely easy. Typical attack artifacts, like crash dumps [141] or logs [142, 143, 101, 144, 145, 94, 146, 147, 148,

95, 149], contain partial, corruptible data [150, 151, 152, 153, 154, 155, 156] with only the *detection* point marked. Concrete inputs may reproduce the symptoms in the production environment, but raise privacy concerns [141] and rarely work for developers [157, 158]. Worse still, developers are known to undervalue a bug’s severity [159] or prioritize other (better understood) issues [160].

Seeking a better solution, we propose a root cause analysis that considers “what if” questions to test the impact of particular inputs on the satisfiability of vulnerable states. The tests are vulnerability-class-specific (e.g., buffer overflows) and enable the analysis to localize vulnerabilities and recommend new enforceable constraints to prevent them, essentially suggesting a patch to developers. Analysis is conducted over the control flow trace of the program flagged by the end-host monitors, testing at each state “what if” any of the vulnerability tests could be satisfied. Notice that this is a divergence from the traditional mindset of replaying [24, 161, 162] or tainting [163, 139]. For example, instead of tainting a string that caused a stack overflow, the developers would most directly benefit from knowing which code block caused the corruption and what additional constraints need to be enforced upon it.¹

Armed with vulnerability-class-specific satisfiability tests, we turn our attention to efficiently collecting control flow traces in production end-hosts, which is challenging due to strict performance expectations. Interestingly, we find that readily available, hardware-supported, PT² offers a novel avenue towards efficient recording. Specifically, we leverage the capability of Intel PT to design a kernel module that can *efficiently capture* the control flow of user programs, storing and forwarding it to an analysis system if the end-host runtime monitor flags the process. Notably, this avoids recording concrete data or attempting to re-execute the program.

We have implemented a system called ARCUS³ — an automated framework for local-

¹Such analysis could also merge redundant alerts stemming from the same bug producing varying symptoms, improving alert fatigue [164, 165, 166].

²Available in Intel[®], AMD[®], and ARM[®] processors.

³Analyzing Root Cause Using Symbex.

izing the root cause of vulnerabilities in executions flagged by end-host runtime monitors. We have evaluated our ARCUS prototype using 27 exploits targeting real-world vulnerabilities, covering stack and heap overflows, integer overflows, allocation bugs like UAF and double free, and format string bugs, across 20 different commodity programs. Surprisingly, ARCUS also discovered 4 new 0-day vulnerabilities that have been issued 3 CVEs, demonstrating an ability to find neighboring programming flaws.⁴ ARCUS demonstrates impressive scalability, handling traces averaging 4,000,000 basic blocks from complicated programs and important web services (GIMP, Redis, Nginx, FTP, PHP), compiled from upwards of 810,000 source lines of C/C++ code. It also achieves 0 false positives and negatives in analyzing traces taken of the over 9,000 test cases provided by the Juliet and RIPE benchmarks for our implemented classes. We show that tracing incurs 7.21% performance overhead on the SPEC CPU 2006 benchmark with a reasonable storage requirement. To promote future work, we have open source ARCUS and our evaluation data.⁵

3.2 Overview

ARCUS' analysis begins when an end-host runtime monitor flags a running process for executing some disallowed or anomalous operation. Three classes of such systems are widely deployed today: CFI monitoring [47, 25, 11, 12, 13, 14, 128, 15, 16, 17], system call/event anomaly detection [129], and segmentation fault/crash reporting [130, 131, 132, 133]). However, ARCUS is not dependant on *how or why the process was flagged*, only that it was flagged. Notice that ARCUS must handle the fact that these systems detect attacks at their *symptom* and not their onset or root cause. In our evaluation, we tested alongside a CFI monitor [47] and segmentation fault handler, both of which provide delayed detection. ARCUS can easily be extended to accept triggers from any end-host runtime monitor.

⁴We reported new vulnerabilities to MITRE for responsible disclosure.

⁵<https://github.com/carter-yagemann/ARCUS>

```

1  int openhost(const char *hname, ...) {
2      char *cp;
3      char name[256];
4
5      cp = hname;
6      if (*cp == '[') {
7          cp++;
8          for (i = 0; *cp && *cp != ']'; cp++, i++)
9              name[i] = *cp; // buffer overflow
10         if (*cp == ']') {
11             name[i] = '\\0';
12             hname = name;
13         } else return 0;
14     /* [...] */

```

Figure 3.1: CVE-2018-12327 in ntpq. A stack overflow occurs if there is no ‘]’ within the first 257 characters of hname.

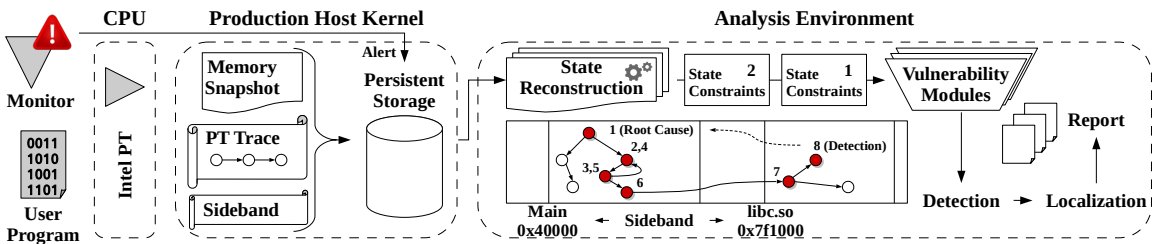


Figure 3.2: ARCUS architecture. The user program executes in the end-host while the ARCUS kernel module snapshots and traces it using Intel PT. When a runtime monitor flags a violation or anomaly, the data is sent to the analysis environment where symbolic states are reconstructed, over which the modules detect, localize, and report vulnerabilities.

3.2.1 Real-World Example

We will briefly walk through how to apply our proposed solution to a real vulnerability: CVE-2018-12327. We pick this example because the bug is concise and straightforward to exploit. Conversely, a case study containing thousands of intermediate function calls is presented in subsection 3.4.5. We will stay at a high level for this subsection and revisit the same example in greater detail in subsection 3.3.2.

CVE-2018-12327 is a stack overflow bug exploitable in `ntpq` to achieve arbitrary code execution. The vulnerability exists because there is no check for the length of the relevant command line argument. We will follow the *source code* in Figure 3.1 for simplicity, but the actual analysis is on *binaries*.

Assume the attacker can manipulate the arguments passed to `ntpq`, allowing him to overwrite the stack with a chain of return addresses that will start a reverse shell — a typical example of ROP. When `ntpq` starts, the ARCUS kernel module snapshots the program’s initial state and configures PT. The malicious input triggers the bug, and a shell is created. A runtime monitor determines that the shell spawning is anomalous and flags the program, causing the kernel module to send the snapshot and trace for analysis.

The analysis sequentially reconstructs a symbolic program state for each executed basic block. All inputs, including command line arguments, are symbolized. As the states are stepped through, a plugin for each implemented bug class checks for memory violations (subsection 3.3.3). Since the attacker’s input is symbolic, when the buggy code corrupts the stack, the return pointer will also become symbolic. The return causes the program counter to become symbolic, which is detected by the stack overflow module as a vulnerability.

ARCUS now switches to localizing the root cause. It identifies the symbolic instruction pointer in memory and finds the prior state that made it become symbolic (compiled from line 9). By examining the control dependencies of this state, ARCUS automatically identifies the guardian basic block that decides when the relevant loop will exit (compiled from line 8). ARCUS determines the loop could have exited sooner and checks what would

happen if it did (the “what if” question, elaborated on in subsection 3.3.2). ARCUS verifies that this alternative state does not have a symbolic return pointer, compares the resulting data constraints to those in the compromised program state, and spots the contradiction — a special delimiter character at a particular offset of an input string. It uses this to automatically recommend a new constraint to enforce at the guardian to fix the overflow.

As output, the human analyst automatically receives a report containing: 1) the basic block that corrupted memory, 2) the guardian that failed to protect against the exploit, and 3) a recommended fix for the guardian.

3.2.2 Threat Model

We consider attacks against user programs and assume that the kernel and hardware in the production system are trustworthy, which is reasonable given that Intel PT is a hardware feature that writes directly to physical memory, bypassing all CPU caches, configurable only in the privileged CPU mode. This is consistent with prior security work relying on Intel PT [25, 167, 47, 168]. We do not alter user space programs in any way. The kernel module also provides a secure way to store and forward recorded data to an analysis system, which may be a separate server for extra isolation.

We expect attackers to target the production system’s programs, but not have direct access to the analysis. We focus on program binaries without assuming access to source code or debug symbols.⁶ Consequently, our approach cannot handle all *data-only* attacks (e.g., selectively corrupting a flag), which may require accurate type information. However, ARCUS can be extended in future work to incorporate this.

3.3 Design

ARCUS consists of two general components, shown in Figure 3.2. A kernel module snapshots the initial state of the monitored program and collects its subsequent control flow via

⁶However, we reference source code in our explanations and figures whenever possible for brevity and clarity.

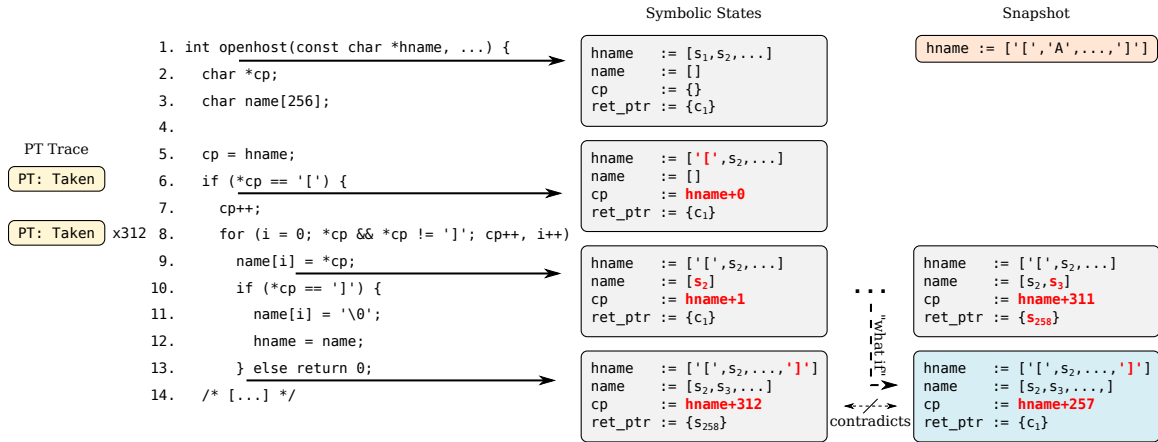


Figure 3.3: Revisiting CVE-2018-12327 in more detail. Part of the snapshot and constraints tracked by ARCUS are shown on the right with registers and addresses substituted with variable names for clarity. PT is on the left.

PT (subsection 3.3.4). The data is recorded to secure storage reserved by the kernel module and if an alarm is raised by a runtime monitor, it is transmitted to the analysis system, which may reside in a separate server. ARCUS is compatible with any end-host runtime monitor that can flag a process ID. We use an asynchronous CFI monitor [47] and a segmentation fault handler in our evaluation for demonstration.

The analysis is facilitated using symbolic execution with pluggable modules for different classes of bugs (subsection 3.3.3). This serves to reconstruct the possible data flows *for a single path*, which enables the system to spot vulnerable conditions (e.g., a large input integer causing a register to overflow) and consider “what if” questions to automatically find contradictory constraints that prune the vulnerable state (subsection 3.3.2). ARCUS then automatically recommends places in the binary to enforce these constraints so that developers can quickly understand and patch the root cause.

3.3.1 Symbolic Execution Along Traced Paths

Once an alarm is raised by a monitor, ARCUS will construct symbolic program states from the data sent by the kernel module. Our insight is to use symbolic analysis, but with special consideration to avoid its greatest shortcoming: state explosion. Put briefly, symbolic anal-

ysis treats data as a combination of *concrete* (one possible value) and *symbolic* (multiple possible values) data. As the analysis explores different paths in the program, it places *constraints* on the symbolic data, altering their set of values. In this way, symbolic analysis tracks the possible data values that can reach a program state.

We use symbolic analysis *not* to statically explore all possible paths, as is the typical use case, but to instead consider *all possible data flows over one particular path*. To do this, we symbolize all input data that could be controlled by the attacker (command line arguments, environment variables, files, sockets, and other standard I/O) and only build constraints for the path that was traced. This sidesteps the biggest problem with performing analysis in a vacuum — state explosion — by leveraging the execution trace leading up to the end-host runtime monitor’s alert.

3.3.2 “What If” Questions

Reasoning over symbolic data also enables ARCUS to consider “what if” questions, which is a key novelty in our root cause analysis. We now revisit CVE-2018-12327 (introduced in subsection 3.2.1) to show how ARCUS uses “what if” questions in detail. In Figure 3.3, part of the snapshot (orange box) and constraints tracked by ARCUS (grey boxes) are shown on the right. We substitute registers and memory addresses with variable names for clarity, but keep in mind that ARCUS operates on binaries without needing debug symbols or source code. A part of the PT trace (yellow boxes) is shown on the left with the source code in the center. We use square brackets to denote array contents and curly to list the possible values for a variable. The notation s_i is for unconstrained symbolic data and c_i is for concrete constants. `ret_ptr` is the return pointer.

ARCUS starts by replacing the attacker-controlled data in the snapshot with symbolic variables. `hname` points to a command line argument, which is why its contents become symbolic. As ARCUS symbolically executes the program, it follows the PT trace, which says to take the branch at line 6 and to repeat the loop 312 times. As the loop iterates,

`cp` increments, and `name` is filled with symbolic values copied from `hname`. By the time line 14 is reached, the return pointer has been overwritten with an unconstrained symbolic value. When the function returns, the program counter becomes symbolic, which means the attacker is capable of directly controlling the program's execution via crafted command line arguments. This is a vulnerability that triggers the stack overflow module in ARCUS to begin root cause analysis.

The full algorithm for this vulnerability class is presented in subsection 3.3.3, so for brevity we will focus on the “what if” question, which comes into play after ARCUS has located the symbolic state prior to `ret_ptr` being corrupted. ARCUS revisits this state and discovers there is another possible path where the loop exits sooner, which requires `cp ≤ hname+257` and the 257th character in `hname` to be `']'`.

What if this path were to be taken by the program? The resulting constraints would contradict the ones that led to the corrupted state, which requires `']'` to occur in `hname` no sooner than offset 258. Thus, by solving the “what if” question, ARCUS has automatically uncovered a fix for the vulnerability. In subsection 3.3.3, we cover how the module then determines where to enforce the new data constraints to make the recommendation more concise and practical. Note that even after applying the recommended fix, line 14 of the program is still reachable. However, because the newly enforced constraints contradict the compromised state, the code can no longer be executed in the context that would give rise to the observed overflow.

3.3.3 Analysis Modules

In this subsection, we expand on our methodology from subsection 3.3.1 and subsection 3.3.2 to describe how serious and prevalent classes of vulnerabilities can be analyzed using ARCUS. Each class has a refined analysis strategy and definition of root cause based on our domain expertise. In our prototype, each technique is implemented as a pluggable module, summarized in Table 3.1. Each module description concludes with a list of con-

Table 3.1: ARCUS Modules Summary

Module	Locating Strategy	Root Cause
Stack Overflow	Symbolic IP	Control Dep.
Heap Overflow	Symbolic IP	Control Dep.
Integer Overflow	Overflowed Reg/Mem	Overflow Site
UAF	R/W Freed Address	Control Dep.
Double Free	Track Frees	Control Dep.
Format String	Symbolic Arguments	Data Dep.

tents generated by ARCUS in its reports.

Stack & Heap Overflow The stack and heap overflow module focuses on analyzing control flow hijacking (recall that data-only attacks are out of scope, subsection 3.2.2), which requires the adversary to gain control over the program counter. As ARCUS reconstructs all the intermediate states along the executed path, the module checks whether the program counter has become symbolic. If it has, this means data from outside the program can exert direct control over which code the program executes, which is indicative of control hijacking.

From this point, the module looks at the previous state to determine what caused symbolic data to enter the program counter. Since hijacking can only occur at indirect control flow transfers, this previous state must have executed a basic block ending in a return, indirect call, or indirect jump. The steps we define for root cause analysis are: 1) identify the code pointer that became symbolic, 2) identify the basic block that wrote it, 3) find basic blocks that control the execution of the write block, and 4) test whether additional constraints at these blocks could have diverted the program away from the buggy behavior (i.e., by introducing a constraint that would contradict the buggy state).

To accomplish the first task, the module uses backward tainting over the previously executed basic block, lifted into an intermediate representation (IR), to identify the registers and then the memory address used to calculate the code pointer. The implementation details are in subsection 3.3.7. Once identified, the module iterates backwards through the previously reconstructed states to find the one where the data contained at the identified

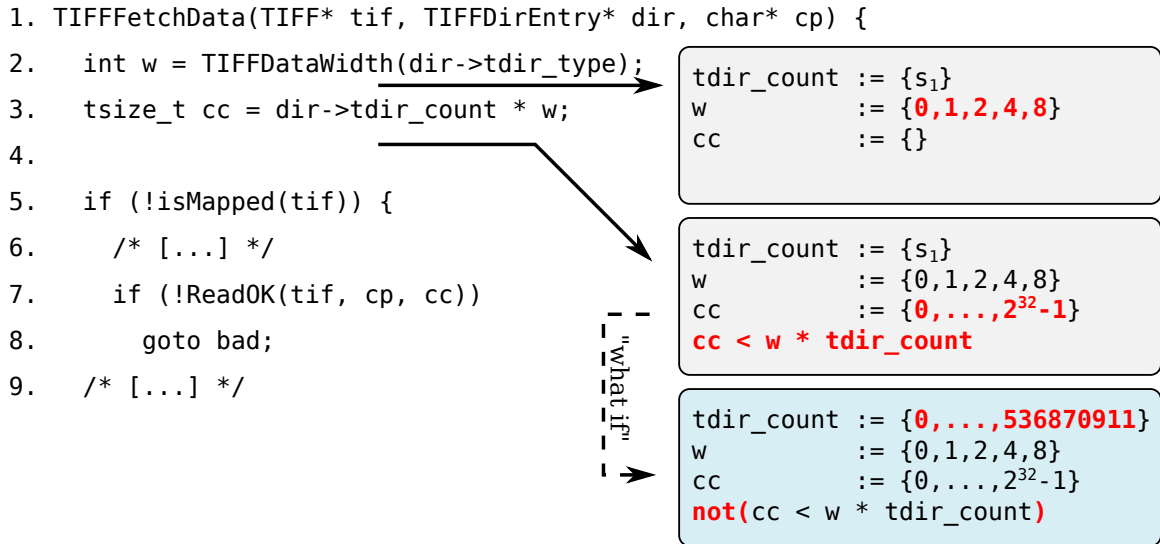


Figure 3.4: CVE-2006-2025. Attacker controls the TIFF image and thus `tdir_count`, which can be used to overflow `cc`. ARCUS automatically finds a new constraint to prevent it.

address changes, which reveals the state that corrupted the pointer. We coin this the *blame state*.

The next step is to identify the basic blocks that control it, which we refer to as *guardians*. The module uses forward analysis over the reconstructed states to generate a control dependency graph (CDG) and find them.⁷ If there are guardians for the blame state, the closest one is picked in terms of shortest path, and the prior state to execute this code is revisited to see if there exists another branch whose constraints contradict the blame state (solving the “what if” question from subsection 3.3.2). If contradicting constraints are found, ARCUS recommends enforcing them at the guardian. Otherwise, only the blame state is reported because an entirely new guardian is required.

For heap overflows, ARCUS needs to ensure that the heap objects are allocated exactly as they were in the flagged execution, which requires careful designing. We elaborate on the details in subsection 3.3.5.

Report: Blame state and, if found, the guardian to modify and new constraints to enforce.

⁷These graph algorithms are readily available in projects like angr.

Integer Overflow & Underflow. The two key challenges with detecting integer overflows and underflows (referred to collectively as overflows for brevity) are: 1) inferring the signedness of register and memory values in the absence of type information and 2) avoiding false positives due to intentional overflowing by developers and compilers.

To conservatively infer signedness, the module uses hints provided by instruction semantics (e.g., zero vs. signed extending [169]), and type information for arguments to known standard library functions (“type-sinking” [170]). If the signedness is still ambiguous for an operand, the arithmetic operation is skipped to err on the side of false negatives.

If an operation can overflow, according to the accumulated data constraints, the result register is flagged and subsequent stores and loads are tracked by the module. However, this is not immediately reported as a bug because the overflow may be intentional (second challenge). Instead, a bug is only reported if flagged data is passed to another function (i.e., following a `call` or `ret` instruction). The intuition is that when data crosses a function boundary, it is likely that the receiver did not consider the possibility of receiving overflowed integers, leading to violated assumptions and bugs. Prior work has measured this phenomenon [171].

Figure 3.4 illustrates how the module handles CVE-2006-2025, showing source code for clarity. In this case, an adversary can craft a TIFF image to overflow the register holding `cc` (defined at line 3) and pass it to `ReadOK` at line 7. Since `cc` is the product of two unsigned values, `cc < w * tdir_count` should not be possible, yet at line 4 the module discovers it is satisfiable, indicating `cc` can overflow. When `cc` is then passed to `ReadOK`, the module flags the bug.

To recommend a fix, the module solves the “what if” question: *what if the prior constraint was not satisfiable?* This requires an additional data constraint to be placed on `tdir_count`. The module includes this in its report along with the basic block that overflowed `cc` and the basic block that passed `cc` to `ReadOK`.

Report: Basic block and IR statement that overflowed the variable, recommended con-

straints, and basic block that passed the overflowed variable to another function.

Use After Free & Double Free. The UAF and double free modules monitor all calls to allocation and free functions, which we assume to know the semantics of in advance. When an allocation call is reached, the size argument is extracted and the returned pointer is evaluated to a concrete value to maintain a list of currently allocated buffers. When a free is reached, the corresponding entry is moved from the allocation list to a freed buffers list. Subsequent allocations can move freed entries back to the allocation list, maintaining mutually exclusive sets. For each state, addresses accessed by memory operations are checked against the freed list to detect the occurrence of UAF, upon which the module reports the starting address, size, and accessed offset. Similarly, the double free module detects freeing of entries already in the freed list. A CDG from the free site to the violating block determines and reports negligent guardians.

Report: Address, size, and offset (if applicable) of the violated buffer. The freeing and violating basic blocks, along with a partial CDG for the path between them.

Format String. Programming best-practice is to always create format strings as constant values in read-only memory. Unfortunately, buggy programs still exist that allow an attacker to control a format string and achieve arbitrary reads or writes. As the analysis reconstructs program states, this module checks for states entering known format string functions (e.g., `printf`) and verifies that: 1) the pointer to the format string is concrete, as it should be if it resides in read-only memory, 2) the string's contents are completely concrete, and 3) all the additional arguments point to mapped memory addresses. If any of these criteria are violated, the module knows data from outside the program can directly influence the format string function, which is a vulnerability.

Once located, the module locates the violating symbolic data in memory and examines prior states to find the one that wrote it. This is the blame state for this category of vulnerability. Since format strings should not be writable in the first place, no further analysis is

necessary.

Report: Contents of the symbolic string, the basic block that wrote it, and where it was passed to a format function.

3.3.4 Capturing the Executed Path

Analyzing the execution flagged by an end-host runtime monitor, which may reside in a different system, requires an efficient way of tracing the program without relying on instrumentation or binary modifications that could degrade performance or be targeted by the attacker. Our solution is to employ a kernel module to manage PT. For simplicity, we will focus on Intel PT, but other modern processors come with their own hardware implementations.

A trace captures the sequence of instructions executed by the CPU, which is large given that modern processors execute millions of instructions per second. To be efficient, Intel PT assumes that the auditor knows the memory layout of the audited program, which our kernel module prepends to the trace as a snapshot, shown on the left side of Figure 3.5 as grey packets. The kernel module also captures and inserts dynamically generated code pages between PT data, allowing complex behaviors like just-in-time (JIT) compilation to be followed. With this, all the auditor needs from the PT hardware is which path to follow when a branch is encountered, shown on the left in blue. For conditional branches, a single *taken-not-taken* bit is recorded. For indirect control flow transfers (return, indirect call, and indirect jump) and asynchronous events (e.g., interrupts, exceptions), the destination is recorded.

Intel PT is configured using model-specific registers (MSRs) that can only be written and read while the CPU is in privileged mode. Since only the kernel executes in this mode, only it can configure Intel PT. The trace is written directly into memory at *physical* addresses specified during configuration, meaning the kernel can make this data inaccessible to all other processes. Intel PT bypasses all caches and memory translation, which mini-

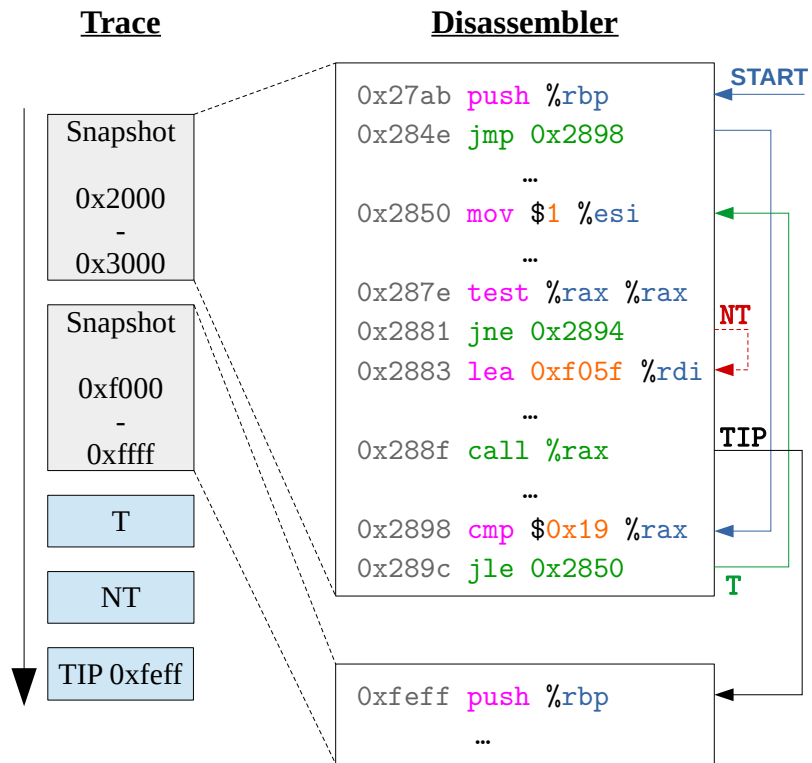


Figure 3.5: Using the trace (left), with snapshot and PT packets, to recover the executed sequence of instructions (right).

Table 3.2: Symbolically Executing CISC Repeat Instructions

Type	Common Usage	Strategy
<code>rep movs</code>	String Copy	Maximize Iterations
<code>rep stos</code>	Memory Initialization	Maximize Iterations
<code>rep cmps</code>	String Search (presence)	Symbolize Register
<code>rep scas</code>	String Search (offset)	Symbolize Register

mizes its impact on the traced program. When the buffer allocated for tracing is filled, the CPU raises a non-maskable interrupt (NMI), which the kernel module handles immediately so no data is lost.

Challenges with PT & Symbolic Execution. Intel PT tries to be as efficient as possible in recording the executed control flow. As a result, only instructions that produce branching paths yield trace packets, which excludes instructions for *repeat string operations* — used to speed up common tasks. For example, `rep mov` sequentially copies bytes from one memory location to another until a condition is met and `repnz scas` can be used as a replacement for `strlen`. These instructions encode an entire traditional loop into a single statement.

When memory is concrete, these complex instructions are deterministic, so Intel PT does not record how many times they “repeat.” This creates a problem for symbolic execution because if these instructions encounter symbolic data in memory or registers, the state will split and the trace will not have information on which successor to follow.

Our solution is to take the path that will most likely lead to a vulnerability, which depends on the type of repeat instruction, shown in Table 3.2. Three repeat types are excluded (`ins`, `outs` and `lods`) because they are typically used by kernel drivers and not user space programs. For move (`movs`) and store (`stos`), the analysis follows the maximum possible iterations given the symbolic constraints to check for overflow bugs. For comparison (`cmps`) and scanning (`scas`), the analysis skips to the next instruction (i.e., it executes zero iterations) and symbolizes the result register. The constraints for this register depend on the instruction. For example, `repnz scasb` in 64-bit mode scans

memory, decreasing RCX by 1 for each scanned byte, until either RCX becomes 0 or the value stored in AL is encountered. The analysis therefore constrains RCX to be between 0 and its starting value.

3.3.5 Snapshots & Memory Consistency

Symbolic execution requires an initial memory state to start its analysis from, which can be created with a custom loader or from a snapshot. The distinction is usually minor, but ends up being vital for ARCUS because it has to follow the path recorded by PT, as opposed to generally exploring the program. We discover that snapshots *are essential* to ARCUS because native loaders have complicated undocumented behaviors that the custom loaders are likely to contradict, creating inconsistencies in memory.

One such discrepancy is in how they resolve *weak symbols*, which can be resolved to one of several possible locations depending on the execution environment. For example, `libc` contains a weak symbol for `memcpy`, which is resolved to point at the most efficient implementation for the processor model. By our count, out of the 2,211 function symbols in `glibc` version 2.28, 30% are weak symbols. Additionally, shared objects can choose to implement their own resolver functions, invoked by the loader, to decide values.⁸

Our solution is for the kernel module to save a concrete snapshot of the program's user space at its entry point — after the initial dynamic loading is complete — and whenever a new thread or process is created. This captures the environment variables, command line arguments, and current *program break* pointer, the latter of which is important for heap placement.

Allocation Consistency. Analyzing attacks requires special care with replicating the spacing and absolute position of dynamically allocated buffers. Inconsistencies could cause overflows between objects or exploited writes to not be reproducible in the analysis.

⁸Example: https://sourceware.org/glibc/wiki/GNU_IFUNC.

The solution is to capture the *program break* (`brk`) pointer in the snapshot, which marks the end of the program’s data segment. When functions like `malloc` do not have enough space to allocate a new buffer, they make a system call to move the break. Consequently, all dynamically allocated objects are placed relative to the starting position of the break. Therefore, by starting with the same break and following the trace, ARCUS can ensure a consistent layout.

3.3.6 Performance Constraints

We prioritize performance in our design, but acknowledge that storage is also a concern for long running programs, to which we create two policies. For task-oriented workers, snapshots are taken as the kernel creates them and the oldest snapshots are discarded if a user defined threshold is exceeded. If a long living thread exceeds the threshold, a snapshot is retaken and the oldest data is discarded. This introduces potential false negatives due to truncation, but we demonstrate useful results with practical thresholds in section 3.4 and leave improvements to future work.

Since the analysis is performed offline only after an alarm is raised, we relax the performance requirements of the analysis system. Our evaluation shows real vulnerabilities are analyzed in minutes, which is sufficient for practical use.

3.3.7 Vex IR Tainting

The code in algorithm 1 shows how we perform backwards tainting on VEX IR lifted from binary code to identify the registers and memory addresses used to calculate a chosen temporary variable. We start by tainting the chosen variable and iterate backwards over the prior statements. Any registers used to store tainted variables (`Put`) become tainted. Whenever tainted variables are assigned a value (`WrTmp`), any registers, memory, or additional variables used to produce the value (i.e., operands) also become tainted. `EvalTmp` uses the symbolic execution engine to resolve memory address pointers. To taint multiple

basic blocks, we clear T between blocks while persisting A and R .

Input: VEX IR statements S starting from last executed.
 Tmp n to taint initially.
Result: Addresses A and registers R used to calculate n .

```

 $A \leftarrow \emptyset$ 
 $R \leftarrow \emptyset$ 
 $T \leftarrow \{n\}$ 
foreach  $s$  in  $S$  do
  | if  $\text{Type}(s)$  is Put and  $\text{Type}(s.data)$  is RdTmp then
  | | if  $s.data.tmp \in T$  then
  | | |  $R \leftarrow R \cup \{s.register\}$ 
  | | end
  | end
  | if  $\text{Type}(s)$  is WrTmp and  $s.tmp \in T$  then
  | | foreach  $a$  in  $s.data.args$  do
  | | | if  $\text{Type}(a)$  is Get then
  | | | |  $R \leftarrow R \cup \{a.register\}$ 
  | | | end
  | | | if  $\text{Type}(a)$  is RdTmp then
  | | | |  $T \leftarrow T \cup \{a.tmp\}$ 
  | | | end
  | | | if  $\text{Type}(a)$  is Load then
  | | | |  $A \leftarrow A \cup \text{EvalTmp}(a.address)$ 
  | | | end
  | | end
  | end
end

```

Algorithm 1: Tainting algorithm to obtain the registers and addresses used to calculate a VEX IR temporary variable.

3.4 Evaluation

We aim to answer the following questions in our evaluation:

1. *Is ARCUS accurate at detecting bugs within our covered classes?* We perform several micro-benchmarks with a ground truth set of over 9,000 test cases from the RIPE [172] and Juliet [173] suites. This ground truth allows us to verify that ARCUS can find root causes for vulnerabilities with 0 false positives and negatives (subsection 3.4.1).

2. *Can ARCUS locate and analyse real-world exploits?* We craft, trace, and have ARCUS analyze exploits for known CVEs and Exploit Database (EDB) vulnerabilities in real programs. ARCUS successfully handles 27 exploits and even discovers 4 new 0-day vulnerabilities, which we examine in additional case studies (subsection 3.4.2 and subsection 3.4.5).
3. *Are ARCUS' root cause reports consistent with real-world advisories and patches?* We manually verify that ARCUS' root cause reports are consistent with public disclosures and, where available, official patches (subsection 3.4.3).
4. *Is ARCUS feasible to deploy in terms of runtime and storage overhead?* We measure the performance and storage overheads of tracing programs using the SPEC CPU 2006 benchmark and Nginx (subsection 3.4.4).

Experimental Setup & Runtime Monitor Selection. We use 2 distinct servers to represent the production and analysis systems, each running Debian Buster and containing an Intel® Core™ i7-7740X processor, 32GB of memory, and solid state storage. To serve as end-host runtime monitors, we use an open source CFI system [47] and our own segmentation fault handler. The former is used for the exploits that leverage code reuse attacks and the latter for crashes. We pick this particular CFI monitor because it is asynchronous and only guarantees detection of control flow violations by the next system call, which requires ARCUS to handle traces containing activity past the initial exploit.

3.4.1 Accuracy on Micro-Benchmarks

Before deploying ARCUS on real-world programs, we evaluate on benchmark test cases where there is known ground truth for the location and behavior of every bug. This is necessary in order to measure false negatives (i.e., executions where a bug is triggered

Table 3.3: RIPE and Juliet Test Cases

Overall Results (Detection by ≥ 1 Strategies)					
RIPE	TP	TN	FP	FN	Acc.
BSS	170	170	0	0	100%
Data	190	190	0	0	100%
Heap	190	190	0	0	100%
Stack	260	260	0	0	100%
Juliet	TP	TN	FP	FN	Acc.
CWE-134	1,200	2,600	0	0	100%
CWE-415	818	2,212	0	0	100%
CWE-416	393	1,222	0	0	100%

By Locating Strategy (RIPE)					
Symbolic IP	TP	TN	FP	FN	Acc.
BSS	154	170	0	16	95.3%
Data	171	190	0	19	95.0%
Heap	154	190	0	36	90.5%
Stack	211	260	0	49	90.6%
Int Overflow	TP	TN	FP	FN	Acc.
BSS	60	170	0	110	67.6%
Data	60	190	0	130	65.8%
Heap	60	190	0	130	65.8%
Stack	150	260	0	110	78.8%

By Locating Strategy (Juliet)					
Symbolic Args.	TP	TN	FP	FN	Acc.
CWE-134	1,200	2,600	0	0	100%
Track Frees	TP	TN	FP	FN	Acc.
CWE-415	818	2,212	0	0	100%
R/W Freed Addr.	TP	TN	FP	FN	Acc.
CWE-416	393	1,222	0	0	100%

but ARCUS yields no report), which cannot be known for real-world programs.⁹ False positives are measurable by manually reviewing reports.

Dataset & Selection Criteria For the overflow modules (stack, heap, and integer), we use the complete RIPE [172] benchmark, which systematically exploits the provided test binary with different bugs (`memcpy`, `strlen`, etc.), strategies (ROP, code injection, etc.), and memory locations (stack, heap, etc.). We port the benchmark to 64-bit and manually create a second patched (bug-free) version of the test binary to measure false positives (FPs), false negatives (FNs), true positives (TPs) and true negatives (TNs). RIPE yields 810 working exploits in our environment.

RIPE does not contain tests for UAF, double free, or format string bugs. We address this shortcoming with the NIST C/C++ Juliet 1.3 suite [173], which contains 2,411 buggy and 6,034 bug-free binaries for CWE-416 (UAF), CWE-415 (double free), and CWE-134 (format string). These are all the test cases provided by Juliet for these CWEs.

Results As presented at the top of Table 3.3, ARCUS correctly analyzes all the test cases across all suites with no FPs or FNAs. That is, each TP is detected by at least 1 module and TN by none. We manually verify that the root cause reports for the TP cases correctly identify the buggy functions and the recommendations prevent the memory corruptions.

On closer investigation, we realize that ARCUS is so accurate on the RIPE cases because there are multiple opportunities for detecting overflows. For example, an integer overflow that corrupts a return pointer can be detected either by the integer overflow module when the register wraps around or by the stack overflow module when the pointer is overwritten. Detecting either behavior (or both) yields an accurate report. Based on this observation, we present the middle and bottom portions of Table 3.3, which separates the RIPE and Juliet results by the locating strategies from Table 3.1. For the modules tested

⁹If we knew the location and behavior of every bug in a real-world program, we could produce a new version that is guaranteed to be bug-free, which is obviously not possible with existing techniques.

by the Juliet cases, their capabilities do not overlap and yield the same numbers as in the overall table. For the strategies relevant to RIPE, we discover that the symbolic instruction pointer (IP) detection is 92.9% accurate, on average, whereas the integer overflow detection is 69.5%. The latter is expected given the challenges described in subsection 3.3.3, like inferring signedness in binaries. We observe that the accuracy is consistent across exploit locations for symbolic IP (4.8% variation), but less so for integer overflow (13%) where it performs better on stack-based tests. Since each strategy yields 0 FPs, their capabilities compliment each other, covering their individual weaknesses and enabling ARCUS to operate effectively.

3.4.2 Locating Real-World Exploits

With ARCUS verified to be working accurately on the micro-benchmarks, we turn our attention to real-world exploits.

Dataset & Selection Criteria We select our vulnerabilities starting with a corpus of proof of compromises (PoCs) gathered from the LinuxFlaw [191] repository and Exploit-DB [192], distilled using the following selection procedure:

1. First, we filter PoCs pertaining to bug classes not covered by our modules (subsection 3.3.3).
2. Next, we filter PoCs that fail to trigger in our evaluation environment.
3. Finally, for PoCs targeting libraries (e.g., `libpng`), we select a large real-world program that utilizes the vulnerable functionality (e.g., `GIMP`) for evaluation.

In total, we consider 34 PoCs pertaining to our covered bug classes (Step 1). Of these, 7 failed to trigger and were filtered (Step 2). The primary cause of failure is older PoCs written for 32-bit that cannot be converted to 64-bit. We decide to use `GIMP` for evaluating

Table 3.4: System Evaluation for Real-World Vulnerabilities

CVE / EDB	Type	Program	# BBs	Size (MB)	Δ Root Cause	Δ Alert	Located	Has Patch	Match
CVE-2004-0597	Heap	GIMP (libpng)	41,625,163	56.0	247	1	Yes	[174]	Yes [†]
CVE-2004-1279	Heap	jpegtoavi	67,772	0.65	26,216	1	Yes	No	-
CVE-2004-1288	Heap	o3read	74,723	0.65	33,211	1	Yes	[175]	Yes
CVE-2009-2629	Heap	nginx	300,071	1.10	28	33,824	Yes	[176]	Yes
CVE-2009-3896	Heap	nginx	283,157	1.10	59	16,821	Yes	[177]	Yes
CVE-2017-9167	Heap	autotrace	75,404	1.01	1,828	2	Yes	No	-
CVE-2018-12326	Heap	Redis	291,275	1.20	8	234	Yes	[178]	Yes
EDB-15705	Heap	ftp	260,986	0.85	19,322	2	Yes	No	-
CVE-2004-1257	Stack	abc2mtx	53,490	0.67	6,319	1	Yes	No	-
CVE-2009-5018	Stack	gif2png	90,738	1.09	1,848	1	Yes	[179]	Yes
CVE-2017-7938	Stack	dmitry	100,186	0.71	4,051	14,402	Yes	No	-
CVE-2018-12327	Stack	nipq	374,830	1.85	122,740	77,990	Yes	[180]	Yes
CVE-2018-18957	Stack	GOOSE (libiec61850)	65,198	0.71	94	30	Yes	[181]	Yes
CVE-2019-14267	Stack	pdfsurrect	128,427	0.66	83,123	1	Yes	[182]	Yes
* EDB-47254	Stack	abc2mtx	53,490	0.67	6,566	-	Yes	No	-
EDB-46807	Stack	MiniFtp	60,849	0.69	335	107	Yes	No	-
CVE-2006-2025	Integer	GIMP (libtiff)	78,419,067	55.0	3	8	Yes	[183]	Yes
CVE-2007-2645	Integer	exif (libexif)	67,697	0.97	1	7	Yes	[184]	Yes
CVE-2013-2028	Integer	nginx	809,977	2.00	1	25,268	Yes	[185]	Yes
CVE-2017-7529	Integer	nginx	1,049,494	1.10	2	780,404	Yes	[186]	Yes
CVE-2017-9186	Integer	autotrace	75,142	1.00	1	1	Yes	No	-
CVE-2017-9196	Integer	autotrace	74,695	1.03	1	203	Yes	No	-
* CVE-2019-19004	Integer	autotrace	132,302	1.02	1	-	Yes	No	-
CVE-2017-11403	UAF	GraphicsMagick	2,316,152	4.61	38	1	Yes	[187]	Yes
CVE-2017-14103	UAF	GraphicsMagick	2,316,133	4.61	38	1	Yes	[187]	Yes
CVE-2017-9182	UAF	autotrace	132,302	1.02	296	58,058	Yes	No	-
* CVE-2019-17582	UAF	PHP (libzip)	5,980,255	6.40	49	-	Yes	[188]	Yes
CVE-2017-12858	DF	PHP (libzip)	5,980,255	6.40	51	719	Yes	[188]	Yes
* CVE-2019-19005	DF	autotrace	132,302	1.02	57,859	-	Yes	No	-
CVE-2005-0105	FS	typespeed	127,209	0.74	1	1	Yes	[189]	Yes
CVE-2012-0809	FS	sudo	108,442	0.69	1	1	Yes	[190]	Yes
Average:			4,568,619	5.07	11,722	36,804	Yes		Yes

* New vulnerability discovered by ARCUS.

[†] Equivalent to applied patch.

image library CVEs, GOOSE Publisher for CVE-2018-18957, exif for CVE-2007-2645, and PHP for CVE-2017-12858 (Step 3).¹⁰

This yields PoCs targeting 27 unique vulnerabilities across 20 programs, covering a diverse range of multimedia libraries, client applications, parsers, and web services. Some are commonly evaluated in related work (e.g., libexif [62]), whereas others align with our motivation of protecting production servers (e.g., Nginx, FTP) and require ARCUS to handle more complex behaviors like multi-threading, inter-process communication, and GUIs (e.g., GIMP). For vulnerabilities that lead to arbitrary code execution, we develop the PoCs into exploits that use code reuse attacks like ROP. We create crashing exploits only as a last resort.

Results Table 3.4 shows that our system is able to successfully localize all 27 exploited vulnerabilities. Surprisingly, ARCUS also uncovers 4 new 0-day vulnerabilities — 3 issued CVEs — that are possible to invoke along the same control flow path, bringing the total count to 31. An example of how this occurs is presented in subsection 3.4.5. For exploited libraries evaluated in the context of a larger program (e.g., CVE-2004-0597), we show the traced program’s name alongside the library.

Table 3.4 includes the number of basic blocks recorded in each trace (“# BBs” column) and size in megabytes (“Size (MB)” column). Traces range from 53,000 basic blocks to over 78,000,000. Sizes are from 600 KB to 56 MB. The larger sizes correlate with programs containing GUIs and complex plug-in frameworks.

The “ Δ Root Cause” column lists how many basic blocks were executed between the state where ARCUS first identifies the vulnerability and its determined root cause point. The numbers vary substantially by class, with heap and stack overflows having distances upwards of 120,000 basic blocks whereas integer overflows and format strings are usually 1.

¹⁰We could not find larger programs in the Debian repositories that trigger CVE-2007-2645 or CVE-2018-18957.

“ Δ Alert” reports the number of blocks between where the runtime monitor flagged the execution and where ARCUS first detected the bug during analysis. In other words, the distance between the monitor alert and the ultimate root cause determined by ARCUS is the sum of “ Δ Root Cause” and “ Δ Alert.” Distances vary depending on which monitor was triggered and the overall program complexity. Some executions were not halted until over 700,000 blocks past the bug’s initial symptoms. 0-days found by ARCUS have no reported value since they were not detected by a monitor.

3.4.3 Consistency to Advisories & Patches

We evaluate the quality of reports for the real-world exploits by manually comparing them against public vulnerability advisories. For example, in CVE-2017-9167, the advisory states that AutoTrace 0.31.1 has a heap-based buffer overflow in the `ReadImage` function defined in `input-bmp.c` on line 337. Accordingly, we expect ARCUS’s root cause report to include the code compiled from this line.

When ARCUS provides a recommendation for extra constraints, we also manually verify that the reported guardian does in fact control the execution of the vulnerable code and that the recommended constraints would prevent the exploit. For example, the ARCUS report for CVE-2018-12327 recommends enforcing at the inner most loop in Figure 3.1 that a ‘]’ character occurs within the first 257 characters of `hname`, as explained in detail in subsection 3.3.2. This does prevent the exploit from succeeding, making the report satisfactory.

Some of the evaluated vulnerabilities have already been fixed in newer versions of the targeted programs. In these cases, we use the patch to further verify the quality of ARCUS’s reports by manually confirming that they identify the same code.

Results The results are shown in the “Located,” “Has Patch,” and “Match” columns of Table 3.4. All 31 reports correctly identify the exploited vulnerable code. There are patches

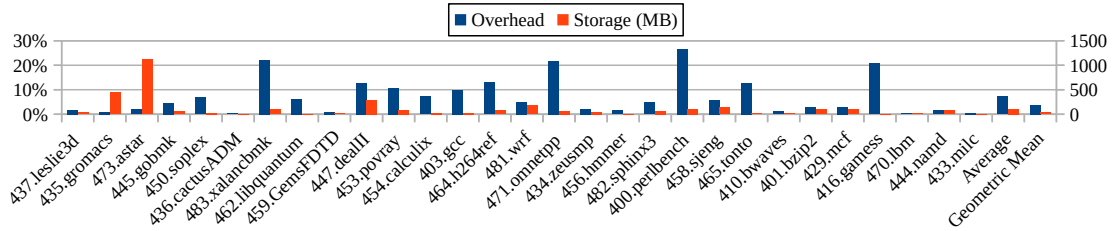


Figure 3.6: Performance overhead and storage size of tracing the SPEC CPU benchmark. The average overhead is 7.21% and the geometric mean is 3.81%. The average trace size is 110 MB and the geometric mean is 38.2 MB.

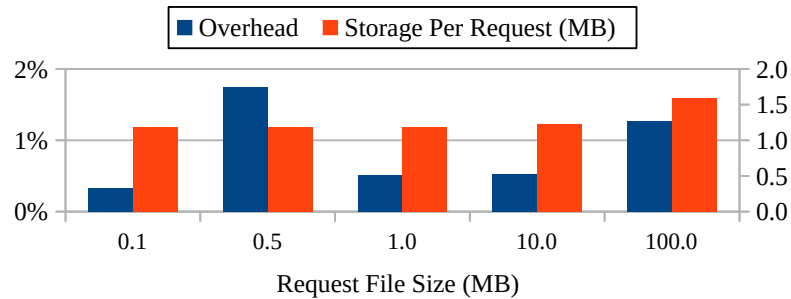


Figure 3.7: Performance overhead and storage required to trace Nginx. The performance overhead is under 2% and the maximum storage is 1.6 MB per request.

available at the time of evaluation for 5 of the 8 heap overflows, 4 of the 8 stack overflows, 4 of the 7 integer overflows, 3 of the 4 use after frees, 1 of the 2 double frees, and all 2 format string vulnerabilities. In all but 1 of the 19 official patches available for our tested vulnerabilities, the report generated by ARCUS is consistent with the applied patch. CVE-2004-0597 is a special case where a parent function calls a child using unsafe parameters, causing the child to overflow a heap buffer. ARCUS correctly identifies the vulnerable code, however the developers chose to patch the parent function, whereas ARCUS suggests adding checks inside the child. Both fixes are correct, so this report is satisfactory despite being slightly different from the official patch. 12 of the evaluated vulnerabilities are not officially patched at the time of evaluation.

3.4.4 Runtime & Storage Overheads

Dataset & Selection Criteria To evaluate the performance and storage overheads of ARCUS, we start with the SPEC CPU 2006 benchmark and a storage threshold of 100 GB. We pick this suite because it is commonly used and intentionally designed to stress CPU performance. Since our design requires control flow tracing, CPU intensive tasks are the most costly to trace. I/O tasks by comparison incur significantly less overhead due to blocking, which we demonstrate using Nginx with PHP. Consequently, we consider the SPEC workloads to represent realistic worst case scenarios for ARCUS.

To simulate long-running services and heavy workloads, we stress Nginx and PHP with default settings using ApacheBench (ab) to generate 50,000 requests for files ranging from 100 KB to 100 MB. This experiment also uses a 100 GB storage threshold.

Results Figure 3.6 shows the performance and storage overheads of tracing the SPEC workloads without the runtime monitors. The average overhead is 7.21% with a geometric mean of 3.81%, which is consistent with other Intel PT systems [47, 25]. A few workloads have overheads upward of 25%, which is also consistent with prior work and is caused by programs with frequent indirect calls and jumps. A workload yields 110 MB of data on average, which at our chosen storage threshold allows us to store 930 invocations of the program before old data is deleted. In the worst case, we can store 83 invocations.

For the Nginx with PHP stress test, shown in Figure 3.7, performance overhead is negligible at under 2%. ARCUS generates at most 1.6 MB of data per request, allowing us to store the past 64,000 requests given our 100 GB storage quota. We observe that file size has little influence over storage requirements, with the smallest file producing 1.2 MB of data per request and the largest producing 1.6 MB.

3.4.5 Case Studies

Discovering Nearby 0-Days ARCUS discovers that version 1.2.0 of `libzip` has a known vulnerability that can be altered into a new, previously undiscovered, 0-day.¹¹ Specifically, there is a buggy memory freeing function that maintains a flag in a parent structure to track whether a substructure has already been freed. Calling the freeing function twice on the same structure, without checking the flag, results in a double free (CVE-2017-12858), exploitable via a malformed input.

However, what was not previously known, but uncovered by ARCUS, is that further corrupting the malformed input can trigger a UAF, which has been assigned CVE-2019-17582. Specifically, after freeing the parent structure, invoking the freeing function again can cause it to access a flag that is no longer properly allocated.

Although both bugs reside in the same function, they are distinct — the known CVE double frees the child structure while the new bug inappropriately accesses the parent structure’s flag. A developer fixing the prior by more carefully checking the flag will not remediate the latter. ARCUS is able to find this new CVE because it considers all data flows over the executed path.

Vulnerabilities Cascading Into 0-Days An interesting example in `autotrace` demonstrates how a patch can address one bug, but fail to fix related “downstream” bugs, which gives ARCUS the opportunity to uncover new vulnerabilities. Version 0.31.1 contains a UAF vulnerability exploitable via a malformed input bitmap image header (CVE-2017-9182). Ultimately, ARCUS discovers two additional *downstream* vulnerabilities: an integer overflow (CVE-2019-19004) and a double free (CVE-2019-19005).

They all stem from a lack of input file validation. When the value of the `bits_per_pixel` field of the image header is invalid, after the known UAF, a previously unreported integer

¹¹Post-evaluation, we discovered that this vulnerability had been described in a previous bug report, however it was never issued a CVE and so we were unaware of it while evaluating ARCUS. Consequently, we were the first to report it to a CVE authority, resulting in the issuance of CVE-2019-17582.

overflow can occur as `autotrace` attempts to calculate the number of bytes per row in the `input_bmp_reader` function. ARCUS then discovers an additional double free that releases *the same freed buffer* the UAF accesses. In short, all 3 vulnerabilities are triggered by the same malformed header field, but each resides in a different code block, meaning a developer fixing one may overlook the others.

Vulnerabilities Over Large Distances Version 0.15 of the program `PDFResurrect` has a buffer overflow vulnerability (CVE-2019-14267) that can be exploited via a malformed PDF to achieve arbitrary code execution. When the function encounters a ‘`%%EOF`’ in the PDF, it scans backwards looking for an ‘`f`’ character, which is supposed to represent the end of ‘`startxref`’. As it scans, a register representing `pos_count` is incremented. An attacker can create a malformed PDF without a ‘`startxref,`’ causing `pos_count` to exceed 256 and overflow `buf`. This bug can be exploited to overwrite the stack and achieve arbitrary code execution.

What is interesting about this example is the vulnerable function loads *all* cross references before returning, any one of which could trigger the described overflow. This means thousands of references can be loaded between the corruption point and the return that starts the arbitrary code execution. In our crafted exploit, this distance is over 83,000 basic blocks (see Table 3.4) and includes almost *17,000* function calls. ARCUS successfully identifies the root cause of the vulnerability despite this distance.

3.5 Discussion & Limitations

False Negatives & Positives Prior work enumerates the possible sources of error in symbolic analysis [193], which are not special to ARCUS. ARCUS is a root cause analysis framework invoked in response to an end-host monitor’s alert, so it relies on the monitor to detect attack symptoms [194]. As described in subsection 3.3.3, some of the modules implemented in ARCUS can incur false negatives.

Only the integer overflow module can yield false positives, due to its combination of forward analysis and heuristics. The sole case we have encountered occurs in `libpng`, where an overflowed value is passed to another function, triggering a detection by ARCUS, but then the receiving function performs additional checks, preventing exploitation. Such patterns of checking for overflows in the receiving function (as opposed to the sending) are atypical [171].

Robustness Recommendations made by ARCUS are based on constraints built from a single execution path, meaning completeness cannot be guaranteed. Human developers are expected to implement the official patch using ARCUS’ recommendation as a starting point. Like most solutions that incorporate symbolic analysis, ARCUS is not well suited to building constraints within cryptography procedures, making the current prototype poorly suited for handling bugs within libraries like OpenSSL (e.g., CVE-2010-2939). However, this does not prevent ARCUS from analyzing programs that import such libraries — because the APIs can be modeled — and there are tailored analysis techniques [195] that ARCUS can adopt in future work. Similarly, we do not expect the current ARCUS prototype to perform well on heavily obfuscated binaries or virtual machines like the Java virtual machine (JVM). The kernel module can trace programs that dynamically generate code, including JIT compilation, however additional API modeling is required for angr to support web browsers. Conversely, ARCUS already successfully handles some complex programs (e.g., GIMP, 810,000 source lines of C/C++), demonstrating potential for future improvement.

Cross-Platform Support The current implementation of ARCUS is for x86-64 Linux, but with engineering effort it can support other platforms. Currently, the analysis uses VEX IR semantics, which is machine independent, and angr can lift several hardware architectures. Our “what-if” approach is also machine independent. The integer overflow module leverages some x86-specific semantics to help infer signedness, but it also contains general

techniques and can be extended in future work. The memory allocation and format string modules require the semantics for allocation and format string functions (e.g., `printf`, `malloc`). The current prototype supports typical libraries like `libc` and `jemalloc` and prior work proposes techniques for custom functions [196], which can be incorporated in future work.

The largest task is the tracing functionality, which requires an OS module. Although Windows[®] 10 has an Intel PT driver for tracing applications [197], it is not intended for third-party use and Microsoft[®] has not released any documentation. While it would be easy for Microsoft to implement ARCUS for Windows, for anyone else, it would require reverse engineering Microsoft’s driver [198].

3.6 Conclusion

This work presents ARCUS, a system for performing concise root cause analysis over traces flagged by end-host runtime monitors in production systems. Using a novel “what if” approach, ARCUS automatically pinpoints a concise root cause and recommends new constraints that demonstrably block uncovered vulnerabilities, enabling system administrators to better inform developers about the issue. Leveraging hardware-supported PT, ARCUS decouples the cost of analysis from end-host performance.

We demonstrate that our approach can construct symbolic program states and analyze several classes of serious and prevalent software vulnerabilities. Our evaluation against 27 vulnerabilities and over 9,000 Juliet and RIPE test cases shows ARCUS can automatically identify the root cause of all tested exploits, uncovering 4 new vulnerabilities in the process, with 0 false positives and negatives. ARCUS incurs a 7.21% performance overhead on the SPEC 2006 CPU benchmark and scales to large programs compiled from over 810,000 lines of C/C++ code.

CHAPTER 4

MARSARA: PREVENTING EXECUTION REPARTITIONING ATTACKS

In this chapter, my collaborators and I present MARSARA, a system designed to protect the integrity of EUP for data provenance against a novel class of attacks called execution repartitioning attacks.

4.1 Introduction

The complexity of interactions within modern computers makes it difficult to detect, prevent, and reverse unwanted system changes, such as in the case of an intrusion. A promising method of understanding suspicious events is causal analysis, in which system audit logs are transformed into a *data provenance graph* that encodes causal dependencies and historical relationships between subjects (processes) and objects (files, sockets, etc.) [108, 109, 90, 124, 199, 91, 105]. The resulting provenance graph can then be used by human analysts or monitoring tools for intrusion detection [102, 103, 104], forensic investigation [86, 87, 89, 90, 91, 92, 94, 95], and more [92, 96, 97, 98, 99, 100].

However, due to the noisy and complex nature of system interactions, provenance graphs are not always sufficient for investigating suspicious activity. Specifically, long-running processes can accumulate causal dependencies over time that become increasingly difficult to unravel; referred to as the *dependency explosion problem* [28] (a.k.a. false provenance). For example, consider a web server handling many requests in parallel. Due to the interwoven system calls invoked by multiple threads, data provenance will falsely conclude that all the files read during a request are causally related to all the currently connected remote IP addresses, which is excessive. However, multi-threading is not the only source of false provenance. Even in a single-threaded web server, a request response will link back to all previously handled requests, even though no actual data flow between

the most recent request and prior responses occurred.

To address dependency explosion, the research community has proposed execution unit partitioning (EUP) [200, 201, 84, 85, 86, 87, 88]. In EUP, audit log events are grouped at the sub-process level, subdividing a monolithic long-running process into autonomous units of work that are easier to trace in the graph. *Signatures* for identifying where to place partitions are typically generated during an offline profiling phase and may be encoded in several ways, such as a state machine of regular expressions to be matched against the audit log [84]. Continuing the web server example, a unit would be the code that processes a single request-reponse pair and the signature is the sequence of system calls and/or application level logs that the code emits. For example, the code might be expected to open a socket and record an access log entry with the source IP address, time, and requested URL at the start of its handling routine. Once a system call closes the socket, this marks the end of that unit. In this way, the data provenance system can distinguish between requests, correctly identifying which objects were accessed or modified on their behalf. In short, EUP is what makes data provenance viable for auditing real-world production systems.

However, all existing EUP solutions [200, 201, 84, 85, 86, 87, 88] make a dangerous implicit assumption, which we are the first to point out. Namely, they assume that *if the audit log events match the expected signatures, the underlying application must be performing the expected execution*. Ensuring this in real-world settings requires complete user program integrity, otherwise a low level bug (e.g., overflow, use-after-free) giving rise to *emergent execution* [33, 34, 30] or *out-of-bounds writes* [202] can produce erroneous signature matches. This in turn can add and remove partitions, reintroducing false dependencies and severing legitimate ones. Potentially, this would make it possible for the attacker to hide their steps from investigators while also framing innocent parties.

Would real-world adversaries be motivated to perform such an attack on EUP-enabled systems? Unsurprisingly, attackers already tamper with audit logs to cover their tracks [203, 204, 205, 206, 207]. Tampering is so prevalent that 72% of incident responders have en-

countered it during real investigations [208, 209], to which numerous *log integrity* defenses have been proposed [150, 151, 152, 153, 210, 154, 155, 156, 211, 115, 116, 114, 113, 212, 213, 214]. However, to our knowledge, all past solutions focus solely on an *offline* threat model, with tampering occurring *after* events are written to the log and are resting on a storage device. This is a distinctly different threat to what we just described, where changes to the user application’s *online* execution yields frustratingly incorrect analysis results.

In this work, we are the first to present two avenues for *online* tampering designed to frustrate provenance analysis without violating traditional notions of log integrity. At a high level, the first technique, *spoofing*, attempts to inject fake log events into the runtime by either maliciously invoking event-emitting code or by tampering with write buffers via an arbitrary write primitive (e.g., format string vulnerability). The second technique, *delaying*, introduces memory corruptions with deferred repercussions, allowing the current unit to finish normally, whereas a subsequent unit (with no discernible causal relationship to the prior) resumes the attack. To demonstrate practicality, we show how to create working examples starting from real-world CVE vulnerabilities.

In response to this new threat, the obvious solution would *seem* to be the deployment of known CFI techniques. However, we surprisingly discover that CFI can only prevent a subset of EUP-targeted attacks, specifically those built on control hijacking. Even then, depending on how subtle the hijack is (e.g., overwriting a code pointer to an arbitrary address versus another valid function), the overhead of enforcing sufficiently fine-grained CFI can be upwards of 47% [47]. Conversely, when data-only exploits are leveraged, prevention exceeds CFI’s scope [202].

Seeking a different solution, we propose a new defense to *validate* the placement of partitions. Specifically, given knowledge about the kinds of events certain parts of the code should yield (data flow), and their expected orderings (control flow), our solution compares runtime execution traces to audit logs to ensure consistency. If the attacker tries to change the ordering with control flow bending, or inject fake event data from another part of the

program, our defense will detect the discrepancy, disregarding the resulting events during partitioning to preserve the integrity of the provenance graph.

However, designing a solution around this idea raises several technical challenges. First, our system has to accurately determine which event sequence to expect for a given execution. Fortunately, rather than having to consider all possible executions, our system can focus on just the ones used offline to generate EUP signatures. Any program paths outside this scope were not intended by the EUP algorithm to yield partitions in the first place. To accomplish this, we propose a binary analysis that combines concrete execution traces with symbolic analysis.

Next, our solution has to collect the necessary additional runtime information to perform validation while minimizing additional overhead compared to prior (insecure) work. To this end, we propose a design that is compatible with the hardware PT available in commodity processors, which a kernel driver can securely control. We then overcome the challenge of connecting low level instruction sequences collected with PT to high level audit log events to accurately perform validation.

To evaluate our design, we implement a prototype for Linux, MARSARA¹, and extensively evaluate it on 14 real-world programs using expertly crafted exploits. MARSARA accurately partitions all the attack provenances while only reintroducing 2.82% of false dependencies, in the *worst case*, with an average performance overhead of 8.7% over traditional auditing frameworks. We also create a new metric for measuring the vulnerability of user programs to EUP attacks, Partitioning Attack Surface (PAS), and show that MARSARA removes 47,642 more gadgets than CFI on our real-world programs, on average per program. To promote further exploration of solutions to the new *online* log integrity problem, we have open sourced our code and data.²

¹Monitor Application Runtimes, Stop Arbitrary Repartitioning Attacks.

²<https://github.com/carter-yagemann/MARSARA>

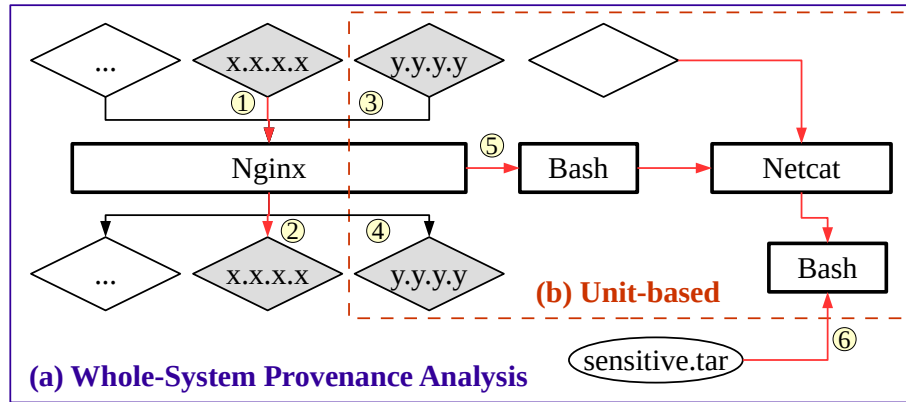


Figure 4.1: Motivating example. The attacker sends a request ① that produces a seemingly normal response ②. However, it has actually employed a delay to trigger the payload ⑤ during a benign request ③ to exfiltrate a sensitive file ⑥, which is further obfuscated using spoofed log messages.

4.2 Background & Motivation

Consider an Nginx web server with several worker processes, hosting a music website that the attacker aims to steal from. He starts by triggering CVE-2013-2028 using a maliciously crafted HTTP request, originating from the IP address $x.x.x.x$ in Figure 4.1. This causes a buffer overflow within one of the worker processes, allowing him to inject shellcode and corrupt a code pointer. However, instead of corrupting any code pointer arbitrarily to point at the shellcode, he cleverly overwrites a particular event handler³ that he knows the worker will not use to complete his request. Consequently, his HTTP request completes with no anomalous system calls or application messages. We call this novel setup a *delay attack*, which we elaborate on in section 4.3.

Later, a request from a benign IP, $y.y.y.y$, is received, causing the worker to access the corrupted code pointer and execute the shellcode. It starts by reading sensitive local files into a buffer. However, instead of immediately transmitting the data back to the attacker’s server, it first writes several forged log entries into Nginx’s access and debug logs to make it look like the current request has ended. This is another novel attack technique, which

³ngx_http_process_request_line

we coin *spoofing* and also elaborate on in section 4.3. With the spoofed messages inserted, the shellcode transmits the buffer of sensitive data back to the attacker and then the worker resumes normal operation.

4.2.1 Existing Defenses & Limitations

Intrusion Detection & Prevention Several aspects of the motivating attack make it difficult to detect or prevent at the onset. First, the initial exploit does not emit any anomalous system calls or application-layer events, rendering host-based defenses reliant on them ineffective. Obfuscation makes it impractical to detect the payloads on the network, and the shellcode may no longer be in memory by the time a symptom of the attack is observed. The corrupted code pointer requires fine-grained CFI to detect because its legitimate value is calculated dynamically during runtime and the necessary instrumentation can yield upwards of 47% execution overhead [47].

Whole-System Provenance Analysis Whole-system provenance tools [108, 109, 90, 124, 199, 91, 105] record system call level events to establish causal dependencies between objects and subjects, resulting in a provenance graph. Figure 4.1(a) shows the provenance graph for our motivating attack scenario *without EUP*. While the attacker’s IP address is contained in the provenance graph, we also see the false dependency problem described in section 4.1, where every open socket is associated with the exfiltrated data, making it inconclusive which connection instigated the attack and which request delivered the exploit and payload. At the same time, every file Nginx touched since its startup (e.g., configurations, temporary files) is also linked to the attack, making it inconclusive what was exfiltrated. In short, human analysts and automated systems do not have a clear picture for answering their forensic questions.

Unit-based Provenance Analysis EUP [126, 93, 84, 200, 88, 86] attempts to solve this dependency explosion problem by partitioning the execution of a long-running process into

autonomous *execution units* in order to provide more precise causal dependency graphs. While EUP is very useful when the adversary is oblivious to how it works, the delay and the spoofing attacks in our motivating example exploit it to further obfuscate what occurred.

Figure 4.1(b) shows the result. The delay attack successfully partitions away the request from the attacker ($x . x . x . x$), causing $y . y . y . y$ to appear as the origin point of the attack. Additionally, the spoofing employed by the shellcode causes the reading of sensitive files to be partitioned separately from its transmission, obfuscating what was actually exfiltrated.

It may be tempting to argue that if the corrupted worker could be identified, then all these problems would be solved, however this is not the case. Since Nginx reuses workers across requests, simply following its PID will wrongly associate unrelated events from prior and future requests, reintroducing false dependencies.

4.2.2 Insights & Lessons Learned

From the above discussion of the motivating example, we observe that data provenance systems that only analyze traditional audit log events will never be able to verify that the recorded, seemingly normal, patterns were emitted by normal program execution, and not by delay or spoofing attacks. Conversely, systems like CFI that rely purely on low level control flow will never be able to answer forensic questions that consider the data contents of reads and writes. Furthermore, we demonstrate in paragraph 4.3.3 that data-only attacks can also leverage delays and spoofing, which is outside CFI's scope to handle.

Instead, our solution is to leverage execution tracing and knowledge gathered during the offline profiling for EUP to recognize the manipulative events introduced by the attacker. In this example, knowing that the worker processing requests executed a program path (due to the delay attack) that was never seen during profiling indicates that it should not be isolated into its own partition. Subsequently, recognizing that several log messages originated from a previously unknown code location (the shellcode), indicates that they should not be considered during partitioning, preventing the attack from separating the

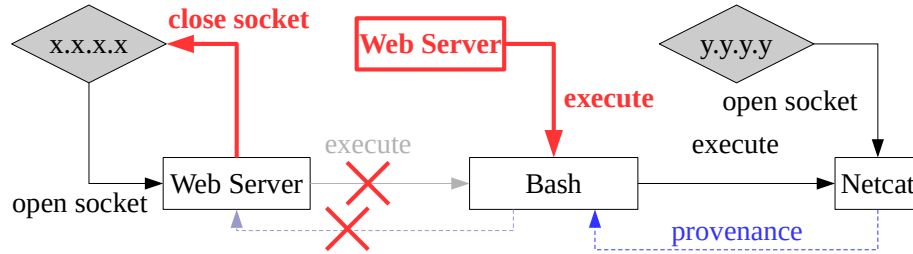


Figure 4.2: High level example of augmenting an exploit with spoofing to thwart data provenance. By adding a close socket system call, the call to execute Bash is partitioned into a different unit, isolating it from the attacker’s exploit.

sensitive file reads from network sends.

In section 4.3, we elaborate on how these novel delay and spoofing techniques can empower existing exploits to hinder provenance analysis. In section 4.4, we formalize the threat model based on our attack techniques and then our proposed defense is presented in section 4.5.

4.3 Execution Repartitioning Attacks

We propose a novel set of techniques for augmenting existing exploits to hinder defenses and forensic tools reliant on data provenance. Our techniques enable exploits to achieve their original goal while simultaneously obfuscating the true sequence of attack events from defenders, making it harder to determine where the attack originated from and what was done to the victim system. The techniques can be divided into two categories, *spoofing* and *delays*, which manipulate the audit events emitted from the target application prior to them being recorded by the auditing framework. Consequently, these techniques cannot be detected with traditional log integrity defenses [150, 151, 152, 153, 210, 154, 155, 156, 211, 115, 116, 114, 113, 212, 213, 214], which only detect changes after the logs are committed to storage.

4.3.1 Spoofing Attacks

Spoofing entails generating artificial system calls and application log messages in order to forge the necessary audit log events to satisfy an EUP signature. Typically, the attacker's exploit begins in the middle of an execution unit, with events linking the unit back to an ingress point. Figure 4.2 shows this for a web server example, with an open socket system call linking the current unit to the attacker's IP address.

Suppose the payload for the exploit is designed to start a reverse shell connected to a remote machine controlled by the attacker, thereby granting them access into the system. If the payload were triggered immediately, data provenance would trivially associate the resulting execute and open socket system calls to the current execution unit. Consequently, a system or human analyst wanting to investigate any of these events can recover the entire sequence using data provenance. For example, if the Netcat process is examined, a backward provenance query will reveal the attacker's IP address and the request used to compromise the web server. Similarly, a forward query will reveal the remote server used to issue commands and any data it exfiltrated.

What would happen if the payload closed the initial socket *before* invoking the execute system call? As it turns out, most existing EUP algorithms for data provenance will mark this as the end of the current execution unit and partition all subsequent audit log events into a new unit, as reflected in Figure 4.2.⁴ With the call to execute Bash now in a new unit, the previously described data provenance query will not include the attacker's IP address, nor contain the request carrying the exploit and payload. In summary, with just one added system call, the attacker has thwarted the ability for data provenance to recover the full attack sequence.

While spoofing is conceptually straightforward, signatures can require many events, all of which have to be spoofed in the correct order to successfully match a signature.

⁴The only exception we know of is BEEP [85] because it instruments programs with an explicit "end-of-unit" event, however this can also be spoofed to perform the attack.

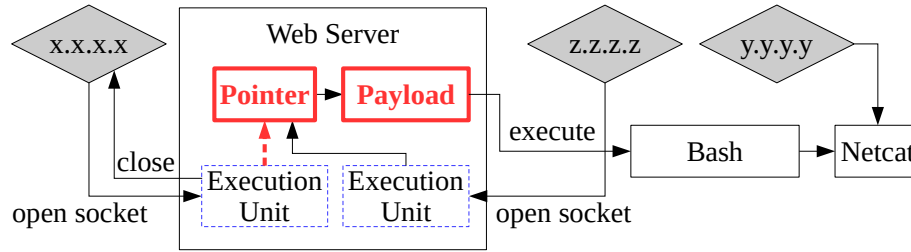


Figure 4.3: High level example of augmenting an exploit with delaying to thwart data provenance. By corrupting a code pointer, rather than directly executing the payload, a different unit can be exploited into triggering the next stage.

Continuing the previous example, for a real server like Nginx, simply closing a socket is not sufficient. There are also dozens of debug messages that have to be spoofed to create a valid signature. In section 4.6, we evaluate an exploit that uses CVE-2009-4769 to target `httpd`'s `toLog` method to conduct a successful attack.

Format string bugs warrant special mention, as they are particularly powerful for spoofing. For example, CVE-2012-0809 in `sudo` can be exploited to yield any string starting with the prefix “`sudo:` ”, making it very flexible for matching signatures. Interpreters that allow scripts to specify format strings (PHP: CVE-2015-8617, CVE-2016-4071) are also ripe for abuse in this manner.

4.3.2 Delay Attacks

Rather than forging fake events to create a partition, the attacker can alternatively augment their exploit to intentionally delay the manifestation of certain actions to later execution units, covertly spanning partitions in a way that will not be reflected in the data provenance. Figure 4.3 visualizes this at a high level, reusing the web server as an example. Rather than directly executing the payload, which would causally link the attacker’s IP address and request to the resulting reverse shell, the exploit instead corrupts a code pointer to point to the payload and then exits normally. When a subsequent (benign) request causes the corrupted pointer to be dereferenced, it will inadvertently trigger the next stage of the attack with no audit log events linking it back to the attacker’s request. This not only decouples

the attacker from the payload, but also frames a benign IP address as being the ingress point.

However, delays do not always require a memory safety violation. For example, event handling loops in many programs can encounter situations where a task must be deferred and rescheduled for handling at a later time (e.g., because a necessary resource is not yet available). Offline analysis can miss these alternate code paths during profiling, creating unintended delay attack primitives.

4.3.3 Crafting Real-World Exploits

Based on our techniques of spoofing and delaying, we present 3 working exploits against real-world programs to encompass the techniques an adversary can use to exploit repartitioning attacks. Our exploits are based on known CVEs, extended using our attack techniques to invoke erroneous data provenance results.

CVE-2013-2028 This CVE stems from a bug in Nginx’s handling of chunked HTTP requests and can be exploited to cause an out-of-bounds write. We use this to target Nginx with the *delay* technique. Specifically, we exploit the original stack overflow to change two local variables that are then used by the buggy function to perform a write, creating an arbitrary write primitive. We exploit this in turn to corrupt one of the program’s global code pointers, implementing the delay primitive. To simplify the payload, we make the program’s heap executable prior to the attack so that the malicious HTTP request can carry its own shellcode. In a real-world setting, the attacker could instead trigger the CVE multiple times to write a ROP chain into memory that corrupts the global pointer.

CVE-2004-0541 This CVE stems from a bug in one of Squid’s remote authentication modules, which can be remotely triggered to cause a buffer overflow. Our attack augments exploits for this CVE with the *spoofing* technique. Specifically, we trigger the overflow in its NTLM authentication child process to inject and trigger a ROP chain, which in turn

messages the logging daemon via an inter-process communication (IPC) channel to print arbitrary log strings. We use this spoof primitive to forge the necessary messages to complete a valid EUP signature, ending the current unit and starting a new one, and then trigger the payload, which is now causally disconnected from the attacker.

CVE-2009-4769 This CVE stems from multiple format string bugs in `httpd`, which can be triggered remotely by a HTTP request to perform arbitrary reads and writes. Specifically, the buggy logging procedure is intended to record details pertaining to the incoming HTTP request (timestamp, IP address, requested file, response code). However, by exploiting it with the spoof technique, an attacker can control the write to inject multiple seemingly legitimate entries into the log, thereby partitioning the attack across several bogus execution units with no causal dependencies. The exploit can then trigger a payload using arbitrary writes or leak data back to the attacker without creating a link to the malicious request.

4.4 Threat Model & Assumptions

Defender The defender's goal is to investigate an intrusion with the aid of a full-system data provenance framework. In order to handle complex real-world long-running programs, it relies on EUP, as is the norm [200, 201, 84, 85, 86, 87, 88]. Conversely, simple short-lived programs that do not incur dependency explosion can have all their events grouped into a single partition and do not require further consideration for this work. In accordance with prior work [200, 201, 84, 85, 86, 87, 88], partitioning signatures do not span multiple programs, so each can be analyzed independently. We assume kernel integrity and correct ordering of audit data, which are standard prerequisites in all full-system auditing [200, 201, 84, 85, 86, 87, 88]. We only consider EUP attacks and note that our proposed solution is compatible with existing approaches to offline tamper-evident logging [150, 151, 152, 153, 210, 154, 155, 156, 211, 115, 116, 114, 113, 212, 213, 214].

CFI has some capacity to coincidentally reduce the EUP attack surface by limiting the

range of unexpected *control* behaviors a program can exhibit. To account for this, we define a metric for quantifying attack surface reduction in subsection 4.4.1 and perform a comparison between CFI and our solution in the evaluation. Our findings show that our design offers more protection than CFI, *against EUP attacks*, across all 14 evaluated real-world programs, eliminating 47,642 additional delay and spoof gadgets per program.

Attacker The attacker’s primary goal is to take control of a target program in order to gain a foothold into the victim’s system. For brevity, we will consider a production server environment where the attack surface is an internet accessible service, such as a HTTP server. Since the attacker expects the defenders to be using an auditing framework that allows for data provenance, he is motivated to augment the attack with the techniques described in section 4.3 to make it as difficult as possible to uncover his activities.

The minimum prerequisite for the attacker to succeed is one vulnerability in the target program that enables control flow hijacking or arbitrary write, along with knowledge of the EUP algorithm being used and a copy of the target program so he can know the partitioning signatures in advance. However, to demonstrate the strength of our proposed defense, we will consider a significantly more powerful adversary who has a complete local copy of the victim system and access to an arbitrary read vulnerability in the target program, granting him complete knowledge of the remote program’s state and the ability to refine his attack to work on the first try, guaranteed. By demonstrating that our defense is able to correctly recover the complete attack provenance of this powerful adversary, we also demonstrate the ability to handle weaker, more realistically constrained attackers.

4.4.1 Quantifying EUP Attack Surface

In order to quantify the surface for EUP attacks and facilitate objective comparisons between defenses, we propose a new metric called PAS. The intuition behind PAS is to quantify how many audit-event-producing sites (e.g., system calls, application log writ-

ing procedures) are reachable from any point in the program based on the policy being enforced by integrity defenses. The more sites that are reachable from the current point in the execution, the more events an attacker can choose from to match a signature.

To measure PAS in real-world programs efficiently, given a graph model representing the enforced policy, we define audit-event-producing sites as nodes that invoke either a system call or write library function (e.g., `printf`). Thus, for each node n in policy N and node e in the set of audit-event-producing nodes E , PAS is defined as:

$$\frac{\sum_{n \in N, e \in E} r(n, e, \{E - e\})}{|N|} \quad (4.1)$$

where r is a function that returns 1 if e is reachable from n without going through any other node in E (i.e., $\{E - e\}$) and returns 0 otherwise. This check is relevant because going through another node in E produces a side-effect that the attacker does not desire. Ultimately, higher PAS values reflect a weaker defense that grants greater flexibility to the attacker.

4.5 Design & Implementation

The high level idea of MARSARA is to use control flow data and knowledge of event-producing code locations (i.e., what messages or system call parameters they can produce) to validate unit signature matches. Figure 4.4 shows our proposed design, which similar to prior work in EUP [200, 84, 88] consists of an offline profiling phase, an online auditing phase, and a post-forensic analysis.

During offline profiling, MARSARA records and analyzes PT traces of the target program, using a binary symbolic analysis, to identify important control and data flows along with possible starting points for execution units (subsection 4.5.2).

During online auditing, MARSARA records the program’s execution and stores it alongside the traditional audit log of system calls and application log messages (subsec-

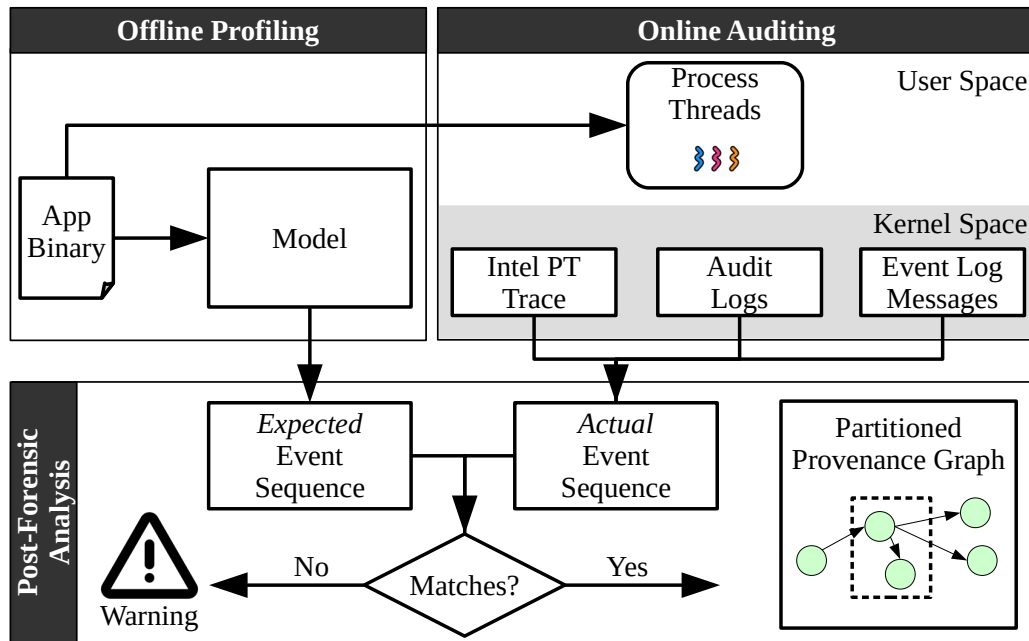


Figure 4.4: MARSARA architecture overview. An offline profiling phase yields a model of expected program behavior, which is used alongside execution traces and audit logs collected during online auditing to perform verified partitioning in post-forensic analysis.

tion 4.5.3).

Lastly, during post-forensic analysis, MARSARA compares the recorded trace against the resulting audit log events to validate each occurring event (subsection 4.5.4) and then uses these verified events to determine where to place partitions, yielding verified execution units (subsection 4.5.5).

At first glance, this approach may seem too restrictive and false positive prone (i.e., rejecting of valid events) to be usable in real-world systems, however it works because:

1. The cost of a false positive is low, merely reintroducing an unnecessary dependency back into the data provenance.
2. Since all EUP work is based on offline profiling [200, 84, 88], no such system can guarantee that signatures are complete in the first place, and yet have demonstrated value in making data provenance usable for real-world systems [92].

Ultimately, MARSARA is effective if it preserves attack provenances while having a false

dependency reduction and performance comparable to previous (insecure) systems.

In this work, we focus on demonstrating the ability for MARSARA to ensure integrity using verified events and execution unit signature matches, as opposed to proving that our EUP algorithm is the most accurate. Readers interested in the latter topic should refer to OmegaLog [200], which implements and evaluates a similar partitioning strategy (without integrity verification).

4.5.1 Intel Processor Trace

Before diving into the phases of MARSARA, it is important to understand how PT works, since we intentionally design our solution to be compatible with it for better performance. PT enables MARSARA to securely audit the basic blocks executed by user space programs and can be controlled with a kernel driver, which we implement as part of MARSARA. For brevity, we will focus on how Intel’s implementation of PT works, which is the architecture supported by our prototype, however our design can be generalized to other PT implementations as well.

When a program for which MARSARA has a model is loaded for execution, it configures Intel PT to trace the execution. The MARSARA kernel maintains per-thread trace buffers, redirecting PT’s data output appropriately during context switches. Anytime a branching or indirect control transfer instruction occurs, PT records an event packet with the outcome. For branches, the packet is a single taken-not-taken (TNT) bit, whereas for indirect transfers (indirect call, indirect jump, and return), the target instruction pointer (TIP) is recorded. The Intel PT hardware automatically applies compression to the written packets to conserve space.

At the start of execution, the MARSARA kernel driver takes a snapshot of the program’s executable pages and then any additional pages loaded into memory afterwards (e.g., `mmap`) are also captured and recorded. This also includes dynamically generated code, such as JIT compilation. The resulting *sideband* data consisting of the initial snap-

shot, subsequently mapped executable memory, and context switch events, are interwoven with the PT data in the thread buffers to yield a linear stream of data.

Each stream contains all the necessary data to recover the program’s execution, down to individual instructions, with the help of a disassembler. However, as we will explain in subsection 4.5.2, not every instruction needs to be recorded for auditing, so to conserve space we distill the instruction sequences using kernel worker threads into relevant events and metadata centered around basic blocks. Since the PT data is not needed until the post-forensic investigation phase, the workers process data asynchronously to minimize overhead.

Intel PT is only configurable in the root CPU privilege level using MSRs and writes directly to physical memory. This allows the kernel to prevent all user space programs from reading or tampering with the trace. It also bypasses CPU caches, eliminating potential side channels and effects on the program’s performance. When the trace buffer is almost full, an NMI is raised, allowing the contents to be flushed without any data loss. As a result, systems leveraging Intel PT have demonstrated low performance overheads (under 7% [25, 168, 47]) and are capable of offering strong security integrity guarantees [25, 168].

4.5.2 Offline Profiling

In the offline phase, we propose to overcome the challenge of accurately determining a program’s control and data flows by using a combination of concrete traces and symbolic analysis. Specifically, MARSARA reads a target binary and generates a model of the program consisting of the possible paths between application log events, systems calls, and function/loop heads. Formally, given a binary b , MARSARA generates a graph $G = \langle V, E \rangle$ where V is a subset of b ’s basic blocks, and $E = V \times V$ is a set of edges such that $(u, v) \in E$ if there exists a path from u to v in b ’s control flow. System call and application log event nodes then get annotated with regular expressions defining their possible data values, calculated using binary single-path symbolic execution over the profiled traces. We

```

Func BUILDMODEL
  Inputs : Binary  $b$ 
  Outputs: Model  $G$ 

   $\mathcal{F} \leftarrow \text{GETLOGGINGPROCEDURES}(b)$ 
   $V \leftarrow \bigcup_{f \in \mathcal{F}} \text{GETCALLSITES}(b, f)$ 
   $V \leftarrow V \cup b.\text{libc\_calls} \cup b.\text{function\_heads} \cup b.\text{loop\_heads} \cup b.\text{function\_returns}$ 

  foreach  $v \in V$  do
     $v.\text{rva} \leftarrow \text{CALCULATERVA}(b, v)$ 
    if  $v$  is log call site then
       $v.\text{logstring} \leftarrow \text{GETLOGFORMATSTRING}(b, v)$ 
    else if  $v$  is loop head then
       $v.\text{is\_infinite\_loop} \leftarrow \text{HASNOEXITEDGES}(b, v)$ 
    end
   $E \leftarrow \{(u \in V, v \in V) \mid \exists \text{ path } u \rightarrow v \text{ in } b\}$ 

```

Algorithm 2: Model generation in MARSARA

use angr [21] for our Linux prototype.

In algorithm 2, we show the steps to produce a model in more detail. First, MARSARA identifies the set \mathcal{F} of logging procedures that produce application level messages. Then, using a first pass on the binary’s CFG, derived from profiled execution traces, MARSARA captures the basic blocks that end in a call to any function in \mathcal{F} . Next, MARSARA collects all basic blocks that correspond to heads of functions/loops and blocks that lead to system calls. In practice, we find that applications rarely make direct system calls, relying instead on standard libraries (e.g., `libc`) that expose equivalent user APIs. To account for this, MARSARA also collects all calls to functions in `libc` and analyzes them to determine the possible system calls they can emit.

To accurately map these basic blocks to events received from PT and audit logs, MARSARA needs to collect further metadata about them. MARSARA first tags each node $v \in V$ with its corresponding type: log, system call, function head, loop head, standard library call. Then, MARSARA calculates the node’s relative virtual address ($v.\text{rva}$), which corresponds to v ’s offset from the binary’s base virtual address. The relative virtual address (RVA) allows MARSARA to recognize addresses reported by PT, which are absolute addresses

affected by ASLR. For each node v that is a call site to a logging procedure, MARSARA uses symbolic execution to produce constraints that are then recorded as the log message’s format specifier ($v.\text{logstring}$). This is essentially a regular expression of all messages this code location is expected to produce. Finally, to be able to identify execution units (subsection 4.5.5) during the later post-forensic analysis phase, MARSARA marks all function and infinite loop heads. We consider such nodes to be possible candidates for starting new execution units since they often correspond to event-handling routines. While this is a heuristic, it has been well studied and considered reliable, appearing in many prior EUP systems [200, 84, 88].

4.5.3 Online Auditing

At runtime, MARSARA leverages PT to capture low level execution events alongside traditional audit logs of system calls and application level log messages. PT provides a hardware-enforced record of the program’s control flow, application log messages reflect data flow, and system calls capture OS events. We pick these sources because they are generated by different layers of the environment (hardware, application, kernel) and are *correlated*. This provides MARSARA a rich perspective from which to verify consistency.

Hardware Processor Trace Pure software solutions for recording runtime execution suffer from high performance overhead and weak security guarantees. PT is a hardware mechanism designed to address this by efficiently and securely capturing instructions as they are executed in the CPU. Intel’s implementation has been included in their processors since 2015, making it a prevalent feature in most computing environments. Although we use Intel’s implementation (subsection 4.5.1) in MARSARA, our design generalizes to other PT hardware as well.

Application Layer Events At runtime, audited programs are loaded with an instrumented standard library that augments the `write` call, as is typical of prior EUP designs [200]. In

addition to writing to the original destination, the new call also forwards messages to the framework used to record system calls. Most standard auditing frameworks (e.g., `auditd`) provide an API with this functionality. To simplify the segmentation of messages during post-forensic analysis, the instrumented write also appends the process/thread IDs and current timestamp to the sent messages.

Although the event logging frameworks used by user space programs are diverse and heterogeneous, the vast majority rely on standard runtime libraries (e.g., `libc`) to efficiently write logs while preserving portability across systems. MARSARA takes advantage of this to capture log messages that indicate various states in the execution units. A more detailed discussion of supporting heterogeneous logging frameworks is presented in prior work [200] and we discuss our prototype’s compatibility with other programming languages with alternative standard libraries in section 4.7.

System Calls Recording for system calls and their parameters are provided by the auditing frameworks MARSARA integrates with, which also include an API for MARSARA to forward application log messages into. For our prototype, we use Linux Audit.

4.5.4 Signature Match Validation

During the post-forensic analysis phase, MARSARA performs two tasks, starting with cross-validation of events received from PT with those from the audit logs, based on the model generated offline in subsection 4.5.2. This yields *validated* audit events that will then be used to produce *verified* execution unit partitions, which we describe in subsection 4.5.5.

In algorithm 3, we formalize our cross-validation matching. It takes three inputs: the generated model G , a PT trace \mathcal{T} , and an audit log \mathcal{A} of system calls and application log messages. For each event e received from the PT trace, MARSARA first determines if it is a system call event or a code block event. If it is a system call, MARSARA extracts e ’s call number and checks that it matches the number on the next event received from the audit

Func VALIDATETRACES**Inputs** : Model G , PT Trace \mathcal{T} , Audit Trace \mathcal{A} , binary b **Outputs**: Validated Events Path \mathcal{P} , Warnings \mathcal{W} $\mathcal{W} \leftarrow \{\Phi\}, \mathcal{P} \leftarrow \{\Phi\}$ /* ω is the last matched node */

*/

 $\omega = \Phi$ **foreach** event $e \in \mathcal{T}$ **do** **if** e is system call **then** $a \leftarrow \text{GETNEXTEVENT}(\mathcal{A})$ **if** $e.\text{syscall_num} = a.\text{syscall_num}$ **then** $\mathcal{P} \leftarrow \mathcal{P} \cup \{(e, a)\}$ **end** **else** $\mathcal{W} \leftarrow \mathcal{W} \cup \{(e, a, \text{critical})\}$ **end** **if** $e.\text{object} \in \{\text{libc}, b\}$ **then** $u \leftarrow \text{GETNODEBYRVA}(e.\text{rva})$ $\omega \leftarrow \text{VALIDATEEANODE}(e, u, a)$ **end** **end** **else** $u \leftarrow \text{GETNODEBYRVA}(e.\text{rva})$ $\omega \leftarrow \text{VALIDATEEANODE}(e, u, \Phi)$ **end****end****Func VALIDATEEANODE****Inputs** : PT event e , nodes ω, u , Audit event a **Outputs**: Last matched node $\text{match} \leftarrow e$ is application log event $\wedge \text{MATCHLOGSTRING}(a.\text{logmessage}, u.\text{logstring})$ **if** $\text{match} \vee (e$ is code block) **then** **if** $(\omega, u) \in E$ **then** $\mathcal{P} \leftarrow \mathcal{P} \cup \{(e, a, u)\}$ **return** u **else** **if** $\ell(u) \in \{\text{function head}\} \vee \ell(\omega) \in \{\text{function return}\}$ **then** $\mathcal{W} \leftarrow \mathcal{W} \cup \{(e, a, u, \text{low})\}$ $\mathcal{P} \leftarrow \mathcal{P} \cup \{(e, a, u)\}$ **return** u **else** $\mathcal{W} \leftarrow \mathcal{W} \cup \{(e, a, u, \text{critical})\}$ **return** Φ **else** $\mathcal{W} \leftarrow \mathcal{W} \cup \{(e, a, u, \text{critical})\}$ **return** Φ **Algorithm 3:** MARSARA's trace validation algorithm.

log. If the two numbers do not match, then the event is invalid and discarded.

Next, if the system call originates from a code block that is either in `libc` or the application's binary, MARSARA obtains the corresponding node in G that matches the event node's RVA. It then validates if the path observed so far matches at least one known signature. Non-system call PT events (i.e, loop heads, function heads, and returns) are treated in a similar manner.

To check for path validity, MARSARA keeps track of the last matched node in the current observed trace. If the newly matched node u is an application log node, MARSARA extracts the node's format specifier ($u.logstring$) from the model, and confirms that it matches the concrete message recorded in the audit log. If a discrepancy is found, the match is invalidated.

When the log matching succeeds, or alternatively, if u is simply a code block, MARSARA checks if there exists an edge $(\omega, u) \in E$ between the last matched and current node. If it exists, MARSARA considers the path to be valid and updates that last matched node to be u . If a discrepancy is found, it is invalidated.

Warning Types When MARSARA detects invalid events, it records warnings of two severity levels: low and critical. Currently, warnings are intended only to provide verbosity so we can empirically evaluate MARSARA's accuracy. They do not need to be considered by investigators and we leave the possibility of using them to aid in investigations to future work.

The severity is based on what kind of discrepancy is detected in the model. In benign experiments where no attack is occurring, if a direct code branch causes a warning, it is ranked low because this is due to a missed path during offline profiling and can be resolved using more data. Recall that all prior work also relies on offline profiling and therefore cannot guarantee completeness.

Conversely, if the inconsistency (in benign experiments) arises from indirect transfers

(indirect jump, indirect call, return), it is ranked critical since this is a limitation in the symbolic analysis used during offline profiling. This represents a limitation that cannot be resolved with more data, which is why we differentiate it from low warnings. Fortunately, as we demonstrate in our evaluation, these are rare, meaning that our design is effective overall.

4.5.5 Execution Partitioning

MARSARA’s partitioning logic relies on the observation that developers of long-running processes create log messages for the important events in each execution unit’s lifecycle. For example, for a web server that handles user requests, it is customary for developers to log the user’s request at the start of each unit. Such log messages often reside at the start of an event-handling function (typically a function pointer) or an infinite loop, which is why our binary analysis in subsection 4.5.2 labeled them explicitly.

However, determining which log messages signal the start of a new execution unit without semantic analysis of the message’s content is a challenging task. To overcome this, we combine information about loops and functions from the offline profiling phase with runtime information about log messages to uncover the heads of execution units.

As discussed in subsection 4.5.2, MARSARA assigns each code block v with a label $\ell(v)$ indicating whether v is an infinite loop or the head of a function. Such blocks become candidates for starting new execution units. MARSARA keeps a running count of the number of times a log messages has been encountered in a priority queue. The intuition behind this approach lies in the observation that application developers, in an effort to reduce the performance overhead of logging, restrict the log messages to important events, the most important of which is the servicing of a new input. Therefore, the log message at the top of the priority queue (i.e., the one with the largest count) likely corresponds to the head of an execution unit. Every time that message is encountered, MARSARA performs a backward search in the current trace and identifies the closest code block that

is either an infinite loop head, or the head of a function with no incoming edges in the model. MARSARA then creates a new execution unit starting from that block and adds all subsequent events to the new unit.

4.6 Evaluation

We evaluate MARSARA with an emphasis on answering the following research questions:

1. What is MARSARA’s accuracy when validating the integrity of partitions? We measure its accuracy in terms of the number of warnings generated over benign inputs in 14 real-world programs and show that only 2.82% of false dependencies are reintroduced at worst.
2. How much does MARSARA reduce the vulnerability of programs to EUP attacks compared to CFI alone? We measure PAS for the same real-world programs while being protected by MARSARA, shadow stack, and function-level CFI. MARSARA removes 47,642 more gadgets per program.
3. Can MARSARA prevent execution repartitioning attacks based on the techniques from section 4.3? We attack several programs using expertly crafted exploits and find that MARSARA successfully preserves the full attack provenance.
4. What is the cost of MARSARA’s forensic analysis? We measure the overhead for the real-world programs and the SPEC CPU 2006 benchmark compared to a standard auditing framework and find it to be 8.7%, on average.

Experimental Setup We evaluate MARSARA using 14 popular real-world applications. These programs have frequently been used to evaluate prior work [88, 93, 84, 200, 85], justifying their inclusion. We use the default configurations and generate workloads with standard benchmark tools, such as Apache Benchmark [215]. We also evaluate against the SPEC CPU 2006 benchmark, with full workloads, for direct comparison with prior work.

Table 4.1: Performance, accuracy, and storage overhead of MARSARA. Time captures the seconds to analyze and validate events. Baseline storage corresponds to running the Linux Audit framework and application log tracking without MARSARA. The low warnings are categorized by the model edge type for additional granularity.

Program	Model			Total Events	Time (sec)	Warnings			Storage (MB)		
	Blocks	Edges	Edges			Low		Critical	FPR	Baseline	MARSARA
						Forward Edges	Backward Edges				
cupsd	4768	32 521	15 592	0.109	1	20	0	0.13%	0.218	0.067	
HAProxy	28 837	188 422	69 009	0.241	131	264	11	0.59%	0.141	0.244	
httpd	7087	25 465	419 532	1.237	187	226	0	0.09%	0.433	1.613	
lighttpd	5680	24 862	508 707	0.967	179	134	5	0.06%	0.277	2.436	
memcached	38 427	200 041	4282	0.082	82	32	7	2.82%	0.300	0.219	
nginx	15 675	99 924	175 239	0.511	265	326	2	0.33%	0.310	0.722	
postfix	146 296	476 904	2968	0.043	28	2	5	1.24%	0.898	0.010	
Proftpd	10 918	70 767	3 050 246	15.214	305	229	4	0.01%	0.630	11.181	
Redis	28 881	161 294	2 681 711	21.357	334	416	3	0.02%	0.483	15.007	
squid	32 516	109 804	116 100	0.436	170	118	2	0.24%	0.652	0.583	
thttpd	32 725	203 385	12 589 818	22.361	48	17	6	0.00%	0.206	33.681	
Transmission	7045	27 765	173 705	0.397	236	154	80	0.27%	0.282	0.031	
wget	6979	49 028	17 624	0.048	74	63	1	0.78%	0.095	0.088	
yafc	3621	18 981	31 170	0.318	60	39	5	0.33%	0.114	0.105	

For practical binary CFI defenses, we consider shadow stack and function-level policies, which are realistic to enforce without source code. Shadow stack prevents control flow hijacking from arising via corrupted return pointers whereas function-level CFI additionally enforces that indirect calls and jumps must target the start of a valid function. More accurate policies have been proposed, but have not seen real-world deployment due to requiring source code, being incompatible with mechanisms like stack unwinding, and/or having overheads upwards of 47% [47].

We conduct our tests on a server-class machine with an Intel® Core™ i7-6700K CPU and 16GB of memory, running Debian 10. Audit logs are collected using Linux Audit with rules covering the most commonly used system calls, such as `read`, `write`, and `execve` (23 in total).

Definition of Errors For the purposes of this evaluation, a *false positive* is defined as a legitimate audit event that is accidentally detected during MARSARA’s integrity check, yielding a warning, and a *false negative* is a spoofed or delayed event that is not. In terms of the resulting provenance graph, a false positive *may* introduce a false dependency edge whereas a false negative *may* remove a true dependency edge.⁵

Calculations Overhead is calculated as $(P - B)/B$ where B is the baseline performance value and P is the value with the evaluated system enabled. FP rate for Table 4.1 is calculated as the sum of all warnings divided by total events. We do not report the time to produce models since this is only done once per program during the offline phase.

4.6.1 Partition Validation Accuracy

Table 4.1 shows the performance and accuracy of MARSARA’s analysis for validating the execution partitions. As expected from algorithm 3, the time to validate is linear to the

⁵Notice that if a false positive happens to be a true dependency, the graph is unaffected, and if a false negative fails to forge a signature match, the graph is also unaffected.

Table 4.2: PAS for several real-world programs and defenses.

Program	ICT	None	SS	Func.	MARSARA
cupsd	4,017	19.42	8.62	8.59	8.33
HAProxy	13,155	2.49	2.18	2.18	2.11
httpd	1,779	40.00	12.14	12.02	9.41
lighttpd	2,858	0.18	0.13	0.13	0.13
memcached	797	2.82	1.10	1.02	0.89
nginx	3,997	0.80	0.37	0.37	0.28
postfix	848	16.00	10.28	9.75	9.42
Proftpd	34,830	2.18	0.82	0.81	0.72
Redis	28,047	7.09	5.37	5.34	5.06
squid	18,412	353.00	196.03	181.11	123.94
thttpd	1,198	1.02	0.14	0.14	0.11
Transmission	17,507	2.89	1.83	1.82	1.75
wget	16,594	6.71	0.85	0.71	0.64
yafc	8,590	0.85	0.64	0.63	0.62
Average:	10,902	32.53	17.18	16.04	11.67

number of events recorded. In the largest observed case (12 million events, `thttpd`), MARSARA analyzes and validates the trace in less than 30 seconds. This is reasonable since verification is only required once per trace and is not performed until an investigation occurs (i.e., the post-forensic analysis phase).

We also report the number of events yielding FP warnings during verification. For 8 of the 14 applications, MARSARA reports no critical FPs, meaning that the symbolic analysis used during the offline profiling phase works well on the evaluated programs. For the remaining programs, the FPs are <6 , highlighting only a few troublesome model edges.

FPs occur mainly for two reasons: due to limitations in binary symbolic analysis and inaccuracies in reporting system calls. In some cases, MARSARA detects system calls that do not map back to nodes in the model. For example, in `Transmission`, unexpected `openat` system calls are recorded. Investigation reveals that the function `tr_variantToFile` makes a call to the `libc` method `mkstemp`. However, when examining the model, we did not find a node for this method, indicating that symbolic execution was not able to analyze it. We further investigated the source code for `mkstemp` in `glibc` and observed that it

is replaced by the compiler with a function called `__gen_tempname`⁶. These kinds of optimizations are not currently handled by the verification algorithm, but will be addressed in future versions.

We also report the number of events yielding low severity warnings, which arise in direct branches not covered by the profiling traces we collected during the offline phase. For additional clarity, we categorize these into forward graph edges (calls, jumps), backward edges (return), and other (unexpected audit log events). The evaluated programs yield between 10 and 600 low warnings, which we explain the impact of next.

Since this experiment does not contain any exploits, all generated warnings are false positives, i.e., legitimate events wrongly detected by MARSARA’s integrity check. This is presented in the table as false positive rate (FPR), calculated as the number of warning-producing events (low and critical) divided by the total number of events. In all cases, FPR is 2.82% or lower. Recall that if a false positive pertains to a false dependency, it will be preserved in the resulting provenance graph as an edge rather than being removed during partitioning. A false positive detection of a true dependency is of no consequence, since it would not have been removed anyway. Consequently, FPR is also the maximum number of false dependencies that can be reintroduced into the graph. For example, if the ideal partitioned provenance graph for a given query contains 1,000 dependencies (edges), the resulting graph with a FPR of 2.82% could contain up to 1,028 edges (28 false dependencies), presenting little difference to analysts or downstream systems. In short, MARSARA almost completely preserves the false dependency reduction of prior (insecure) EUP techniques with the added benefit of integrity.

4.6.2 Partitioning Attack Surface Reduction

Table 4.2 presents MARSARA’s PAS for the real-world programs compared to the unprotected binaries and several practical binary CFI policies, along with the number of indirect

⁶Observed in `glibc/misc/mkstemp.c` at line 33.

control transfers (ICTs) in each program. Recall from subsection 4.4.1 that smaller values equate to greater protection against EUP attacks.

Across all measured programs, MARSARA's PAS is better than any of the CFI defenses. Since most programs contain over 1,000 ICTs, even small reductions in PAS are significant. For example, MARSARA reduces Proftpd's PAS by 0.09 versus function-level CFI, which over 34,830 ICTs equates to eliminating 3,134 events that an attacker could otherwise leverage to spoof EUP signatures. In the simpler programs, the benefits are more modest. For example, `lighttpd` gains little added protection from MARSARA, or function-level CFI for that matter, due to not having any indirect calls or jumps. The biggest benefit is observed in Squid, where its modular design presents the opportunity for MARSARA to reduce PAS by 57.17 over function-level CFI, eliminating over 1,052,614 event gadgets. On average, 47,642 additional gadgets are removed compared to function-level CFI. In short, MARSARA successfully eliminates thousands (and sometimes millions) of options for an attacker attempting to spoof an EUP signature, even in programs already protected by binary CFI.

4.6.3 Attack Investigation

To evaluate MARSARA's integrity, we use the expertly crafted exploits described in subsection 4.3.3 to attack real-world programs. Specifically, we first run EUP without PT or MARSARA's partition verification (essentially placing partitions as prior systems would, creating a baseline for comparison) to confirm that the exploits produce valid (malicious) signatures for partitioning. As expected, all 3 attacks successfully manipulated prior EUP algorithms into fragmenting the attacker's exploit and resulting symptoms across disjoint partitions. In short, without MARSARA, provenance queries made by investigators will be answered with seemingly legitimate (but actually misleading and incomplete) results.

We then rerun the attacks, now with MARSARA. In the 2 control hijacking cases (CVE-2013-2028, CVE-2004-0541), we observe critical warnings at the point where the exploits

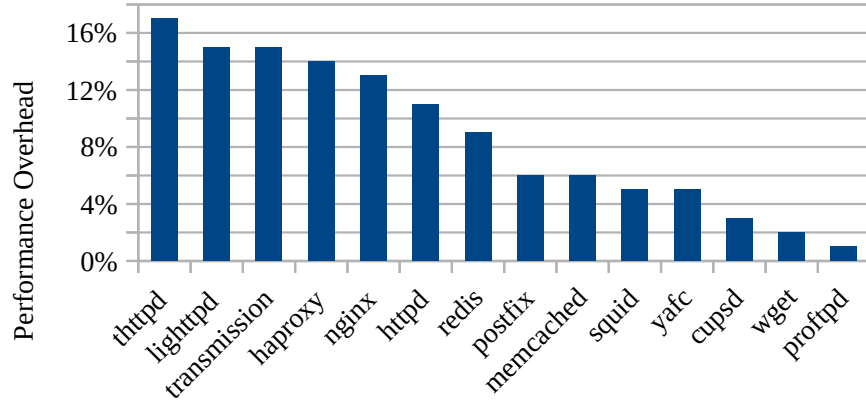


Figure 4.5: Performance overhead for the real-world programs. The average is 8.7%.

redirect control of the execution. For CVE-2009-4769, the critical warning arises because the model reveals, based on the call site to the logging method, that the resulting message in the audit log contradicts the expected format. Consequently, MARSARA does not fragment the attacker’s network requests from the rest of the symptoms, yielding complete provenance attack graphs that contain all the relevant events. For example, for CVE-2013-2028, which pertains to our motivating example originally visualized in Figure 4.1, MARSARA’s partition includes both the events pertaining to $x.x.x.x$ and $y.y.y.y$. *In short, this experiment yields no false negatives.*

4.6.4 Runtime & Space Overhead

Real-World Programs We report the storage requirements for MARSARA’s analysis in the last two columns of Table 4.1. Our baseline represents the amount of compressed data needed to store the events generated by the Linux Audit framework. We compare that to the amount of extra storage (also in compressed form) that MARSARA requires for PT.

For 9 of the 14 applications we evaluated, MARSARA’s storage requirement is in the same order as the baseline (e.g., 1.6 MB for 500K events in the case of `httpd`). However, for each Linux Audit trace, the corresponding PT trace can be discarded after MARSARA’s validation is completed. This renders the PT storage overhead as only a temporary cost.

For 3 applications (`thttpd`, `Proftpd`, and `Redis`), a large number of PT events

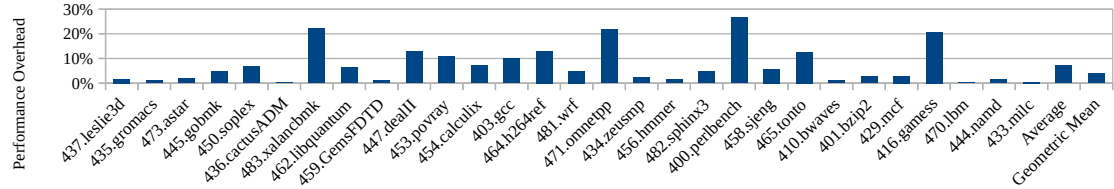


Figure 4.6: Performance overhead for the SPEC CPU 2006 benchmark. The average is 7.21% and the geometric mean is 3.81%.

are generated, requiring significantly more temporary storage. Investigating further, we discover that MARSARA reports on events pertaining to several loop blocks engaged in “busy-waiting” behavior for initializing large arrays. For example, `thttpd` creates an array for storing all the possible file descriptors (1024 in our evaluation environment) and then initializes each element to `-1`. Consequently, every time this code block is executed, PT records a path consisting of 1024 blocks, significantly increasing the number of events generated. We discuss possible solutions to PT’s storage requirements in section 4.7.

Figure 4.5 shows MARSARA’s runtime overhead compared to the baseline of Linux Audit framework with no PT event tracking. MARSARA’s average runtime overhead is 8.7%, which is consistent with prior PT systems [25, 47, 168]. The overhead observed varies depending on the profiled application’s behavior. For example, applications that are mostly IO-bound, such as caching servers (`memcached`, `squid`), file, mail, printing servers (`proftpd`, `postfix`, and `cupsd`), and key-value stores (`redis`) exhibit low runtime overhead, ranging from 1% for `proftpd` to 9% for `Redis`. Conversely, applications that are more CPU-intensive, such as web servers and load balancers, incur a larger overhead (up to 17% for `thttpd`) since PT yields more events. We will consider alternative methods to reduce PT’s runtime overhead for CPU-intensive applications in future work.

SPEC CPU 2006 To provide an additional standard benchmark for comparison, we also report the performance overhead of monitoring the SPEC CPU 2006 benchmark programs

over all provided workloads, visualized in Figure 4.6. Across the SPEC programs, MARSARA yields an average performance overhead of 7.21%, which is consistent with the results from monitoring the 14 real-world programs that are typically used in provenance system evaluations. However, we also note that some of the SPEC programs produce noticeably higher overhead due to the amount of PT data they produce. This is to be expected since the benchmark is designed to stress CPUs, making the workloads CPU-bound, whereas the other programs we evaluate are mostly I/O-bound. We believe the non-SPEC workloads are more representative of the programs an EUP attack would target, so we conclude that the SPEC performance results are tolerable.

4.7 Discussion

Improving Model Accuracy The current MARSARA prototype relies on binary single-path symbolic execution to generate the model during offline profiling. This results in an under-approximated set of paths. Although we consider improving the state of binary analysis to be outside our scope, several possible solutions exist to improve its accuracy.

For example, because MARSARA already records the full PT trace and system call audit for protected programs, it is possible to use the collected data to guide an offline replay. Specifically, when MARSARA encounters an inconsistency due to a missing edge in the model, an existing record-and-replay system [91, 26, 27] can re-execute the program offline with additional instrumentation (e.g., Valgrind [22]) to detect the presence of memory corruptions and then refine the model appropriately. Although memory-safe record-and-replay is expensive, the cost would be paid in an offline analysis and each newly encountered path would only need to be tested once. In time, the model would converge to the ground truth graph with a priority towards refining execution paths actually observed in real-world executions.

We also note that symbolic execution does not scale to all programs, particularly complicated ones like web browsers. However, by evaluating a prototype that uses application

message and system call auditing, designed as an extension of the most recent work, we demonstrate that our approach of using PT and binary symbolic analysis to verify signatures can benefit the security of all EUP-dependent systems, not just our prototype. We also demonstrate that even in its current form, MARSARA protects logs derived from important web services.

Improving Storage Overhead While most of the tested binaries produce audit logs comparable in size to the baseline system considered in section 4.6, we encounter some cases where sizes are an order of magnitude larger. We discover the cause of this phenomenon to be non-blocking event loops (i.e., “busy waiting”), which yield many control flow events of little significance (i.e., checking a flag and then returning to the loop head). This can be addressed as the PT trace is decoded by summarizing loops or using compression tailored to our problem context. Note that decreasing the PT trace size will also benefit performance, since less data has to be processed.

In a similar vein, while our PT-enabled kernel is capable of tracing programs with dynamically generated code (e.g., JIT in browsers), doing so is likely to yield higher performance overhead as each generated code page has to be captured in the sideband data. We leave these optimizations to future work.

Compatibility with Other Languages MARSARA’s reliance on an instrumented `libc` means it will not be able to capture application messages for all possible Linux programs. However, fixating on this detail overlooks two points that are more significant. First, our prototype’s EUP signatures contain messages *and* system calls. Even when the former is unavailable due to compatibility, the latter can still be used to identify units of execution, albeit at a coarser granularity. EUP is still valuable in such cases [84, 86]. Second, the purpose for including application messages in our design is to demonstrate the flexibility of our modeling to serve a wide range of analyses that require EUP, not just those reliant on one data source (e.g., system calls).

Compatibility with “At Rest” Integrity In this work, we focus on protecting log integrity against a novel form of online tampering based on EUP attacks. This is outside the scope of prior work, which focuses on tampering performed to data *at rest* on storage. Our proposed defense complements the protection offered by these past solutions and MARSARA can be extended to incorporate them into a holistic system. For example, solutions based on cryptography can be readily applied to the data produced by MARSARA, thereby adding storage integrity. Similarly, MARSARA can control where data is stored, allowing it to leverage trusted storage solutions like WORM drives or central logging servers.

4.8 Conclusion

This work presents the first formal exploration of online anti-forensic attacks against data provenance leveraging software exploits. We demonstrate that attackers can break the causal links in data provenance graphs used for forensic investigation, and even frame benign subjects, without triggering existing tamper-evident logging defenses. We propose MARSARA to verify EUP signature matches and demonstrate that it resists expertly crafted exploits while reintroducing no more than 2.82% of false dependencies, across 14 real-world programs, with a performance overhead of 8.7%. Compared to CFI, MARSARA removes 47,642 more gadgets per program.

CHAPTER 5

BUNKERBUSTER: PROACTIVE BUG HUNTING AND REPORTING

In this chapter, my collaborators and I present Bunkerbuster, a system designed to proactively hunt for and localize memory corruption bugs using benign user data recorded from multiple end-host systems.

5.1 Introduction

As pressure on companies to swiftly identify and remediate system vulnerabilities has increased [216], corporations have adopted *bug hunting* strategies. They *proactively* search for and remediate problems in their adopted software *before* adversaries can exploit them in an attack [217]. Unfortunately, the path from corporate bug hunting to developer software patch is cumbersome and laborious, leaving less-equipped companies vulnerable.

Human bug hunters, lacking good inputs to test programs, rely on fuzz testing (fuzzing) [4, 7, 1, 2, 3] to brute force test cases, starting from seeds provided by the developers (e.g., regression tests) or scraped from public databases (e.g., ImageNet [218]) that offer limited coverage. Such tools often require manually written scaffolding code to reach deep libraries or APIs [6, 5, 219] and rely on crashes to signal buggy behavior [63, 58, 70, 64], which is not always reliable [220, 54, 9]. The process is further complicated by binaries that lack source code, requiring bug hunters to engage in extensive reverse engineering [221, 8, 10].

Worse still, the bug hunter then needs to share their findings with the software's developers. Crashes can corrupt artifacts [142, 143, 101, 144, 145, 94, 146, 147, 148, 95, 149, 150, 151, 152, 153, 154, 155, 156] and bugs can be difficult to reproduce due to environment differences. Capturing stack traces or re-executing the crashing input with instrumentation [222, 59, 163, 139] offers some insights, but as we discover in an in-depth case study, the results can be incomplete, hindering triage. Prior work shows that develop-

ers consistently undervalue or ignore issues they do not understand [159, 160], but without their aid, the only other remediation choices are incomplete stopgaps like input filters [136, 137, 138, 139] or selective function hardening [140], which incur significant overhead [23].

However, we observe that software testing need not occur in a vacuum. Namely, companies already have employees constantly using the software in question, and their *real-world usage already drives the program into deep behaviors within realistic environments*. The data to automate bug hunting and reporting is already within their reach, so why are they not using it?

We hypothesize that the disconnect that occurs is due to the traditional definition of “seed as program input” being insufficient. While program inputs are easy to collect, they offer little insight into how to build scaffolding, how to get from sound program states to buggy ones, and how to explain those bugs in a meaningful way. Instead, we hypothesize that *control flow traces* are the better seeds for automating bug hunting and reporting because they can reveal the solution to all the above problems while still being efficient enough to collect from real user environments.

To demonstrate this, we first propose how to segment control flow traces and save sequential memory snapshots to guide symbolic analysis through code where it is otherwise susceptible to path explosion [223, 52]. We hypothesize that this *control oriented* record and replay of user sessions is suitable for discovering serious classes of memory corruption, such as those arising from overflows, use-after-free (UAF), double free (DF), and format string (FS) bugs. Better still, thanks to the prevalence of hardware assisted PT, production systems can securely capture traces with user transparency and tolerable overhead.

Notice that while prior work has demonstrated the value of snapshots for bug hunting [224], they did not combine them with traces. Without the accompanying segmented control flow traces leveraged in our design, such systems are still susceptible to path explosion due to loops and string manipulation, limiting their scalability in real-world settings.

However, collecting a corpus of new seeds is only half the battle. To reach new buggy

program states, we also propose a technique to selectively symbolize predicate data, based on the recorded control flow traces, to facilitate *constrained* exploration that *prioritizes* certain paths while managing path explosion. To inspect deep API calls, we propose an analysis to automatically recover parameter prototypes, eliminating the need for human analysts to implement scaffolding. To find bugs from benign recordings, we employ bug-class-specific search strategies and detection techniques that check uncovered states for symbolic indicators of buggy behavior.

The above technical contribution also brings an additional benefit to our design, which is that the same symbolic constraints can also be used to perform symbolic root cause analysis [225]. This recently proposed technique for localizing memory corrupting bugs has only been demonstrated in single path symbolic analysis, starting from the program entry point, limiting its possible applications. Our design shows how it can be used in a multi-path setting, starting from the main program entry point *or* entry points to imported library APIs, increasing its applicability.

We implement our design as a Linux prototype, named Bunkerbuster, and evaluate it on 15 programs, some of which contain binaries compiled from over 810,000 lines of C/C++ code, invoking 1,710 imported functions and producing traces 19,392,602 basic blocks long, on average. Bunkerbuster successfully uncovers 39 bugs, of which 8 are newly discovered by our approach. 1 EDB and 3 CVE IDs have been issued and patched by developers, using Bunkerbuster’s reports to independently verifying their novelty.¹ In a side-by-side comparison, Bunkerbuster finds 8 bugs missed by AFL and QSYM, and correctly classifies 4 that AddressSanitizer mislabeled. Our prototype accomplishes this with 7.21% recording overhead and manageable storage requirements. We have open sourced our prototype and data to facilitate future work.²

¹We report all bugs to developers, MITRE, and Offensive Security for responsible disclosure.

²<https://github.com/carter-yagemann/arcus>

```

/* coders/png.c */
ReadMNGImage() {
5129:    previous = image; // heap object
5130:    mng_info->image = image;
5138:    ReadOneJNGImage(mng_info); // 1st free
5143:    DestroyImageList(previous); // 2nd free
}

/* coders/png.c */
ReadOneJNGImage(MngInfo *mng_info) {
3126:    DestroyImageList(mng_info->image);
}

/* magick/list.c */
DestroyImageList(Image* images) {
239:    DestroyImage(images); // calls free
}

```

Figure 5.1: Source code pertaining to CVE-2017-11403 in GraphicsMagick, summarized. ReadMNGImage calls ReadOneJNGImage without realizing that it may free image, making Line 5,143 a double free bug for some paths.

5.2 Overview

Bunkerbuster’s analysis replaces the laborious process of *proactively* hunting for and reporting software bugs in enterprise networks. Bug hunting should not be confused with IDS, which requires *reacting* swiftly to ongoing attacks. In place of a human security expert creating a testbed to fuzz programs or library APIs, Bunkerbuster gathers data from end-hosts using a kernel driver, cleverly inferring input structures and segmenting traces to achieve offline binary symbolic execution.

5.2.1 Real-World Example

To show how Bunkerbuster benefits a bug hunter tasked with finding problems in software, consider the following example based on CVE-2017-11403, a UAF vulnerability found in GraphicsMagick. For clarity, we will explain this example using the source code shown in Figure 5.1, however the real analysis is on binaries. In this instance, the func-

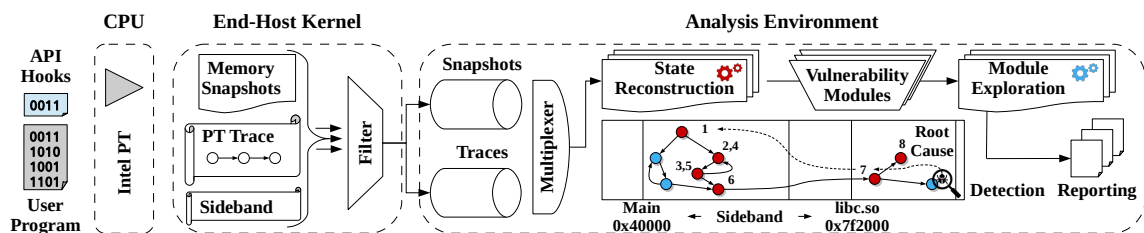


Figure 5.2: Bunkerbuster architecture. End-hosts with PT-enabled kernel drivers collect and filter traces of the target program, forwarding novel segments to the analysis environment. Symbolic states are reconstructed and then expanded by exploration plugins. When a bug is detected, symbolic root cause analysis pinpoints the source and produces a report.

tion `ReadMNGImage` always frees the heap object `image` before returning, but what it does not account for is that a child function it invokes, `ReadOneJNGImage`, can also free `image` after a certain error, causing a DF.

Suppose that the bug hunter, having heard about all the recent vulnerabilities found in image processing libraries, wants to analyze a program his employees are using that imports the `GraphicsMagick` library. Unfortunately, he is not familiar with obscure image formats like MNG, so building fuzzer scaffolding for all of `GraphicsMagick`'s APIs would be tedious, and fuzzing the entire program from startup would be inefficient due to its complexity.

Instead, he gives the name of the target program to the Bunkerbuster analysis system, which in turn forwards it to all the end-hosts with the Bunkerbuster kernel driver installed, as shown in Figure 5.2. These systems observe the processes being created locally and anytime the target program starts, they configure PT for recording. As the data is collected at the end-host, it locally segments the trace at calls to imported library functions and hashes them on-the-fly. Each hash is checked against a filter, and if the segment is novel, it is forwarded for analysis.

Back at the analysis system, Bunkerbuster uses the incoming traces along with symbolic execution to reconstruct symbolic states for each executed basic block. Since the conditions leading to CVE-2017-11403 are rare, these segments do not directly reach the DF bug, but some contain invocations of the vulnerable function. Using its search plugins containing

bug-class-specific exploration strategies, Bunkerbuster symbolically expands the set of reconstructed states, yielding additional states within the same function, including the one containing the CVE. When Bunkerbuster checks them for memory corruption, it finds the state containing the DF. It then switches to localizing a concise root cause. Bunkerbuster compares the constraints leading to this buggy state against others sharing the same predecessor guardian (i.e., conditional check) and determines the difference that makes the DF reachable. It then traces this back through the predecessor states, pinpointing the error checking branch. The end result is a concise, human-readable report, identifying the site of the first and second frees, and the input error check in GraphicsMagick that caused the DF.

Notice that if no end-user ever loads an MNG image, the analysis will not find this DF because the traces will not have any invocations of the vulnerable function to reference. However, code that is never invoked is a prime candidate for debloating [226, 227, 228, 229, 230], which is outside the scope of this work. Conversely, Bunkerbuster *will* cover all the code used by monitored users.

5.2.2 Goals & Assumptions

We focus on discovering and localizing overflow, UAF, DF, and FS bugs within unobfuscated, benign Linux programs without access to source code or debug symbols. The limitations imposed by this scope are discussed further in section 5.5.

We assume that the end-hosts contain PT-enabled CPUs, which also form our trusted computing base (TCB). PT is a hardware feature that writes directly to physical memory, bypassing all CPU caches, and is only configurable in the privileged CPU mode, making it a trusted platform in numerous security systems [168, 25, 231, 232, 233]. We expect collected data to encode benign behaviors, motivating the need for bug hunting. In the event that an end-host captures malicious activity, detection becomes easier. We envision our system being deployed on enterprise computers and servers, leaving mobile and embedded devices for future work.

To recover the structure of inputs to APIs as accurately as possible while covering the diverse range of possible use cases, we consider two scenarios. The first targets open source C/C++ libraries, where we assume access to stub code or source headers that define the API. This is a necessary part of any public release to allow other developers to integrate their systems with the API. For all other cases, we assume the most conservative scenario where only the binary is available.

5.2.3 Bug Class Definitions

Use-After-Free UAF is a type of temporal memory safety violation that can lead to memory corruption. Instances follow a specific pattern [234, 235], starting with a block of memory being freed by the program. Once freed, any pointers in memory to the freed block that are not cleared become dangling pointers. If a dangling pointer is dereferenced for any operation, this becomes a UAF. By this definition, DF is a subclass where the violating use is another free [236].

Overflow Overflows are a type of spacial memory safety violation whereby a looping sequence of memory writes exceeds the bounds of a finite buffer and corrupts neighboring memory locations. Conceptually, overflows are possible if a loop fails to correctly consider the address bounds of the buffer being written to. Vulnerable loops typically terminate based on conditions derived from counting (e.g., `fwrite`) and delimiters (e.g., `strcpy`) [237]. Since our analysis is performed at binary level, where variable type information is unavailable, we focus on control flow hijacking overflows, which corrupt code pointers and lead to arbitrary code execution [23, 70, 238, 71, 7], as opposed to data-only attacks [239]. Future work can extend our design to cover data-only attacks by either inferring types [169, 170] or incorporating source code or debug symbols.

Format String FS is another type of spacial memory safety violation where input data directly controls the format specifier used in functions like `printf`, allowing for arbitrary

reading and writing of memory. This can lead to memory corruption or be used to leak sensitive data. We consider both cases, including data-only attacks, because unlike overflows, FS can be analyzed without additional type information.

5.3 Design & Implementation

In this section, we elaborate on the steps in Bunkerbuster’s recording and analysis, initially presented at a high level in subsection 5.2.1. Stepping through the workflow sequentially, subsection 5.3.1 describes how the end-hosts record and filter the PT traces that the analysis uses to recover valid program execution paths. Next, subsection 5.3.2 describes how memory snapshots are taken at the end-host and how the analysis selectively symbolizes them to bootstrap symbolic execution. Given a symbolized snapshot as a starting state and a matching trace segment, subsection 5.3.3 describes how to recover symbolic representations of all the intermediate program states along the traced path at basic block granularity.

With a linear sequence of symbolic states for the recorded path constructed, we then describe how to explore additional paths, prioritized using search strategies based on our domain knowledge of our target bug classes. We also describe how Bunkerbuster uses the symbolic constraints for the states to detect and then localize bugs. Since our techniques are bug-class-specific, we split our description between UAF/DF, which arise from *temporal* memory safety violations, and overflow/FS, which arise from *spatial* memory safety violations, in subsection 5.3.4 and subsection 5.3.5, respectively.

5.3.1 Capturing & Filtering Traces

One technical challenge Bunkerbuster has to overcome is how to efficiently, securely, and transparently record user sessions. To this end, we center our design around PT, and then propose a novel way of hashing recorded segments so redundant ones can be discarded. However, before explaining filtering, it is important to understand what PT is and how Bunkerbuster uses it. For brevity, we will focus on Intel’s implementation of PT, however

similar features exist in processors made by ARM, AMD, and others.

Intel PT records traces of user space execution directly to physical memory, where it can then be forwarded by a kernel driver to persistent storage or remote endpoints. Its recording can be restricted to a particular process at the hardware level using a configuration register that accepts a CR3 value representing the process' page table address.

Traces consist of a stream of packets, each recording the outcome of a branching instruction, indirect call/jump, return, or interrupt. Binary branches are recorded as a single TNT bit, whereas other events yield a TIP. To decode the trace into an instruction sequence, the decoder also needs additional *side-band data* about the traced process' memory space and thread scheduling, which we describe next.

First, the decoder needs the process' executable pages in order to recover instructions. Bunkerbuster's kernel driver handles this by hooking relevant system calls (e.g., `mmap`, `mprotect`) and recording memory pages alongside the PT trace. Bunkerbuster can then linearly disassemble the memory, starting at the program's entry point and consulting the next PT packet whenever a branch is encountered, to recover every executed instruction.

Second, in order to distinguish threads that share the same page table (CR3 value), the kernel driver also hooks context switches to record when threads are swapped in and out of CPU cores. The driver also hooks the `fork` and `exec` system calls so it can detect and trace child processes created by the target program.

Trace Filtering Unlike prior PT systems, Bunkerbuster has to account for the fact that users and services may engage in repetitive tasks, yielding partially redundant execution traces. To address this, our driver quickly hashes trace segments on-the-fly and compares them against a global map, discarding ones that have already been observed, using the following algorithm:

$$(u, v) \in T : u \ll 1 \oplus v \bmod S \tag{5.1}$$

where u and v are virtual address offsets, relative to their object bases to account for ASLR, recovered from trace T . The result is a bit offset within a map of size S bits corresponding to the edge (u, v) . The global map is initialized with all bits set to 0 and then as edges are decoded from the PT trace, their corresponding bits are set to 1. If a trace segment adds any novel bits to the global map, it is forwarded for analysis, otherwise it is discarded.

5.3.2 Symbolizing Memory Snapshots

Alongside the data described in subsection 5.3.1, the end-host driver also records snapshots of register values and memory that will serve as starting states for symbolic execution. Specifically, when the program is loaded at runtime, the driver hooks the program's main entry point and any entrances to imported APIs (i.e., library functions) by placing traps in the process' procedure linkage table (PLT). Once captured, Bunkerbuster symbolizes the input data, which for the main entry point is the program's input arguments and for APIs are the called function's parameters. This data is replaced with unconstrained symbolic variables, enabling Bunkerbuster to reason about all possible input values to the program and imported APIs. For this reason, each trap only needs to be used once, and is then removed, minimizing runtime overhead. This also allows Bunkerbuster to analyze snapshots (and their corresponding trace segments) in any order because there are no prior constraints.

Generally speaking, under-constrained symbolic execution can result in false positive detections, i.e., bugs that cannot actually be reached in real executions. However, because we are careful to only snapshot the entry points to the program and its imported libraries, Bunkerbuster's results do not have this issue. Bugs found using snapshots of the program's entry point will be inherently reachable, and of relevance to the program's developers. Conversely, for API snapshots, so long as Bunkerbuster halts its analysis at the return from the called function,³ any discovered bugs *may not* be reachable within the context of the program that was recorded, but *may* be reachable by other programs that also import the

³Analysis beyond this point can yield false positives because the returned value is under-constrained.


```

0000000000001142 <foobar>:
...
114a: mov    %rdi,-0x18(%rbp)  u:{rdi}
114e: mov    %esi,-0x1c(%rbp)  u:{rdi,esi}
1151: mov    %rdx,-0x28(%rbp)  u:{rdi,esi,rdx}
...
1164: mov    -0x18(%rbp),%rax
1168: add    %rdx,%rax
116b: movzbl (%rax),%eax
116e: movsbl %al,%eax          u:{rdi[s8],esi,rdx}
...
1183: mov    -0x28(%rbp),%rax
...
118c: callq  *%rax             u:{rdi[s8],esi,rdx[c]}

```

Figure 5.3: Binary-only scenario, with color added for clarity. The boxes show the usage of non-clobbered values. The first snippet reveals `foobar` has 3 arguments, the next reveals that the RDI argument is a `char` pointer (denoted `[s8]`), and the last reveals RDX is a code pointer (`[c]`).

same library, making the results relevant to library developers. In this way, Bunkerbuster decomposes long traces into smaller segments, simplifying the symbolic execution.

One small caveat we discovered while designing Bunkerbuster is that while most inputs within snapshots should be symbolized, *code pointers passed to APIs should not*. The reason is that some APIs are designed to accept code pointers, which may serve as callback functions, helper functions, and more. If these are replaced with unconstrained symbolic variables, then their use will be difficult to distinguish from control flow hijacking, despite being intended behavior. The reason why will become clearer in subsection 5.3.5, which describes how Bunkerbuster detects overflow bugs.

Whereas program arguments adhere to a fixed memory layout, as specified by the operating system, the locations and types of API arguments has to be recovered by Bunkerbuster. Recall from subsection 5.2.2 that we aim to handle both public and private APIs. Consequently, we propose two approaches for inferring and symbolizing the input arguments, one based on parsing C/C++ headers and the other based on binary-only analysis.

Source-Based Inference When source headers are available, we use a C/C++ parser to read the API’s function prototype into an abstract syntax tree (AST), terminating with basic

data types of known size (e.g., `int`, void pointer). All non-pointer types are treated as data. For pointers, if the type is a function prototype, then it is a code pointer. Similarly, pointers to basic data types are data. However, it is ambiguous when the type is void, which could point to data or code. In such cases, the parser assumes the pointer points to code to remain conservative. The result is a data structure defining the offset, size, and type of each element for each argument. This is then combined with the calling convention for the architecture being analyzed to pinpoint these elements in registers and memory. Notice that because libraries are shared between programs, factors like padding are treated consistently across systems and is easy to account for. When data pointers point to buffers of arbitrary length, they are replaced with new large buffers of unconstrained symbolic bytes to test for overflows.

Binary-Based Inference When headers are unavailable, our analysis leverages the recorded trace, shown with a concrete example in Figure 5.3. Bunkerbuster steps through the traced basic blocks in order and tracks where registers and stack values are used in operations versus being clobbered by writes. If a non-clobbered value is used, it is likely an argument. The type is inferred based on how the loaded value is manipulated. If it appears in a call, it is treated as a code pointer. If it is used in subsequent loads, it is a data pointer. Otherwise it is treated as a basic data type. It is possible for this approach to miss a parameter if it is never used, however we did not observe this in our evaluation.

During implementation, we tested the robustness of this approach by comparing its outputs against those of the source-based technique and verifying that they match. We include a breakdown of the tested libraries in Table A.1 of the appendix.

5.3.3 Symbolic State Reconstruction

Once Bunkerbuster has a symbolized snapshot for a starting state (subsection 5.3.2), and a corresponding trace segment (subsection 5.3.1), it then needs to recover the intermediate

program states that cover the recorded execution path. Notice that Bunkerbuster cannot simply take more snapshots because doing so comes at a performance cost, so instead our solution is to use symbolic execution to recover the missing states. As an added benefit, this will also enable Bunkerbuster to consider states beyond what was concretely executed, potentially finding additional bugs.

To perform the reconstruction, each instruction is emulated and constraints are added to the programs state to encode all possible data that can reach the current point in the execution. When a branching instruction is encountered, a *satisfiability modulo theories* (SMT) solver evaluates the accumulated constraints to yield reachable successor states. However, Bunkerbuster initially focuses on only recovering the path that was recorded, so it only keeps the successor that matches the next address in the trace. In this way, there is only 1 active state per step.

CPU Architecture-Specific Considerations Although following a linear sequence of executed addresses is conceptually intuitive, in practice real-world encoding schemes can introduce ambiguities that must be resolved carefully. One prevalent case occurs in processors supporting extended instruction sets (ISAs), such as IA64 and AMD64. Among the added instructions are complex operations like Intel’s “repeat” instructions, which allow compilers to implement an entire loop in one instruction.⁴ When executing concrete memory in a real processor, these instructions are deterministic, so Intel PT ignores them. However, in symbolic analysis, two successor states become reachable if symbolic memory is accessed: one that completes the instruction and another that continues its iterating. Since the trace offers no guidance, our solution is to “iterate” on the repeat instruction as many times as possible, given the symbolic constraints, because this is most likely to reveal to an overflow bug. Once the analysis must advance past the complex instruction, it synchronizes back to the trace and continues.

⁴`strlen` can be implemented in IA64 using a single `repnz scas` instruction.

5.3.4 Use-After-Free & Double Free Bugs

The UAF module (also covering DF) relies on a value set analysis (VSA) over the symbolic states. However, unlike a typical VSA that tracks the *concrete* pointers to allocated and freed memory buffers, Bunkerbuster’s VSA is performed using *symbolic* pointers, constrained by the symbolic execution to encode all possible values at the current program state. This carries several advantages. For example, in the evaluation presented in subsection 5.4.3, we encounter a case where AddressSanitizer, having access to only a single concrete input provided by a fuzzer, concluded that a pointer passed to `free` could cause an invalid free, since the pointer’s value was an address that was not allocated. However Bunkerbuster, using the symbolic representation of that same pointer, detected that there were other satisfiable values for it, some of which corresponded to addresses that *were* allocated, revealing the bug to really be a UAF, which is more severe.

Detection To perform the VSA, we assume knowledge of the syntax of memory management functions in advance, which is easily achievable in practice because most programs rely on a few standard implementations. Even when a wrapper is placed around memory management functions for portability across systems, we find that Bunkerbuster can track the underlying standard library while disregarding the wrapper. In the case of the real-world programs in our evaluation dataset, they all rely on either `libc` or `jemalloc`. There are also algorithms to automatically detect and infer memory management functions [196], which can be incorporated in future work, but are not implemented in our initial prototype.

When the program calls into an allocation function, Bunkerbuster records the locations of the pointer, the allocated buffer, and its size, in an *allocated* set. If the size is symbolic, Bunkerbuster evaluates it to its maximum satisfiable value. When a pointer is passed to a `free` function, Bunkerbuster evaluates the symbolic constraints to determine which buffer is being referenced and moves it into a *freed* set. Notice that the referenced buffer could be one that is already freed, in which case a DF bug is detected. Similarly, Bunkerbuster

checks any dereferenced pointers in each discovered state, and if one can point to a freed buffer, it is a UAF bug.

Search With symbolic buffer and pointer metadata recovered via VSA, Bunkerbuster’s search strategy first recovers function boundaries, which are determined based on the calls and returns contained within the trace, and then labels which functions manipulate heap based on the collected pointer metadata. The implementation of the algorithm to recover all the accessed memory addresses for a basic block is provided in the appendix as algorithm 4. Bunkerbuster then searches these functions for additional states using depth-first-search to see if they can cause a UAF or DF. By sticking only to functions reached during tracing, Bunkerbuster can avoid path explosion by returning to any of the traced states reconstructed in subsection 5.3.3.

Figure 5.4 shows a partial CFG for a UAF bug found with this strategy. The initial states from the trace are shown in white, connected by black edges. Nearby states found during exploration are shown in blue, revealing the blue path to a free. Further exploration of this and other traced functions then reveals the red path leading to a UAF.

Root Cause Once detected, the symbolic root cause report prepared for the developers contains the basic block that allocated the accessed buffer, the one that freed it, and the one that performed the buggy access. To propose a preliminary patch, the module constructs a CDG over the path leading to the UAF, revealing all the conditional branches the violating basic block’s reachability depends on. The branch nearest to the violator is selected (based on shortest path) and the state for the alternate branch (which did *not* cause a bug) is checked for its constraints. If these constraints contradict the UAF state, this becomes the preliminary patch, otherwise the report advises the developers to place a new guard condition before the violating basic block.

5.3.5 Overflow & Format String Bugs

Detection Bunkerbuster’s overflow detection module focuses on bugs that can manifest into control flow hijacking, taking advantage of the fact that all external input data is symbolized in the starting memory snapshot (subsection 5.3.2). Consequently, if the program counter for a state ever becomes symbolic due to one of these variables, this means external input can directly control the execution of the code via crafted inputs, which is a serious vulnerability. Notice that symbolic constraints are already propagated by the symbolic execution, so detection is performed by querying the SMT solver to check whether the program counter is symbolic (i.e., has more than 1 satisfiable value). If it is, an overflow has occurred.

Search Bunkerbuster searches for overflows by identifying all the loops that appear in the trace, which is accomplished by transforming the linear execution into a CFG and then using a depth-first search to find all the backward edges in the graph.⁵ Once identified, the module’s search strategy is to *stress* the known loops by iterating through them as much as possible (given the symbolic constraints) and then observe the side effects in subsequent successor states. However, stressing every loop encountered in the trace is time consuming, so Bunkerbuster employs two strategies to prioritize loops that are more likely to lead to overflows.

First, not all loops write to memory and for the ones that do, not all writes rely on a changing pointer value or offset, which is necessary to cause an overflow. We coin this behavior as *stepping* and Bunkerbuster checks for instances of it in the recorded trace. Specifically, for each visit to each loop in the reconstructed CFG, Bunkerbuster collects the target memory address of each write instruction and examines how its target changes over each iteration. If there exists a write instruction such that each invocation targets an always increasing (or decreasing) memory address, then the loop is prioritized as a candidate for

⁵See NetworkX’s `find_cycle` algorithm for a suitable implementation.

overflow analysis. An implementation of this algorithm is provided in the appendix as algorithm 5. Notice that since symbolic states are examined, the pointers can have multiple satisfiable values, so the satisfiability test for the stepping criteria is performed by the SMT solver.

Next, the module takes into special consideration loops that engage in *counting* behavior because subsequent overflow candidates may have control dependencies to the computed value. For example, a string copying method can be implemented as two loops, the first counting how many bytes are in the string and the second copying them, as shown in Figure 5.5. If our algorithm blindly stresses the counting loop, its final written value will be maximized and then the subsequent copying loop will have to iterate the appropriate number of times. However, once the module detects that a code or return pointer in memory has been corrupted, continuing to stress the loop is excessive. Our solution is to detect counting loops, similarly to how stepping is detected, and replace the final value with a new symbolic variable constrained to all of the original's intermediate values. For example, if the counting loop in Figure 5.5 can iterate up to 4,096 times, rather than constraining `length` to 4096, it is replaced with the symbolic integer set $[1, 4096]$. This allows it to discover the subsequent bug in fewer steps.

Once candidate loops have been stressed by iterating them as much as possible (or until a return pointer on the stack is overwritten), the module explores successor states until a return executes. If a control flow hijack is not detected by this point, it moves on to the next candidate until none remain.

Root Cause To generate a report, the module first includes the basic block where the hijack occurred. Next, it identifies the memory location of the symbolic pointer that triggered the hijack using the symbolic constraints. An implementation of this algorithm is in the appendix (algorithm 6). Next, it rewinds backwards through the predecessor states until it finds the one that first made the pointer symbolic and adds it to the report. The

module then generates a CDG for the execution path leading to this state, selects the nearest conditional branch in terms of shortest path, and checks the alternate branching states for contradicting constraints. If any are found, they become the preliminary patch for the developers, otherwise a new guarding branch should be placed before the corrupting state.

Format String Bugs We find that unlike UAFs, DFs, and overflows, FS bugs are usually not as constrained by control flow. Specifically, if a call site contains a FS vulnerability, reaching it via *any* path is sufficient for discovering the bug. For this reason, we do not employ a tailored search strategy for FS and instead perform detection over the states found by the other exploration modules. In practice, format specifier strings should always be constant, turning them into read-only data at compile time. Consequently, for each call to a known format string function (e.g., `printf`), the module checks whether the specifier pointer or any of its content is symbolic. If it is, this means input data is able to directly control the specifier, which is a bug. In such cases, the root cause report identifies the caller of the format string function and the predecessor state that wrote to the specifier.

5.4 Evaluation

We aim to answer the following questions in our evaluation:

1. *Is Bunkerbuster able to detect bugs within our covered classes?* We select 15 widely-used commodity programs and generate a corpus of benign inputs. After analysis, Bunkerbuster finds 39 bugs, of which 8 are new, never before reported cases. We manually verify the presence of all bugs. 1 EDB and 3 CVE IDs have been issued and patched by developers using Bunkerbuster’s reports. We also measure Bunkerbuster’s code coverage to show that its exploration converges.
2. *Is Bunkerbuster’s exploration effective compared to prior techniques?* We compare against AFL [20] and QSYM [64] on our target programs, starting from similar seeds. After 1 week, Bunkerbuster finds 8 bugs missed by the other systems.

Table 5.1: System Evaluation for Real-World Programs

ID	Type	Program	Component	Version	# Traces	# Novel	(%)	# Snaps	# BBs	# APIs	Find (s)
EDB-47254	Ovf	abc2mtex	main	1.6.1	1,209	166	13.7	166	124,248	26	27
CVE-2004-0597	Ovf	Butteraugli	libpng	1.2.5	176	78	44.3	78	1,648,790	112	15
CVE-2004-1257	Ovf	abc2mtex	main	1.6.1	1,209	166	13.7	166	50,120	26	97,188
CVE-2004-1279	Ovf	jpegtovai	main	1.5	333	18	5.4	18	46,313	15	82,050
CVE-2013-2028	Ovf	Nginx	main	1.4.0	5	4	80.0	312	809,977	64	4,538
CVE-2009-5018	Ovf	gif2png	main	2.5.3	1,709	39	2.3	39	24,210,156	12	10
CVE-2017-7938	Ovf	dmitry	main	1.3a	10	10	100.0	10	56,488,245	20	78
CVE-2017-9167	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	23,764,196	84	2,659
CVE-2017-9168	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	74,252	84	3,868
CVE-2017-9169	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	74,753	84	728
CVE-2017-9170	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	74,543	84	2,868
CVE-2017-9171	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	33,965,824	84	786
CVE-2017-9172	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	95,561,159	84	6,038
CVE-2017-9173	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	33,965,824	84	5,995
CVE-2017-9191	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	23,070,692	84	5,364
CVE-2017-9192	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	23,764,196	84	3,010
CVE-2018-12326	Ovf	redis-cli	main	4.0.9	1,253	31	2.5	31	112,144	40	49
CVE-2018-12327	Ovf	ntpq	main	4.2.8p11	15	11	73.3	11	194,489	42	1,316
CVE-2018-18957	Ovf	GOOSE	libec61850	1.3	5	2	40.0	6	65,198	9	11
CVE-2019-14267	Ovf	pdfressurect	main	0.15	199	76	38.2	76	8,901,803	18	16,171
* CVE-2020-9549	Ovf	pdfressurect	main	0.19	199	76	38.2	76	9,497,364	18	14,744
* CVE-2020-14931	Ovf	dmitry	main	1.3a	10	10	100.0	10	165,235	20	123
Will Not Fix	Ovf	GIMP	glibc	2.2.5	26	25	99.2	21,572	46,757,444	278	1
* CVE-2020-35457	Ovf	GIMP	glib	2.58.3	26	25	99.2	21,572	60,406,299	278	12
EDB-46807	Ovf	MiniFTP	main	1.0	7	3	42.8	33	60,849	45	7
* Patched	FS	dmitry	main	1.3a	10	10	100.0	10	125,662	20	16
CVE-2017-9162	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	95,561,159	84	3,253
CVE-2017-9163	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	90,334	84	3,253
CVE-2017-9182	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	30,386,474	84	2,873
CVE-2017-9183	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	413,022	84	3,253
CVE-2017-9190	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	40,806,309	84	3,253
CVE-2017-14103	UAF	GraphicsMagick	main	1.3.26	4	3	75.0	3	2,520,481	62	646
CVE-2019-17582	UAF	PHP	libzip	7.4.14	6	6	100.0	145	5,980,255	312	72
* Reported	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	40,632,944	84	3,253
* Reported	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	74,506	84	3,253
* Reported	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	40,632,944	84	3,253
* EDB-49259	UAF	GIMP	babl	0.1.62	26	25	96.2	21,572	46,757,444	278	15
CVE-2017-11403	DF	GraphicsMagick	main	1.3.26	4	3	75.0	3	2,513,590	62	634
CVE-2017-12858	DF	PHP	libzip	7.4.14	6	6	100.0	145	5,980,255	312	72
					Average:	189	64	1,710	19,392,602	90	7,045

* New vulnerability discovered by Bunkerbuster.

3. *Is Bunkerbuster’s root cause analysis valuable compared to existing instrumentation?* We compare Bunkerbuster’s root cause reports for Autotrace against those from QSYM with AddressSanitizer [222]. Bunkerbuster provides more accurate class labels in 4 cases.
4. *Are Bunkerbuster’s exploration heuristics effective?* We compare the exploration techniques described in section 5.3 against breadth-first and depth-first search and find that Bunkerbuster outperforms across all trials by better managing path exploration.
5. *Is Bunkerbuster feasible to deploy in terms of runtime and storage overhead?* We measure the performance and storage overheads of tracing programs using the SPEC CPU 2006 benchmark and Nginx, averaging 7.21% runtime overhead.
6. *Is Bunkerbuster’s symbolic root cause analysis over partial paths correct?* We repeat the main experiment from the original symbolic root cause analysis work [225] using Bunkerbuster and verify our prototype produces the same results.

Experimental Setup We use 1 computer to represent the end-host for tracing and 1 server to perform the analysis. Each device runs Debian Buster and contains an Intel Core i7-7740X processor, 32GB of memory, and solid state storage. Our prototype uses angr [21] as its symbolic execution engine and is implemented in 7,062 Python and 1,208 C source lines of code (SLoC).

Dataset & Selection Criteria To select our target programs for evaluation, we start by considering the packages offered in Debian’s APT repository, filtered using the C/C++, CLI, and GUI tags, to ensure we only consider standalone programs written in languages that can contain memory corruption bugs. We then cross-reference MITRE’s CVE database to isolate programs that contain or import (via libraries) code with known prior overflow,

UAF, DF, and FS vulnerabilities, as these may contain more that have yet to be discovered. From this, we randomly pick 15 programs for testing.

We also manually assemble a corpus of benign inputs for each program by examining test cases and documentation. For CLI programs, we ensure the corpus has at least one case for each possible flag. For GUI programs, we manually perform some basic actions, such as opening, modifying, and saving files. When programs require complex input formats (e.g., images), we collect valid inputs from public sources like ImageNet [218].

5.4.1 Bug Hunting in Real-World Programs

Methodology For each of the 15 real-world programs in our dataset, we allow Bunkerbuster to trace and analyze our corpus of benign inputs for 1 week. We also measure Bunkerbuster’s code coverage over forwarded traces to test whether it converges, which is relevant to determining its usability in real-world deployments.

Results Table 5.1 shows the results produced by Bunkerbuster’s analysis for the gathered data using our target programs and input corpus. In total, 39 bugs were found across the 15 tested programs. The “ID” column shows that 31 of the found bugs pertain to already publicly known vulnerabilities, whereas 8 have never been reported before. We manually inspect these cases to verify their presence. In 1 case, our prototype found a previously reported bug that the developers decided not to fix due to its performance consequences versus the relatively low security impact. 1 bug has been issued an EDB ID by Offensive Security and 3 CVE IDs by MITRE. Developers have patched them, using our system’s reports to independently review and verify their novelty and impact. Some of these bugs were highly exploitable, including a now patched remote code execution (RCE) vulnerability, triggered via a WHOIS response.

The “Type” column lists the type of each bug. In total, Bunkerbuster found 25 overflows (Ovf), 1 FS bug, and 13 UAFs/DFs. The “Program” and “Component” columns re-

port where the bugs reside, with “main” denoting the main executable object. 24 bugs were found within import libraries and 15 were inside the main object. We also report the version number of the vulnerable component for completeness. We observe that Autotrace is particularly buggy, with 17 vulnerabilities residing within the main object. Conversely, while GIMP is associated with 3 bugs, they were all found within imported libraries, demonstrating the importance of being able to analyze these APIs.

The “# APIs” column counts how many unique function imports were segmented by Bunkerbuster, using its symbolized memory snapshots and automatic prototype recovery. In other words, this is the number of APIs a human analyst would have to build scaffolding for if they were *not* using Bunkerbuster and wanted similar results. On average, 90 unique APIs were segmented per program, with counts ranging from 9 (GOOSE) to 278 (GIMP). Bunkerbuster eliminates the need to manually perform this laborious task.

The “# Traces” column reports how many traces (not segments) were recorded. On average, the end-host monitored 189 execution sessions per program. “# Novel” is the number of traces that contained at least 1 novel segment forwarded for analysis. On average, 36 were novel per program. For most programs, even with as few as 4 traces, at least 1 was filtered, demonstrating the importance of being able to identify and remove redundant data. The “# Snaps” column shows the number of trace segments and snapshots forwarded. On average, 1,710 were forwarded for analysis per program. In the case of GIMP, our input corpus yielded a comparatively high number of API snapshots. This is due to GIMP being one of the largest programs in our dataset, compiled from over 810,000 lines of C/C++ code, with a sophisticated architecture where each plugin is itself a standalone executable with additional library dependencies. For example, one of the `babl` functions found to contain a vulnerability was not invoked by GIMP directly, but rather by its plugin for loading PNG images. Trying to naively symbolically execute 46,757,444 basic blocks (from GIMP’s entry point, through the PNG plugin, into `babl`) would be difficult for prior work. Bunkerbuster succeeds thanks to its ability to segment.

“# BBs” records the number of traced basic blocks and “To Find” reports the number of seconds it took for the analysis to make its discovery. On average, traces containing bugs were 19,392,602 basic blocks long and bugs were found in 7,045 seconds, i.e., within 2 hours or so. Some bugs were found in as little as 1 second, while others took over 4 hours, depending on the complexity of the recorded behavior. Interestingly, because Bunkerbuster is able to segment and snapshot APIs, there is little correlation between trace length and the time to find bugs. For example, despite one GIMP trace being 60,406,299 basic blocks long, the bug it revealed was uncovered in 12 seconds. Conversely, several Autotrace traces of about 40,000,000 basic blocks each uncovered bugs in about 1 hour.

Figure 5.6 presents the coverage of our analysis over forwarded traces (i.e., after end-host-side filtering). To normalize each program’s curve, we present a cumulative distribution function (CDF) of the percentage of novel basic blocks discovered versus the percentage of traces analyzed. For all target programs, by the time 50% of the traces were analyzed, at least 80% of the total discovered basic blocks had been found, demonstrating that Bunkerbuster’s analysis converges. This is also consistent with the change in ratio of segments being filtered by the end-host over time.

5.4.2 Comparing Prior Exploration Techniques

Methodology We compare Bunkerbuster against AFL [20], a highly popular greybox fuzzer, and QSYM [64], a recent concolic execution hybrid fuzzer, for this experiment. We pick these systems because they work in the binary-only setting for a wide range of bug classes, whereas other prior work requires source code [62] or is limited to a single class [68, 52], which would make for an unfair comparison. For consistency, we run each system on each target program for 1 week, starting from the same corpus of seeds. For each *unique* crash (as determined by AFL and QSYM), we manually inspect it to determine the bug class and root cause. We measure which bugs are detected by each system and how many reports are generated. We present the results for Autotrace in subsection 5.4.3 as an

Table 5.2: Bunkerbuster Vs. AFL & QSYM

ID	Type	Program	BB	AFL	QSYM
EDB-47254	Ovf	abc2mtex	1	0	0
CVE-2004-1257	Ovf	abc2mtex	1	350	246
Patched	FS	dmitry	1	0	0
CVE-2020-14931	Ovf	dmitry	1	5	35
CVE-2020-9549	Ovf	pdfresurrect	1	0	0
CVE-2019-14267	Ovf	pdfresurrect	1	88	108
CVE-2017-11403	UAF	GraphicsM.	1	0	25
CVE-2017-14103	UAF	GraphicsM.	1	0	0
CVE-2018-12327	Ovf	ntpq	1	15	27
CVE-2018-12326	Ovf	redis-cli	1	18	44
CVE-2009-5018	Ovf	gif2png	1	88	163
CVE-2004-1279	Ovf	jpegtoavi	1	0	0
CVE-2004-0597	Ovf	Butteraugli	1	72	65
CVE-2018-18957	Ovf	GOOSE	1	1	1
CVE-2013-2028	Ovf	Nginx	1	0	0
EDB-46807	Ovf	MiniFTP	1	32	29
Will Not Fix	Ovf	GIMP	1	0	0
CVE-2020-35457	Ovf	GIMP	1	0	0
EDB-49259	UAF	GIMP	1	0	0
CVE-2019-17582	UAF	PHP	1	0	0
CVE-2017-12858	DF	PHP	1	7	7

extended case with crashes analyzed by AddressSanitizer.

Results The results are presented in Table 5.2. In several cases, AFL and QSYM were unable to detect vulnerabilities found by Bunkerbuster. For example, they were unable to find the FS bug in DMitry because it requires a specific set of command line arguments to reliably cause a crash. Conversely, Bunkerbuster detected that symbolic format specifiers were being passed to `libc`, alerting it to the bug even in non-crashing cases. In general, we observed that the mutation algorithms used by AFL and QSYM are not well suited for fuzzing CLIs, which is also noted in AFL’s documentation. We also observe that of the 4 UAFs listed in Table 5.2, QSYM only found 1 and AFL none. QSYM and AFL also struggled to handle GIMP and GraphicsMagick due to their size and complexity, causing them to miss 10 and 11 bugs, respectively. It is possible that these tools would perform better if an expert human analyst created scaffolding around the imported libraries, but

Table 5.3: Bunkerbuster Vs. AddressSanitizer

ID	Location	BB	QSYM + AS
CVE-2017-9167	input-bmp.c-337	Ovf	Ovf
CVE-2017-9168	input-bmp.c-353	Ovf	Ovf
CVE-2017-9169	input-bmp.c-355	Ovf	Ovf
CVE-2017-9170	input-bmp.c-370	Ovf	Ovf
CVE-2017-9171	input-bmp.c-492	Ovf	Ovf
CVE-2017-9172	input-bmp.c-496	Ovf	Ovf
CVE-2017-9173	input-bmp.c-497	Ovf	Ovf
CVE-2017-9191	input-tga.c-252	Ovf	Ovf
CVE-2017-9192	input-tga.c-528	Ovf	Ovf
CVE-2017-9162	autotrace.c-191	UAF	UNDEF
CVE-2017-9163	pxl-outline.c-106	UAF	UNDEF
CVE-2017-9182	color.c-16	UAF	UAF
CVE-2017-9183	autotrace.c-309	UAF	UNDEF
CVE-2017-9190	bitmap.c-24	UAF	BADFREE
Reported	pxl-outline.c-140	UAF	-
Reported	pxl-outline.c-609	UAF	-
Reported	color.c-10	UAF	-

in GIMP’s case, there are *70 unique libraries* with *1,288 exported functions* to consider. Bunkerbuster relieves the analyst of this task.

In almost all of the cases where the prior systems found the same bug as Bunkerbuster, the former generated over 15 redundant reports. This is because AFL and QSYM rely on stack traces to determine the uniqueness of crashes, which are sometimes unreliable, such as when dealing with overflows. For example, QSYM generated *108 reports* for CVE-2019-14267 and *246* for CVE-2004-1257 because a stack corruption misled it to classify each crash as unique. Bunkerbuster avoids this fatigue inducing redundancy using its symbolic root cause analysis, resulting in only 1 report per bug. Curiously, while QSYM generated more unique crashes than AFL overall, it only led to the discovery of 1 additional bug. This is likely due to the sparsity of bugs in real-world programs.

5.4.3 Comparing Prior Root Cause Techniques

Methodology In this experiment, we perform the same evaluation as described in subsection 5.4.2, with two adjustments made. First, we focus explicitly on Autotrace for this

experiment because it yields by far the most bugs out of all the real-world programs. Second, we use AddressSanitizer (AS) to automatically triage the crashes uncovered by AFL and QSYM, as is common practice in real-world bug hunting. This allows us to compare the quality of Bunkerbuster’s root cause analysis to AS.

Over the course of this experiment, QSYM and AFL found 1 crash identified by AS as integer overflow and 1 out-of-bounds read, which we exclude from the results since these are classes outside the current scope of Bunkerbuster. For clearer presentation, we translate binary addresses in our figures to source code line numbers using debug symbols, postmortem. *No system had access to the symbols during the experiment.* In our results, AFL and QSYM found the same set of bugs, so we only present QSYM for brevity.

Results After 1 week of analysis, Bunkerbuster yields 17 bug findings. Conversely, QSYM yields 14 bugs after triaging by AS. The two sets of reports are presented side-by-side in Table 5.3. Bunkerbuster finds all of the UAFs and overflows identified in the AS reports along with 3 UAFs never before reported. Upon investigation, we discover that the new UAFs reside in code branches missed by QSYM’s exploration. We believe that given more time, QSYM would eventually find inputs to reach these branches, whereupon AS would be able to triage them correctly. However, QSYM did not accomplish this within the allotted time whereas Bunkerbuster did.

Another interesting observation is that for 4 CVEs, Bunkerbuster is able to give more precise classifications than AS (bold in Table 5.3). In 3 cases, AS reports undefined behavior (UNDEF), meaning that despite QSYM detecting a crash and providing a concrete input to AS for analysis, AS still could not decide on a class for the bug. Conversely, Bunkerbuster correctly identifies the bugs to be UAFs. In 1 case, AS reports a bad free (BADFREE), meaning that the address being freed was never allocated, but Bunkerbuster, using its symbolic constraints, is able to correctly identify that a more carefully chosen input can turn this bug into a UAF. In summary, our system finds 3 UAFs missed by QSYM

and yields more accurate classifications than AS in 4 cases.

5.4.4 Effectiveness of Exploration Techniques

Methodology To validate whether our proposed exploration techniques enable Bunkerbuster to better search program states while avoiding path explosion, we compare against two baselines: breadth-first and depth-first search (BFS, DFS).⁶ Notice that DFS is the default exploration technique used by popular symbolic analysis frameworks [21].

To conduct the experiment, we randomly pick 1 trace for each of the real-world programs from our dataset and allow each technique (BFS, DFS, and ours) to explore states for 1 hour per program. Once the time limit has expired, we halt Bunkerbuster and count the number of unique basic blocks discovered by each technique. Since some target programs are slower to explore than others, we normalize our results by dividing the counts by the total number of unique blocks discovered globally, across all evaluated techniques, yielding a percentage from 0% to 100%.

Results The results of our experiment are presented in Figure 5.7. Across all 15 real-world programs, Bunkerbuster’s exploration techniques outperform BFS and DFS. Specifically, for about half of the programs, Bunkerbuster’s techniques find all the basic blocks BFS and DFS find, and more. Bunkerbuster also finds more than double the number of basic blocks than the baselines in many cases, such as in Dmitry and MiniFTP.

The biggest contrast occurs in Butteraugli, where BFS and DFS only find about 2% of the blocks discovered by Bunkerbuster. Upon investigation, we discover that BFS and DFS both get stuck in `libz`’s CRC32 checksumming function. Such functions are notorious for inducing path explosion [240]. Bunkerbuster’s techniques avoid this function using heuristics to recognize that the contained code is unlikely to cause our targeted bug classes (e.g., the contained loops do not perform stepping writes, subsection 5.3.5).

⁶We include these baselines in the open sourced code repository for reproducibility.

Another stark contrast occurs in MiniFTP, where the baselines only find about 10% of the blocks Bunkerbuster finds. In this case, BFS, DFS, and Bunkerbuster all focus on MiniFTP’s function for loading the settings file, which is expensive to explore because the code is densely packed with string comparisons, another well-known source of path explosion. However, whereas BFS and DFS explore this function naively, yielding lower code coverage and uncovering no bugs within the allotted time, Bunkerbuster prioritizes the contained loops using our described heuristics and finds EDB-46807 in under 10 seconds.

In summary, the heuristics we propose for Bunkerbuster do in fact help it explore more code in our evaluated dataset in less time than BFS or DFS. In many cases, the contrast is significant, with Bunkerbuster’s exploration techniques discovering more than double the number of basic blocks within the allotted time.

5.4.5 Performance & Storage

Methodology To measure the performance and storage overheads of Bunkerbuster, we start with the SPEC CPU 2006 benchmark with a storage quota of 10 GB per end-host. We use the 2006 version deliberately so our numbers can be directly compared against other prior full-trace⁷ PT systems [25, 47]. Since these workloads are CPU intensive, we consider this to be the worst realistic case for our system. For another comparison point, we also evaluate Nginx running PHP with default settings, stressed using ApacheBench to serve 50,000 HTTP requests for files ranging in size from 100 KB to 100 MB, which we consider to be an I/O bound workload. Performance overhead is measured with tracing and API snapshots enabled versus running without the kernel driver installed for the baseline. Storage is the at-rest size of all collected data. Overheads are calculated as $(P - B)/B$ where B is the baseline metric and P is with Bunkerbuster.

Results Figure 5.8 shows the metrics for the SPEC benchmark. The average tracing overhead is 7.21% with a geometric mean of 3.83%, which is within 1% of prior systems

⁷As opposed to systems that use small finite buffers [233, 232].

Table 5.4: Symbolic Root Cause Verification

CVE / EDB	Type	# BBs	ΔRC	L	P	M
CVE-2004-0597	Ovf	41,625,163	247	Y	[174]	Y
CVE-2004-1257	Ovf	53,490	6,319	Y	-	-
CVE-2004-1279	Ovf	67,772	26,216	Y	-	-
CVE-2004-1288	Ovf	74,723	33,211	Y	[175]	Y
CVE-2009-2629	Ovf	300,071	28	Y	[176]	Y
CVE-2009-3896	Ovf	283,157	59	Y	[177]	Y
CVE-2009-5018	Ovf	90,738	1,848	Y	[179]	Y
CVE-2017-7938	Ovf	100,186	4,051	Y	-	-
CVE-2017-9167	Ovf	75,404	1,828	Y	-	-
CVE-2018-12326	Ovf	291,275	8	Y	[178]	Y
CVE-2018-12327	Ovf	374,830	122,740	Y	[180]	Y
CVE-2018-18957	Ovf	65,198	94	Y	[181]	Y
CVE-2019-14267	Ovf	128,427	83,123	Y	[182]	Y
EDB-15705	Ovf	260,986	19,322	Y	-	-
EDB-46807	Ovf	60,849	335	Y	-	-
CVE-2017-9182	UAF	132,302	296	Y	-	-
CVE-2017-11403	UAF	2,316,152	38	Y	[187]	Y
CVE-2017-14103	UAF	2,316,133	38	Y	[187]	Y
CVE-2017-12858	DF	5,980,255	51	Y	[188]	Y
CVE-2005-0105	FS	127,209	1	Y	[189]	Y
CVE-2012-0809	FS	108,442	1	Y	[190]	Y

that record full PT traces, demonstrating that the filtering and snapshot steps performed by Bunkerbuster incur negligible additional overhead. Similar to prior work, the storage requirement is also large for some cases, averaging 1,348 MB/min, however all tests completed in under 1 minute, so the average final size is 110 MB per workload. We believe this is tolerable given that the data is forwarded to a storage server and with a 10 GB quota per end-host, dozens of executions can be stored at a time for analysis. Recall that this storage is temporary. Once a trace is analyzed, it can be discarded to free space. The bandwidth required to transfer traces currently makes Bunkerbuster better suited to enterprise LANs/WANs as opposed to end-hosts distributed across the internet.

Figure 5.9 shows the results for our Nginx benchmark. Here the average performance overhead is only 2% with 1.6 MB of data generated, on average, per HTTP request. With a quota of 10 GB, traces corresponding to thousands of requests can be buffered at a time. Requested file size had little impact on our results.

5.4.6 Verifying the Root Cause Analysis

Methodology We trace proof of compromise exploits targeting overflow, UAF, DF, and FS vulnerabilities, for the same dataset used in the original symbolic root cause analysis work [225]. We then analyze the recorded traces with Bunkerbuster. In each case, we verified that our detection modules pinpoint the concise root cause of the vulnerability, in accordance with the prior work’s results.

Results The results of our evaluation are summarized in Table 5.4, which shows the number of basic blocks in each trace, the number of blocks between where the bug was detected and its determined root cause, whether the root cause was correctly located, whether a patch exists, and if so, whether the recommended constraints match the official patch. In total, 21 bugs were evaluated. As the table shows, Bunkerbuster’s detection modules are able to accurately detect and localize all 21 of the tested exploits, even when traces are over 1,000,000 basic blocks long and contain bugs that do not manifest into an observable corruption until over 100,000 blocks from the root cause. This gives us confidence that our symbolic root cause analysis is correctly designed, despite now working over partial traces in multi-path exploration.

5.5 Limitations & Threats to Validity

Scope of Target Programs Our current prototype is evaluated on benign, unobfuscated, Linux binaries. Further work is required to handle malware, packing, and virtualization, which fall outside the intended scope for this system. The current prototype also skips dynamically generated code (e.g., JIT compilation), however our driver is capable of recording and decoding it. Although our prototype focuses on Linux, the analysis is implemented for VEX IR, which is architecture independent and can be ported to other OSes that support PT, assuming the necessary system calls are modeled.

Scope of Detected Bug Classes Bunkerbuster currently supports detection of overflow, UAF, DF, and FS bugs, but these are not the only types of memory corruption that can occur in programs written in unsafe languages like C/C++. However, all approaches to bug detection have class limitations. For example, the systems we compare against (AFL, QSYM) rely on crashes as indicators of buggy behavior, and consequently cannot detect non-crashing bugs, such as ones caught by exception handlers. Conversely, it is possible for Bunkerbuster to miss bugs that reside in program states that it cannot reach within the allotted time. It is also possible for Bunkerbuster to miss overflows that cannot corrupt the program counter. Detecting UAF, DF, and FS bugs relies on knowing which functions manage dynamic memory and accept format specifier strings in advance. The search strategies proposed in section 5.3 are used only to prioritize certain paths and therefore do not limit Bunkerbuster’s total detection capabilities.

Reachability of Detected Bugs As explained in subsection 5.3.2, bugs found using snapshots taken from the program’s entry point are inherently reachable via input arguments. Conversely, bugs found via API snapshots may not be reachable via the analyzed program, but may be reachable by other programs that also import the same library. In such cases, we reported the bugs to the library maintainers, who decided to patch in most cases.

Severity of Detected Bugs Our prototype does not currently analyze the exploitability of uncovered bugs, however our approach is compatible with automatic exploit generation techniques [70]. Our system has found confirmed 0-day RCE vulnerabilities, demonstrating the security relevance of our techniques.

In one case, Bunkerbuster found a bug that the developers decided not to patch, labeled “Will Not Fix” in Table 5.1. In this lone case, the developers acknowledged the bug’s existence, but decided that the performance cost of fixing it was too high, and instead cautioned downstream developers to take care in validating the inputs passed to the relevant library API.

Trace Length vs. Analysis Time Given the computationally expensive nature of symbolic execution, it is not surprising that minutes of real-time execution can take hours to analyze. However, our experiments in section 5.4 show that the analysis converges, making novel traces less frequent.

Snapshot Frequency vs. Performance Overhead In Bunkerbuster’s current implementation, one snapshot is taken per API per execution. This is reflected in the evaluation results presented in subsection 5.4.5 and is not impacted by prior traces or analysis because in order to perform filtering, the end-host must first capture and decode the trace. In other words, the runtime overhead is the same regardless of whether a particular trace segment or snapshot is ultimately filtered or not.

5.6 Privacy & Legal Considerations

In the evaluation, we setup an end-host and an analysis server as separate machines to emphasize the decoupled nature of Bunkerbuster’s design. However, it is important to point out that analyzing control flow reveals some information about the values of data variables due to program control dependencies.

The threat of control flow leaking sensitive data has been well-studied by the side-channel research community [55], and some sensitive applications (e.g., cryptography) use hardened code to mitigate, however leakage in the context of traces recorded by PT has *not* been formally studied, to the best of our knowledge. Consequently, we envision the end-hosts and analysis servers belonging to the same or trusted parties where leakage is not an issue. However, it is possible for these machines to belong to different parties, raising privacy and legal concerns (e.g., Europe’s General Data Protection Regulation, a.k.a., GDPR). Further research is required to fully understand this risk, which is outside the scope of this work. Notice however that there exists prior work on sanitizing artifacts like crash dumps [141], some large corporations may already be recording PT traces from

end-users [241], and once the analysis is distilled into a root cause report, its privacy risk diminishes, as can be seen in the example report shown in the appendix.

5.7 Conclusion

We propose Bunkerbuster, a system for automated data-driven bug hunting of memory corruption bugs using symbolic root cause analysis. Our design leverages PT and sparse memory snapshots to symbolically reconstruct execution traces and explore nearby paths to uncover overflow, UAF, DF, and FS vulnerabilities. We implement our prototype and evaluate it on 15 real-world Linux programs, where it finds 39 bugs, 8 of which are never before reported. 3 have been independently verified by MITRE, issued CVE IDs, and patched by developers using Bunkerbuster’s reports, validating our prototype’s usefulness. Bunkerbuster finds 8 bugs missed by AFL and QSYM in our target programs and correctly classifies 4 more bugs that AS mislabeled. Bunkerbuster achieves this with 7.21% performance overhead and reasonable storage requirements.

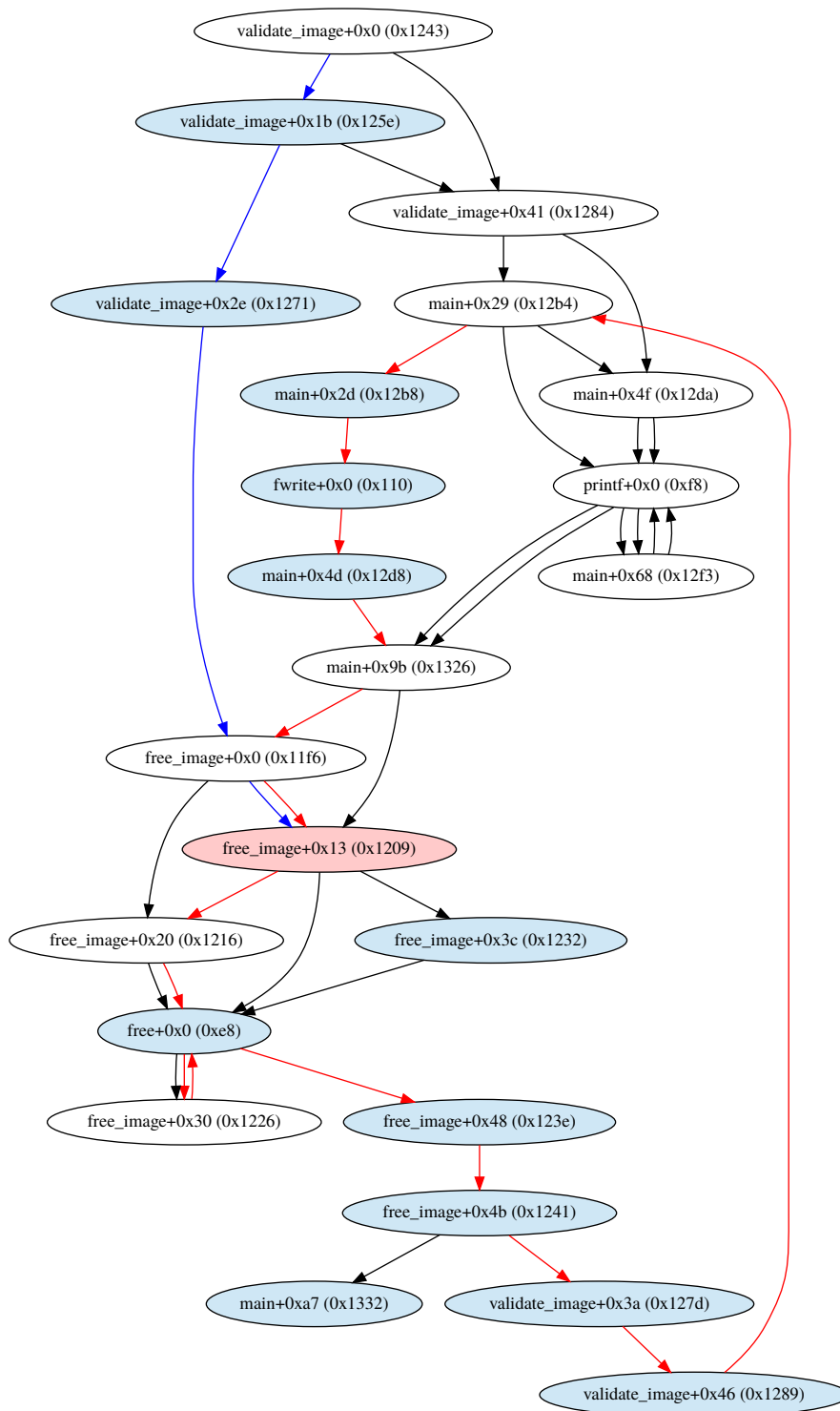



Figure 5.4: CFG created by the UAF module for a real-world case (subgraph shown for brevity). Black edges are the path traced by PT and blue nodes are states the module discovered. The blue edges show a discovered path leading to a free, followed by the red path leading to a UAF bug (red node).


```

1. void my_strcpy(char *src, char *dst) {
2.     int length = 0;
3.     char *ptr = src;
4.     // "counting" loop
5.     while (*ptr) {
6.         ptr++;
7.         length++;
8.     }
9.     
10.    // "stepping loop"
11.    for (int i = 0; i < length; i++) {
12.        dst[i] = src[i];
13.    }
14.    dst[length] = 0;
15. }
16.
17. void foobar() {
18.     char *m_dst[128];
19.     char *m_src = {'A' * 4096, 0};
20.     my_strcpy(src, dst);
21. }

```

~~RCX := [4096]~~
RCX := [1, 2, ..., 4096]

~~RCX := [1, 2, ..., 4096]~~
RCX := 132

Figure 5.5: Counting loop example. Here the number of iterations of Line 12 depends on length, set by the loop starting at Line 5. When foobar passes my_strcpy a 4097 byte string, the register holding length (RCX) would normally become 4096 by Line 9. Our module overwrites RCX with a symbolic variable, allowing Line 11 to exit sooner, and then verifies the control hijack via a corrupted return pointer at Line 21.

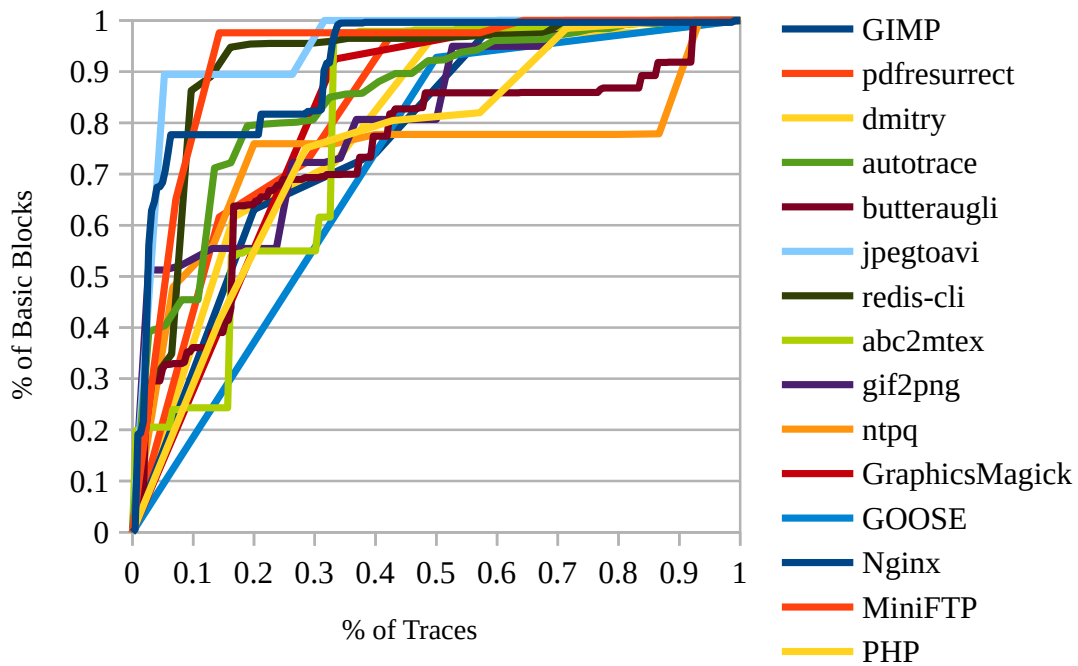


Figure 5.6: Basic block coverage for traces forwarded to the analysis, cumulatively.

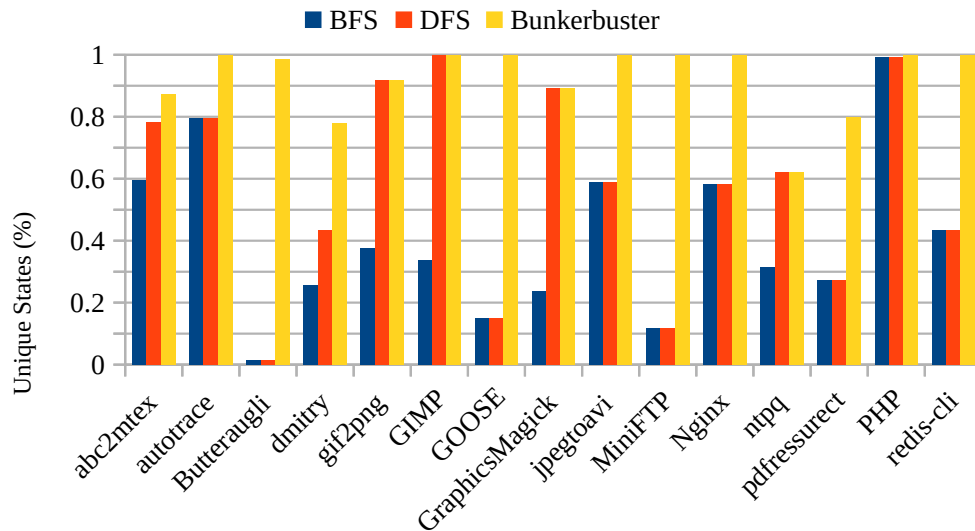


Figure 5.7: Percentage of unique basic blocks discovered using breadth-first search, depth-first search, and our proposed exploration techniques. Our techniques outperform the base-lines across our entire dataset of 15 real-world programs.

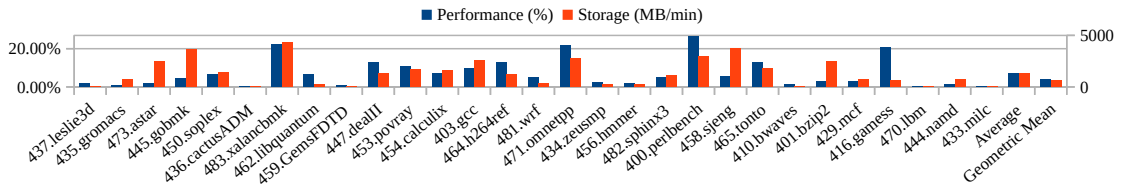


Figure 5.8: Performance and storage for tracing the SPEC CPU 2006 benchmark. The average overhead is 7.21% and the geometric mean is 3.83%. The average trace size is 1,348 MB/min and the geometric mean is 602 MB/min.

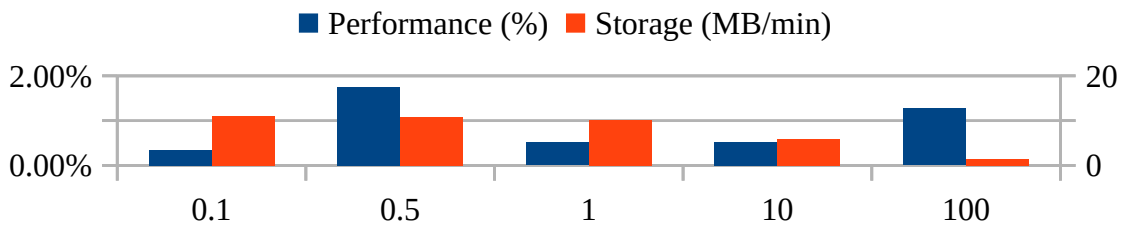


Figure 5.9: Overheads for tracing Nginx. The performance overhead is under 2% and the maximum storage is 1.6 MB per request.

CHAPTER 6

CONCLUSION

In this dissertation, I presented the growing problem of publicly disclosed vulnerabilities not being addressed in a timely manner, granting adversaries a window of opportunity for exploitation. In response, I proposed the need for new systems and techniques that can automate the steps between bug discover and bug remediation to bridge the automation gap and deny adversaries the opportunity to cause harm. In this context, I proposed a novel hardware-assisted approach to detecting, localizing, and preventing software bugs and attacks centered around the idea of control-oriented record and replay. Specifically, my approach makes novel use of PT, an advanced hardware feature available in most commodity processors, to record directly from production systems the necessary telemetry about attacks and benign user activities to extract the information system administrators and developers need to secure their programs and systems.

To demonstrate the versatility of my approach, I presented three unique systems — ARCUS, MARSARA, and Bunkerbuster — targeting applications in explaining IDS alerts to admins and developers, preventing attackers from shaping their exploits to tamper with data provenance-based forensic analysis, and proactively hunting for and explaining bugs. These systems serve both developers and system administrators while incurring overheads that are low enough to be deployable on production systems. In terms of real-world impact, these systems have led to the detection and remediation of over 18 novel 0-day vulnerabilities in real-world software. Thanks to the human readable reporting by Bunkerbuster, the third and most general of my presented solutions, all these vulnerabilities have been fixed by software developers.

Moving forward, Bunkerbuster continues to analyze traces of software available in the Debian ecosystem and report findings to developers for patching. All three systems are

also open source, including their evaluation datasets, to help promote future work towards PT-based solutions to the automation gap I presented, as well as other broader security problems. In particular, the code repository for ARCUS and Bunkerbuster has been forked by 14 researchers at academic institutes across the world. In the broader scope of systems security, solutions based on hardware-software co-design, like the ones I presented here, are going to play an increasing role in securing computing infrastructure. While the presented solutions lay the groundwork for demonstrating the feasibility of hardware-assisted, control-oriented record and replay using PT, there is still more work that can be done to increase its generality and to transition these ideas into practice.

First, while I have proposed ways to manage the amount of trace data yielded by PT, there is still more work that can be done to reduce trace sizes and to encode traces in an encoding that is searchable without requiring decompression. Advances in this direction will reduce the storage requirements of PT-based defenses, reduce performance overhead caused by I/O bottlenecks, and reduce latency for online applications that target prevention.

Second, the work presented here demonstrates feasibility only in the context of typical user programs. There is still more work that needs to be done in applying the proposed concepts to additional important domains like kernels, embedded systems, and cyber-physical systems. These domains incur new technical challenges arising from different execution models and additional constraints, namely energy usage. For example, Bunkerbuster's end-host recording is not suitable yet for deployment on mobile phones.

Third, while ARCUS and Bunkerbuster demonstrate the feasibility of addressing low-level memory bugs, and MARSARA demonstrates a high-level semantic bug, there are still plenty of other important bug classes that require modeling to increase the scope of these systems. One such example is race conditions, which is a particularly difficult bug class to model that can nevertheless give rise to severe vulnerabilities. Another is vulnerabilities that can give rise to working exploits without hijacking control flow. Detecting such bugs requires more accurate information about software data types than is currently available in

the proposed systems.

In the longer term, the viability of hardware-assisted security solutions can be improved by a redesign of the PT hardware. When companies like Intel originally designed their PT implementations, they did so with a focus on helping developers debug tricky software bugs in their development environments. The way that I (and others [25]) leverage PT for security exceeds this scope, giving rise to the technical challenges overcome by ARCUS, MARSARA, and Bunkerbuster. A redesigned PT that guarantees no data loss at the hardware level and that can selectively record memory data values at runtime would go a long way towards improving the efficiency and security guarantees of future work.

However, even with the current challenges and limitations, the systems I presented are already able to have a significant positive impact on the security of the software ecosystem, addressing severe bug classes like overflows, use-after-free, double-free, and format string bugs, as well as more advanced semantic level bugs like the presented execution repartitioning attacks. I plan to continue collaborations with academic, industry, and government partners to expand and improve upon the ideas presented here.

Appendices

APPENDIX A

BUNKERBUSTER ALGORITHMS & ADDITIONAL EXPERIMENTS

```
A, T ← ∅
foreach i ∈ I do
  if Type(i) = Store then
    if Type(i.addr) = Const then
      // Write to constant address
      A ← A ∪ i.addr
    end
    else
      // Write to variable address
      T ← T ∪ i.addr
    end
  end
  if Type(i) = WrTmp ∧ Type(i.data) = Load then
    if Type(i.data.addr) = Const then
      // Read from constant address
      A ← A ∪ i.data.addr
    end
    else
      // Read from variable address
      T ← T ∪ i.data.addr
    end
  end
end
end
// Use S to avoid recomputing ASTs
foreach t ∈ T do
  A ← A ∪ EvalTmp(S, t)
end
```

Algorithm 4: Retrieve all memory reads and writes for a VEX IRSB I , using successor state S , producing A .

Figure A.1: Example root cause report for CVE-2018-12326.

```

Trace: openhost+0x2a4 in ntpq (0xbae4)
Trace: openhost+0x218 in ntpq (0xba58)
Trace: openhost+0x3bc in ntpq (0xbbfc)
Trace: __stack_chk_fail+0x0
We've triggered a bug
Analyzing exit at openhost+0x218
Blaming: openhost+0x2dd in ntpq (0xbb1d)
Recommendation: Add [argv[232] == ']'] to
<CFGENode openhost+0x2d8 0x55857a27db18[5]>
Vulnerability Hooks Details:
Hash: 1cfad
Addr: 0x55857b000028 => __stack_chk_fail+0x0
      0x55857a27dc01 => openhost+0x3c1

```

Table A.1: Manually Verified APIs for Binary-Only Recovery

Library	# Functions	# Variables	# Pointers	Match?
libpng	71	183	117	Yes
libz	11	14	2	Yes
glib	125	283	202	Yes
libc	22	29	19	Yes
libbabi	70	163	104	Yes
libx11	5	247	137	Yes
libjpeg-turbo	25	15	12	Yes
libcyrus-sasl	1	3	1	Yes
libpoppler	3	7	4	Yes
libgegl	33	52	44	Yes
libghostpdl	29	47	40	Yes
libgimp	36	50	48	Yes
libgtk	21	41	26	Yes
libkeyutils	4	8	6	Yes
libidn2	1	2	1	Yes
libXpm	5	18	5	Yes
libopenjpeg	22	40	24	Yes
Total:	484	1,202	792	

```

R ← False
I ← ∅
foreach s ∈ S do
  foreach i ∈ s.irsb.statements do
    if IsTmpStore(i) then
      | I[i.addr] ← I[i.addr] ∪ i.addr.tmp
    end
  end
end
foreach a ∈ I do
  | l ← I[a].size
  if l > 1 then
    | if I[a][0] ≤ I[a][1] ≤ ... ≤ I[a][l] then
    | | R ← True
    end
    | if I[a][0] ≥ I[a][1] ≥ ... ≥ I[a][l] then
    | | R ← True
    end
  end
end

```

Algorithm 5: Detect stepping behavior in a sequence of states S , iterating a loop. `IsTmpStore` is true when the VEX IRSB instruction is a `WrTmp` and its expression is `Store`.

Input: VEX IR statements S starting from last executed.

Tmp n to taint initially.

Result: Addresses A and registers R used to calculate n .

$A, R \leftarrow \emptyset$

$T \leftarrow \{n\}$

foreach s in S **do**

if $\text{Type}(s) = \text{Put}$ and $\text{Type}(s.data) = \text{RdTmp}$ **then**

if $s.data.tmp \in T$ **then**

$R \leftarrow R \cup \{s.register\}$

end

end

if $\text{Type}(s) = \text{WrTmp}$ and $s.tmp \in T$ **then**

foreach a in $s.data.args$ **do**

if $\text{Type}(a) = \text{Get}$ **then**

$R \leftarrow R \cup \{a.register\}$

end

if $\text{Type}(a) = \text{RdTmp}$ **then**

$T \leftarrow T \cup \{a.tmp\}$

end

if $\text{Type}(a) = \text{Load}$ **then**

$A \leftarrow A \cup \text{EvalTmp}(a.address)$

end

end

end

end

Algorithm 6: Tainting algorithm to obtain the registers and addresses used to calculate a VEX IR temporary variable.

REFERENCES

- [1] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, “Optimizing seed selection for fuzzing,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 861–875.
- [2] S. K. Cha, M. Woo, and D. Brumley, “Program-adaptive mutational fuzzing,” in *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 725–741.
- [3] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing.,” in *NDSS*, vol. 17, 2017, pp. 1–14.
- [4] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, “Path-exploration lifting: Hi-fi tests for lo-fi emulators,” in *ACM SIGARCH Computer Architecture News*, ACM, vol. 40, 2012, pp. 337–348.
- [5] J. Jang and H. K. Kim, “Fuzzbuilder: Automated building greybox fuzzing environment for c/c++ library,” in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 627–637.
- [6] R. Hodován and Á. Kiss, “Fuzzing javascript engine apis,” in *International Conference on Integrated Formal Methods*, Springer, 2016, pp. 425–438.
- [7] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, “Semfuzz: Semantics-based automatic generation of proof-of-concept exploits,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 2139–2154.
- [8] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, “Kameleonfuzz: Evolutionary fuzzing for black-box xss detection,” in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 2014, pp. 37–48.
- [9] A. Fioraldi, D. C. D’Elia, and L. Querzoni, “Fuzzing binaries for memory safety errors with qasan,” in *2020 IEEE Secure Development (SecDev)*, IEEE, 2020, pp. 23–30.
- [10] P. Godefroid, “Random testing for security: Blackbox vs. whitebox fuzzing,” in *Proceedings of the 2nd International Workshop on Random Testing*, 2007, pp. 1–1.
- [11] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, “Adaptive call-site sensitive control flow integrity,” in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2019, pp. 95–110.

- [12] W. He, S. Das, W. Zhang, and Y. Liu, “Bbb-cfi: Lightweight cfi approach against code-reuse attacks using basic block information,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 1, pp. 1–22, 2020.
- [13] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, “Ccfi: Cryptographically enforced control flow integrity,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 941–951.
- [14] M. Zhang and R. Sekar, “Control flow integrity for cots binaries,” in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.
- [15] B. Niu and G. Tan, “Modular control-flow integrity,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.
- [16] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, “Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace,” in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 173–184.
- [17] B. Niu and G. Tan, “Rockjit: Securing just-in-time compilation using modular control-flow integrity,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1317–1328.
- [18] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.,” in *USENIX Security Symposium*, San Antonio, TX, vol. 98, 1998, pp. 63–78.
- [19] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, *et al.*, “Meltdown: Reading kernel memory from user space,” in *USENIX Security Symposium*, 2018, pp. 973–990.
- [20] M. Zalewski, “American fuzzy lop,” <http://lcamtuf.coredump.cx/afl>, 2017.
- [21] 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 *Proceedings of the 37th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.
- [22] Valgrind Developers, *Valgrind*, <http://www.valgrind.org/>, 2017.
- [23] Y. Kwon, B. Saltaformaggio, I. L. Kim, K. H. Lee, X. Zhang, and D. Xu, “A2c: Self destructing exploit executions via input perturbation,” in *Network and Distributed Systems Security (NDSS) Symposium 2017*, 2017.

- [24] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, “Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, Apr. 2010.
- [25] X. Ge, W. Cui, and T. Jaeger, “Griffin: Guarding control flows using intel processor trace,” in *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi’an, China, Apr. 2017.
- [26] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, “Eidetic systems,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 525–540.
- [27] R. O’Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, “Engineering record and replay for deployability,” in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 377–389.
- [28] D. Tariq, M. Ali, and A. Gehani, “Towards Automated Collection of Application-Level Data Provenance,” in *4th USENIX Workshop on the Theory and Practice of Provenance*, Boston, MA: USENIX, 2012.
- [29] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C,” in *Proceedings of the 30th ACM SIG-PLAN Conference on Programming Language Design and Implementation*, 2009.
- [30] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, “Hacking Blind,” in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.
- [31] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-Oriented Programming: A New Class of Code-reuse Attack,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.
- [32] E. Bosman and H. Bos, “Framing Signals - A Return to Portable Shellcode,” in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.
- [33] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-Oriented Programming Without Returns,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.
- [34] 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*, 2007.

- [35] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, “Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization,” in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.
- [36] PaX Team, *PaX Address Space Layout Randomization (ASLR)*, <http://pax.grsecurity.net/docs/aslr.txt>, 2003.
- [37] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, “ASLR on the Line: Practical Cache Attacks on the MMU,” in *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, 2017.
- [38] K. Lu, S. Nürnberger, M. Backes, and W. Lee, “How to Make ASLR Win the Clone Wars: Runtime Re-Randomization,” in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, 2016.
- [39] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, “ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.
- [40] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow Integrity,” in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.
- [41] M. Zhang and R. Sekar, “Control Flow Integrity for COTS Binaries,” in *Proceedings of the 22nd USENIX Security Symposium*, 2013.
- [42] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical Control Flow Integrity and Randomization for Binary Executables,” in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.
- [43] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-Flow Bending: On the Effectiveness of Control-Flow Integrity,” in *Proceedings of the 24th USENIX Security Symposium*, 2015.
- [44] B. Niu and G. Tan, “Modular Control-flow Integrity,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.
- [45] 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 the 23rd USENIX Security Symposium*, 2014.
- [46] V. van der Veen, E. Goktas, 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 the 37th IEEE Symposium on Security and Privacy*, 2016.

- [47] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. Harris, T. Kim, and W. Lee, “Enforcing unique code target property for control-flow integrity,” in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.
- [48] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, “Confirm: Evaluating compatibility and relevance of control-flow integrity protections for modern software,” in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, 2019, pp. 1805–1821, ISBN: 978-1-939133-06-9.
- [49] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
- [50] J. Newsome, D. Brumley, J. Franklin, and D. Song, “Replayer: Automatic protocol replay by binary analysis,” in *Proceedings of the 13th ACM conference on Computer and communications security*, ACM, 2006, pp. 311–321.
- [51] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “Bitblaze: A new approach to computer security via binary analysis,” in *International Conference on Information Systems Security*, Springer, 2008, pp. 1–25.
- [52] P. Saxena, P. Poosankam, S. McCamant, and D. Song, “Loop-extended symbolic execution on binary programs,” in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ACM, 2009, pp. 225–236.
- [53] D. A. Molnar and D. Wagner, “Catchconv: Symbolic execution and run-time type inference for integer conversion errors,” *UC Berkeley EECS*, 2007.
- [54] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, “Automated whitebox fuzz testing,” in *NDSS*, Citeseer, vol. 8, 2008, pp. 151–166.
- [55] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, “Casym: Cache aware symbolic execution for side channel detection and mitigation,” in *CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation*, IEEE, 2019.
- [56] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmallice-automatic detection of authentication bypass vulnerabilities in binary firmware.” in *NDSS*, 2015.
- [57] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, “Symcerts: Practical symbolic execution for exposing noncompliance in x. 509

- certificate validation implementations,” in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 503–520.
- [58] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, *et al.*, “Mechanical phish: Resilient autonomous hacking,” *IEEE Security & Privacy*, vol. 16, no. 2, pp. 12–22, 2018.
- [59] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, “Dta++: Dynamic taint analysis with targeted control-flow propagation.” in *NDSS*, 2011.
- [60] K. Sen, D. Marinov, and G. Agha, “Cute: A concolic unit testing engine for c,” in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 30, 2005, pp. 263–272.
- [61] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *Proceedings of the 33rd Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.
- [62] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, “Dowsing for overflows: A guided fuzzer to find buffer boundary violations,” in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 49–64.
- [63] 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 *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.
- [64] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “Qsym: A practical concolic execution engine tailored for hybrid fuzzing,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.
- [65] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, L. Lu, *et al.*, “Savior: Towards bug-driven hybrid testing,” *arXiv preprint arXiv:1906.07327*, 2019.
- [66] K. Böttinger and C. Eckert, “Deepfuzz: Triggering vulnerabilities deeply hidden in binaries,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2016, pp. 25–34.
- [67] V.-T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury, “Hercules: Reproducing crashes in real-world application binaries,” in *37th IEEE International Conference on Software Engineering*, IEEE, vol. 1, 2015, pp. 891–901.
- [68] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, “Towards efficient heap overflow discovery,” in *26th USENIX Security Symposium*, 2017, pp. 989–1006.

- [69] 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.
- [70] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, “Automatic exploit generation,” Carnegie Mellon University, 2018.
- [71] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, “Revery: From proof-of-concept to exploitable,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 1914–1927.
- [72] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, “Fudge: Fuzz driver generation at scale,” in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985.
- [73] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, “Fuzzgen: Automatic fuzzer generation,” in *29th USENIX Security Symposium*, 2020, pp. 2271–2287.
- [74] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, “Winnie: Fuzzing windows applications with harness synthesis and fast cloning,” in *Network and Distributed System Security Symposium*, 2021.
- [75] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.
- [76] J.-D. Choi and A. Zeller, “Isolating failure-inducing thread schedules,” in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 27, 2002, pp. 210–220.
- [77] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, “Using likely invariants for automated software fault localization,” in *ACM SIGARCH Computer Architecture News*, ACM, vol. 41, 2013, pp. 139–152.
- [78] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, “Pres: Probabilistic replay with execution sketching on multiprocessors,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, 2009, pp. 177–192.
- [79] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, “Failure sketching: A technique for automated root cause diagnosis of in-production failures,” in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

- [80] V.-T. Pham, S. Khurana, S. Roy, and A. Roychoudhury, “Bucketing failing tests via symbolic analysis,” in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2017, pp. 43–59.
- [81] R. van Tonder, J. Kotheimer, and C. Le Goues, “Semantic crash bucketing,” in *33rd IEEE International Conference on Automated Software Engineering*, IEEE, 2018, pp. 612–622.
- [82] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, “Aurora: Statistical crash analysis for automated root cause explanation,” in *29th USENIX Security Symposium*, 2020, pp. 235–252.
- [83] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, “Postmortem program analysis with hardware-enhanced post-crash artifacts,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 17–32.
- [84] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie, A. Gehani, and V. Yegneswaran, “Mci: Modeling-based causality inference in audit logging for attack investigation,” in *Proc. of the 25th Network and Distributed System Security Symposium (NDSS’18)*, 2018.
- [85] K. H. Lee, X. Zhang, and D. Xu, “High Accuracy Attack Provenance via Binary-based Execution Partition,” in *Proceedings of NDSS ’13*, San Diego, CA, Feb. 2013.
- [86] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, “Accurate, low cost and instrumentation-free security audit logging for windows,” in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015, Los Angeles, CA, USA: ACM, 2015, pp. 401–410, ISBN: 978-1-4503-3682-6.
- [87] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, “Kernel-supported cost-effective audit logging for causality tracking,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA: USENIX Association, 2018, pp. 241–254, ISBN: 978-1-931971-44-7.
- [88] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, “MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning,” in *26th USENIX Security Symposium*, Aug. 2017.
- [89] A. Bates, W. U. Hassan, K. R. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear, “Transparent Web Service Auditing via Network Provenance Functions,” in *26th World Wide Web Conference*, ser. WWW’17, Perth, Australia, Apr. 2017.

- [90] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, “Towards a Timely Causality Analysis for Enterprise Security,” in *Proceedings of the 25th ISOC Network and Distributed System Security Symposium*, ser. NDSS’18, San Diego, CA, USA, Feb. 2018.
- [91] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. 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*, ser. CCS ’17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 377–390, ISBN: 9781450349468.
- [92] W. U. Hassan, A. Bates, and D. Marino, “Tactical Provenance Analysis for Endpoint Detection and Response Systems,” in *41st IEEE Symposium on Security and Privacy (SP)*, ser. Oakland’20, May 2020.
- [93] S. Ma, X. Zhang, and D. Xu, “ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting,” in *Proceedings of NDSS ’16*, San Diego, CA, Feb. 2016.
- [94] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrisnan, “Holmes: Real-time apt detection through correlation of suspicious information flows,” in *2019 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA: IEEE Computer Society, 2019.
- [95] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Zhen, W. Cheng, C. A. Gunter, and H. chen, “You are what you do: Hunting stealthy malware via data provenance analysis,” in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS’20, 2020.
- [96] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, “Differential provenance: Better network diagnostics with reference events,” in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets’15)*, Philadelphia, PA, Nov. 2015.
- [97] A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, “One primitive to diagnose them all: Architectural support for internet diagnostics,” in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys ’17, Belgrade, Serbia: ACM, 2017, pp. 374–388, ISBN: 978-1-4503-4938-3.
- [98] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, “The good, the bad, and the differences: Better network diagnostics with differential provenance,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16, Florianopolis, Brazil: ACM, 2016, pp. 115–128, ISBN: 978-1-4503-4193-6.

- [99] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou, “Let SDN Be Your Eyes: Secure Forensics in Data Center Networks,” in *NDSS Workshop on Security of Emerging Networking Technologies*, ser. SENT’14, Feb. 2014.
- [100] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, “Secure Network Provenance,” in *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.
- [101] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, “Nodoze: Combatting threat alert fatigue with automated provenance triage,” in *26th ISOC Network and Distributed System Security Symposium*, ser. NDSS’19, 2019.
- [102] A. Bates, K. R. B. Butler, and T. Moyer, “Take Only What You Need: Leveraging Mandatory Access Control Policy to Reduce Provenance Storage Costs,” in *7th Workshop on the Theory and Practice of Provenance*, ser. TaPP’15, Edinburgh, Scotland, Jul. 2015.
- [103] J. Park, D. Nguyen, and R. Sandhu, “A Provenance-Based Access Control Model,” in *Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (PST)*, 2012, pp. 137–144.
- [104] F. Capobianco, C. Skalka, and T. Jaeger, “Accessprov: Tracking the provenance of access control decisions,” in *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*, 2017.
- [105] S. T. King and P. M. Chen, “Backtracking intrusions,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’03, Bolton Landing, NY, USA: ACM, 2003, pp. 223–236, ISBN: 1-58113-757-5.
- [106] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, “Hi-Fi: Collecting High-Fidelity Whole-System Provenance,” in *Proceedings of the 2012 Annual Computer Security Applications Conference*, ser. ACSAC ’12, Orlando, FL, USA, 2012.
- [107] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, “Intrusion recovery using selective re-execution,” in *OSDI*, USENIX Association, 2010.
- [108] A. Bates, D. Tian, K. R. Butler, and T. Moyer, “Trustworthy Whole-System Provenance for the Linux Kernel,” in *Proceedings of 24th USENIX Security Symposium*, Washington, D.C., Aug. 2015.
- [109] A. Gehani and D. Tariq, “SPADE: Support for Provenance Auditing in Distributed Environments,” in *Proceedings of the 13th International Middleware Conference*, ser. Middleware ’12, Montreal, Quebec, Canada, Dec. 2012.

- [110] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, “Fear and Logging in the Internet of Things,” in *Proceedings of the 25th ISOC Network and Distributed System Security Symposium*, ser. NDSS’18, Feb. 2017.
- [111] M. A. Borkin, A. A. Vo, Z. Bylinskii, P. Isola, S. Sunkavalli, A. Oliva, and H. Pfister, “What makes a visualization memorable?” *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2306–2315, 2013.
- [112] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, and H. Pfister, “Evaluation of filesystem provenance visualization tools,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2476–2485, 2013.
- [113] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian, “Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution,” in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS’20, Feb. 2020.
- [114] V. Karande, E. Bauman, Z. Lin, and L. Khan, “SGX-Log: Securing System Logs With SGX,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’17, 2017.
- [115] S. A. Crosby and D. S. Wallach, “Efficient data structures for tamper-evident logging,” in *In Proceedings of the 18th USENIX Security Symposium*, 2009.
- [116] T. Pulls and R. Peeters, “Balloon: A forward-secure append-only persistent authenticated data structure,” in *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, 2015.
- [117] Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, and D. D. Long, “A Hybrid Approach for Efficient Provenance Storage,” in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, ser. CIKM ’12, Maui, Hawaii, USA, 2012.
- [118] A. Chapman, H. Jagadish, and P. Ramanan, “Efficient Provenance Storage,” in *Proceedings of the 2008 ACM Special Interest Group on Management of Data Conference*, ser. SIGMOD’08, Vancouver, Canada, Jun. 2008.
- [119] C. Chen, H. T. Lehri, L. Kuan Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou, “Distributed provenance compression,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD ’17, Chicago, Illinois, USA: ACM, 2017, pp. 203–218, ISBN: 978-1-4503-4197-4.
- [120] A. Gehani, M. Kim, and J. Zhang, “Steps Toward Managing Lineage Metadata in Grid Clusters,” in *1st Workshop on the Theory and Practice of Provenance*, ser. TaPP’09, San Francisco, CA, Feb. 2009.

- [121] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, Y. Li, and D. D. E. Long, "Evaluation of a Hybrid Approach for Efficient Provenance Storage," *Trans. Storage*, vol. 9, no. 4, 14:1–14:29, Nov. 2013.
- [122] A. Bates, D. Tian, G. Hernandez, T. Moyer, K. R. Butler, and T. Jaeger, "Taming the Costs of Trustworthy Provenance through Policy Reduction," *ACM Trans. on Internet Technology*, vol. 17, no. 4, 34:1–34:21, Sep. 2017.
- [123] Y. Tang, D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li, "Nodemerge: Template based efficient data reduction for big-data causality analysis," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: ACM, 2018, pp. 1324–1337, ISBN: 978-1-4503-5693-0.
- [124] W. U. Hassan, N. Aguse, M. Lemay, T. Moyer, and A. Bates, "Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs," in *Proceedings of the 25th ISOC Network and Distributed System Security Symposium*, ser. NDSS'18, San Diego, CA, USA, Feb. 2018.
- [125] Y. Ben, Y. Han, N. Cai, W. An, and Z. Xu, "T-tracker: Compressing system audit log by taint tracking," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2018, pp. 1–9.
- [126] K. H. Lee, X. Zhang, and D. Xu, "LogGC: Garbage Collecting Audit Log," in *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, ser. CCS '13, Berlin, Germany: ACM, 2013, pp. 1005–1016, ISBN: 978-1-4503-2477-9.
- [127] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 504–516, ISBN: 978-1-4503-4139-4.
- [128] L. Feng, J. Huang, J. Hu, and A. Reddy, "Fastcfi: Real-time control flow integrity using fpga without code instrumentation," in *International Conference on Runtime Verification*, Springer, 2019, pp. 221–238.
- [129] S. Forrest, S. Hofmeyr, and A. Somayaji, "The evolution of system-call monitoring," in *2008 Annual Computer Security Applications Conference (ACSAC)*, IEEE, 2008, pp. 418–430.
- [130] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," 2010.

- [131] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.
- [132] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, “Language-independent sandboxing of just-in-time compilation and self-modifying code,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 355–366.
- [133] J. A. Kroll, G. Stewart, and A. W. Appel, “Portable software fault isolation,” in *2014 IEEE 27th Computer Security Foundations Symposium*, IEEE, 2014, pp. 18–32.
- [134] B. Patel, *Intel Releases New Technology Specifications to Protect Against ROP attacks*, <https://software.intel.com/content/www/us/en/develop/blogs/intel-release-new-technology-specifications-protect-rop-attacks.html>, [Online; accessed 26-June-2020].
- [135] *Control Flow Guard*, <https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard>, [Online; accessed 26-June-2020].
- [136] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, “Vigilante: End-to-end containment of internet worms,” in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 133–147.
- [137] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, “Towards automatic generation of vulnerability-based signatures,” in *2006 IEEE Symposium on Security and Privacy (S&P’06)*, IEEE, 2006.
- [138] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah, “Vulnerability-specific execution filtering for exploit prevention on commodity software,” in *NDSS*, 2006.
- [139] A. Slowinska and H. Bos, “The age of data: Pinpointing guilty bytes in polymorphic buffer overflows on heap or stack,” in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, IEEE, 2007, pp. 487–500.
- [140] K. Bhat, E. Van Der Kouwe, H. Bos, and C. Giuffrida, “Probeguard: Mitigating probing attacks through reactive program transformations,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 545–558.
- [141] R. Ding, H. Hu, W. Xu, and T. Kim, “Desensitization: Privacy-aware and attack-preserving crash report,” in *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

- [142] F. Capobianco, R. George, K. Huang, T. Jaeger, S. Krishnamurthy, Z. Qian, M. Payer, and P. Yu, “Employing Attack Graphs for Intrusion Detection,” in *New Security Paradigms Workshop*, ser. NSPW’19, 2019.
- [143] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A sense of self for unix processes,” in *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996, pp. 120–128.
- [144] X. Han, T. Pasqueir, A. Bates, J. Mickens, and M. Seltzer, “Unicorn: Runtime provenance-based detector for advanced persistent threats,” in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS’20, 2020.
- [145] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, “Enriching intrusion alerts through multi-host causality,” in *Proceedings of the 12th ISOC Network and Distributed System Security Symposium*, ser. NDSS’05, 2005.
- [146] X. Shu, D. (Yao, N. Ramakrishnan, and T. Jaeger, “Long-span program behavior modeling and attack detection,” *ACM Transactions on Privacy and Security*, vol. 20, 2017.
- [147] A. Wespi, M. Dacier, and H. Debar, “Intrusion detection using variable-length audit trail patterns,” in *Recent Advances in Intrusion Detection*, Springer, 2000, pp. 110–129.
- [148] C. Warrender, S. Forrest, and B. Pearlmutter, “Detecting intrusions using system calls: Alternative data models,” in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*, 1999, pp. 133–145.
- [149] K. Xu, K. Tian, D. Yao, and B. G. Ryder, “A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity,” in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 467–478.
- [150] M. Bellare and B. Yee, “Forward integrity for secure audit logs,” Computer Science and Engineering Department, University of California at San Diego, Tech. Rep., 1997.
- [151] J. E. Holt, “Logcrypt: Forward security and public verification for secure audit logs,” in *Proceedings of the Australasian Information Security Workshop (AISW-NetSec)*, 2006.
- [152] R. Paccagnella, K. Liao, D. (Tian, and A. Bates, “Logging to the danger zone: Race condition attacks and defenses on system audit frameworks,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS’20, 2020.

- [153] B. Schneier and J. Kelsey, “Cryptographic support for secure logs on untrusted machines.,” in *Proceedings of the USENIX Security Symposium (USENIX)*, 1998.
- [154] D. Ma and G. Tsudik, “A new approach to secure logging,” *ACM Transactions on Storage (TOS)*, vol. 5, no. 1, 2009.
- [155] A. A. Yavuz and P. Ning, “Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.
- [156] A. A. Yavuz, P. Ning, and M. K. Reiter, “Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging,” in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2012.
- [157] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, “Finding and reproducing heisenbugs in concurrent programs.,” in *OSDI*, vol. 8, 2008, pp. 267–280.
- [158] I. Ahmed, N. Mohan, and C. Jensen, “The impact of automatic crash reports on bug triaging and development in mozilla,” in *Proceedings of The International Symposium on Open Collaboration*, 2014, pp. 1–8.
- [159] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg, “Security impact ratings considered harmful,” *arXiv preprint arXiv:0904.4058*, 2009.
- [160] P. J. Guo and D. R. Engler, “Linux kernel developer responses to static analysis bug reports.,” in *USENIX Annual Technical Conference*, 2009, pp. 285–292.
- [161] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song, “Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions,” in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, Jun. 2016.
- [162] J. Chow, T. Garfinkel, and P. M. Chen, “Decoupling dynamic program analysis from execution in virtual environments,” in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008, pp. 1–14.
- [163] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, “Rain: Refinable attack investigation with on-demand inter-process information flow tracking,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.
- [164] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, “Nodoze: Combatting threat alert fatigue with automated provenance triage.,” in *NDSS*, 2019.

- [165] M. E. Aminanto, L. Zhu, T. Ban, R. Isawa, T. Takahashi, and D. Inoue, “Automated threat-alert screening for battling alert fatigue with temporal isolation forest,” in *2019 17th International Conference on Privacy, Security and Trust (PST)*, IEEE, 2019, pp. 1–3.
- [166] S. McElwee, J. Heaton, J. Fraley, and J. Cannady, “Deep learning for prioritizing and responding to intrusion detection alerts,” in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, IEEE, 2017, pp. 1–5.
- [167] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, “Efficient protection of path-sensitive control security,” in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.
- [168] C. Yagemann, S. Sultana, L. Chen, and W. Lee, “Barnum: Detecting document malware via control flow anomalies in hardware traces,” in *Proceedings of the 25th Information Security Conference (ISC)*, New York, NY, USA, 2019.
- [169] J. Lee, T. Avgerinos, and D. Brumley, “Tie: Principled reverse engineering of types in binary programs,” 2011.
- [170] Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.
- [171] T. Wang, C. Song, and W. Lee, “Diagnosis and emergency patch generation for integer overflow exploits,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2014, pp. 255–275.
- [172] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, “Ripe: Runtime intrusion prevention evaluator,” in *In Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, ACM, 2011.
- [173] P. E. Black and P. E. Black, *Juliet 1.3 Test Suite: Changes From 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018.
- [174] *CVE-2004-0597 Patch*, <https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2004-0597#patch>, [Online; accessed 25-October-2019].
- [175] *CVE-2004-1288 Patch*, <https://pastebin.com/raw/fsFkspFF>, [Online; accessed 25-October-2019].
- [176] *Red Hat Bugzilla – Attachment 360889 Details for Bug 523105*, <https://bugzilla.redhat.com/attachment.cgi?id=360889&action=diff>, [Online; accessed 07-January-2020].

- [177] *Debian Bug report logs - #552035*, <https://bugs.debian.org/cgi-bin/bugreport.cgi?att=1;bug=552035;filename=diff;msg=16>, [Online; accessed 10-January-2020].
- [178] *Commit 9fdcc15962f9ff4baebe6fdd947816f43f730d50*, <https://github.com/antirez/redis/commit/9fdcc15962f9ff4baebe6fdd947816f43f730d50>, [Online; accessed 16-January-2020].
- [179] gif2png, *Command Line Buffer Overflow*, <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=550978#50>, [Online; accessed 25-October-2019], 2009.
- [180] ntp, *Stack-based buffer overflow in ntpq and ntpdc allows denial of service or code execution*, https://bugzilla.redhat.com/show_bug.cgi?id=1593580, [Online; accessed 25-October-2019], 2018.
- [181] *Commit 074f7a8cd19b7661a59047e9257691df5470551c*, <https://github.com/mz-automation/libiec61850/commit/074f7a8cd19b7661a59047e9257691df5470551c>, [Online; accessed 09-April-2020].
- [182] pdfresurrect, *Prevent a buffer overflow in possibly corrupt PDFs*, <https://github.com/enferex/pdfresurrect/commit/4ea7a6f4f51d0440da651d099247e2273f811dbc>, [Online; accessed 25-October-2019], 2019.
- [183] libtiff, *Multiple libtiff Issues*, <https://bugzilla.redhat.com/attachment.cgi?id=128255&action=diff>, [Online; accessed 25-October-2019], 2006.
- [184] EXIF Tag Parsing Library, *#70 serious security bug in exif_data_load_data_entry()*, <https://sourceforge.net/p/libexif/bugs/70/>, [Online; accessed 25-October-2019], 2007.
- [185] *patch.2013.chunked.txt*, <https://nginx.org/download/patch.2013.chunked.txt>, [Online; accessed 16-January-2020].
- [186] *patch.2017.ranges.txt*, <https://nginx.org/download/patch.2017.ranges.txt>, [Online; accessed 16-January-2020].
- [187] GraphicsMagick, *Attempt to Fix Issue 440*, <http://hg.code.sf.net/p/graphicsmagick/code/rev/98721124e51f>, [Online; accessed 25-October-2019], 2017.
- [188] libzip, *Fix double free*, <https://tinyurl.com/jce6afsh>, [Online; accessed 25-October-2019], 2017.
- [189] *CVE-2005-0105 Patch*, <https://pastebin.com/raw/GHm1k1Rk>, [Online; accessed 25-October-2019].

- [190] sudo, *Format String Vulnerability*, <https://bugs.gentoo.org/401533>, [Online; accessed 25-October-2019], 2012.
- [191] *LinuxFlaw*, <https://github.com/VulnReproduction/LinuxFlaw>, [Online; accessed 06-January-2020].
- [192] *Exploit Database*, <https://www.exploit-db.com/>, [Online; accessed 06-January-2020].
- [193] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, and W. Wang, “Detecting vulnerabilities in c programs using trace-based testing,” in *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010.
- [194] V. van der Veen, D. Andriessse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffridia, “The dynamics of innocent flesh on the bone: Code reuse ten years later,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1675–1689.
- [195] S. Y. Chau, M. Yahyazadeh, O. Chowdhury, A. Kate, and N. Li, “Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification.,” in *NDSS*, 2019.
- [196] X. Chen, A. Slowinska, and H. Bos, “Who allocated my memory? detecting custom memory allocators in c binaries,” in *2013 20th Working Conference on Reverse Engineering (WCRE)*, IEEE, 2013, pp. 22–31.
- [197] *WindowsIntelPT*, <https://github.com/intelpt/WindowsIntelPT>, [Online; accessed 12-June-2020].
- [198] *winipt*, <https://github.com/ionescu007/winipt>, [Online; accessed 12-June-2020].
- [199] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrisnan, “SLEUTH: Real-time attack scenario reconstruction from COTS audit data,” in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, 2017, pp. 487–504, ISBN: 978-1-931971-40-9.
- [200] W. U. Hassan, M. Nouredine, P. Datta, and A. Bates, “OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis,” in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS’20, Feb. 2020.
- [201] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, “Ldx: Causality inference by lightweight dual execution,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Program-*

ming Languages and Operating Systems, ser. ASPLOS '16, Atlanta, Georgia, USA: ACM, 2016, pp. 503–515, ISBN: 978-1-4503-4091-5.

- [202] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *IEEE Symposium on Security and Privacy*, IEEE, 2016, pp. 969–986.
- [203] OccupytheWeb, *How to cover your tracks & leave no trace behind on the target system*, <https://tinyurl.com/ygqte9p>, Last accessed 04-20-2019, 2013.
- [204] The MITRE Corporation, *Capec-81: Web logs tampering*, <https://capec.mitre.org/data/definitions/81.html>, Last accessed 04-20-2019, 2017.
- [205] JustLinux Forums, *Server hacked!! /var/log deleted. how can i trace hacker!?! http://forums.justlinux.com/showthread.php?123851-server-hacked-var-log-deleted-how-can-i-trace-hacker*, Last accessed 04-20-2019.
- [206] Rapid7, *Metasploit, the world's most used penetration testing framework*, <https://www.metasploit.com/>, Last accessed 04-20-2019.
- [207] S. Hales, *Last door log wiper*, <https://packetstormsecurity.com/files/118922/LastDoor.tar>, Last accessed 04-20-2019.
- [208] Carbon Black, *Global incident response threat report*, <https://www.carbonblack.com/global-incident-response-threat-report/november-2018/>, Last accessed 04-20-2019, Nov. 2018.
- [209] C. Cimpanu, *Hackers are increasingly destroying logs to hide attacks*, <https://www.zdnet.com/article/hackers-are-increasingly-destroying-logs-to-hide-attacks/>, Last accessed 04-20-2019.
- [210] B. Schneier and J. Kelsey, “Secure audit logs to support computer forensics,” *ACM Transactions on Information and System Security (TISSEC)*, 1999.
- [211] G. Hartung, B. Kaidel, A. Koch, J. Koch, and D. Hartmann, “Practical and robust secure logging from fault-tolerant sequential aggregate signatures,” in *Proc. of the International Conference on Provable Security (ProvSec)*, 2017.
- [212] K. Karen and S. Murugiah, *NIST special publication 800-92, guide to computer security log management*, 2006.
- [213] IBM Knowledge Center, *Storage and analysis of audit logs*, https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.sec.doc/doc/c0052328.html, Last accessed 04-20-2019.

- [214] National Institute of Standards and Technology, *NIST special publication 800-53 (rev. 4), security controls and assessment procedures for federal information systems and organizations*, 2013.
- [215] A. S. Foundation, *Apache HTTP server benchmarking tool*, <https://httpd.apache.org/docs/2.4/programs/ab.html>.
- [216] D. Habusha, *Vulnerability prioritization tops security pros' challenges*, <https://tinyurl.com/y6os685b>, [Online; accessed 18-November-2020].
- [217] P. Low, "Insuring against cyber-attacks," *Computer Fraud & Security*, vol. 2017, no. 4, pp. 18–20, 2017.
- [218] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2009, pp. 248–255.
- [219] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 748–758.
- [220] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1497–1511.
- [221] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*, Springer, 2015, pp. 330–347.
- [222] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC)*, 2012, pp. 309–318.
- [223] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 189–200.
- [224] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, "Towards efficient heap overflow discovery," in *26th USENIX Security Symposium*, 2017, pp. 989–1006.
- [225] C. Yagemann, M. Pruet, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "Arcus: Symbolic root cause analysis of exploits in production systems," in *USENIX Security Symposium*, 2021.

- [226] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, “Razor: A framework for post-deployment software debloating,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1733–1750.
- [227] A. Quach, A. Prakash, and L. Yan, “Debloating software through piece-wise compilation and loading,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 869–886.
- [228] H. Koo, S. Ghavamnia, and M. Polychronakis, “Configuration-driven software debloating,” in *Proceedings of the 12th European Workshop on Systems Security*, 2019, pp. 1–6.
- [229] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, “Trimmer: Application specialization for code debloating,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 329–339.
- [230] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, “Effective program debloating via reinforcement learning,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 380–394.
- [231] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. Kemerlis, “Retracer: Triaging crashes by reverse execution from partial memory dumps,” in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, Texas, May 2016.
- [232] B. Kasikci, W. Cui, X. Ge, and B. Niu, “Lazy diagnosis of in-production concurrency bugs,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ACM, 2017, pp. 582–598.
- [233] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, “Rept: Reverse debugging of failures in deployed software,” in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, Oct. 2018.
- [234] J. Feist, L. Mounier, and M.-L. Potet, “Statically detecting use after free on binary code,” *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.
- [235] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, “Typestate-guided fuzzer for discovering use-after-free vulnerabilities,” in *42nd International Conference on Software Engineering*, ACM, 2020.
- [236] J. Caballero, G. Grieco, M. Marron, and A. Nappa, “Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities,” in *Proceed-*

ings of the 2012 International Symposium on Software Testing and Analysis, 2012, pp. 133–143.

- [237] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, “Tupni: Automatic reverse engineering of input formats,” in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 391–402.
- [238] S. Heelan, “Automatic generation of control flow hijacking exploits for software vulnerabilities,” Ph.D. dissertation, University of Oxford, 2009.
- [239] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 969–986.
- [240] S. Y. Chau, M. Yahyazadeh, O. Chowdhury, A. Kate, and N. Li, “Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification,” in *NDSS*, 2019.
- [241] X. Ge, B. Niu, and W. Cui, “Reverse debugging of kernel failures in deployed systems,” in *USENIX Annual Technical Conference*, 2020, pp. 281–292.

VITA

Carter Yagemann was born in Pasadena, California in 1993 and works for the Institute of Information Security and Privacy at the Georgia Institute of Technology under the supervision of Prof. Wenke Lee. His work has been published at top academic conferences, including at the ACM Conference on Computer and Communications Security (CCS) and the USENIX Security Symposium. Prior to joining Georgia Tech, Carter worked for the Department of Electrical Engineering and Computer Science at Syracuse University under the supervision of Prof. Wenliang (Kevin) Du, where he received his B.S. and M.S. degrees in Computer Science. He also worked at JPMorgan Chase & Co. on cyber-threat intelligence. His research interests include systems security, automated program analysis, machine learning, and threats to open markets. After graduation from Georgia Tech, Carter plans to join the Ohio State University as an Assistant Professor.