AUTOMATED BINARY SOFTWARE ATTACK SURFACE REDUCTION

by

Masoud Ghaffarinia

APPROVED BY SUPERVISORY COMMITTEE:

Kevin W. Hamlen, Chair

Bhavani Thuraisingham

Latifur Khan

Farokh Bastani

To my wife, Anahita.

AUTOMATED BINARY SOFTWARE ATTACK SURFACE REDUCTION

by

MASOUD GHAFFARINIA, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2020

ACKNOWLEDGMENTS

I would like to express my profound appreciation to my advisor, Kevin Hamlen, for his unconditional support, his dedicated and passionate teaching style, and his patient guidance, without any of which this dissertation would have not been possible. I am blessed that I was supervised by him who has an unbounded technical breadth and yet is infectiously humble.

I also like to thank my dissertation committee members, Latifur Khan, Bhavani Thuraisingham, and Farokh Bastani, for their support and contributions and their help for this dissertation to be accomplished. I also would like to thank Rhonda Walls for her invaluable support and help with proposal, defense and conference travel arrangements and for always being available when I needed her help. I deeply thank Ravi Prakash for writing valuable recommendations and for his moral support during my PhD.

I want to extend my sincere thanks to my research partners, Xiaoyang Xu and Wenhao Wang and my other labmates, Benjamin Ferrell, Jun Duan, and Erick Bauman for the memories and discussions we shared. Heartfelt thanks to my true friends Farhad Shakerin and Somayeh Mohammadpour for always being there for us. I also want to thank Afshin Taghavi, Aliehsan Samiei, and all who made this a memorable journey.

Hearty thanks to my mother, brothers and my sister for their unconditional love and support. I especially want to thank my brother, Mahmood, whose support in life was more than I needed and for never giving up on me. My grateful thanks to Aida Mahzari and Saina Namazifard for their inspiration.

Above all, I wish to express my love and appreciation for my wife, Anahita, for her continued and unfailing love, unwavering support and for standing by me through all my travails. Understanding me as a PhD herself, Anahita has been my best friend and helped me to keep things in perspective.

This dissertation was supported in part by the Office of Naval Research (ONR) under awards N00014-14-1-0030 and N00014-17-1-2995, the National Science Foundation (NSF) under CAREER award #1054629 and award #1513704, DARPA under award FA8750-19-C-0006, NSF Industry-University Collaborative Research Center (I/UCRC) awards from Raytheon and Lockheed-Martin Corporations and an endowment from the Eugene McDermott family. All opinions, recommendations, and conclusions expressed are those of the author and not necessarily of the ONR, NSF, DARPA, Raytheon, Lockheed-Martin or Eugene McDermott family. I'd like to thank all of them.

March 2020

AUTOMATED BINARY SOFTWARE ATTACK SURFACE REDUCTION

Masoud Ghaffarinia, PhD
The University of Texas at Dallas, 2020

Supervising Professor: Kevin W. Hamlen, Chair

Automated removal of potentially exploitable, abusable, or unwanted code features from binary software is potentially valuable in scenarios where security-sensitive organizations wish to employ general-purpose, closed-source, commercial software in specialized computing contexts where some of the software's functionalities are unneeded.

This dissertation proposes *binary control-flow trimming*, a new method of automatically reducing the attack surfaces of binary software, affording code consumers the power to remove features that are unwanted or unused in a particular deployment context. The approach targets stripped binary native code with no source-derived metadata or symbols, can remove semantic features irrespective of whether they were intended and/or known to code developers, and anticipates consumers who can demonstrate desired features (e.g., via unit testing), but who may not know the existence of specific unwanted features, and who lack any formal specifications of the code's semantics.

Through a combination of runtime tracing, machine learning, in-lined reference monitoring, and contextual control-flow integrity enforcement, it is demonstrated that automated code feature removal is nevertheless feasible under these constraints, even for complex programs such as compilers and servers. The approach additionally accommodates consumers whose demonstration of desired features is incomplete; a tunable entropy-based metric detects coverage lapses and conservatively preserves unexercised but probably desired flows.

A prototype implementation for Intel x86-64 exhibits low runtime overhead for trimmed binaries (about 1.87%), and case studies show that consumer-side control-flow trimming can successfully eliminate zero-day vulnerabilities.

Binary control-flow trimming relies foundationally upon *control-flow integrity* (CFI) enforcement, which has become a mainstay of protecting certain classes of software from code-reuse attacks. Using CFI to enforce the highly complex, context-sensitive security policies needed for feature removal requires a detailed analysis of CFI's compatibility with large, binary software products. However, prior analyses of CFI in the literature have primarily focused on assessing CFI's security weaknesses and performance characteristics, not its ability to preserve intended program functionalities (*semantic transparency*) of large classes of diverse, mainstream software products. This is in part because although there exist many performance and security benchmarking suites, there remains no standard regimen for assessing compatibility. Researchers must often therefore resort to anecdotal assessments, consisting of tests on homogeneous software collections with limited variety (e.g., GNU Coreutils), or on CPU benchmarks (e.g., SPEC) whose limited code features are not representative of large, mainstream software products.

To fill this void, this dissertation presents ConFIRM (CONtrol-Flow Integrity Relevance Metrics), a new evaluation methodology and microbenchmarking suite for assessing compatibility, applicability, and relevance of CFI protections for preserving the intended semantics of software while protecting it from abuse. Reevaluation of CFI solutions using ConFIRM reveals that there remain significant unsolved challenges in securing many large classes of software products with CFI, including software for market-dominant OSes (e.g., Windows) and code employing certain ubiquitous coding idioms (e.g., event-driven callbacks and exceptions). An estimated 47% of CFI-relevant code features with high compatibility impact remain incompletely supported by existing CFI algorithms, or receive weakened controls that leave prevalent threats unaddressed (e.g., return-oriented programming attacks). Discussion of these open problems highlights issues that future research must address to bridge these important gaps between CFI theory and practice.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Humanity's relationship with computers has progressed to the point where nearly every aspect of human life is now somehow exposed to computer systems. Personal medical information is now routinely digitized and analyzed by AI software. We are living in the age of mobile wallets that facilitate basic financial transactions through the aid of smartphones. Computers are manifested in our children digital toys, our cars, as well as military missiles and drones.

As computers increasingly permeate our lives, cyber attacks inevitably have more potential impact for harm. We are witnessing a dramatic growth in cybersecurity breaches, and the growth is accelerating. In one of the biggest data breaches ever, a hacker gained access to more than 100 million Capital One customers' accounts and credit card applications (Krebs on Security, 2019). WannaCry ransomware encrypted hundred of thousands computers over 150 countries in a matter of hours (Whittaker, 2019). Israeli cyber intelligence company NSO created a spyware, called Pegasus, that was used to infect senior government officials as well as journalists and human rights defenders. It secretly jailbreaks iPhones and exfiltrates user communications and location information (Brewster, 2016). These are just few examples that highlight the importance of cybersecurity for everyone in the world today.

Although many efforts have been devoted to the study and practice of cybersecurity, more attention both in academia and industry is nevertheless needed to meet the rising threats. Automated tools are built that help developers to catch exploitable bugs before product release. Software testing is extensively being exercised to achieve the same goal, identifying security lapses contrary to software's requirements. Software companies consistently provide security patches and updates to fix vulnerabilities found after release.

For many of these efforts, security-sensitive organizations, such as military agencies, tend to use commercial off-the-shelf (COTS) code, which usually comes with more stability, reliability, and support than in-house developed alternatives. However, the security of software is widely believed to be inversely related to its complexity (cf., Walden et al., 2014; Zimmermann et al., 2010). With more software features, larger implementations, and more behavioral variety inevitably come more opportunities for programmer error, malicious code introduction, and unforeseen interactions between components.

Unfortunately, this danger stands at odds with one of the major economic forces in the software market—the need to mass-produce ever more general-purpose software in order to tractably meet the broadening needs of the expanding base of software consumers worldwide. Software developers understandably seek to create products that appeal to the widest possible clientele, in order to maximize sales and reduce overheads. This has led to commercial software products of increasing complexity, as developers pack more features into each product they release. As a result, software is becoming more and more difficult to reliably secure as software becomes more multi-purpose and more complex.

Code-reuse attacks (Solar Designer, 1997; Bletsch et al., 2011; Carlini and Wagner, 2014; Schwartz et al., 2011; Shacham, 2007) are one example of the inherent security risks that such feature-accumulation can introduce. In a code-reuse attack, a malicious software user amplifies an otherwise low-severity software bug, such as a memory corruption bug, to hijack the victim software and control it to perform arbitrary actions. For example, *return-oriented programming* (ROP) attacks (Roemer et al., 2012) abuse such data corruptions to overwrite the stack with a series of attacker-supplied code addresses. This causes the victim program to execute attacker-specified *gadgets* (i.e., code fragments at the attacker-supplied addresses) when it consults the stack to return to the current method's caller. The potential potency of a ROP attack therefore depends in part on the variety of gadgets that reside in the victim program's executable memory as it runs (Homescu et al., 2012; Göktaş et al., 2014). The larger the software, the more code gadgets are potentially available to be co-opted by the attacker, and the more malicious behaviors can potentially be triggered.

Chapter 2 and 3 of this dissertation propose *binary control-flow trimming*, a new approach to the feature-bloat problem that combines machine learning with in-lined reference monitors (IRMs) (Schneider, 2000) to remove consumer-undesired features from binary software in order to reduce the attack surface of programs. Unlike prior work that reduces software attack surfaces during the software development process or with developer assistance, binary control-flow trimming focuses on empowering code recipients to remove potentially dangerous software features that developers are not motivated to remove (e.g., because doing so might impact sales) but that pose unnecessary risks for individual consumers because they are not needed in certain deployments. For example, software that inadvertently contains an exploitable vulnerability in the portion of the code that supports AMD processors can be secured for a consumer that uses only Intel processors by removing the unneeded AMD support code. This focus contrasts with prior work on *control-flow integrity (CFI)* (Abadi et al., 2005, 2009; Zhang and Sekar, 2013; Tice et al., 2014; Niu and Tan, 2014a, 2015; van der Veen et al., 2015; Mashtizadeh et al., 2015; Akritidis et al., 2008) and *software fault isolation (SFI)* (Wahbe et al., 1993; Yee et al., 2009; McCamant and Morrisett, 2006), which constrain software control-flow graph to a subgraph of edges that were originally intended by the *software's creators* to be reachable.

Binary control-flow trimming builds upon and generalizes CFI and SFI to enforce context-sensitive policies specified indirectly by end-users through quality assurance testing. Once a suitable trimming policy is learned, the defense instruments programs with extra security guard code that validates the destinations of jump instructions at runtime. Jumps that attempt to traverse a graph edge not permitted by the security policy are blocked, thereby detecting and averting the attack. This is more reliable than probabilistic approaches that randomize the code to make unwanted control-flow edges unlikely to be reachable (Zhang et al., 2013; Mohan et al., 2015; Larsen et al., 2014; Homescu et al., 2013).

A critical challenge for this new approach is discovering an adequate policy to enforce. Typically such policies are expressed as a graph that whitelists the control-flow graph edges that the policy declares to be safe. When program source code is available, such a whitelist can sometimes be derived from the program source code (e.g., Erlingsson et al., 2006). Alternatively, a conservative, heuristic disassembly of the binary code can be used in binary-only settings (e.g., Wartell et al., 2014, 2012b; Zhang and Sekar, 2013).

Unfortunately, recent attacks, such as *counterfeit object-oriented programming* (COOP) (Schuster et al., 2015), have demonstrated the exceptional difficulty of deriving control-flow policies conservative enough to preclude hijackings. For example, the semantics of object-oriented programming idioms tend to intentionally embody large control-flow graphs that are prone to attack even when all unintended edges are dropped (Bounov et al., 2016; Zhang et al., 2016, 2015; Prakash et al., 2015; Jang et al., 2014). One prominent source of abusable edges is method inheritance and overriding, which introduces control-flow edges from all child method call sites to all parent object methods of the same name and/or type signature. This is often enough flexibility for attackers to craft counterfeit objects that traverse only these "intended" control-flow edges—but in an order or with arguments unforeseen by the developers—to hijack the program.

Our experiences with such attacks inspired us to pursue the alternative approaches proposed in this dissertation, which derive and enforce control-flow policies that exclude even some of the developer-intended flows of programs. As an example, consider a command-line file compression/decompression utility deployed in a security-sensitive operational context where only the decompression logic is ever used. In such a context, removing the compression logic (and any vulnerabilities it may contain) from the binary code of the utility could achieve important reductions in the attack surface of the system. Other plausible scenarios include removal of multi-OS compatibility code in contexts where compatibility with only one particular OS is needed, or removal of parser logic for file/media formats never used in a particular computing environment. In all these cases, control-flow trimming makes the control-flows that implement the undesired functionality unrealizable by instrumenting all computed jump instructions in the program with logic that prohibits that flow. Code blocks specific to unrealizable flows can be deleted entirely, further reducing the attack surface and de-bloating the software.

To evaluate our solution and to motivate future work, Chapter 4 introduces the first testing methodology and benchmarking suite for exposing and evaluating potential compatibility barriers for CFI and binary control-flow trimming enforcement on large software products. There has been prolific new research on CFI-based defenses in recent years, mainly aimed at improving performance, enforcing richer policies, obtaining higher assurance of policy-compliance, and protecting against more subtle and sophisticated attacks. For example, between 2015–2018 over 25 new CFI algorithms appeared in the top four applied security conferences alone. These new frameworks are generally evaluated and compared in terms of performance and security. Performance overhead is commonly evaluated in terms of the CPU benchmark suites (e.g., SPEC), and security is often assessed using the RIPE test suite (Wilander et al., 2011) or with manually crafted proof-of-concept attacks (e.g., Schuster et al., 2015). For example, a recent survey systematically compared various CFI mechanisms against these metrics for precision, security, and performance (Burow et al., 2017).

While this attention to performance and security has stimulated rapid gains in the ability of CFI solutions to efficiently enforce powerful, precise security policies, less attention has been devoted to systematically examining which general classes of software can receive CFI protection without suffering compatibility problems. Historically, CFI research has struggled to bridge the gap between theory and practice (cf., Zhang et al., 2013) because code hardening transformations inevitably run at least some risk of corrupting desired, policy-permitted program functionalities. For example, introspective programs that read their own code bytes at runtime (e.g., many VMs, JIT compilers, hot-patchers, and dynamic linkers) can break after their code bytes have been modified or relocated by CFI.

Compatibility issues of this sort have dangerous security ramifications if they prevent protection of software needed in mission-critical contexts, or if the protections must be weakened in order to achieve compatibility. For example, due in part to potential incompatibilities related to *return address introspection* (wherein some callees read return addresses as arguments) the three most widely deployed compiler-based CFI solutions (LLVM-CFI (Tice et al., 2014), GCC-VTV (Tice et al., 2014), and Microsoft Visual Studio MCFG (Tang, 2015)) all presently leave return addresses unprotected, potentially leaving code vulnerable to ROP attacks—the most prevalent form of code-reuse.

Understanding these compatibility limitations, including their impacts on real-world software performance and security, requires a new suite of CFI functional tests with substantially different characteristics than benchmarks typically used to assess compiler or hardware performance. In particular, CFI relevance and effectiveness is typically constrained by the nature and complexity of the target program's *control-flow paths* and *control data dependencies*. Such complexities are not well represented by SPEC benchmarks, which are designed to exercise CPU computational units using only simple control-flow graphs, or by utility suites (e.g., GNU Coreutils) that were all written in a fairly homogeneous programming style for a limited set of compilers, and that use a very limited set of standard libraries chosen for exceptionally high cross-compatibility.

To better understand the compatibility and applicability limitations of modern CFI solutions on diverse, modern software products, and to identify the coding idioms and features that constitute the greatest barriers to more widespread CFI adoption, Chapter 4 presents ConFIRM (CONtrol-Flow Integrity Relevance Metrics), a new suite of CFI tests designed to exhibit code features most relevant to CFI evaluation.[1] Each test is designed to exhibit one or more control-flow features that CFI solutions must guard in order to enforce integrity, that are found in a large number of commodity software products, but that pose potential problems for CFI implementations.

[1]https://github.com/SoftwareLanguagesSecurityLab/ConFIRM

It is infeasible to capture in a single test set the full diversity of modern software, which embodies myriad coding styles, build processes (e.g., languages, compilers, optimizers, obfuscators, etc.), and quality levels. We therefore submit ConFIRM as an extensible baseline for testing CFI compatibility, consisting of code features drawn from experiences building and evaluating CFI and randomization systems for several architectures, including Linux, Windows, Intel x86/x64, and ARM32 in academia and industry (Wartell et al., 2012a,b, 2014; Mohan et al., 2015; Wang et al., 2017; Bauman et al., 2018; Gu et al., 2017; Muntean et al., 2018).

We used ConFIRM to reevaluate 12 publicly available CFI implementations published in the open literature. The results show that about 47% of solution-test pairs exhibit incompatible or insecure operation for code features needed to support mainstream software products, and a *cross-thread stack-smashing* attack defeats all tested CFI defenses.

In summary, the remainder of this dissertation continues as follows: Chapter 2 introduces binary control-Flow trimming (Ghaffarinia and Hamlen, 2019) design. Chapter 3 details the implementation and evaluation of binary control-Flow trimming for Intel x86-64 native codes. Chapter 4 presents, ConFIRM (Xu et al., 2019), a new evaluation methodology and micro-benchmarking suite for assessing compatibility, applicability, and relevance of control-flow integrity (CFI) protections. At the end, relevant related works are presented in Chapter 5 and Chapter 6 concludes this dissertation.

CHAPTER 2

BINARY CONTROL-FLOW TRIMMING[1]

2.1 Introduction

Security of software is widely believed to be inversely related to its complexity (cf., Walden et al., 2014; Zimmermann et al., 2010). With more features, larger implementations, and more behavioral variety come more opportunities for programmer error, malicious code introduction, and unforeseen component interactions.

Unfortunately, economic forces have a history of driving complexity increases in commercial software (sometimes dubbed *Zawinski's law of software envelopment* (Raymond, 2003)). Software developers understandably seek to create products that appeal to the widest possible clientele. This "one-size-fits-all" business model has led to commercial software products of increasing complexity, as developers pack more features into each product they release. As a result, software becomes more multi-purpose and more complex, its attack surface broadens, and more potential opportunities for malicious compromise become available to adversaries. For security-sensitive (e.g., critical infrastructure or military) consumers who leave many product features unused but critically rely on others, these security dangers are often unacceptable. Yet because of the market dominance, low cost, and high availability of one-size-fits-all COTS software, bloated software continues to pervade many mission-critical software networks despite the security disadvantages.

Code-reuse attacks (Solar Designer, 1997; Bletsch et al., 2011; Carlini and Wagner, 2014; Schwartz et al., 2011; Shacham, 2007) are another example of the inherent security risks that code-bloat can introduce. The potential potency of these attacks depends on the variety of code fragments (*gadgets*) in the victim program's executable memory (Göktaş et al., 2014;

[1]This chapter contains material previously published as: Masoud Ghaffarinia and Kevin W. Hamlen. "Binary Control-flow Trimming". In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pp. 1009–1022, November 2019

Homescu et al., 2012), which the attacks abuse to cause damage. Feature-bloated code offers adversaries a larger code-reuse attack surface to exploit.

Control-flow integrity (CFI) protections (Abadi et al., 2005, 2009; Zhang and Sekar, 2013; Tice et al., 2014; Niu and Tan, 2014a, 2015; van der Veen et al., 2015; Mashtizadeh et al., 2015; Akritidis et al., 2008) defend against such attacks by constraining software to a policy of control-flow graph (CFG) edges that is *defined by the programmer* (Abadi et al., 2009; van der Veen et al., 2015) (e.g., derived from the semantics of the programmer-defined source code, or a recovery of those semantics from the program binary). They therefore do not learn or enforce policies that defend against undocumented feature vulnerabilities like Shellshock, whose control-flows are sanctioned by the source semantics and are therefore admitted by CFI controls. Prior CFI solutions failure to address this problem is because they were designed to infer and enforce policies that whitelist developer-intended control-flows, not automatically de-bloat hidden features.

To address this unsolved problem, our research introduces *binary control-flow trimming*, a new technology for automatically specializing binary software products to exclude semantic features undesired by consumers, irrespective of whether the features are intended or even known to developers, or whether they are part of a product's source-level design. Control-flow trimming is envisioned as an extra layer of consumer-side defense (i.e., CFG policy tightening) that identifies and excises unwanted software functionalities and gadgets that are beyond the reach of CFI alone.

Learning consumer-unwanted but developer-intended functionalities cannot be achieved with purely non-probabilistic CFG recovery algorithms, such as those central to CFI, because such algorithms approximate a ground truth policy that is a strict supergraph of the policy needed for feature removal. Tightening that supergraph based on incomplete, consumer-supplied tests requires coupling CFI with trace-based machine-learning. The resulting policy is a more complex, probabilistically constructed, *contextual CFG* (CCFG), which considers fine-grained branch history to distinguish consumer-wanted flows from a sea of developer-intended flows.

In addition, our work assumes that consumers who lack source code probably have no way of formally specifying semantic features or control-flows that they wish to retain, and might not even be aware of all features whose disuse make them candidates for trimming. We therefore assume that consumers merely demonstrate *desired* software features via unit tests (e.g., inputs or user interactions that test behaviors for quality assurance purposes). Such testing is inevitably incomplete and inexhaustive for programs whose input spaces are large (often infinite); so in order to tolerate this incompleteness, we introduce an entropy-based method of detecting points of uncertainty in CCFGs derived from unit tests, and a strategy for relaxing enforcement at such points. Consumers can strengthen the enforcement by supplying additional unit tests that exercise these points more thoroughly.

In summary, we contribute the following in this chapter:

- We present a method to reduce the size and complexity of binary software by removing functionalities unwanted by code-consumers (but possibly intended by code-producers) without any reliance on source code or debug metadata.

- We present a new binary-only context-sensitive control-flow graph (CCFG) integrity policy formalism derivable from execution traces.

- We propose a machine learning approach to construct CCFGs from runtime trace sets.

Section 2.2 first gives a high level overview of our system. Sections 2.3 details our technical approaches to feature identification.

2.2 Overview

2.2.1 Contextual Control-flow Graph Policies

Our approach assumes that feature-trimming specifications are informal, taking the form of unit tests that exercise only the consumer-desired features of the software. Such testing is commonly practiced by security-sensitive consumers. One obvious approach to trimming unwanted features entails simply erasing all code bytes that remain unexecuted by the tests. However, our early experimentation taught us that this blunt approach fails for at least two reasons: (1) It requires an unrealistically comprehensive unit test set, lest some code bytes associated with wanted features go unexercised and get improperly erased. Such comprehensive testing is very difficult to achieve without source code. (2) It often retains unwanted features due to the modular design of complex software, which reuses each individual code block to implement multiple semantic features—some wanted and some unwanted. When all code blocks for an unwanted feature are each needed by some wanted feature, the unwanted feature cannot be trimmed via code byte erasure without corrupting the wanted features.

These experiences led us to adopt the more general approach of control-flow trimming. Control-flow trimming removes semantic features by making the control-flow paths that implement the feature unreachable—e.g., by instrumenting all computed jump instructions in the program with logic that prohibits that flow. This generalizes the code byte erasure approach because, in the special case that the trimmed CFG contains no edges at all to a particular code block, that block can be erased entirely.

We also discovered that control-flow policies that successfully distinguish consumer-undesired (yet developer-intended) code features from consumer-desired features tend to be significantly more complex and powerful than any prior CFI solution can efficiently enforce. In particular, policy decisions must be highly context-sensitive, considering a detailed *history* of prior CFG edges traversed by the program in addition to the next branch target when deciding whether to permit an impending control transfer. Since trace histories of real-world programs are large (e.g., unbounded), these decisions must be implemented in a highly efficient manner to avoid unreasonable performance penalties for the defense.

To illustrate, assume critical functionality F_1 executes code blocks $c_1; c_2; c_3; c_4$ in order, whereas undesired functionality F_2 executes $c_1; c_3; c_3; c_4$. A strict code byte erasure approach cannot safely remove any blocks in this case, since all are needed by F_1. However, control-flow trimming can potentially delete CFG edges (c_1, c_3) and (c_3, c_3) to make functionality F_2 unrealizable without affecting F_1.

Extending the example to include context-sensitivity, consider an additional critical functionality F_3 implemented by sequence $c_2; c_3; c_3; c_1; c_3; c_4$. This prevents removal of edges (c_1, c_3) and (c_3, c_3) from the CFG, since doing so would break F_3. But an enforcement that permits edge (c_3, c_3) conditional on it being immediately preceded by edge (c_2, c_3) successfully removes F_2 without harming F_3. In general, extending the context to consider the last n edges traversed lends the technique greater precision as n increases, though typically at the cost of higher space and time overheads. A balance between precision and performance is therefore needed for best results, which we explore in §3.3.

2.2.2 Automated, In-lined CCFG Enforcement

Figure 2.1 depicts our control-flow trimming architecture. The input to our system consists of stripped x86-64 binaries along with sample execution traces that exercise functionalities wanted by the consumer. The rewriter automatically disassembles, analyzes, and transforms them into a new binary whose control-flows are constrained to those exhibited by the traces, possibly along with some additional flows that could not be safely trimmed due to uncertainty in the trace set or due to performance limitations. We assume that no source code, debug symbols, or other source-derived metadata are provided. Prior work on *reassemblable*

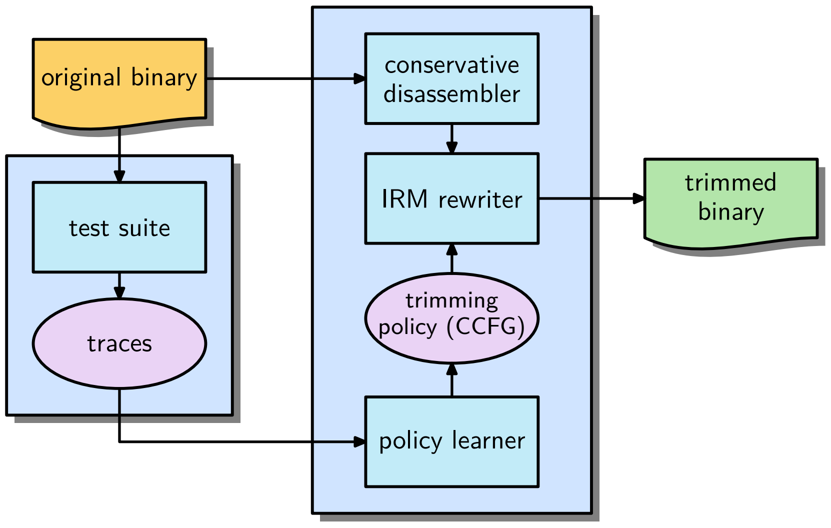


Figure 2.1. Binary control-flow trimming system architecture

disassembly (Wang et al., 2015) has established the feasibility of recovering (raw, unannotated) assembly files from binaries for easier code transformation, allowing us to use assembly files as input to our prototype during evaluations (§3.3).

Discerning a consumer-desired CCFG policy based on traces without access to sources is challenging. Our approach applies machine learning to traces generated from the test suite to learn a subgraph of the developer-intended flows. The output of this step is a decision tree forest, with one tree for each control-flow transfer point in the disassembled program. Each decision tree consults the history of immediately previous branch destinations, along with the impending branch target, to decide whether to permit the impending branch. The forest therefore defines a CCFG policy.

Since decision trees tend to overfit the training, it is important to detect overfitting and relax the policy to permit traces that were not exhibited during training, but whose removal might break consumer-desired functionalities. We therefore assign an entropy-based confidence score to each node of the decision forest. Nodes with unacceptable confidence receive relaxed enforcement by pruning their children from the tree. In the extreme case, pruning all trees to a height of 1 results in a non-contextual CFG that matches the policy enforced by most non-contextual (backward- and forward-edge) CFI. Trimming therefore always enforces a policy that is at least as strict as non-contextual CFI, and usually stricter.

After deriving a suitable CCFG, the policy is enforced via in-lined reference monitoring. Specifically, we surround each control-flow transfer instruction in the program with guard code that maintains and updates a truncated history of branch targets expressed as a hash code. A read-only hash table determines whether the impending branch is permitted. Policy-violating branches yield a security violation warning and premature termination of the program.

2.2.3 Threat Model

Like prior research on CFI and artificial diversity, success of our approach can be measured in terms of two independent criteria: (1) inference of an accurate policy to enforce, and (2) enforcement of the inferred policy. For example, COOP attacks (Schuster et al., 2015) exploit lapses in the first criterion; they hijack software by traversing only edges permitted by the policy, which is insufficiently precise. In contrast, coarse-grained CFI approaches are susceptible to lapses in the second criterion; to achieve high performance, they enforce a policy approximation, which sometimes allows attackers to exploit approximation errors to hijack the code (e.g., Carlini et al., 2015). Artificial diversity defenses can experience similar failures, as in the case of implementation disclosure attacks (Bittau et al., 2014; Davi et al., 2015; Seibert et al., 2014; Snow et al., 2013; Gawlik et al., 2016).

With regard to the first criterion, our approach is probabilistic, so success is evaluated empirically in §3.3 in terms of false negatives and false positives. (The false classification rates measure accuracy against a policy that differs from CFI policies, however, since control-flow trimming has a stricter model of ground truth than CFI, as described in §2.1.) With regard to the second criterion, we assume a relatively strong threat model in which attackers have complete read-access to the program image as it executes, and even have write-access to all writable data pages, but lack the power to directly change page access permissions. Thus, attackers know the policy being enforced but lack the ability to change it since its runtime encoding resides in read-only memory. (We assume that DEP or W$\bigoplus$X protections prevent writes to code and static data sections.) Attackers also cannot directly corrupt CPU machine registers, affording our defense a safe place to store security state.

Since our defense enforces a control-flow policy, non-control data attacks are out of scope for this work. We defer mitigations of such attacks to other defense layers.

2.3 Design

2.3.1 Learning CCFG Policies

Since it is usually easier for code-consumers to exhibit all features they wish to preserve (e.g., through software quality testing), rather than discovering those they wish to remove, we adopt a whitelisting approach when learning consumer control-flow policies:

A *trace* $e_1, e_2, e_3, \ldots$ is defined as the sequence of control-flow edge traversals during one run of the program, where e_i is the ith edge taken. We include in the edge set all binary control-flow transfers except for unconditional branches and fall-throughs of non-branching instructions (whose destinations are fixed and therefore not useful to monitor). Thus, the edge set includes targets of conditional branches, indirect (computed) branches, and returns.

Let T_1 be a set of program execution traces that exhibit only software features that must be preserved, and let T_2 be a set that includes traces for both wanted and unwanted features. T_1 is provided by the user, and is assumed to be noise-free; every trace exhibited during training is a critical one that must be preserved after control-flow trimming. However, we assume there may be additional critical traces requiring preservation that do not appear in T_1. The learning algorithm must therefore conservatively generalize T_1 in an effort to retain desired functionalities. T_2 is assumed to be unavailable during training, and is used only for evaluation purposes to assess whether our training methodology learns accurate policies.

Control-flow contexts are defined as finite-length sub-sequences of traces. A CCFG policy can therefore be defined as a set of permissible control-flow contexts. While the logic for precisely enforcing an entire CCFG policy could be large, the logic needed to enforce the policy at any particular branch origin need only consider the subset of the policy whose final edge begins at that branch origin. This distributes and specializes the logic needed to enforce the policy at any given branch site in the program.

Context lengths are not fixed in our model. While an upper bound on context lengths is typically established for practical reasons, our approach considers different context lengths at different branch sites based on an estimate of the benefits, as measured by information gain. In our design, we first suppose there is a fixed size (possibly large) for the contexts, and then proceeded to accommodate variable-sized contexts.

To maximize effectiveness, contexts must include as much policy-relevant control-flow information as possible without being polluted with uninformative edges. Indirect branches and returns are the primary sources of control-flow hijacks, so are included. Direct calls and jumps are also included even though they have fixed destinations, because we found that doing so allows the training to learn a form of call-return matching that improves accuracy. We also include conditional branch destinations in the contexts, since they often implement series of tests that conditionally activate software features that may be targets of trimming.

The learning algorithm is a binary classification that decides for each control-flow edge whether it is permissible, based on the last k edges currently in the context. We chose decision trees as our learning model, since they are relatively simple and efficient to implement at the binary level. While decision trees can suffer from overfitting, such overfitting is potentially advantageous for our problem because every trace in T_1 must be preserved. Higher security therefore results from a conservatively tight model that can be conditionally relaxed at points of uncertainty.

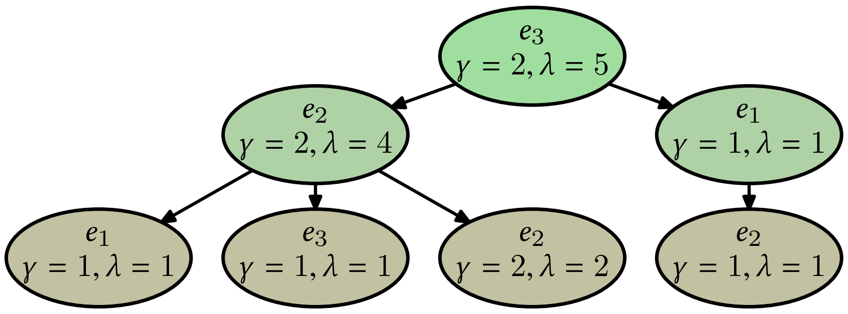

e_3 $\gamma = 2, \lambda = 5$
e_2 $\gamma = 2, \lambda = 4$
e_1 $\gamma = 1, \lambda = 1$
e_1 $\gamma = 1, \lambda = 1$
e_3 $\gamma = 1, \lambda = 1$
e_2 $\gamma = 2, \lambda = 2$
e_2 $\gamma = 1, \lambda = 1$

Figure 2.2. Sample decision tree for edge e_3

For a given edge e, the learning algorithm creates a decision tree as follows: The root is labeled with e and the depth of the tree is k, where k is the maximum size of the context. Each node at level $i \geq 1$ of the tree is labeled with the edge e' appearing immediately before the context defined by the path from the node's parent at level i up to the root. It is additionally annotated with the number of traces γ and number of contexts λ in which that particular edge-label occurs at that context position. These numbers are used during uncertainty detection and policy relaxation (§2.3.2).

Every leaf of this tree represents a permissible control-flow history encoded by the path from it to the root. The feature encoded by a node at level $i+1$ is the i-to-last edge in the context when the edge labeled at the root is reached. So, given a context χ we can check whether it is permissible as follows: The last edge in χ must be a root of some tree in our learned decision tree forest; otherwise the impending branch is rejected. The penultimate edge in χ should be one of that root's children; otherwise the impending branch is rejected. We continue to check the context edges in χ in reverse order until we reach a decision tree leaf. Reaching a leaf implies policy-compliance, and the impending branch is permitted.

To illustrate, consider a hypothetical program with two sample traces: one containing sub-sequences $[e_1, e_2, e_3]$, $[e_2, e_2, e_3]$ and $[e_3, e_2, e_3]$; and the other containing sub-sequences $[e_2, e_1, e_3]$ and $[e_2, e_2, e_3]$. Figure 2.2 shows the decision tree made for edge e_3 out of these sub-traces. The root is labeled with $(e_3, \gamma = 2, \lambda = 5)$, since there are 2 traces and 5 histories having edge e_3. Edge e_2 is the penultimate edge in 4 of those cases, and e_1 is the penultimate edge in 1 case, causing nodes $(e_2, \gamma = 2, \lambda = 4)$, and $(e_1, \gamma = 1, \lambda = 1)$ to comprise the next level of the tree. In the same way, the nodes at the bottom level correspond to the antepenultimate edges appearing in each context. Edges e_1, e_3, and e_2 are antepenultimate when e_2 is penultimate, and e_2 is antepenultimate when e_1 is penultimate. Observe that the labels are not unique; the same label or edge can be assigned to some other node of the same tree. In addition, for any node, λ is the sum of its child λ's, while γ is not.

2.3.2 CCFG Policy Relaxation

To cope with the inevitable incompleteness of training data that is assumed to be amassed without guidance from source code, we next consider the problem of generalizing the decision tree forest to include more child nodes than were explicitly observed during training. In general, if training observes many diverse jump destinations for a specific subtree, that subtree may have a complex behavior that was not exhaustively covered by training. There is therefore a high chance that additional consumer-desired destinations for that branch site exist that were not explicitly observed.

The same is true for diverse collections of contexts. If the contextual information at a given tree node is highly diverse and offers little information gain, this indicates that the context at that position is not a useful predictor of whether the impending branch is permissible. For example, the branch may be the start of what the user considers an independent semantic feature of the software, in which case the context is reflecting a previous semantic feature that has little relevance to the permissibility of this branch point. Thus, nodes with numerous low-frequency child nodes should be considered with low confidence.

To estimate this confidence level, we use entropy to calculate an uncertainty metric using the number of times different child nodes of a node appear in the training. Nodes with diverse children have higher entropy. The confidence score of a node n is computed as

$$\mathit{confidence}(n) = \frac{\gamma}{N} \times -\frac{1}{M^2} \sum_{m=1}^{M} \frac{\lambda_m}{\lambda} \log_M \left(\frac{\lambda_m}{\lambda} \right) \tag{2.1}$$

where (e, γ, λ) is node n's label, M is the number of node n's children, $(e_m, \gamma_m, \lambda_m)$ is child m's label, and N is the total number of traces.

This formula combines the probability of a node being in a trace, the entropy of its children λ, and the number of its children. It is inversely related to entropy because, for any given number of children M, we have higher confidence if the distribution of child frequencies is relatively flat. For example, if we observe two children with λ's 5 and 5, we have higher confidence than if we observe two children with λ's 1 and 9. The former indicates a well-covered, predictable behavior, whereas the latter is indicative of a behavior with rare outliers that were not covered well during training. Fewer children likewise engender higher confidence in the node.

An ideal confidence threshold t^* that maximizes accuracy on the training set is computed using crossfold validation (see §3.3), and all children with confidence below t^* are pruned from the forest. In the worst case, pruning all the trees to a height of 1 yields a non-contextual CFG that is the policy that would be enforced by typical non-contextual CFI (i.e., no debloating). Pruning therefore finds a middle ground between trimming only the developer-unintended features and over-trimming the consumer-wanted features.

For example, in Figure 2.2 the confidence score of the root and the node labeled $(e_2, \gamma = 2, \lambda = 4)$ are 0.36 and 0.31, respectively. If our confidence threshold exceeds a node's confidence score, then context is disregarded when making policy decisions at that origin. So in our example, a confidence threshold of 0.35 prunes the tree after node $(e_2, \gamma = 2, \lambda = 4)$, making that node a leaf. This refines the policy by disregarding policy-irrelevant context information.

```
level 0 target:-0x1a6f, Gamma:86 Lambda:86 M:2 Score:0.427090102576
  level 1 target:-0x1f7f, Gamma:24 Lambda:24 M:1 Score:0.279069767442
    level 2 target:-0x1fa3, Gamma:24 Lambda:24 M:1 Score:0.279069767442
      level 3 target:-0x1fb7, Gamma:24 Lambda:24 M:1 Score:0.279069767442
        level 4 target:-0x1fdf, Gamma:24 Lambda:24 M:0 Score:0.279069767442
  level 1 target:-0x1f74, Gamma:62 Lambda:62 M:1 Score:0.720930232558
    level 2 target:-0x1fb7, Gamma:62 Lambda:62 M:1 Score:0.720930232558
      level 3 target:-0x1fdf, Gamma:62 Lambda:62 M:1 Score:0.720930232558
        level 4 target:-0x1bc9, Gamma:62 Lambda:62 M:0 Score:0.720930232558
```

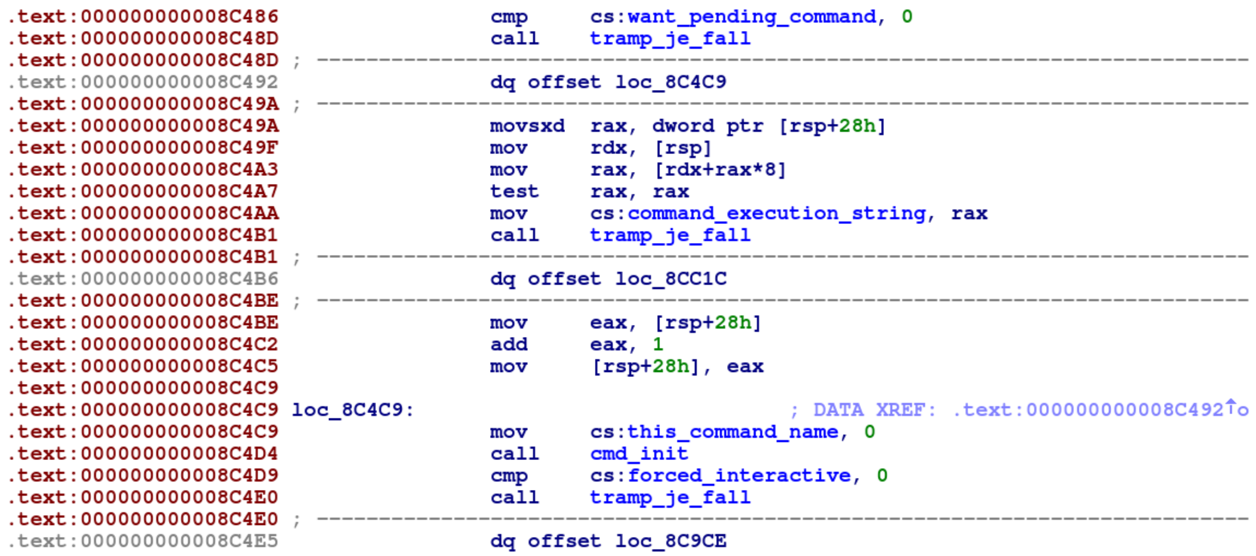
Figure 2.3. CCFG tree for target address `0x55580cce` in Bash 4.1.

```
if (want_pending_command)
  {
    command_execution_string = argv[arg_index];
    if (command_execution_string == 0)
      {
        report_error (_("%s: option requires an argument"), "-c");
        exit (EX_BADUSAGE);
      }
    arg_index++;
  }
this_command_name = (char *)NULL;

cmd_init();

if (forced_interactive ||             /* -i flag */
    (!command_execution_string &&     /* No -c command and ... */
     wordexp_only == 0 &&             /* No --wordexp and ... */
     ((arg_index == argc) ||          /*   no remaining args or... */
      read_from_stdin) &&             /*   -s flag with args, and */
     isatty (fileno (stdin)) &&       /* Input is a terminal and */
     isatty (fileno (stderr))))       /* error output is a terminal. */
  init_interactive ();
else
  init_noninteractive ();
```

Figure 2.4. Part of Bash 4.1 `main` function source code.



```
.text:000000000008C486                 cmp     cs:want_pending_command, 0
.text:000000000008C48D                 call    tramp_je_fall
.text:000000000008C48D ; ---------------------------------------------------------------------------
.text:000000000008C492                 dq offset loc_8C4C9
.text:000000000008C49A ; ---------------------------------------------------------------------------
.text:000000000008C49A                 movsxd  rax, dword ptr [rsp+28h]
.text:000000000008C49F                 mov     rdx, [rsp]
.text:000000000008C4A3                 mov     rax, [rdx+rax*8]
.text:000000000008C4A7                 test    rax, rax
.text:000000000008C4AA                 mov     cs:command_execution_string, rax
.text:000000000008C4B1                 call    tramp_je_fall
.text:000000000008C4B1 ; ---------------------------------------------------------------------------
.text:000000000008C4B6                 dq offset loc_8CC1C
.text:000000000008C4BE ; ---------------------------------------------------------------------------
.text:000000000008C4BE                 mov     eax, [rsp+28h]
.text:000000000008C4C2                 add     eax, 1
.text:000000000008C4C5                 mov     [rsp+28h], eax
.text:000000000008C4C9
.text:000000000008C4C9 loc_8C4C9:                              ; DATA XREF: .text:000000000008C492↑o
.text:000000000008C4C9                 mov     cs:this_command_name, 0
.text:000000000008C4D4                 call    cmd_init
.text:000000000008C4D9                 cmp     cs:forced_interactive, 0
.text:000000000008C4E0                 call    tramp_je_fall
.text:000000000008C4E0 ; ---------------------------------------------------------------------------
.text:000000000008C4E5                 dq offset loc_8C9CE
```

Figure 2.5. Part of Figure 2.4 assembly code.

2.3.3 CCFG Real-world Policy Example

Figure 2.3 exhibits a real-world CCFG policy by depicting a decision tree for target address `0x555555580cce` of Bash 4.1. Node levels are expressed via indentations and are also written in the beginning of each line, and scores are calculated with $N = 86$. Figure 2.4 is part of Bash 4.1's main function source code, whose assembly code is partially shown in Figure 2.5. The base label, `local_base_label`, is located at address `0x8e43d` of the `.text` section in Figure 2.5. The location of offset `-0x1fb7` in the disassembled code is obtained by adding it to the base: `-0x1fb7` + `0x8e43d` = `0x8c486`. This is the first line of code in Figure 2.5, and therefore locates the `if` statement at the first line of Figure 2.4. Similarly, `-0x1fa3` and `-0x1f7f` are the address offsets for `0x8c49a` and `0x8c4be`, which are the fallthrough and the jump targets of this conditional branch. We can decode the tree and describe the policy as follows: Whenever the conditional branch located at `0x8c4e0` falls through, the policy requires that `want_pending_command` was false or `command_execution_string` was not 0. The policy therefore prohibits calls to `report_error` in Figure 2.4 followed suddenly by execution of the `cmd_init` line and so forth.

2.3.4 Enforcing CCFG Policies

In-lining guard code that enforces a highly context-sensitive policy at every computed branch without incurring prohibitive overheads raises some difficult implementation challenges. To track and maintain contexts, our enforcement must additionally instrument all direct calls, conditional branches, and interrupt handlers with context-update logic. Space-efficiency is a challenge because CCFG policies are potentially large—code with b branch sites and context-length bound k can have CCFG policies of size $O(b^k)$ in the worst case. Time-efficiency is a challenge because policy decisions for CCFGs potentially require $O(k)$ operations, in contrast to non-contextual CFG policies, which engender constant-time decisions.

To obtain acceptable overheads in the face of these challenges, our implementation compactly represents contexts as hash codes, and represents CCFG policies as sparse hash tables of bits, where an entry of 1 indicates a permitted context. The hash function need not be secure since our enforcement protects hash values via access controls (see §3.2), but it must be efficiently computable and uniform. We therefore use the relatively simple hash function given by

$$hash(\chi) = \bigoplus_{i=1}^{|\chi|} ((\pi_2 \chi_i) \ll (|\chi| - i)s) \tag{2.2}$$

where $\bigoplus$ is xor, $|\chi|$ is the length of context χ, $\pi_2 \chi_i$ is the destination (second projection) of the ith edge in χ, $\ll$ is bit-shift-left, and $s \geq 0$ is a shift constant. This has the advantage of being computable in an amortized fashion based on the following recursion:

$$hash(\chi e) = (hash(\chi) \ll s) \oplus (\pi_2 e) \tag{2.3}$$

The CCFG hash table is constructed by storing a 1 at the hash of every policy-permitted context. This can introduce some imprecision in the form of hash collisions, since a policy-violating context can have the same hash code as a policy-permitted context, causing both to be accepted. However, this collision rate can be arbitrarily reduced by increasing shift-constant s and the bit-width w of shift operation $\ll$. For example, setting s to the address-width a and using $w = ka$ guarantees no collisions, at the expense of creating a large table of size 2^{ka-3} bytes. On 64-bit architectures, we found that using $s = 1$ and $w \approx \log_2 c$ where c is the code segment size works well, since all branch destination *offsets* (into their respective code segments) are less than c, and the offset portion of the address is where the most policy-relevant bits reside. This yields a hash table of size $O(c)$, which scales linearly with program size.

CHAPTER 3

BINARY CONTROL-FLOW TRIMMING IMPLEMENTATION AND EVALUATION FOR INTEL X86-64 NATIVE CODES[1]

3.1 Introduction

As discussed earlier in Chapter 2, security of software stands at odds with one of the major economic forces in the software market—the need to mass-produce ever more general-purpose software in order to tractibly meet the broadening needs of the expanding base of software consumers worldwide. Software developers understandably seek to create products that appeal to the widest possible clientele, in order to maximize sales and reduce overheads. This has led to commercial software products of increasing complexity, as developers pack more features into each product they release. As a result, software is becoming more and more difficult to reliably secure as software becomes more multi-purpose and more complex.

As a high-profile example of such feature bloat, in 2014 the bash command interpreter, which is a core component of nearly all Posix-compliant operating systems, was found to contain a series of obscure, undocumented features in its parser (Vaughan-Nichols, 2014) that afforded attackers near-arbitrary remote code execution capabilities. Though sometimes referred to as the Shellshock "bug," the vulnerabilities were likely intended as features related to function inheritance when bash was originally written in the 1980s (Wheeler, 2015). Their inclusion in a rarely analyzed part of the code caused them to elude detection for a quarter century, exposing millions of security-sensitive systems to potential compromise. This demonstrates that high-complexity software can contain obscure features that may have been *intended by software developers,* but that are unknown to consumers and pose security risks in certain deployment contexts.

[1]This chapter contains material previously published as: Masoud Ghaffarinia and Kevin W. Hamlen. "Binary Control-flow Trimming". In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pp. 1009–1022, November 2019

Table 3.1. CVEs of security-evaluated products

Program	**CVE numbers**
Bash	CVE-2014-6271, -6277, -6278, -7169
ImageMagic	CVE-2016-3714, -3715, -3716, -3717, -3718
Proftpd	CVE-2015-3306
Node.js	CVE-2017-5941
Exim	CVE-2016-1531

Beside unknown and undocumented features, recent attacks, such as *counterfeit object-oriented programming (COOP)* (Schuster et al., 2015), have demonstrated the exceptional difficulty of deriving control-flow policies conservative enough to preclude hijackings as well. For example, the semantics of object-oriented programming idioms tend to intentionally embody large control-flow graphs that are prone to attack even when all unintended edges are dropped (Bounov et al., 2016; Zhang et al., 2016, 2015; Prakash et al., 2015; Jang et al., 2014). One prominent source of abusable edges is method inheritance and overriding, which introduces control-flow edges from all child method call sites to all parent object methods of the same name and/or type signature. This is often enough flexibility for attackers to craft counterfeit objects that traverse only these "intended" control-flow edges—but in an order or with arguments unforeseen by the developers—to hijack the program.

Context-sensitive CFI policies as discussed earlier in Chapter 2 can prevent such attacks. No prior CFI approach can enforce policies of this complexity because their sensitivity is presently limited to only a few code features (e.g., system API calls (Pappas et al., 2013)), they rely on machine registers or OS modifications unavailable to user code (Cheng et al., 2014; van der Veen et al., 2015), or they require source code access (Mashtizadeh et al., 2015) which is often unavailable to consumers. To enforce these policies, we therefore introduce a new contextual CFI enforcement strategy that efficiently encodes contexts as hash codes safely maintainable in user-level machine registers.

In summary, we contribute the following in this chapter:

Table 3.2. Finding the address of "`__libc_start_main`"

```
lea   (__base_symbol), %rdx
addl  2(%rdx), %edx
add   $6, %rdx
mov   (%rdx), %rdx
movd  (%rdx), %xmm12
```

- We showcase a fully functional prototype that automatically instruments native code with an in-lined reference monitor (IRM) (Schneider, 2000) that enforces the CCFG policy.
- We demonstrate that binary control-flow trimming is accurate enough to exhibit a 0% false positive rate for complicated programs such as compilers and web servers.
- Experiments show that control-flow trimming can eliminate zero-day vulnerabilities associated with removed functionalities, and that the approach exhibits low runtime overheads of about 1.87%.

Section 3.2 details our policy enforcement implementation for Intel x86-64 native codes. Section 3.3 evaluates the approach in terms of accuracy and performance, and Section 3.4 concludes this chapter.

3.2 Implementation

To generate sample traces, we use Pin (Luk et al., 2005) and DynamoRIO (Bruening, 2004) to track all branches during each run of each test program. We then apply our machine learning algorithm with the hash function defined in §2.3.4 to generate the CCFG hash table. The hash table is added in a relocatable, read-only data section accessible from shared libraries while protecting it from malicious corruption.

We next in-line some initializer code into the program that pre-computes section base addresses (after any load-time randomization due to ASLR) needed to encode CFG edges as

Table 3.3. Callee trampoline for inter-module calls

`leaq`	`base_label_local(%rip), %r11`
`movd`	`%r11d, %xmm12`
`call`	*callee_function*
`jmp`	`caller_tramp`

Table 3.4. Caller trampoline for inter-module calls

`leaq`	`base_label_local(%rip), %rcx`
`movd`	`%ecx, %xmm12`

Table 3.5. Guard checks for each kind of branch type

Description	**Original code**	**Rewritten Code**
Conditional Jumps	*jcc l*	`call` *jcc*`_fall` `.quad` *l*

Indirect calls	`call` *r*/[*m*]	`mov` *r*/[*m*]`, %rax`
		`call indirect_call`

Indirect Jumps	`jmp` *r*/[*m*]	`mov %rax, -16(%rsp)`
		`mov` *r*/[*m*]`, %rax`
		`call indirect_jump`

Variable Returns	`ret` *n*	`pop %rdx`
		`lea` *n*`(%rsp), %rsp`
		`push %rdx`
		`jmp return`

Returns	`ret`	`mov (%rsp), %rdx`
		`jmp return`

base-offset pairs, before calling the main program's entry point. When excluding libraries from control-flow trimming, all branches except returns in our test applications have destinations in the program's `.text`, `.plt`, or `.got` sections. In our experiments we found all returns also target these sections, except the return that ends the program, which jumps into `libc`. Therefore, this particular `ret` instruction is the only instruction whose destination offset is calculated relative to another symbol.

In our implementation we used a an artificial symbol as an origin address for all of these section base addresses. (This causes some offsets for section addresses such as `.plt` to be

Table 3.6. Trampolines used in guards referred in Table 3.5

Label	Assembly Code
`indirect_jump:`	`push  %rax`
	`common-guard`
	`mov   -8(%rsp), %rax`
	`ret`
`indirect_call:`	`push  %rax`
	`common-guard`
	`ret`
`return:`	`common-guard`
	`ret`
jcc`_fall:`	*jcc* `jump_l`
	`jmp   fall_l`
jcc`_back:`	*jcc* `jump_l`
	`jmp   back_l`
`jump_l:`	`xchg  (%rsp), %rax`
	`mov   (%rax), %rax`
	`jmp   condition_jump`
`fall_l:`	`xchg  (%rsp), %rax`
	`lea   8(%rax), %rax`
	`jmp   condition_jump`
`back_l:`	`xchg  (%rsp), %rax`
	`lea   8(%rax), %rax`
	`xchg  (%rsp), %rax`
	`ret`
`condition_jump:`	`push  %rax`
	`common-guard`
	`pop   %rax`
	`xchg  (%rsp), %rax`
	`ret`

negative, but that does not affect our algorithm.) Table 3.2 provides the code to load address of "`__libc_start_main`", which our IRM executes after returning from the program's main entry point but before returning into `libc`. The loader exchanges this symbol with the address of a near-jump instruction in `.plt` that jumps to the library; therefore, we extract the address from the jump instruction's encoding.

One particular case for programs with shared libraries is when one module calls a function in another module. For those cases, the base address should change at the callee site before any progress. Also, a return from one module to another module requires a change of base address to the caller base address. Therefore, for every possible pair of callee function and a caller module we define a pair of trampoline shown in 3.3 and 3.4 tables that change the base address accordingly. A subtle change that make the trampolines work is to swap the original `call` instruction to the callee by a `jmp` instruction to the corresponding callee trampoline.

we define a set of artificial entry points (trampolines) each for any pair of callee function and a module that possibly calls the callee. Our rewriter, determine all possible such pairs in advance and add the trampolines for each library. They are needed as the return address of a call from one module to another needs two times

Table 3.5 transforms each type of branch instruction (column 2) to guard code (column 3). To reduce code size overhead, the guard code is modularized into trampolines that jump to a policy-check before jumping to each target. This trades smaller code size for slightly higher runtime overhead. Table 3.6 shows the details of the trampoline code called by branch guards (Table 3.5), which invoke policy checks and state updates (Table 3.7).

Guard code for conditional jumps must carefully preserve all CPU status flags until the branch decision is made. Since sequences of n consecutive conditional jumps can implement an n-way branch, we avoid corrupting status flags by updating the context before the sequence is complete, in-lining only one fall-through trampoline for the sequence. This is achieved by using another trampoline `j`*cc*`_back` for the first $n - 1$ instructions, which fall-through without checking the destination because the guards in Table 3.7 are the only parts that affect flags. A similar strategy applies to conditional branches followed by Intel conditional-moves (`set`*cc* and `cmov`*cc*). This results in a maximum of 67 trampolines for all possible conditional jumps (2×32 for the two directions of each of the 32 possible conditional jump instructions on x86-64, plus 3 other trampolines `fall_l`, `back_l`, and `jump_l`).

Table 3.7 shows the common guard invoked by the trampolines, which updates the context and consults the hash table to enforce the policy. Two implementations are provided: the center column uses SSE instructions, which are widely available on Intel-based processors; while the rightmost column provides a more efficient implementation that leverages SHA-extensions (`sha1msg1` and `sha1msg2`) that are presently only available on a few processor

Table 3.7. Guard checks implementation for trampolines referred as `common-guard` in Table 3.6

	Guard Code			
Guard Name	**Legacy-mode**		**SHA-extension**	
`before-check`	1:`movd`	r`, %xmm12`	1:`movd`	r`, %xmm12`
	2:`psubd`	`%xmm13, %xmm12`	2:`psubd`	`%xmm13, %xmm12`
			3:`sha1msg1`	`%xmm14, %xmm15`
			4:`sha1msg2`	`%xmm15, %xmm15`
			5:`pslrdq`	`$4, %xmm15`
	3:`pxor`	`%xmm12, %xmm15`	6:`pxor`	`%xmm12, %xmm15`
`check`	4:`movd`	`%xmm15,` r	7:`movd`	`%xmm15,` r
	5:`and`	$max\_hash - 1$`,` r	8:`and`	$max\_hash - 1$`,` r
	6:`bt`	r`, (HASH_TABLE)`	9:`bt`	r`, (HASH_TABLE)`
	7:`jnb`	`TRAP`	10:`jnb`	`TRAP`
`after-check`	8:`pextrd`	`$3, %xmm14,` r	11:`pslldq`	`$4, %xmm14`
	9:`pslldq`	`$4, %xmm14`	12:`psllw`	`$1, %xmm14`
	10:`pxor`	`%xmm12, %xmm14`	13:`pxor`	`%xmm12, %xmm14`
	11:`movd`	r`, %xmm12`		
	12:`pxor`	`%xmm12, %xmm15`		
	13:`pslld`	`$1, %xmm15`		
	14:`pslld`	`$1, %xmm14`		

lines (Al-Qudsi, 2017). Our experiments and the descriptions that follow use the legacy-mode implementation, but we expect improved performance of our algorithm as SHA extensions become more available.

For efficiency and safety, we store contexts in 128-bit `xmm` registers rather than memory. Register `%xmm14` maintains a length-4 context as four packed 32-bit unsigned integers, and `%xmm15` maintains the context hash. On entry to the `before-check` code, `%xmm13` contains the section base address and general (64-bit) register r holds the impending branch target to check. Register r varies depending on the branch type (`%rdx` for returns and `%rax` for others).

This implementation strategy requires the target program to have at most 12 live `xmm` registers (out of 16 total) at each program point, leaving at least 2 to globally maintain context and context-hash, plus 2 more for scratch use at each guard site. More constrained `xmm` register usage is rare, but can be supported by spilling `xmm` registers to general-purpose registers or to memory. Two of the evaluated programs in §3.3 require this special treatment (postgres and postmaster), and exhibited slightly higher than average overheads of 3% as a result.

Lines 1–2 of `before-check` calculate the target offset. Line 3 then updates the hash code using Equation 2.3. After this, `%xmm12` and `%xmm15` have the target offset and the new hash, respectively.

The `check` operation implements the policy check. Line 5 truncates the hash value to the size of the hash table. Finally, line 6 finds the bit corresponding to the hash value in the table, and line 7 jumps to the trap in case it is unset, indicating a policy rejection.

The `after-check` code updates the history in `%xmm14` and the hash code in `%xmm15`. It does so by extracting the oldest context entry about to be evicted (line 8), shifting the context left to evict the oldest entry and make space for a new one (line 9), adding the new entry (line 10), and leveraging involutivity of xor to remove the evicted entry from the hash code (lines 11–12). Finally, lines 13–14 left-shift the context and hash code by one bit in preparation for the next context and hash update.

One important deployment consideration is whether to exclude library control-flows from the program flow, since they are shared, and it may therefore be infeasible to learn appropriate policies for them based on profiling only some applications that load them. On the other hand, if security is a priority, the user may be interested in generating a specialized, non-shared version of the shared library specifically for use by each security-sensitive application. For this work, we enforce the policy on all branches from any portion of the program code section and all the shared libraries shipped with it, but we leave system shared libraries unaltered. The latter can optionally be trimmed by making a local copy to which the policy is applied, though the result is obviously no longer a library that can be shared across multiple applications.

Table 3.8. An example for code transformation of computed calls and returns in lighttpd

Original Code	**Rewritten Code**
`data_config_insert_dup:`	`data_config_insert_dup:`
`.LFB89:`	`.LFB89:`
`.cfi_startproc`	`.cfi_startproc`
`subq  $8, %rsp`	`subq  $8, %rsp`
`.cfi_def_cfa_offset 16`	`.cfi_def_cfa_offset 16`
`movq  16(%rsi), %rax`	`movq  16(%rsi), %rax`
`movq  %rsi, %rdi`	`movq  %rsi, %rdi`
	`movq  %r11, %xmm10`
	`mov   16(%rax), %r11`
`call  *16(%rax)`	`call  indirect_call`
`xorl  %eax, %eax`	`xorl  %eax, %eax`
`addq  $8, %rsp`	`addq  $8, %rsp`
`.cfi_def_cfa_offset 8`	`.cfi_def_cfa_offset 8`
	`movq  %rcx, %xmm10`
	`mov   (%rsp), %rcx`
`ret`	`jmp   return_guard`
`.cfi_endproc`	`.cfi_endproc`

When rewriting app-included shared libraries, we add trampolines to each image, and declare them with `.hidden` visibility to avoid symbol name-clashes between the images. The hash table can be specialized to each image or centralized for all. For this work we use one centralized table for all interoperating images, accessed via the `.got` table for concurrent, shared access between modules.

3.2.1 Transforming Real-world Assembly Code

To better understand how real world assembly code is transformed under our rewriter, we illustrate using some examples of rewriting the `lighttpd` web-server and Bash. The code that implements function `data_config_insert_dup` is within the `src/lighttpd-data_config.o` object file. Table 3.8 shows the transformation of this code. The `call` and `ret` instructions are the two control-flow transfer instructions within this code that require security guards in order to enforce the policy. Since the `call` is indirect, its target is only discoverable at

Table 3.9. An example for code transformation of computed jumps in lighttpd

Original Code	Rewritten Code
`connection_write_cq:`	`connection_write_cq:`
`.LFB100:`	`.LFB100:`
`.cfi_startproc`	`.cfi_startproc`
`movl  88(%rsi), %esi`	`movl  88(%rsi), %esi`
	`movq  %r11, %xmm10`
	`mov   792(%rdi), %r11`
`jmp   *792(%rdi)`	`jmp   indirect_jump`
`.cfi_endproc`	`.cfi_endproc`

Table 3.10. An example for code transformation of conditional jumps in lighttpd

Original Code	Rewritten Code
`cmpb  $32, %cl`	`cmpb  $32, %cl`
`je    .L81`	`call  tramp_je_back`
	`quad  .L81`
`jle   .L139`	`call  tramp_je_fall`
	`quad  .L139`
`cmpb  $44, %cl`	`cmpb  $44, %cl`
`je    .L81`	`call  tramp_je_fall`
	`quad  .L81`
`cmpb  $87, %cl`	`cmpb  $87, %cl`
`je    .L90`	`call  tramp_je_fall`
	`quad  .L90`
`cmpb  $34, %cl`	`cmpb  $34, %cl`
`je    .L140`	`call  tramp_je_fall`
	`quad  .L140`

runtime by reading the `%r11` register. Before modifying `%r11`, we must save it as it may be used later. Memory operations are less efficient than register-only operations, so our algorithm here saves `%r11` in `%xmm10`, which is free throughout the lighttpd program. The `ret` instruction receives similar treatment, except that the target is in `(%rsp)` and we save the target in `%rcx`.

As another example, Table 3.9 depicts the code transformation for `connection_write_cq` function defined in `src/lighttpd-connections.o` object file. As with an indirect call, to

Table 3.11. An example for code transformation in Bash for the policy example provided in §2.3.3

Original Code		**Rewritten Code**	
`.L202:`		`.L202:`	
`cmpb`	`$0, want_pending_command(%rip)`	`cmpb`	`$0, want_pending_command(%rip)`
`je`	`.L201`	`call`	`tramp_je_fall`
		`.quad`	`.L201`
`movslq`	`40(%rsp), %rax`	`movslq`	`40(%rsp), %rax`
`movq`	`(%rsp), %rdx`	`movq`	`(%rsp), %rdx`
`movq`	`(%rdx,%rax,8), %rax`	`movq`	`(%rdx,%rax,8), %rax`
`testq`	`%rax, %rax`	`testq`	`%rax, %rax`
`movq`	`%rax, command_...._string(%rip)`	`movq`	`%rax, command_...._string(%rip)`
`je`	`.L392`	`call`	`tramp_je_fall`
		`.quad`	`.L392`
`movl`	`40(%rsp), %eax`	`movl`	`40(%rsp), %eax`
`addl`	`$1, %eax`	`addl`	`$1, %eax`
`movl`	`%eax, 40(%rsp)`	`movl`	`%eax, 40(%rsp)`
`.L201:`		`.L201:`	
`movq`	`$0, this_command_name(%rip)`	`movq`	`$0, this_command_name(%rip)`
`call`	`cmd_initPLT`	`call`	`cmd_initPLT`
`cmpl`	`$0, forced_interactive(%rip)`	`cmpl`	`$0, forced_interactive(%rip)`
`je`	`.L204`	`call`	`tramp_je_fall`
		`.quad`	`.L204`

guard an indirect jump instruction we first save the `%r11` register in `%xmm10` and then move the target to `%r11` to further be used in the jump trampoline. As discussed earlier in the previous section, guard code for conditional jumps must carefully preserve all CPU status flags until the branch decision is made.

Table 3.10 contains code snippets from the `src/lighttpd-etag.o` object file in which the first two conditional jumps are consecutive without any instruction that affects CPU status flags in between, while the rest of conditional jumps are preceded by such instructions. Therefore, the first conditional jump is swapped by a call to the `tramp_je_back` trampoline, which does not check the target if the conditional for taking the branch does not hold, thus a fallthrough. The second conditional jump is the last one in this run, and therefore it is swapped by a call to the `tramp_je_fall` trampoline; thus a check is performed irrespective of whether it is a fallthrough. The rest of conditional jumps are all singular, so all of them are transformed to a call to the `tramp_je_fall` trampoline.

Our last example, shown in Table 3.11, is from Bash 4.1 and contains the enforcement code for the example policy detailed in Section 2.3.3. Direct calls to `.PLT` section functions do not require guards, but all three conditional jumps are guarded. Our implementation omits guards for conditional jumps that only fall through or always branch by transforming them into direct branches. Furthermore, if text code size is not a concern, guard code can be in-lined to achieve better runtime performance.

3.3 Evaluation

We experimentally evaluated our control-flow trimming system in terms of performance, security, and accuracy. Performance evaluation measures the overhead that our system imposes in terms of space and runtime. Our security analysis examines the system's ability to withstand the threats modeled in §2.2.3. Security failures therefore correspond to false negatives in the classification. Finally, accuracy is measured in terms of false positives—premature aborts of the trimmed program when no policy violation occurred.

Test programs consist of the real-world software products in Table 3.12, plus bash, gcc, ImageMagic, the epiphany and uzbl browsers, and the SPEC2017 benchmarks. We also successfully applied our prototype to rewrite the full GNU Coreutils 8.30 collection. The browsers were chosen primarily for their compatibility with Pin and DynamoRIO, which we use for trace collection and replay.

To evaluate accuracy, we created or obtained test suites for each program. For example, in the gcc evaluations, we used the gcc source code as its own input for unit testing. That test suite therefore consists of all `C` source files needed to compile gcc on the the experiment machine. For ImageMagic, we randomly gathered hundreds of JPEG and PNG images. We unit-tested ftp servers by downloading and uploading randomly selected files interspersed

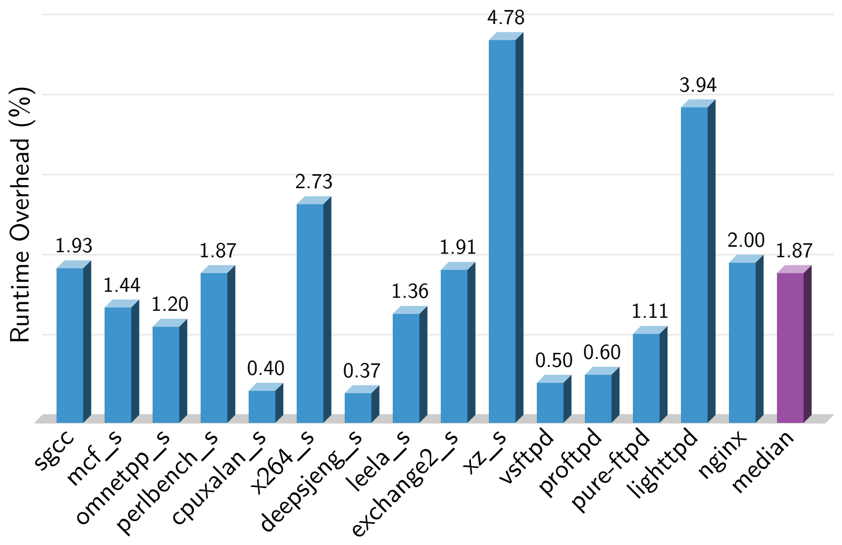


Figure 3.1. Runtime overhead for SPEC2017 intspeed suite and some ftp- and web-servers

with random ftp commands (e.g., cd, mkdir, ls, append, and rename). For exim we used a script to launch sendmail and randomly send an email to a specific address. Browser experiments entail loading pages randomly drawn from the Quantcast top 475K urls, and uzbl experiments additionally include random user interactions (e.g., back/forward navigation, scrolling in all directions, zoom in/out, search, etc.). All results were obtained using a DELL T7500 machine with 24G of RAM and Intel Xeon E5645 processor.

3.3.1 Performance Overhead

Figure 3.1 graphs the runtime overhead for SPEC2017 benchmarks and several ftp- and web-servers. We used Apache benchmark (Apache, 2019) to issue 25,000 requests with concurrency level of 10 for benchmarking lighttpd and nginx. To benchmark the FTP servers, we wrote a Python script based on the pyftpdlib benchmark (Rodola, 2018) to make 100 concurrent clients, each of which request 100 1KB-sized files.

The median runtime overhead is 1.87%, and all benchmarks exhibit an overhead of 0.37–4.78%. The good performance is partially attributable to Table 3.7's reliance on SIMD instructions, which tend to exercise CPU execution units independent of those constrained

Table 3.12. Space overhead for SPEC2017 intspeed suite benchmarks and some real-world applications

	Original Size (KB)		**Size Increase** (%)	
Binary	**File**	**Code**	**File**	**Code**
perlbench_s	10686	1992	10.17	35.14
sgcc	63243	8499	12.76	59.15
mcf_s	131	19	8.80	35.20
omnetpp_s	28159	1567	5.15	55.37
cpuxalan_s	80762	4701	4.19	48.25
x264_s	3320	567	6.41	23.40
deepsjeng_s	508	85	10.23	42.17
leela_s	3819	191	2.15	45.14
exchange2_s	182	111	16.01	18.61
xz_s	1082	146	0.69	2.12
exim	1407	1187	32.14	14.70
lighttpd	1304	294	13.12	27.12
memcached	746	156	13.50	23.89
nginx	1674	1444	29.76	19.07
openssh	2467	638	15.12	21.40
proftpd	3310	803	16.34	29.12
pureftpd	470	118	17.12	27.04
vsftpd	143	133	25.78	28.99
postgresrl	757	544	41.35	33.53
node.js	36758	30059	28.63	17.84
median	1541	556	16.42	28.06

by the mostly general-purpose instructions in the surrounding code. This allows out-of-order execution (OoOE) hardware optimizations in modern processors (Intel, 2019) to parallelize many guard code μops with those of prior and subsequent instructions in the stream.

Table 3.12 shows the space overhead for the SPEC2017 benchmarks and a sampling of the other tested binaries. On average, the test binaries increase in size by 16.42% and their code sizes increase by 28.06%. The main size contributions are the extra control-flow security guard code in-lined into code sections, and the addition of the hash table that encodes the CCFG policy.

Although these size increases are an important consideration for memory and disk resources needed to support our approach, we emphasize that they are not an accurate measure of the resulting software attack surface, since many of the added bytes are non-executable or erased (exception-throwing) opcodes (e.g., `int3`). Attack surface must therefore be measured in terms of reachable code bytes, not raw file or code section size.

To evaluate this, Table 3.13 measures the reachable, executable code from the decision trees for binaries with a test suite. Despite the increase in total file and code sizes, the amount of reachable code is reduced by an average of 36%. For example, the attack surface of ImageMagic convert is reduced by 94.5%. (The method of computing Table 3.13 is detailed in §3.3.3 .)

3.3.2 Security

Vulnerability Removal

A primary motivation for control-flow trimming is the possible removal of defender-unknown vulnerabilities within code features of no interest to code consumers. To test the efficacy of our approach for removing such zero-days, we tested the effects of control-flow trimming on unpatched versions of Bash 4.2, ImageMagic 6.8.6–10, Proftpd 1.3.5, Node.js 8.12, and Exim 4.86 that are vulnerable to the CVEs shown in Table 3.1, including Shellshock and ImageTragick.

Shellshock attacks exploit a bug in the bash command-line parser to execute arbitrary shellcode. The bug erroneously executes text following function definitions in environment variables as code. This affords adversaries who control inputs to environment variables remote code execution capabilities. Because of its severity, prevalence, and the fact that it remained exploitable for over 20 years before it was discovered, Shellshock has been identified as one of the highest impact vulnerabilities in history (Delamore and Ko, 2015).

ImageMagick is used by web services to process images and is also pre-installed in many commonly used Linux distributions such as Ubuntu 18.04. ImageTragick vulnerabilities afford attackers remote code execution; delete, move, and read access to arbitrary files;

and server-side request forgery (SSRF) attack capabilities in ImageMagic versions before 6.9.3–10, and in 7.x before 7.0.1-1.

ProFTPD 1.3.5 allows remote attackers to read and write from/to arbitrary files via `SITE CPFR` and `SITE CPTO` commands. In node serialize package 0.0.4, the `unserialize` function can be exploited by being passed a maliciously crafted JS object to achieve arbitrary code execution. Exim before 4.86.2 allows a local attacker to gain root privilege when Exim is compiled with Perl support and contains a `perl_startup` configuration variable.

Unit tests for the bash experiment consist of the test scripts in the bash source package, which were created and distributed with bash before Shellshock became known. The tests therefore reflect the quality assurance process of users for whom Shellshock is a zero-day. For the remaining programs, we manually exposed each to a variety of inputs representative of common usages. For ImageMagic, our unit tests execute the application's convert utility to convert images to other formats. We unit-tested ProFTPD by exposing it to a variety of commands (e.g., `FEAT`, `HASH`), excluding the `SITE` command. For Node.js we wrote some JS code that does not leverage node-serialize package. We ran Exim without a `perl_startup` configuration variable.

Using these test suites, we applied the procedure described in §2.3 to learn a CCFG policy for these five vulnerable programs, and automatically in-lined an enforcement of that policy approximated as a bit hash table. No source code was used in any of the experiments.

Control-flow trimming these programs with these test suites has the effect of removing all the listed vulnerabilities. For example, Shellshock-exploiting environment variable definitions push bash's control-flow to an obscure portion of the parser logic that is trimmed by the learned CCFG policy, and that the in-lined guard code therefore rejects. Similar policy rejections occur when attempting to trigger the vulnerabilities in the other binaries. This demonstrates that control-flow trimming can effectively remove zero-days if the vulnerability is unique to a semantic feature that remains unexercised by unit testing.

Table 3.13. False positive ratios (%). Zero threshold means no pruning (most conservative) (§2.3.2).

			Context Anomalies			**Origin Anomalies**			**Trace Anomalies**			
Program	**Samples**	**t***	**t=0.00**	**t=0.25**	**t=t***	**t=0.00**	**t=0.25**	**t=t***	**t=0.00**	**t=0.25**	**t=t***	**Reachable Code** (%)
proftpd	10	0.48	3.04	2.37	1.75	4.51	3.95	2.81	45.00	30.00	25.00	47.31
	100	0.37	0.43	0.17	0.05	1.68	1.02	0.37	3.00	1.50	1.00	47.81
	500	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	47.85
vsftpd	10	0.38	2.45	2.16	1.60	3.74	3.23	1.80	35.00	25.00	25.00	51.11
	100	0.23	0.33	0.07	0.14	0.91	0.17	0.22	2.00	1.50	1.50	51.47
	500	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	51.47
pure-ftpd	10	0.41	2.23	1.96	1.43	3.61	3.14	2.83	25.00	25.00	10.00	49.89
	100	0.28	0.04	0.00	0.00	0.15	0.00	0.00	2.50	1.50	1.50	50.03
	500	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	50.05
exim	10	0.25	2.72	1.12	1.88	5.12	4.06	4.81	35.00	15.00	20.00	10.31
	100	0.53	0.58	0.01	0.00	1.36	0.01	0.00	7.50	1.00	0.00	10.63
	200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	10.65
ImageMagic	10	0.64	0.21	0.10	0.04	1.51	1.23	0.91	20.00	15.00	10.00	5.27
convert	100	0.54	0.09	0.07	0.00	0.17	0.10	0.00	2.50	1.00	0.00	5.53
	200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	5.55
gcc	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	7.66
	100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	7.66
	200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	7.66
epiphany	10	0.93	10.91	0.22	0.00	19.60	1.29	0.00	85.00	40.00	0.00	23.41
	100	0.81	10.76	0.20	0.08	15.50	1.14	0.57	40.00	10.00	6.50	23.73
	500	0.33	2.94	0.01	0.01	12.14	0.09	0.08	8.70	0.40	0.30	24.01
	1000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	24.01
uzbl	10	0.92	2.16	0.25	0.12	18.90	1.30	0.81	90.00	40.00	30.00	30.81
	100	0.83	2.09	0.04	0.03	17.36	0.96	0.75	50.50	3.50	2.50	30.83
	500	0.65	0.57	0.01	0.01	9.08	0.34	0.17	10.70	0.90	0.60	30.91
	1000	0.45	0.46	0.03	0.02	7.94	0.52	0.33	4.30	0.85	0.35	30.91

Unit-tested features of the test programs all remain functional after CCFG learning and enforcement. Section 3.3.3 evaluates the accuracy more precisely by measuring false positive rates under a variety of conditions.

Gadget Analysis

Although control-flow trimming is primarily envisioned as a semantic feature removal method and not a gadget removal method, gadget chains are nevertheless one example of a class of unwanted semantic features that control-flow trimming might remove. To study the effect of control-flow trimming on gadget reachability, we used ROPgadget (Salwan, 2018) to find all gadgets in the rewritten test binaries. Since our threat model pessimistically assumes attackers have unrestricted write-access to the stack and heap, our gadget reachability analysis assumes that the attacker can cause any of these gadget addresses to flow as input to any indirect branch or return instruction in the original program. Our defense substitutes all such instructions with guarded-branches and replaces unreachable instructions with `int3`; thus, in order to circumvent (i.e., jump over) the guards to reach a hijackable instruction, the attacker must first supply at least one malicious gadget address that the guards accept, in order to initiate the chain.

To evaluate whether this is possible, for each program we collected all contexts of length $k-1$ observed during training and testing, appended each discovered gadget address to each, and computed the hashes of the resulting length-k contexts. We then examined whether the hash table that approximates the CCFG policy contains a 0 or 1 for each hash value. In all cases, the hash table entry was 0, prompting a policy rejection of the hijacking attempt. We also simulated the attacks discovered by ROPgadget using Pin and verified that the guards indeed block the attacks in practice.

We can also study the theoretical probability of realizing a gadget chain. The probability of finding a gadget address that can pass the guard code to initiate a gadget chain is approximately equal to the ratio p of 1's to 0's in the hash table that encodes the CCFG policy. This can be reduced almost arbitrarily small by increasing the hash table size relative to the code size (see §2.3). For example, in gcc this ratio is as small as 0.004. Only 650KB of the original 8499KB code section is visited by the unit tests and remains reachable after control-flow trimming—an attack surface reduction of 92%.

Moreover, if control-flow trimming is coupled with software fault isolation (SFI) (Wahbe et al., 1993; Yee et al., 2009; McCamant and Morrisett, 2006) to enforce indivisible basic blocks for the guarded-jump trampolines in Table 3.6, then the probability of realizing a length-n gadget chain reduces to p^n. Since SFI is much easier to realize than CFI for source-free binaries (because it enforces a very simple CFG recoverable by binary disassembly), and tends to impose very low runtime overhead, we consider such a pairing to be a promising direction of future work.

3.3.3 Accuracy

Specificity

To measure our approach's accuracy in retaining consumer-desired features while excluding undesired ones, we used the programs in Table 3.13, including several real-world ftp servers, exim, ImageMagic convert, gcc, and two web browsers, since they constitute large, complex pieces of software.

To test gcc, we trained by compiling its own source code to a 64-bit binary, and tested by attempting to compile many C programs to various architectures (32-bit and 64-bit) using the trimmed binary.

For other programs we used the test suites described earlier. In the ImageMagic experiments, the desired functionality is converting a JPG picture to PNG format, and the undesired functionality is resizing a picture. For ftp servers, the undesired functionalities are the `SITE` and `DELETE` commands, and the remaining commands are desired. Ftp file content and command order were randomized during training and evaluation. For exim, the undesired functionality is `-oMs` (which sets the sender host name instead of looking it up). The undesired functionalities for epiphany and uzbl-browser are incognito mode and cookie add/delete, respectively.

Positives in the classification are execution failures during testing, as signaled by premature abort with a security violation warning. *False negatives* are runs that exercise a consumer-undesired semantic feature even after trimming. In contrast, a *false positive* occurs when the defense aborts a consumer-desired functionality.

For all these experiments, *the false negative rate is zero.* That is, no consumer-unwanted functionality is available in any of the test binaries after trimming. For example, after instrumenting `gcc` with training data that uses the `-m64` command-line flag to target 64-bit architectures, the trimmed binary is unable to compile any program for 32-bit architectures; specifying `-m32` on the command-line yields a security abort. This is because our method is a whitelisting approach that prefers overfitting to maintain high assurance.

A classification's susceptibility to false positives can be measured in terms of its *false positive ratio* (i.e., the complement of its *specificity*). The false positive ratio of control-flow trimming is driven by the unit testing's ability to comprehensively model the set of semantic features desired by the consumer. We therefore measure accuracy as the false positive ratio, broken down into three measures: the percentage of contexts incorrectly identified as anomalies, the percentage of branch origins that at least one context anomaly incorrectly detected at that branch site, and the total percentage of traces in which at least one anomaly was incorrectly detected.

Table 3.13 shows the resulting false positive ratios. Each entry in the table is averaged over 10 different experiments in which trace samples are randomly drawn. Since the training phase's accuracy depends significantly on the size of the training data, we conducted experiments with 10–1000 samples for training, evaluation, and testing with a ratio of 3 : 1 : 1.

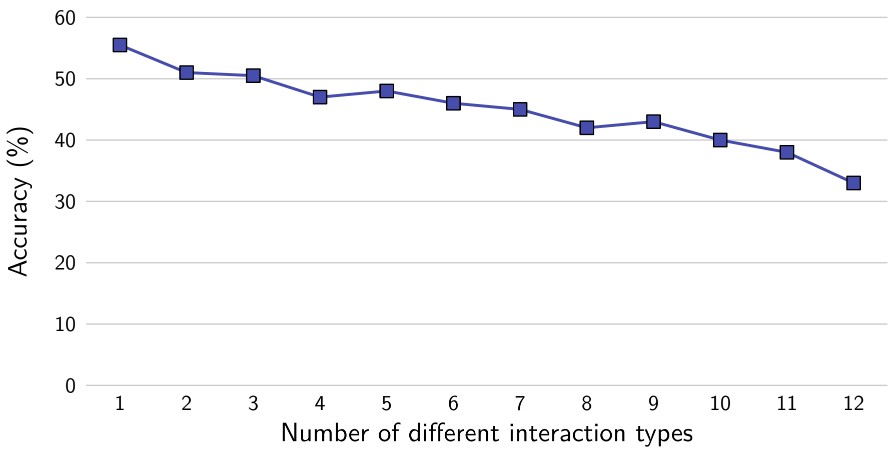


Figure 3.2. Accuracy vs. interaction diversity with uzbl, using a fixed training set size of 100 and $t = 0.0$

The experiments consider the effects of two different confidence thresholds for CCFG pruning (see §2.3.2): 0.0, 0.25, and an optimal threshold t^* experimentally determined as the minimum threshold that achieves zero false negatives for evaluation sample traces. A threshold of 0.0 means no pruning, which is the most conservative CCFI policy (no relaxation). All experiments use contexts of length 4 as described in §3.2.

As expected, increasing the training data size significantly improves classification accuracy, until at higher training sizes, almost all experiments exhibit perfect accuracy. More aggressive CCFG policy pruning via lower confidence thresholds helps to offset the effects of overfitting when limited training data is available. Increasing context size has a reverse effect; the increased discriminatory power of the classifier (due to widening its feature space by a multiplicative factor for each additional context entry) creates a more complex concept for it to learn. More comprehensive training is therefore typically required to learn the concept.

Interactive Experiments

As a whitelisting approach, our design primarily targets software whose desired features are well known to the consumer, and can therefore be fully exercised during training. Table 3.13

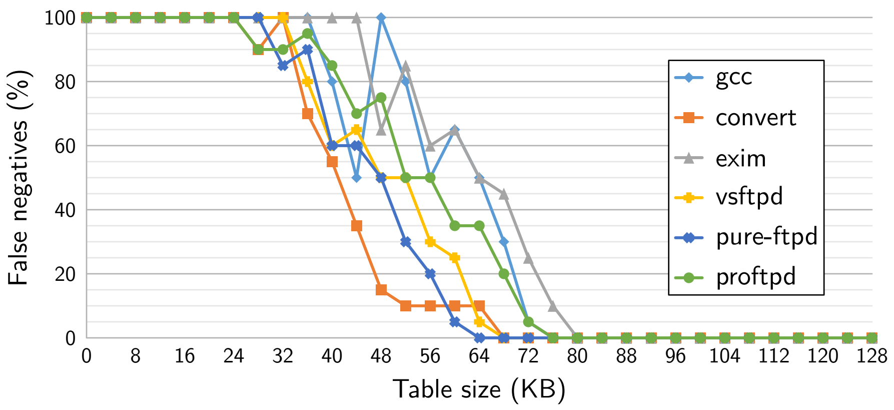


Figure 3.3. False negative ratios with varying table sizes

shows that highly interactive products, such as browsers, might require more training to learn all their features as a result of this design. Experiments on epiphany and uzbl require about 500 traces to obtain high accuracy, with a few rare corner cases for epiphany only discovered after about 1000 traces, and uzbl never quite reaching perfect accuracy.

To better understand the relationship between interaction diversity and training burden for such products, Figure 3.2 plots the accuracy rate for uzbl as the diversity of interactions increases, with the training set size held fixed at 100 traces. Each data point characterizes an experiment in which training and testing are limited to $x \in [1, 12]$ different types of user interactions (e.g., using forward-backward navigation but not page-zoom). The results show an approximately linear decline in accuracy as the diversity of interactions increases, indicating that more training is needed to learn the consumer's more complex policy.

Table Size

For efficiency purposes, our enforcement approximates the policy being enforced as a hash table (see §2.3.4). Poor approximations that use an overly small hash table could permit dangerous false negatives (e.g., undetected attacks), since the enforcement would inadvertently accept policy-violating contexts whose hashes happen to collide with at least one policy-permitted context. To investigate the minimum table sizes needed to avoid these risks, we therefore performed an additional series of experiments wherein we varied the hash table size without changing the policy, and measured the false negative ratio for each size.

Figure 3.3 plots the results for six of the programs with test suites, with hash table size on the x-axis and false negative ratio on the y-axis. The results show that even hash table sizes as small as 128 bytes (1024 bit-entries) reliably achieve a zero false negative rate. This is because policy-accepted contexts are so rare relative to the space of all possible contexts that almost any sequence of contexts that implements an undesired feature quickly witnesses at least one context that is policy-violating, whereupon it is rejected.

Our experiments nevertheless use larger table sizes than this minimum in order to minimize population ratio p, which §3.3.2 shows is important for resisting implementation-aware code-reuse attacks. Specifically, table sizes that scale with the code section size are recommended for security-sensitive scenarios where the threat model anticipates that adversaries have read-access to the table, and might use that knowledge to craft gadget chains.

3.4 Conclusion

Control-flow trimming is the first work to offer an automated, source-free solution for excluding developer-intended but consumer-unwanted functionalities expressible as CCFGs from binary software products with complex input spaces, such as command-lines, files, user interactivity, or data structures. Using only traces that exercise consumer-desired behaviors, the system learns a contextual CFG policy that whitelists desired semantic features, and in-lines an enforcement of that policy in the style of context-sensitive CFI into the target binary. A prototype implementation for Intel x86-64 native code architectures exhibits low runtime overhead (about 1.87%) and high accuracy (zero misclassifications) for training sets as small as 100–500 samples). Experiments on real-world software demonstrate that control-flow trimming can eliminate zero-day vulnerabilities associated with consumer-unwanted features, and resist control-flow hijacking attacks based on code-reuse.

CHAPTER 4

CONFIRM: EVALUATING COMPATIBILITY AND RELEVANCE OF CONTROL-FLOW INTEGRITY PROTECTIONS FOR MODERN SOFTWARE[1]

4.1 Introduction

Control-flow integrity (CFI) (Abadi et al., 2005) (supported by vtable protection (Gawlik and Holz, 2014) and/or software fault isolation (Wahbe et al., 1993)), has emerged as one of the strongest known defenses against modern control-flow hijacking attacks, including return-oriented programming (ROP) (Roemer et al., 2012) and other code-reuse attacks. These attacks trigger dataflow vulnerabilities (e.g., buffer overflows) to manipulate control data (e.g., return addresses) to hijack victim software. By restricting program execution to a set of legitimate control-flow targets at runtime, CFI can mitigate many of these threats. Furthermore, CFI as it was shown in Chapter 2 can be used as a way to reduce software attack surface. Razor (Qian et al., 2019) is another debloating framework that targets binary programs using CFI.

Inspired by the initial CFI work (Abadi et al., 2005), there has been prolific new research on CFI in recent years, mainly aimed at improving performance, enforcing richer policies, obtaining higher assurance of policy-compliance, and protecting against more subtle and sophisticated attacks. For example, between 2015–2018 over 25 new CFI algorithms appeared in the top four applied security conferences alone. These new frameworks are generally evaluated and compared in terms of performance and security. Performance overhead is commonly evaluated in terms of the CPU benchmark suites (e.g., SPEC), and security is often assessed using the RIPE test suite (Wilander et al., 2011) or with manually crafted

[1]This chapter contains material previously published as: Xu, X., M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin (2019). ConFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *Proceeding 28th USENIX Security Symposium*, pp. 1805–1821.

proof-of-concept attacks (e.g., COOP (Schuster et al., 2015)). For example, a recent survey systematically compared various CFI mechanisms against these metrics for precision, security, and performance (Burow et al., 2017).

While this attention to performance and security has stimulated rapid gains in the ability of CFI solutions to efficiently enforce powerful, precise security policies, less attention has been devoted to systematically examining which general classes of software can receive CFI protection without suffering compatibility problems. Historically, CFI research has struggled to bridge the gap between theory and practice (cf., Zhang et al., 2013) because code hardening transformations inevitably run at least some risk of corrupting desired, policy-permitted program functionalities. For example, introspective programs that read their own code bytes at runtime (e.g., many VMs, JIT compilers, hot-patchers, and dynamic linkers) can break after their code bytes have been modified or relocated by CFI.

Compatibility issues of this sort have dangerous security ramifications if they prevent protection of software needed in mission-critical contexts, or if the protections must be weakened in order to achieve compatibility. For example, due in part to potential incompatibilities related to *return address introspection* (wherein some callees read return addresses as arguments) the three most widely deployed compiler-based CFI solutions (LLVM-CFI (Tice et al., 2014), GCC-VTV (Tice et al., 2014), and Microsoft Visual Studio MCFG (Tang, 2015)) all presently leave return addresses unprotected, potentially leaving code vulnerable to ROP attacks—the most prevalent form of code-reuse.

Understanding these compatibility limitations, including their impacts on real-world software performance and security, requires a new suite of CFI functional tests with substantially different characteristics than benchmarks typically used to assess compiler or hardware performance. In particular, CFI relevance and effectiveness is typically constrained by the nature and complexity of the target program's *control-flow paths* and *control data dependencies*. Such complexities are not well represented by SPEC benchmarks, which are designed to exercise CPU computational units using only simple control-flow graphs, or by utility suites (e.g., GNU Coreutils) that were all written in a fairly homogeneous programming style for a limited set of compilers, and that use a very limited set of standard libraries chosen for exceptionally high cross-compatibility.

To better understand the compatibility and applicability limitations of modern CFI solutions on diverse, modern software products, and to identify the coding idioms and features that constitute the greatest barriers to more widespread CFI adoption, we present ConFIRM (CONtrol-Flow Integrity Relevance Metrics), a new suite of CFI tests designed to exhibit code features most relevant to CFI evaluation.[2] Each test is designed to exhibit one or more control-flow features that CFI solutions must guard in order to enforce integrity, that are found in a large number of commodity software products, but that pose potential problems for CFI implementations.

It is infeasible to capture in a single test set the full diversity of modern software, which embodies myriad coding styles, build processes (e.g., languages, compilers, optimizers, obfuscators, etc.), and quality levels. We therefore submit ConFIRM as an extensible baseline for testing CFI compatibility, consisting of code features drawn from experiences building and evaluating CFI and randomization systems for several architectures, including Linux, Windows, Intel x86/x64, and ARM32 in academia and industry (Wartell et al., 2012a,b, 2014; Mohan et al., 2015; Wang et al., 2017; Bauman et al., 2018; Gu et al., 2017; Muntean et al., 2018).

Our work is envisioned as having the following qualitative impacts: (1) CFI designers (e.g., compiler developers) can use ConFIRM to detect compatibility flaws in their designs that are currently hard to anticipate prior to full scale productization. This can lower the currently steep barrier between prototype and distributable product. (2) Defenders (e.g., developers of secure software) can use ConFIRM to better evaluate code-reuse defenses,

[2]https://github.com/SoftwareLanguagesSecurityLab/ConFIRM

in order to avoid false senses of security. (3) The research community can use ConFIRM to identify and prioritize missing protections as important open problems worthy of future investigation.

We used ConFIRM to reevaluate 12 publicly available CFI implementations published in the open literature. The results show that about 47% of solution-test pairs exhibit incompatible or insecure operation for code features needed to support mainstream software products, and a *cross-thread stack-smashing* attack defeats all tested CFI defenses. Microbenchmarking additionally reveals some performance/compatibility trade-offs not revealed by purely CPU-based benchmarking.

In summary, our contributions include the following:

- We present ConFIRM, the first testing suite designed specifically to test compatibility characteristics relevant to control-flow security hardening evaluation.

- A set of 20 code features and coding idioms are identified, that are widely found in deployed, commodity software products, and that pose compatibility, performance, or security challenges for modern CFI solutions.

- Evaluation of 12 CFI implementations using ConFIRM reveals that existing CFI implementations are compatible with only about half of code features and coding idioms needed for broad compatibility, and that microbenchmarking using ConFIRM reveals performance trade-offs not exhibited by SPEC benchmarks.

- Discussion and analysis of these results highlights significant unsolved obstacles to realizing CFI protections for widely deployed, mainstream, commodity products.

Section 4.2 begins with a summary of technical CFI attack and defense details important for understanding the evaluation approach. Section 4.3 next presents ConFIRM's evaluation metrics in detail, including a rationale behind why each metric was chosen, and how it impacts potential defense solutions; and Section 4.4 describes implementation of the resulting tests. Section 4.5 reports our evaluation of CFI solutions using ConFIRM and discusses significant findings. Finally, Section 4.6 concludes.

4.2 Overview

CFI defenses first emerged from an arms race against early code-injection attacks, which exploit memory corruptions to inject and execute malicious code. To thwart these malicious code-injections, hardware and OS developers introduced Data Execution Prevention (DEP), which blocks execution of injected code. Adversaries proceeded to bypass DEP with "return-to-libc" attacks, which redirect control to existing, abusable code fragments (often in the C standard libraries) without introducing attacker-supplied code. In response, defenders introduced Address Space Layout Randomization (ASLR), which randomizes code layout to frustrate its abuse. DEP and ASLR motivated adversaries to craft even more elaborate attacks, including ROP and Jump-Oriented Programming (JOP) (Bletsch et al., 2011), which locate, chain, and execute short instruction sequences (gadgets) of benign code to implement malicious payloads.

CFI emerged as a more comprehensive and principled defense against this malicious code-reuse. Most realizations consist of two main phases: (1) A program-specific *control-flow policy* is first formalized as a (possibly dynamic) control-flow graph (CFG) that whitelists the code's permissible control-flow transfers. (2) To constrain all control flows to the CFG, the program code is instrumented with guard code at all computed (e.g., indirect) control-flow transfer sites. The guard code decides at runtime whether each impending transfer satisfies the policy, and blocks it if not. The guards are designed to be uncircumventable by confronting attackers with a chicken-and-egg problem: To circumvent a guard, an attack must first hijack a control transfer; but since all control transfers are guarded, hijacking a control transfer requires first circumventing a guard.

Both CFI phases can be source-aware (implemented as a source-to-source transformation, or introduced during compilation), or source-free (implemented as a binary-to-binary transformation). Source-aware solutions typically benefit from source-level information to derive more precise policies, and can often perform more optimization to achieve better performance. Examples include WIT (Akritidis et al., 2008), NaCl (Yee et al., 2009), CFL (Bletsch et al., 2011), MIP (Niu and Tan, 2013), MCFI (Niu and Tan, 2014a), RockJIT (Niu and Tan, 2014b), Forward CFI (Tice et al., 2014), CCFI (Mashtizadeh et al., 2015), πCFI (Niu and Tan, 2015), MCFG (Tang, 2015) CFIXX (Burow et al., 2018) and μCFI (Hu et al., 2018). In contrast, source-free solutions are potentially applicable to a wider domain of software products (e.g., closed-source), and have a more flexible deployment model (e.g., consumer-side enforcement without developer assistance). These include XFI (Erlingsson et al., 2006), Reins (Wartell et al., 2012b), STIR (Wartell et al., 2012a), CCFIR (Zhang et al., 2013), bin-CFI (Zhang and Sekar, 2013), BinCC (Wang et al., 2015), Lockdown (Payer et al., 2015), TypeArmor (van der Veen et al., 2016), OCFI (Mohan et al., 2015), OFI (Wang et al., 2017) and τCFI (Muntean et al., 2018).

The advent of CFI is a significant step forward for defenders, but was not the end of the arms race. In particular, each CFI phase introduces potential loopholes for attackers to exploit. First, it is not always clear which policy should be enforced to fully protect the code. Production software often includes complex control-flow structures, such as those introduced by object-oriented programming (OOP) idioms, from which it is difficult (even undecidable) to derive a CFG that precisely captures the policy desired by human developers and users. Second, the instrumentation phase must take care not to introduce guard code whose decision procedures constitute unacceptably slow runtime computations (Hamlen et al., 2006). This often results in an enforcement that imprecisely approximates the policy. Attackers have taken advantage of these loopholes with ever more sophisticated attacks, including Counterfeit Object Oriented Programming (COOP) (Schuster et al., 2015), Control Jujutsu (Evans et al., 2015), and Control-Flow Bending (Carlini et al., 2015).

These weaknesses and threats have inspired an array of new and improved CFI algorithms and supporting technologies in recent years. For example, to address loopholes associated with OOP, *vtable protections* prevent or detect virtual method table corruption at or before control-flow transfers that depend on method pointers. Source-aware vtable protections include GNU VTV (Tice, 2012), CPI (Kuznetsov et al., 2014), SafeDispatch (Jang et al., 2014), Readactor++ (Crane et al., 2015), and VTrust (Zhang et al., 2016); whereas source-free instantiations include T-VIP (Gawlik and Holz, 2014), VTint (Zhang et al., 2015), and VfGuard (Prakash et al., 2015).

However, while the security and performance trade-offs of various CFI solutions have remained actively tracked and studied by defenders throughout the arms race, attackers are increasingly taking advantage of CFI compatibility limitations to exploit unprotected software, thereby avoiding CFI defenses entirely. For example, 88% of CFI defenses cited herein have only been realized for Linux software, but over 95% of desktops worldwide are non-Linux.[3] These include many mission-critical systems, including over 75% of control systems in the U.S. (Konkel, 2017), and storage repositories for top secret military data (Office of Inspector General, 2018). None of the top 10 vulnerabilities exploited by cybercriminals in 2017 target Linux software (Donnelly, 2018).

4.2.1 Prior CFI Evaluations

We surveyed 54 CFI algorithms and implementations published between 2005–2019 to prepare ConFIRM, over half of which were published within 2015–2019. Of these, 66% evaluate performance overheads by applying SPEC CPU benchmarking programs. Examples of such performance evaluations include those of PittSFIeld (McCamant and Morrisett, 2006), NaCl (Yee et al., 2009), CPI (Kuznetsov et al., 2014), Reins (Wartell et al., 2012b), bin-CFI (Zhang and Sekar, 2013), control flow locking (Bletsch et al., 2011), MIP (Niu and

[3]http://gs.statcounter.com/os-market-share/desktop/worldwide

Tan, 2013), CCFIR (Zhang et al., 2013), ROPecker (Cheng et al., 2014), T-VIP (Gawlik and Holz, 2014), GCC-VTV (Tice et al., 2014), MCFI (Niu and Tan, 2014a), VTint (Zhang et al., 2015), Lockdown (Payer et al., 2015), O-CFI (Mohan et al., 2015), CCFI (Mashtizadeh et al., 2015), PathArmor (van der Veen et al., 2015), BinCC (Wang et al., 2015), πCFI (Niu and Tan, 2015), VTI (Bounov et al., 2016), VTrust (Zhang et al., 2016), VTPin (Sarbinowski et al., 2016), TypeArmor (van der Veen et al., 2016), PittyPat (Ding et al., 2017), RA-Guard (Zhang et al., 2017), GRIFFIN (Ge et al., 2017), OFI (Wang et al., 2017), PT-CFI (Gu et al., 2017), HCIC (Zhang et al., 2019), μCFI (Hu et al., 2018), CFIXX (Burow et al., 2018), and τCFI (Muntean et al., 2018).

The remaining 34% of CFI technologies that are not evaluated on SPEC benchmarks primarily concern specialized application scenarios, including JIT compiler hardening (Niu and Tan, 2014b), hypervisor security (Wang and Jiang, 2010; Kwon et al., 2018), iOS mobile code security (Davi et al., 2012; Pewny and Holz, 2013), embedded systems security (Abera et al., 2016; Abbasi et al., 2017; Adepu et al., 2018), and operating system kernel security (Kemerlis et al., 2012; Criswell et al., 2014; Ge et al., 2016). These therefore adopt analogous test suites and tools specific to those domains (Coker, 2016; The Wine Committee, 2020; Postmark, 2013; Pozo and Miller, 2016; de Melo, 2009).

Several of the more recently published works additionally evaluate their solutions on one or more large, real-world applications, including browsers, web servers, FTP servers, and email servers. For example, VTable protections primarily choose browsers as their enforcement targets, and therefore leverage browser benchmarks to evaluate performance. The main browser benchmarks used for this purpose are Microsoft's Lite-Brite (Microsoft, 2013) Google's Octane (Google, 2013), Mozilla's Kraken (Mozilla, 2013), Apple's Sunspider (Apple, 2013), and RightWare's BrowserMark (RightWare, 2019).

Since compatibility problems frequently raise difficult challenges for evaluations of larger software products, these larger-scale evaluations tend to have smaller sample sizes. Overall, 88% of surveyed works report evaluations on 3 or fewer large, independent applications, with TypeArmor (van der Veen et al., 2016) having the most comprehensive evaluation we studied, consisting of three FTP servers, two web servers, an SSH server, an email server, two SQL servers, a JavaScript runtime, and a general-purpose distributed memory caching system.

To demonstrate security, prior CFI mechanisms are typically tested against proof-of-concept attacks or CVE exploits. The most widely tested attack class in recent years is COOP. Examples of security evaluations against COOP attacks include those reported for μCFI (Hu et al., 2018), τCFI (Muntean et al., 2018), CFIXX (Burow et al., 2018), OFI (Wang et al., 2017), PittyPat (Ding et al., 2017), VTrust (Zhang et al., 2016), PathArmor (van der Veen et al., 2015), and πCFI (Niu and Tan, 2015).

The RIPE test suite (Wilander et al., 2011) is also widely used by many researchers to measure CFI security and precision. RIPE consists of 850 buffer overflow attack forms. It aims to provide a standard way to quantify the security coverage of general defense mechanisms. In contrast, ConFIRM focuses on a larger variety of code features that are needed by many applications to implement non-malicious functionalities, but that pose particular problems for CFI defenses. These include a combination of benign behaviors and attacks.

While there is a hope that small-scale prototyping will result in principles and approaches that eventually scale to more architectures and larger software products, follow-on works that attempt to bridge this gap routinely face significant unforeseen roadblocks. We believe many of these obstacles remain unforeseen because of the difficulty of isolating and studying many of the problematic software features lurking within large, commodity products, which are not well represented in open-source codes commonly available for study by researchers during prototyping.

The goal of this research is therefore to describe and analyze a significant collection of code features that are routinely found in large software products, but that pose challenges to effective CFI enforcement; and to make available a suite of CFI test programs that exhibit each of these features on a small scale amenable to prototype development. The next section discusses this feature set in detail.

4.3 Compatibility Metrics

To measure compatibility of CFI mechanisms, we propose a set of metrics that each includes one or more code features from either C/C++ source code or compiled assembly code. We derived this feature set by attempting to apply many CFI solutions to large software products, then manually testing the functionalities of the resulting hardened software for correctness, and finally debugging each broken functionality step-wise at the assembly level to determine what caused the hardened code to fail. Since many failures manifest as subtle forms of register or memory corruption that only cause the program to crash or malfunction long after the failed operation completes, this debugging constitutes many hundreds of person-hours amassed over several years of development experience involving CFI-protected software.

Table 4.1 presents the resulting list of code features organized into one row for each root cause of failure. Column two additionally lists some widely available, commodity software products where each of these features can be observed in non-malicious software in the wild. This demonstrates that each feature is representative of real-world software functionalities that must be preserved by CFI implementations in order for their protections to be usable and relevant in contexts that deploy these and similar products.

4.3.1 Indirect Branches

We first discuss compatibility metrics related to the code feature of greatest relevance to most CFI works: indirect branches. Indirect branches are control-flow transfers whose destination addresses are computed at runtime—via pointer arithmetic and/or memory-reads. Such

Table 4.1. ConFIRM compatibility metrics

Compatibility metric	**Real-world software examples**
Function Pointers	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
Callbacks	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, TeXstudio, Visual Studio, Windows Defender, WinSCP
Dynamic Linking	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
Delay-Loading	Adobe Reader, Calculator, Chrome, Firefox, JVM, MS Paint, MS PowerPoint, PotPlayer, Visual Studio, WinSCP
Exporting/Importing Data	7-Zip, Apache, Calculator, Chrome, Dropbox, Firefox, MS Paint, MS PowerPoint, PowerShell, TeXstudio, UPX, Visual Studio
Virtual Functions	7-Zip, Adobe Reader, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, TeXstudio, Visual Studio, Windows Defender, WinSCP
CODE-COOP Attack	Programs built on GTK+ or Microsoft COM can pass objects to trusted modules as arguments.
Tail Calls	Mainstream compilers provide options for tail call optimization. e.g. `/O2` in MSVC, `-O2` in GCC, and `-O2` in LLVM.
Switch-Case Statements	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, MS Paint, MS PowerPoint, PotPlayer, PuTTY, TeXstudio, Visual Studio, WinSCP
Returns	Every benign program has returns.
Unmatched Call/Return Pairs	Adobe Reader, Apache, Chrome, Firefox, JVM, MS PowerPoint, Visual Studio
Exceptions	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, Visual Studio, Windows Defender, WinSCP
Calling Conventions	Every program adopts one or more calling convention.
Multithreading	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
TLS Callbacks	Adobe Reader, Chrome, Firefox, MS Paint, TeXstudio, UPX
Position-Independent Code	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
Memory Protection	7-Zip, Adobe Reader, Apache, Chrome, Dropbox, Firefox, MS PowerPoint, PotPlayer, TeXstudio, Visual Studio, Windows Defender, WinSCP
JIT Compiler	Adobe Flash, Chrome, Dropbox, Firefox, JVM, MS PowerPoint, PotPlayer, PowerShell, Skype, Visual Studio, WinSCP
Self-Unpacking	Programs decompressed by self-extractors (e.g., UPX, NSIS).
Windows API Hooking	Microsoft Office software, including MS Excel, MS PowerPoint, etc.

Table 4.2. Source code compiled to indirect call

Source code	Assembly code
1 void foo() { return; }	
2 void bar() { return; }	
3 void main() {	
4 void (*fptr)();	1 ...
5 int n = input();	2 call _input
6 if (n)	3 test eax, eax
7 fptr = foo;	4 mov edx, offset_foo
8 else	5 mov ecx, offset_bar
9 fptr = bar;	6 cmovnz ecx, edx
10 fptr();	7 call ecx
11 }	8 ...

transfers tend to be of high interest to attackers, since computed destinations are more prone to manipulation. CFI defenses therefore guard indirect branches to ensure that they target permissible destinations at runtime. Indirect branches are commonly categorized into three classes: indirect calls, indirect jumps, and returns.

Table 4.2 shows a simple example of source code being compiled to an indirect call. The function called at source line 5 depends on user input. This prevents the compiler from generating a direct branch that targets a fixed memory address at compile time. Instead, the compiler generates a register-indirect call (assembly line 7) whose target is computed at runtime. While this is one common example of how indirect branches arise, in practice they are a result of many different programming idioms, discussed below.

Function Pointers. Calls through function pointers typically compile to indirect calls. For example, using gcc with the `-O2` option generates register-indirect calls for function pointers, and MSVC does so by default.

Callbacks. Event-driven programs frequently pass function pointers to external modules or the OS, which the receiving code later dereferences and calls in response to an event. These callback pointers are generally implemented by using function pointers in C, or as method references in C++. Callbacks can pose special problems for CFI, since the call site is not within the module that generated the pointer. If the call site is within a module that cannot easily be modified (e.g., the OS kernel), it must be protected in some other way, such as by sanitizing and securing the pointer before it is passed.

Dynamic Linking. Dynamically linked shared libraries reduce program size and improve locality. But dynamic linking has been a challenge for CFI compatibility because CFG edges that span modules may be unavailable statically.

In Windows, *dynamically linked libraries* (DLLs) can be loaded into memory at load time or runtime. In load-time dynamic linking, a function call from a module to an exported DLL function is usually compiled to a memory-indirect call targeting an address stored in the module's *import address table* (IAT). But if this function is called more than once, the compiler first moves the target address to a register, and then generates register-indirect calls to improve execution performance. In run-time dynamic linking, a module calls APIs, such as `LoadLibrary()`, to load the DLL at runtime. When loaded into memory, the module calls the `GetProcAddress()` API to retrieve the address of the exported function, and then calls the exported function using the function pointer returned by `GetProcAddress()`.

Additionally, MSVC (since version 6.0) provides linker support for delay-loaded DLLs using the `/DELAYLOAD` linker option. These DLLs are not loaded into memory until one of their exported functions is invoked.

In Linux, a module calls functions exported by a shared library by calling a stub in its *procedure linkage table* (PLT). Each stub contains a memory-indirect jump whose target depends on the writable, lazy-bound *global offset table* (GOT). As in Windows, an application can also load a module at runtime using function `dlopen()`, and retrieve an exported symbol using function `dlsym()`.

Supporting dynamic and delay-load linkage is further complicated by the fact that shared libraries can also export data pointers within their export tables in both Linux and Windows. CFI solutions that modify export tables must usually treat code and data pointers differently, and must therefore somehow distinguish the two types to avoid data corruptions.

Virtual Functions. Polymorphism is a key feature of OOP languages, such as C++. Virtual functions are used to support runtime polymorphism, and are implemented by C++ compilers using a form of late binding embodied as *virtual tables* (vtables). The tables are populated by code pointers to virtual function bodies. When an object calls a virtual function, it indexes its vtable by a function-specific constant, and flows control to the memory address read from the table. At the assembly level, this manifests as a memory-indirect call. The ubiquity and complexity of this process has made vtable hijacking a favorite exploit strategy of attackers.

Some CFI and vtable protections address vtable hijacking threats by guarding call sites that read vtables, thereby detecting potential vtable corruption at time-of-use. Others seek to protect vtable integrity directly by guarding writes to them. However, both strategies are potentially susceptible to COOP (Schuster et al., 2015) and CODE-COOP (Wang et al., 2017) attacks, which replace one vtable with another that is legal but is not the one the original code intended to call. The defense problem is further complicated by the fact that many large classes of software (e.g., GTK+ and Microsoft COM) rely upon dynamically generated vtables. CFI solutions that write-protect vtables or whose guards check against a static list of permitted vtables are incompatible with such software.

Tail Calls. Modern C/C++ compilers can optimize tail-calls by replacing them with jumps. Row 8 of Table 4.1 lists relevant compiler options. With these options, callees can return directly to ancestors of their callers in the call graph, rather than to their callers. These mismatched call/return pairs affect precision of some CFG recovery algorithms.

Switch-case Statements. Many C/C++ compilers optimize switch-case statements via a static dispatch table populated with pointers to case-blocks. When the switch is executed, it calculates a dispatch table index, fetches the indexed code pointer, and jumps to the correct case-block. This introduces memory-indirect jumps that refer to code pointers not contained in any vtable, and that do not point to function boundaries. CFI solutions that compare code pointers to a whitelist of function boundaries can therefore cause the switch-case code to malfunction. Solutions that permit unrestricted indirect jumps within each local function risk unsafety, since large functions can contain abusable gadgets.

Returns. Nearly every benign program has returns. Unlike indirect branches whose target addresses are stored in registers or non-writable data sections, return instructions read their destination addresses from the stack. Since stacks are typically writable, this makes return addresses prime targets for malicious corruption.

On Intel-based CISC architectures, return instructions have one of the shortest encodings (1 byte), complicating the efforts of source-free solutions to replace them in-line with secured equivalent instruction sequences. Additionally, many hardware architectures heavily optimize the behavior of returns (e.g., via speculative execution powered by shadow stacks for call/return matching). Source-aware CFI solutions that replace returns with some other instruction sequence can therefore face stiff performance penalties by losing these optimization advantages.

Unmatched call/return Pairs. Control-flow transfer mechanisms, including exceptions and setjmp/longjmp, can yield flows in which the relation between executed call instructions and executed return instructions is not one-to-one. For example, exception-handling implementations often pop stack frames from multiple calls, followed by a single return to the parent of the popped call chain. Shadow stack defenses that are implemented based on traditional call/return matching may be incompatible with such mechanisms.

4.3.2 Other Metrics

While indirect branches tend to be the primary code feature of interest to CFI attacks and defenses, there are many other code features that can also pose control-flow security problems, or that can become inadvertently corrupted by CFI code transformation algorithms, and that therefore pose compatibility limitations. Some important examples are discussed below.

Multithreading. With the rise of multicore hardware, multithreading has become a centerpiece of software efficiency. Unfortunately, concurrent code execution poses some serious safety problems for many CFI algorithms.

For example, in order to take advantage of hardware call-return optimization (see §4.3.1), most CFI algorithms produce code containing guarded return instructions. The guards check the return address before executing the return. However, on parallelized architectures with flat memory spaces, this is unsafe because any thread can potentially write to any other (concurrently executing) thread's return address at any time. This introduces a TOCTOU vulnerability in which an attacker-manipulated thread corrupts a victim thread's return address after the victim thread's guard code has checked it but before the guarded return executes. We term this a cross-thread stack-smashing attack. Since nearly all modern architectures combine concurrency, flat memory spaces, and returns, this leaves almost all CFI solutions either inapplicable, unsafe, or unacceptably inefficient for a large percentage of modern production software.

Position-Independent Code. *Position-independent code* (PIC) is designed to be relocatable after it is statically generated, and is a standard practice in the creation of shared libraries. Unfortunately, the mechanisms that implement PIC often prove brittle to code transformations commonly employed for source-free CFI enforcement. For example, PIC often achieves its position independence by dynamically computing its own virtual memory address (e.g., by performing a call to itself and reading the pushed return address from the stack), and then performing pointer arithmetic to locate other code or data at fixed offsets relative to itself. This procedure assumes that the relative positions of PIC code and data are invariant even if the base address of the PIC block changes.

However, CFI transforms typically violate this assumption by introducing guard code that changes the sizes of code blocks, and therefore their relative positions. To solve this, PIC-compatible CFI solutions must detect the introspection and pointer arithmetic operations that implement PIC and adjust them to compute corrected pointer values. Since there are typically an unlimited number of ways to perform these computations at both the source and native code levels, CFI detection of these computations is inevitably heuristic, allowing some PIC instantiations to malfunction.

Exceptions. Exception raising and handling is a mainstay of modern software design, but introduces control-flow patterns that can be problematic for CFI policy inference and enforcement. Object-oriented languages, such as C++, boast first-class exception machinery, whereas standard C programs typically realize exceptional control-flows with gotos, longjumps, and signals. In Linux, compilers (e.g., gcc) implement C++ exception handling in a table-driven approach. The compiler statically generates read-only tables that hold exception-handling information. For instance, gcc produces a `gcc_except_table` comprised of *language-specific data areas* (LSDAs). Each LSDA contains various exception-related information, including pointers to exception handlers.

In Windows, *structured exception handling* (SEH) extends the standard C language with first-class support for both hardware and software exceptions. SEH uses stack-based exception nodes, wherein exception handlers form a linked list on the stack, and the list head is stored in the *thread information block* (TIB). Whenever an exception occurs, the OS fetches the list head and walks through the SEH list to find a suitable handler for the thrown exception. Without proper protection, these exception handlers on the stack can potentially be overwritten by an attacker. By triggering an exception, the attacker can then redirect the control-flow to arbitrary code. CFI protection against these SEH attacks is complicated by the fact that code outside the vulnerable module (e.g., in the OS and/or system libraries) uses pointer arithmetic to fetch, decode, and call these pointers during exception handling. Thus, suitable protections must typically span multiple modules, and perhaps the OS kernel.

From Windows XP onward, applications have additionally leveraged *vectored exception handling* (VEH). Unlike SEH, VEH is not stack-based; applications register a global handler chain for VEH exceptions with the OS, and these handlers are invoked by the OS by interrupting the application's current execution, no matter where the exception occurs within a frame.

There are at least two features of VEH that are potentially exploitable by attackers. First, to register a vectored exception handler, the application calls an API `AddVecoredExceptionHandler()` that accepts a callback function pointer parameter that points to the handler code. Securing this pointer requires some form of inter-module callback protection.

Second, the VEH handler-chain data structure is stored in the application's writable heap memory, making the handler chain data directly susceptible to data corruption attacks. Windows protects the handlers somewhat by obfuscating them using the `EncodePointer()` API. However, `EncodePointer()` does not implement a cryptographically secure function (since doing so would impose high overhead); it typically returns the XOR of the input pointer with a process-specific secret. This secret is not protected against memory disclosure attacks; it is potentially derivable from disclosure of any encoded pointer with value known to the attacker (since XOR is invertible), and it is stored in the *process environment block* (PEB), which is readable by the process and therefore by an attacker armed with an information disclosure exploit. With this secret, the attacker can overwrite the heap with a properly obfuscated malicious pointer, and thereby take control of the application.

From a compatibility perspective, CFI protections that do not include first-class support for these various exception-handling mechanisms often conservatively block unusual control-flows associated with exceptions. This can break important application functionalities, making the protections unusable for large classes of software that use exceptions.

Calling Conventions. CFI guard code typically instruments call and return sites in the target program. In order to preserve the original program's functionality, this guard code must therefore respect the various calling conventions that might be implemented by calls and returns. Unfortunately, many solutions to this problem make simplifying assumptions about the potential diversity of calling conventions in order to achieve acceptable performance.

For example, a CFI solution whose guard code uses `EDX` as a scratch register might suddenly fail when applied to code whose calling convention passes arguments in `EDX`. Adapting the solution to save and restore `EDX` to support the new calling convention can lead to tens of additional instructions per call, including additional memory accesses, and therefore much higher overhead.

The C standard calling convention (`cdecl`) is caller-pop, pushes arguments right-to-left onto the stack, and returns primitive values in an architecture-specific register (`EAX` on Intel). Each architecture also specifies a set of caller-save and callee-save registers. Caller-popped calling conventions are important for implementing variadic functions, since callees can remain unaware of argument list lengths.

Callee-popped conventions include `stdcall`, which is the standard convention of the Win32 API, and `fastcall`, which passes the first two arguments via registers rather than the stack to improve execution speed. In OOP languages, every nonstatic member function has a hidden *this pointer* argument that points to the current object. The `thiscall` convention passes the *this pointer* in a register (`ECX` on Intel).

Calling conventions on 64-bit architectures implement several refinements of the 32-bit conventions. Linux and Windows pass up to 14 and 4 parameters, respectively, in registers rather than on the stack. To allow callees to optionally spill these parameters, the caller additionally reserves a *red zone* (Linux) or 32-byte *shadow space* (Windows) for callee temporary storage.

Highly optimized programs also occasionally adopt non-standard, undocumented calling conventions, or even blur function boundaries entirely (e.g., by performing various forms of function in-lining). For example, some C compilers support language extensions (e.g., MSVC's `naked` declaration) that yield binary functions with no prologue or epilogue code, and therefore no standard calling convention. Such code can have subtle dependencies on non-register processor elements, such as requiring that certain Intel status flags be preserved across calls. Many CFI solutions break such code by in-lining call site guards that violate these undocumented conventions.

TLS Callbacks. Multithreaded programs require efficient means to manipulate thread-local data without expensive locking. Using *thread local storage* (TLS), applications export one or more TLS callback functions that are invoked by the OS for thread initialization or termination. These functions form a null-terminated table whose base is stored in the PE header. For compiler-based CFI solutions, the TLS callback functions do not usually need extra protection, since both the PE header and the TLS callback table are in unwritable memory. But source-free solutions must ensure that TLS callbacks constitute policy-permitted control-flows at runtime.

Memory Protection. Modern OSes provide APIs for memory page allocation (e.g., `VirtualAlloc` and `mmap`) and permission changes (e.g., `VirtualProtect` and `mprotect`). However, memory pages changed from writable to executable, or to simultaneously writable and executable, can potentially be abused by attackers to bypass DEP defenses and execute attacker-injected code. Many software applications nevertheless rely upon these APIs for legitimate purposes (see Table 4.1), so conservatively disallowing access to them introduces many compatibility problems. Relevant CFI mechanisms must therefore carefully enforce memory access policies that permit virtual memory management but block code-injection attacks.

Runtime Code Generation. Most CFI algorithms achieve acceptable overheads by performing code generation strictly statically. The statically generated code includes fixed runtime guards that perform small, optimized computations to validate dynamic control-flows. However, this strategy breaks down when target programs generate new code dynamically and attempt to execute it, since the generated code might not include CFI guards. *Runtime code generation* (RCG) is therefore conservatively disallowed by most CFI solutions, with the expectation that RCG is only common in a few, specialized application domains, which can receive specialized protections.

Unfortunately, our analysis of commodity software products indicates that RCG is becoming more prevalent than is commonly recognized. In general, we encountered RCG compatibility limitations in at least three main forms across a variety of COTS products:

1. Although typically associated with web browsers, *just-in-time* (JIT) compilation has become increasingly relevant as an optimization strategy for many languages, including Python, Java, the Microsoft .NET family of languages (e.g., C#), and Ruby. Software containing any component or module written in any JIT-compiled language frequently cannot be protected with CFI.

2. Mobile code is increasingly space-optimized for quick transport across networks. *Self-unpacking executables* are therefore a widespread source of RCG. At runtime, self-unpacking executables first decompress archived data sections to code, and then map the code into writable and executable memory. This entails a dynamic creation of fresh code bytes. Large, component-driven programs sometimes store rarely used components as self-unpacking code that decompresses into memory whenever needed, and is deallocated after use. For example, NSIS installers pack separate modules supporting different install configurations, and unpack them at runtime as-needed for reduced size. Antivirus defenses hence struggle to distinguish benign NSIS installers from malicious ones (Crofford and McKee, 2017).

3. Component-driven software also often performs a variety of obscure *API hooking* initializations during component loading and clean-up, which are implemented using RCG. As an example, Microsoft Office software dynamically redirects all calls to certain system API functions within its address space to dynamically generated wrapper functions. This allows it to modify the behaviors of late-loaded components without having to recompile them all each time the main application is updated.

To hook a function f within an imported system DLL (e.g., `ntdll.dll`), it first allocates a fresh memory page f' and sets it both writable and executable. It next copies the first five code bytes from f to f', and writes an instruction at $f' + 5$ that jumps to $f + 5$. Finally, it changes f to be writable and executable, and overwrites the first five code bytes of f with an instruction that jumps to f'. All subsequent calls to f are thereby redirected to f', where new functionality can later be added dynamically before f' jumps to the preserved portion of f.

Such hooking introduces many dangers that are difficult for CFI protections to secure without breaking the application or its components. Memory pages that are simultaneously writable and executable are susceptible to code-injection attacks, as described previously. The RCG that implements the hooks includes unprotected jumps, which must be secured by CFI guard code. However, the guard code itself must be designed to be rewritable by more hooking, including placing instruction boundaries at addresses expected by the hooking code ($f + 5$ in the above example). No known CFI algorithm can presently handle these complexities.

4.3.3 Compositional Defense Evaluation

Some CFI solutions compose CFI controls with other defense layers, such as randomization-based defenses (e.g., Cowan et al., 1998; Bhatkar et al., 2003; Berger and Zorn, 2006; Novark and Berger, 2010; Mohan et al., 2015; Wartell et al., 2012a). Randomization defenses can be susceptible to other forms of attack, such as memory disclosure attacks (e.g., Strackx et al., 2009; Snow et al., 2013; Evans et al., 2015; Shacham et al., 2004). ConFIRM does not test such attacks, since their implementations are usually specific to each defense and not easy to generalize.

Evaluation of composed defenses should therefore be conducted by composing other attacks with ConFIRM tests. For example, to test a CFI defense composed with stack canaries, one should first simulate attacks that attempt to steal the canary secret, and then modify any stack-smashing ConFIRM tests to use the stolen secret. Incompatibilities of the evaluated defense generally consist of the union of the incompatibilities of the composed defenses.

4.4 Implementation

To facilitate easier evaluation of the compatibility considerations outlined in Section 4.3 along with their impact on security and performance, we developed the ConFIRM suite of CFI tests. ConFIRM consists of 24 programs written in C++ totalling about $2,300$ lines of code. Each test isolates one of the compatibility metrics of Section 4.3 (or in some cases a few closely related metrics) by emulating behaviors of COTS software products. Source-aware solutions can be evaluated by applying CFI code transforms to the source codes, whereas source-free solutions can be applied to native code after compilation with a compatible compiler (e.g., gcc, LLVM, or MSVC). Loop iteration counts are configurable, allowing some tests to be used as microbenchmarks. The tests are described as follows:

fptr. This tests whether function calls through function pointers are suitably guarded or can be hijacked. Overhead is measured by calling a function through a function pointer in an intensive loop.

callback. As discussed in Section 4.3, call sites of callback functions can be either guarded by a CFI mechanism directly, or located in immutable kernel modules that require some form of indirect control-flow protections. We therefore test whether a CFI mechanism can secure callback function calls in both cases. Overhead is measured by calling a function that takes a callback pointer parameter in an intensive loop.

load_time_dynlnk. Load-time dynamic linking tests determine whether function calls to symbols that are exported by a dynamically linked library are suitably protected. Overhead is measured by calling a function that is exported by a dynamically linked library in an intensive loop.

run_time_dynlnk. This tests whether a CFI mechanism supports runtime dynamic linking, whether it supports retrieving symbols from the dynamically linked library at runtime, and whether it guards function calls to the retrieved symbol. Overhead is measured by loading a dynamically linked library at runtime, calling a function exported by the library, and unloading the library in an intensive loop.

delay_load *(Windows only)***.** CFI compatibility with delay-loaded DLLs is tested, including whether function calls to symbols that are exported by the delay-loaded DLLs are protected. Overhead is measured by calling a function that is exported by a delay-loaded DLL in an intensive loop.

data_symbl. Data and function symbol imports and exports are tested, to determine whether any controls preserve their accessibility and operation.

vtbl_call. Virtual function calls are exercised, whose call sites can be directly instrumented. Overhead is measured by calling virtual functions in an intensive loop.

code_coop. This tests whether a CFI mechanism is robust against CODE-COOP attacks. For the object-oriented interfaces required to launch a CODE-COOP attack, we choose Microsoft COM API functions in Windows, and gtkmm API calls that are part of the C++ interface for GTK+ in Linux.

tail_call. Tail call optimizations of indirect jumps are tested. Overhead is measured by tail-calling a function in a loop.

switch. Indirect jumps associated with switch-case control-flow structures are tested, including their supporting data structures. Overhead is measured by executing a switch-case statement in an intensive loop.

ret. Validation of return addresses (e.g., dynamically via shadow stack implementation, or statically by labeling call sites and callees with equivalence classes) is tested. Overhead is measured by calling a function that does nothing but return in an intensive loop.

unmatched_pair. Unmatched call/return pairs resulting from exceptions, setjmp, and longjmp are tested.

signal. This test uses signal-handling in C to implement error-handling and exceptional control-flows.

cppeh. C++ exception handling structures and control-flows are exercised.

seh *(Windows only)***.** SEH-style exception handling is tested for both hardware and software exceptions. This test also checks whether the CFI mechanism protects the exception handlers stored on the stack.

veh *(Windows only)***.** VEH-style exception handling is tested for both hardware and software exceptions. This test also checks whether the CFI mechanism protects callback function pointers passed to `AddVecoredExceptionHandler()`.

convention. Several different calling conventions are tested, including conventions widely used in C/C++ languages on 32-bit and 64-bit x86 processors.

multithreading. Safety of concurrent thread executions is tested. Specifically, one thread simulates a memory corruption exploit that attempts to smash another thread's stack and break out of the CFI-enforced sandbox.

tls_callback *(Windows source-free only)***.** This tests whether static TLS callback table corruption is detected and blocked by the protection mechanism.

pic. Semantic preservation of position-independent code is tested.

mem. This test performs memory management API calls for legitimate and malicious purposes, and tests whether security controls permit the former but block the latter.

jit. This test generates JIT code by first allocating writable memory pages, writing JIT code into those pages, making the pages executable, and then running the JIT code. To emulate behaviors of real-world JIT compilers, the JIT code performs different types of control-flow transfers, including calling back to the code of JIT compiler and calling functions located in other modules.

api_hook *(Windows only)***.** Dynamic API hooking is performed in the style described in Section 4.3.

unpacking *(source-free only)***.** Self-unpacking executable code is implemented using RCG.

4.5 Evaluation

4.5.1 Evaluation of CFI Solutions

To examine ConFIRM's effect on real CFI defenses, we used it to reevaluate 12 major CFI implementations for Linux and Windows that are either publicly available or were obtainable in a self-contained, operational form from their authors at the time of writing. Our purpose in performing this evaluation is not to judge which compatibility features solutions should be expected to support, but merely to accurately document which features are currently supported and to what degree, and to demonstrate that ConFIRM can be used to conduct such evaluations.

Table 4.3 reports the evaluation results. Columns 2–6 report results for Windows CFI approaches, and columns 7–14 report those for Linux CFI. All Windows experiments were performed on an Intel Xeon E5645 workstation with 24 GB of RAM running 64-bit Windows 10. Linux experiments were conducted on different versions of Ubuntu VM machines corresponding to the version tested by each CFI framework's original developers. All the VM machines had 16GB of RAM with 6 Intel Xeon CPU cores. The overheads for source-free approaches were evaluated using test binaries compiled with most recent version of gcc

Table 4.3. Tested results for CFI solutions on ConFIRM

	LLVM (Windows)						LLVM (Linux)						
Test	**CFI**	**ShadowStack**	**MCFG**	**OFI**	**Reins**	**GCC-VTV**	**CFI**	**ShadowStack**	**MCFI**	**πCFI**	**πCFI (nto)**	**PathArmor**	**Lockdown**
fptr	6.35%	⚠	20.13%	4.35%	4.08%	⚠	6.97%	⚠	✗	−14.00%	−13.79%	⚠	174.92%
callback	⚠	⚠	⚠	128.39%	114.84%	⚠	⚠	⚠	✗	✗	✗	⚠	✗
load_time_dynlnk	2.74%	⚠	8.83%	3.36%	2.66%	⚠	1.33%	⚠	30.83%	31.52%	34.05%	74.54%	1.45%
run_time_dynlnk	⚠	⚠	17.63%	12.57%	11.48%	⚠	4.44%	⚠	✗	✗	✗	1,221.48%	✗
delay_load$^{\boxplus}$	N/A	N/A	8.16%	3.61%	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
data_symbl	✓	⚠	✓	✓	✗	✓	✓	⚠	✓	✓	✓	✓	✓
vtbl_call	5.62%	⚠	27.71%	35.94%	31.17%	33.56%	5.94%	⚠	✗	−8.19%	−9.31%	⚠	227.82%
code_coop	⚠	⚠	⚠	✓	✗	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠
tail_call	6.17%	⚠	9.51%	0.05%	0.05%	⚠	6.82%	⚠	✗	−17.69%	−17.37%	⚠	178.06%
switch	−5.80%	⚠	3.51%	22.82%	17.69%	⚠	−6.93%	⚠	−29.01%	−27.19%	−28.46%	⚠	85.85%
ret	⚠	18.04%	⚠	49.34%	48.49%	⚠	⚠	20.88%	70.72%	72.40%	71.52%	⚠	106.71%
unmatched_pair	⚠	⚠	⚠	✓	✓	⚠	⚠	⚠	✓	✓	✓	⚠	⚠
signal	✓	⚠	✓	✗	✗	✓	✓	⚠	✓	✓	✓	✗	✓
cppeh	✓	⚠	✓	✓	✗	✓	✓	⚠	✓	✓	✓	✗	✓
seh$^{\boxplus}$	✓	⚠	✓	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
veh$^{\boxplus}$	⚠	⚠	⚠	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
convention	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
multithreading	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠
tls_callback$^{\boxplus,\$}$	N/A	N/A	N/A	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
pic	✓	✓	✓	⚠	⚠	✓	✓	✓	✓	✓	✓	✓	✓
mem	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	✗	✗	✗	✓	✗
jit	⚠	⚠	⚠	✗	✗	⚠	⚠	⚠	✗	✗	✗	⚠	✗
unpacking$^{\$}$	N/A	N/A	N/A	✗	✗	N/A	N/A	N/A	N/A	N/A	N/A	✗	✗
api_hook$^{\boxplus}$	⚠	⚠	⚠	✗	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

(nto) stands for *no tail-call optimization*
%: CFI defense passes compatibility and security test, and microbenchmark yields indicated performance overhead
✓: same as %, but this test provides no performance number
⚠: CFI defense passes compatibility but not security check
✗: test does not compile (compilation error), or crashes at runtime
N/A: test is not applicable to the CFI mechanism being tested
$\boxplus$: test is Windows-only
$: test is only for source-free defenses

available for each test platform. All source-aware approaches were applied before or during compilation with the most recent version of LLVM for each test platform (since LLVM provides greatest compatibility between the tested source-aware solutions).

Two forms of compatibility are assessed in the evaluation: A CFI solution is categorized as *permissively compatible* with a test if it produces an output program that does not crash and exhibits the original test program's non-malicious functionality. It is *effectively compatible* if it is permissively compatible and any malicious functionalities are blocked. Effective compatibility therefore indicates secure and transparent support for the code features exhibited by the test.

In Table 4.3, Columns 2–3 begin with an evaluation of LLVM CFI and LLVM ShadowCallStack on Windows. With both CFI and ShadowCallStack enabled, LLVM on Windows enforces policies that constrain impending control-flow transfers at every call site, except calls to functions that are exported by runtime-loaded DLLs. Additionally, LLVM on Windows does not secure callback pointers passed to external modules not compiled with LLVM, leaving it vulnerable CODE-COOP attacks. Although ShadowCallStack protects against return address overwrites, its shadow stack is incompatible with unmatched call/return pairs.

Column 4 of Table 4.3 reports evaluation of Microsoft's MCFG, which is integrated into the MSVC compiler. MCFG provides security checks for function pointer calls, vtable calls, tail calls, and switch-case statements. It also passes all tests related to dynamic linking, including load_time_dynlnk, run_time_dynlnk, delay_load, and data_symbl. As a part of MSVC, MCFG provides transparency for generating position-independent code and handling various calling conventions. With respect to exception handling, MCFG is permissively compatible with all relevant features, but does not protect vectored exception handlers. MCFG's most significant shortcoming is its weak protection of return addresses. In addition, it generates call site guard code at compile-time only. Therefore, code that links to immutable modules or modules compiled with a different protection scheme remains potentially insecure. This results in failures against callback corruption and CODE-COOP attacks.

Table 4.4. Overall compatibility of CFI solutions

Tests	**LLVM (Windows)**[*]	**MCFG**	**OFI**	**Reins**	**GCC-VTV**	**LLVM (Linux)**[*]	**MCFI**	**πCFI**	**πCFI (nto)**	**Path-Armor**	**Lock-down**
applicable	21	22	24	24	18	18	18	18	18	19	19
permissively compatible	21	22	20	12	18	18	11	14	14	16	14
effectively compatible	12	13	17	9	6	12	9	12	12	6	11
Permissive compatibility	100.00%	100.00%	83.33%	50.00%	100.00%	100.00%	61.11%	77.78%	77.78%	84.21%	73.68%
Effective compatibility	57.14%	59.09%	70.83%	37.50%	33.33%	66.67%	50.00%	66.67%	66.67%	31.58%	57.89%

[*]Compatibility of LLVM is measured with both CFI and ShadowCallStack enabled.

Columns 5–6 of Table 4.3 report compatibility testing results for Reins and OFI, which are source-free solutions for Windows. Reins validates control-flow transfer targets for function pointer calls, vtable calls, tail calls, switch-case statements, and returns. It supports dynamic linking at load time and runtime, and is one of the only solutions we tested that secures callback functions whose call sites cannot be directly instrumented (with a high overhead of 114.84%). Like MCFG, Reins fails against CODE-COOP attacks. However, OFI extends Reins with additional protections that succeed against CODE-COOP. OFI also exhibits improved compatibility with delay-loaded DLLs, data exports, all three styles of exception handling, all tested calling conventions, and TLS callbacks. Both Reins and OFI nevertheless proved vulnerable against attacks that abuse position-independent code and memory management API functions.

The GNU C-compiler does not yet have built-in CFI support, but includes *virtual table verification* (VTV). VTV is first introduced in gcc 4.9.0. It checks at virtual call sites whether the vtable pointer is valid based on the object type. This blocks many important OOP vtable corruption attacks, although type-aware COOP attacks can still succeed by calling a different virtual function of the same type (e.g., supertype). As shown in column 7 of Table 4.3, VTV does not protect other types of control-flow transfers, including function pointers, callbacks, dynamic linking for both load-time and run-time, tail calls, switch-case jumps, return addresses, error handling control-flows, or JIT code. However, it is permissively compatible with all the applicable tests, and can compile any feature functionality we considered.

As reported in Columns 8–9, LLVM on Linux shows similar evaluation results as LLVM on Windows. It has better effective compatibility by providing proper security checks for calls to functions that are exported by runtime loaded DLLs. LLVM on Linux overheads range from -6.93% (for switch control structures) to 20.88% (for protecting returns).

MCFI and πCFI are source-aware control-flow techniques. We tested them on x64 Ubuntu 14.04.5 with LLVM 3.5. The results are shown in columns 10–12 of Table 4.3. ΠCFI comes with an option to turn off tail call optimization, which increases the precision at the price of a small overhead increase. We therefore tested both configurations, observing no compatibility differences between πCFI with and without tail call optimizations. Incompatibilities were observed in both MCFI and πCFI related to callbacks and runtime dynamic linking. MCFI additionally suffered incompatibilities with the function pointer and virtual table call tests. For callbacks, both solutions incorrectly terminate the process reporting a CFI violation. In terms of effective compatibility, MCFI and πCFI both securely support dynamic linking, switch jumps, return addresses, and unmatched call/return pairs, but are susceptible to CODE-COOP attacks. In our performance analysis, we did not measure any considerable overheads for πCFI's tail call option (only 0.3%). This option decreases the performance for dynamic linking but increases the performance of vtable calls, switch-case, and return tests. Overall, πCFI scores more compatible and more secure relative to MCFI, but with slightly higher performance overhead.

PathArmor offers improved power and precision over the other tested solutions in the form of contextual CFI policy support. Contextual CFI protects dangerous system API calls by tracking and consulting the control-flow history that precedes each call. Efficient context-checking is implemented as an OS kernel module that consults the last branch record (LBR) CPU registers (which are only readable at ring 0) to check the last 16 branches before the impending protected branch. As reported in column 13, our evaluation demonstrated high permissive compatibility, only observing crashes on tests for C++ exception handling and signal handlers. However, our tests were able to violate CFI policies using function pointers, callbacks, virtual table pointers, tail-calls, switch-cases, return addresses, and unmatched call/return pairs, resulting in a lower effective compatibility score. Its careful guarding of system calls also comes with high overhead for those calls (1221.48%). This affects feasibility of dynamic loading, whose associated system calls all receive a high performance penalty per call. Similarly, load-time dynamic linking exhibits a relatively high 74.54% overhead.

Lockdown enforces a dynamic control-flow integrity policy for binaries with the help of symbol tables of shared libraries and executables. Although Lockdown is a binary approach, it requires symbol tables not available for stripped binaries without sources, so we evaluated it using test programs specially compiled with symbol information added. Its loader leverages the additional symbol information to more precisely sandbox interactions between interoperating binary modules. Lockdown is permissively compatible with most tests except callbacks and runtime dynamic linking, for which it crashes. In terms of security, it robustly secures function pointers, virtual calls, switch tables, and return addresses. These security advantages incur somewhat higher performance overheads of 85.85–227.82% (but with only 1.45% load-time dynamic loading overhead). Like most of the other tested solutions, Lockdown remains vulnerable to CODE-COOP and multithreading attacks. Additionally, Lockdown implements a shadow stack to protect return addresses, and thus is incompatible with unmatched call/return pairs.

4.5.2 Evaluation Trends

ConFIRM evaluation of these CFI solutions reveals some notable gaps in the current state-of-the-art. For example, all tested solutions fail to protect software from our cross-thread stack-smashing attack, in which one thread corrupts another thread's return address. We hypothesize that no CFI solution yet evaluated in the literature can block this attack except by eliminating all return instructions from hardened programs, which probably incurs prohibitive overheads. By repeatedly exploiting a data corruption vulnerability in a loop, our test program can reliably break all tested CFI defenses within seconds using this approach.

Since concurrency, flat memory spaces, returns, and writable stacks are all ubiquitous in almost all mainstream architectures, such attacks should be considered a significant open problem. Intel Control-flow Enforcement Technology (CET) (Intel, 2017) has been proposed as a potential hardware-based solution to this; but since it is not yet available for testing, it

Table 4.5. Correlation between SPEC CPU and ConFIRM performance

SPEC CPU Benchmark	CFI Solution									Benchmark Correlation
	MCFG	**Reins**	**GCC-VTV**	**LLVM-CFI**	**MCFI**	**πCFI**	**πCFI** (nto)	**PathArmor**	**Lockdown**	
perlbench				2.4	5.0	5.0	5.3	15.0	150.0	**0.09**
bzip2	−0.3	9.2		−0.7	1.0	1.0	0.8	0.0	8.0	**−0.12**
gcc					4.5	4.5	10.5	9.0	50.0	**0.02**
mcf	0.5	9.1		3.6	4.5	4.5	1.8	1.0	2.0	**−0.39**
gobmk	−0.2			0.2	7.0	7.5	11.8	0.0	43.0	**−0.09**
hmmer	0.7			0.1	0.0	0.0	−0.1	1.0	3.0	**0.33**
sjeng	3.4			1.6	5.0	5.0	11.9	0.0	80.0	**−0.03**
h264ref	5.4			5.3	6.0	6.0	8.3	1.0	43.0	**−0.09**
libquantum				−6.9	0.0	−0.3	−1.0	3.0	5.0	**0.51**
omnetpp	3.8		5.8		5.0	5.0	18.8			**−0.52**
astar	0.1		3.6	0.9	3.5	4.0	2.9		17.0	**0.92**
xalancbmk	5.5		24.0	7.2	7.0	7.0	17.6		118.0	**0.94**
milc	2.0			0.2	2.0	2.0	1.4	4.0	8.0	**0.40**
namd	0.1		−0.1	0.1	−0.5	−0.5	−0.5	3.0		**0.98**
dealII	−0.1		0.7	7.9	4.5	4.5	4.4			**−0.36**
soplex	2.3		0.5	−0.3	−4.0	−4.0	0.9	12.0		**0.89**
povray	10.8		−0.6	8.9	10.0	10.5	17.4		90.0	**0.88**
lbm	4.2			−0.2	1.0	1.0	−0.5	0.0	2.0	**−0.22**
sphinx3	−0.1			−0.8	1.5	1.5	2.4	3.0	8.0	**0.31**
ConFIRM median	9.51	4.59	33.56	5.19	30.83	−11.10	−11.60	648.01	140.82	**0.36**

is unclear whether its hardware shadow stack will be compatible with software idioms that exhibit unmatched call-return pairs.

Memory management abuse is another major root of CFI incompatibilities and insecurities uncovered by our experiments. Real-world programs need access to the system memory management API in order to function properly, making CFI approaches that prohibit it impractical. However, memory API arguments are high value targets for attackers, since they potentially unlock a plethora of follow-on attack stages, including code injections. CFI solutions that fail to guard these APIs are therefore insecure. Of the tested solutions, only PathArmor manages to strike an acceptable balance between these two extremes, but only at the cost of high overheads.

A third outstanding open challenge concerns RCG in the form of JIT-compiled code, dynamic code unpacking, and runtime API hooking. RockJIT (Niu and Tan, 2014b) is the only language-based CFI algorithm proposed in the literature that yet supports any form of RCG, and its approach entails compiler-specific modifications to source code, making it difficult to apply on large scales to the many diverse forms of RCG that appear in the wild. New, more general approaches are needed to lend CFI support to the increasing array of software products built atop JIT-compiled languages or linked using RCG-based mechanisms—including many of the top applications targeted by cybercriminals (e.g., Microsoft Office).

Table 4.4 measures the overall compatibility of all the tested CFI solutions. Permissive and effective compatibility are measured as the ratio of applicable tests to permissively and effectively compatible ones, respectively. All CFI techniques embedded in compilers (*viz.* LLVM on Linux and Windows, MCFG, and GCC-VTV), are 100% permissively compatible, avoiding all crashes. LLVM on Linux, LLVM on Windows, and MCFG secure at least 57% of applicable tests, while GCC-VTV only secures 33%.

OFI scores high overall compatibility, achieving 83% permissive compatibility and 71% effective compatibility on 24 applicable tests. Reins has the lowest permissive compatibility score of only 50%. PathArmor and Lockdown are permissively compatible with 84% and 74% of 19 applicable tests. However PathArmor can only secure 32% of the tests, giving it the lowest effective compatibility score.

4.5.3 Performance Evaluation Correlation

Prior performance evaluations of CFI solutions primarily rely upon SPEC CPU benchmarks as a standard of comparison. This is based on a widely held expectation that CFI overheads on SPEC benchmarks are indicative of their overheads on real-world, security-sensitive software to which they might be applied in practical deployments. However no prior work has attempted to quantify a correlation between SPEC benchmark scores and overheads observed for the addition of CFI controls to large, production software products. If, for example, CFI introduces high overheads for code features not well represented in SPEC benchmarks (e.g., because they are not performance bottlenecks for CFI-free software and were therefore not prioritized by SPEC), but that become real-world bottlenecks once their overheads are inflated by CFI controls, then SPEC benchmarks might not be good predictors of real-world CFI overheads. Recent work has argued that prior CFI research has unjustifiably drawn conclusions about real-world software overheads from microbenchmarking results (van der Kouwe et al., 2019), making this an important open question.

To better understand the relationship between CFI-specific operation overheads and SPEC benchmark scores, we therefore computed the correlation between median performance of CFI solutions on ConFIRM benchmarks with their performances reported on SPEC benchmarks (as reported in the prior literature). Although ConFIRM benchmarks are not real-world software, they can serve as microbenchmarks of features particularly relevant to CFI. High correlations therefore indicate to what degree SPEC benchmarks exercise code features whose performance are affected by CFI controls.

Table 4.5 reports the results, in which correlations between each SPEC CPU benchmark and ConFIRM median values are computed as Pearson correlation coefficients:

$$\rho_{x,y} = \frac{(\sum_{i=1}^{n} x_i \times y_i) - (n \times \bar{x} \times \bar{y})}{(n-1) \times \sigma_x \times \sigma_y} \tag{4.1}$$

where x_i and y_i are the CPU SPEC overhead and ConFIRM median overhead scores for solution i, $\bar{x}$ and $\bar{y}$ are the means, and σ_x and σ_y are the sample standard deviations of x and y, respectively. High linear correlations are indicated by $|\rho|$ values near to 1, and direct and inverse relationships are indicated by positive and negative ρ, respectively.

The results show that although a few SPEC benchmarks have strong correlations (namd, xalancbmk, astar, soplex, and povray being the highest), in general SPEC CPU benchmarks exhibit a poor correlation of only 0.36 on average with tests that exercise CFI-relevant code features. Almost half the SPEC benchmarks even have negative correlations. This indicates that SPEC benchmarks consist largely of code features unrelated to CFI overheads. While this does not resolve the question of whether SPEC overheads are predictive of real-world overheads for CFI, it reinforces the need for additional research connecting CFI overheads on SPEC benchmarks to those on large, production software.

4.6 Conclusion

ConFIRM is the first evaluation methodology and microbenchmarking suite that is designed to measure applicability, compatibility, and performance characteristics relevant to control-flow security hardening evaluation. The ConFIRM suite provides 24 tests of various CFI-relevant code features and coding idioms, which are widely found in deployed COTS software products.

Twelve publicly available CFI mechanisms are reevaluated using ConFIRM. The evaluation results reveal that state-of-the-art CFI solutions are compatible with only about 53% of the CFI-relevant code features and coding idioms needed to protect large, production software systems that are frequently targeted by cybercriminals. Compatibility and security limitations related to multithreading, custom memory management, and various forms of runtime code generation are identified as presenting some of the greatest barriers to adoption.

In addition, using ConFIRM for microbenchmarking reveals performance characteristics not captured by metrics widely used to evaluate CFI overheads. In particular, SPEC CPU benchmarks designed to assess CPU computational overhead exhibit an only 0.36 correlation with benchmarks that exercise code features relevant to CFI. This suggests a need for more CFI-specific benchmarking to identify important sources of performance bottlenecks, and their ramifications for CFI security and practicality.

CHAPTER 5
RELATED WORK

5.1 Code Surface Reduction

Software debloating has been used in the past to reduce code sizes for performance and security. Such techniques were initially applied to Linux kernels to save memory on embedded systems (Lee et al., 2003; Chanet et al., 2005; He et al., 2007). Later the focus shifted to reducing the kernel's attack surface to improve security (Kurmus et al., 2011; Tartler et al., 2012; Kurmus et al., 2013, 2014; Gu et al., 2014). Prior work has shown that certain Linux kernel deployments leave 90% of kernel functions unused (Kurmus et al., 2014). kRazor learns the set of used functions based on runtime traces, and limits the code reachability using a kernel module. Face-Change (Gu et al., 2014) makes multiple minimized kernels in a VM and exposes each minimized kernel to a particular application upon context-switching. In contrast to these works, our approach is not kernel-specific, can enforce context-sensitive control-flow policies, and can debloat code at instruction-level granularity.

Code surface reduction has recently started to be applied to user-level libraries and programs. Winnowing (Malecha et al., 2015) is a source-aware static analysis and code specialization technique that uses partial evaluation to preserve developer-intended semantics of programs. It implements Occam, which performs both intra-module and inter-module winnowing atop LLVM, and produces specific version of the program based on the deployment setup. Piecewise Debloating (Quach et al., 2018) uses piece-wise compilation to maintain intra-modular dependencies, and a piece-wise loader that generates an inter-modular dependency graph. The loader removes all code that is not in the dependency graph. Chisel (Heo et al., 2018) debloats the program given a high-level specification from the user. The specification identifies wanted and unwanted program input/output pairs, and requires the source code and the compilation toolchain. To accelerate program reduction, Chisel uses reinforcement learning. It repeats a trial and error approach to make a more precise Markov

Decision Process that corresponds to the specification. Razor (Qian et al., 2019) is the only debloating framework that targets binary programs other than this dissertation. Same as proposed in this dissertation, Razor exerts runtime traces, and removes code using heuristics, however, it cannot remove functionalities that have contextual dependencies.

Source-free, binary code reduction has been achieved for certain closed-source Windows applications by removing unimported functions in shared libraries at load time (Mulliner and Neugschwandtner, 2015). The approach requires *image freezing*, which prevents any new code section or executable memory page from being added. Shredder (Mishra and Polychronakis, 2018) is another source-free approach that specializes the API interface available to the application. It combines inter-procedural backwards data flow analysis and lightweight symbolic execution to learn a policy for each function in the program. Although these approaches boast source-freedom, they can only permit or exclude program behaviors at the granularity of functions with well-defined interfaces. Many critical security vulnerabilities, including Shellshock, cannot be isolated to individual functions, so cannot be pruned in this way without removing desired program behaviors. Our approach therefore learns and enforces policies definable as arbitrary CCFGs irrespective of function boundaries or even the feasibility of recovering function abstractions from the binary.

5.2 Control-flow Integrity

SFI (Wahbe et al., 1993) and CFI (Abadi et al., 2005) confine software to a whitelist of permitted control-flow edges by guarding control-transfer instructions with dynamic checks that validate their destinations. In SFI, the policy is typically a sandboxing property that isolates the software to a subset of the address space, whereas CFI approaches typically enforce stronger properties that restrict each module's internal flows. In both cases the policy is designed to prohibit flows unintended or unwanted by software developers (e.g., developer-unwanted component interactions or control-flow hijacks). Since the original works, the research community have proposed many variations (e.g., Erlingsson et al., 2006; Akritidis et al., 2008; Yee et al., 2009; Davi et al., 2012; Zhang and Sekar, 2013; Niu and Tan, 2013; Zhang et al., 2013; Tice et al., 2014; Niu and Tan, 2014b,a; Davi et al., 2015; Mohan et al., 2015; Wang et al., 2015; Mashtizadeh et al., 2015; van der Veen et al., 2015; Niu and Tan, 2015; Payer et al., 2015; van der Veen et al., 2016), most of which improve security, performance, compatibility, and/or applicability to various code domains and architectures.

CFI algorithms come in context-sensitive and context-insensitive varieties. Context-sensitivity elevates the power of the policy language using contextual information, such as return address history or type information, usually in a protected shadow stack. The price of such power is usually lower performance due to maintaining, consulting, and securing the contexts. Low overhead solutions must usually relax policies, introducing a sacrifice of assurance.

For example, kBouncer (Pappas et al., 2013) enforces a context-sensitive policy that considers the previous 16 jump destinations at each system call. Unfortunately, enforcing the policy only at system calls makes the defense susceptible to history-flushing attacks (Carlini and Wagner, 2014), wherein attackers make 16 benign redundant jumps followed by a system call. ROPecker (Cheng et al., 2014) and PathArmor (van der Veen et al., 2015) implements OS kernel modules that consult last branch record (LBR) CPU registers to achieve lower performance, which are only available at ring 0. Both systems implement sparse checking regimens to save overhead, in which not every branch is checked. CCFI (Mashtizadeh et al., 2015) uses message authentication codes (MACs) to protect important pointers, such as return addresses, function pointers, and vtable pointers, to enforce call-return matching policies.

CFI methodologies can also be partitioned into source-aware and source-agnostic approaches. Source-aware approaches are typically more powerful and more efficient, because they leverage source code information to infer more precise policies and optimize code. However, they are inapplicable to consumers who receive closed-source software in strictly binary form, and who wish to enforce consumer-specific policies. They likewise raise difficulties for software products that link to closed-source library modules. These difficulties have motivated source-agnostic approaches.

WIT (Akritidis et al., 2008), MIP (Niu and Tan, 2013), MCFI (Niu and Tan, 2014a), Forward CFI (Tice et al., 2014), RockJIT (Niu and Tan, 2014b), CCFI (Mashtizadeh et al., 2015), π-CFI (Niu and Tan, 2015), VTrust (Zhang et al., 2016), VTable Interleaving (Bounov et al., 2016), PittyPat (Ding et al., 2017), CFIXX (Burow et al., 2018), and μCFI(Hu et al., 2018) are examples of source-aware CFI. XFI (Erlingsson et al., 2006), Native Client (Yee et al., 2009), MoCFI (Davi et al., 2012), CCFIR (Zhang et al., 2013), bin-CFI (Zhang and Sekar, 2013), O-CFI (Mohan et al., 2015), BinCC (Wang et al., 2015), Lockdown (Payer et al., 2015), PathArmor (van der Veen et al., 2015), TypeArmor (van der Veen et al., 2016), C-FLAT (Abera et al., 2016), OFI (Wang et al., 2017), and τCFI (Muntean et al., 2018) are all examples of source-free approaches.

Our research addresses the problem of consumer-side software feature trimming and customization, which calls for a combination of source-agnosticism and context-sensitivity. Binary control-flow trimming is therefore the first work to target this difficult combination for fine-grained CCFG learning and enforcement. Table emphasizes the difference between this problem and the problems targeted by prior works. For example, PathArmor enforces contextual CFG policies, but maintains a much sparser context that is only checked at system API calls. This suffices to block exploitation of developer-unintended features, but not abusable developer-intended functionalities.

5.3 Partial Evaluation

Partial evaluation (Jones et al., 1993) is a program analysis and transformation that specializes code designed to accommodate many inputs to instead accommodate only a specific subset of possible inputs. This can have the effect of shrinking and optimizing the code, at the expense of deriving code of less generality. Although partial evaluation has traditionally only been applied to source code programs, recent work has applied it to de-bloat native codes without sources. WiPEr (Srinivasan and Reps, 2015a; Driscoll and Johnson, 2016) lifts Intel IA-32 native code to CodeSurfer/x86 intermediate form (Balakrishnan et al., 2005), converts it to a quantifier-free bit-vector logic amenable to specialization, and then synthesizes specialized native code using McSynth (Srinivasan and Reps, 2015b). While the approach is promising, it is currently only applicable to relatively small binary programs with clearly demarcated inputs, such as integers. Larger inputs, such as string command-lines or user-interactive behaviors, prevent the slicing algorithm from effectively extracting and eliminating concept-irrelevant portions of the code automatically.

5.4 Abnormal Behavior Detection

Our approach to learning CCFG policies from traces is a form of anomaly-based intrusion detection, which also has security applications for malware detection and software behavior prediction.

Malware Detection and Code Reuse

Static and dynamic analyses are both used in modern malware detection. Static analysis can be based on source code or binaries, and does not use any runtime information. For example, Apposcopy (Feng et al., 2014) uses static taint analysis and inter-component call graphs to match applications with malware signatures specified in a high level language that describes semantic characteristics of malware. Static code analysis for malware detection has been proved to be undecidable in general, as witnessed by opaque constants (Moser et al., 2007), which can obfuscate register-load operations from static analyses. As a result, most of the recent works in this genre use dynamic or hybrid static-dynamic analyses (e.g., Kolbitsch et al., 2009; Anderson et al., 2011; Park et al., 2013). As an example of dynamic analysis, Crowdroid (Burguera et al., 2011) uses system calls, information flow tracking, and network monitoring to detect malware and trojans as they are being executed. TaintDroid (Enck et al., 2010) is another Android application that constantly monitors the system and detects leaks of user-sensitive information using dynamic taint analysis.

Software Behavior Prediction

Prior works have leveraged machine learning to classify program traces. Markov models trained on execution traces can learn a classifier of program behaviors (Bowring et al., 2004). Random forests are another effective technique (Haran et al., 2005). Software behavioral anomalies have also be identified via intra-component CFGs constructed from templates mined from execution traces (Nandi et al., 2016). Recent work has also applied clustering of input/output pairs and their amalgamations for this purpose (Almaghairbe and Roper, 2017). Our approach adopts a decision tree forest model because of its efficient implementation as in-lined native code (see) and its amenability to relaxation and specialization at control-flow transfer points (see).

CHAPTER 6

CONCLUSION

This dissertation presents a new method to automatically remove potentially exploitable, abusable, or unwanted code features from binary software. Furthermore, it introduces a new practice to evaluate CFI works in terms of compatibility as well as performance. First, we discuss some technical observations and aspects of control-flow trimming, followed by a section with a high-level summary of the dissertation's main contributions and discoveries.

6.1 Discussion and Future Work

6.1.1 Control-flow Obfuscation

Although our evaluation presently only targets non-obfuscated binary code, we conjecture that control-flow trimming via CCFG enforcement has potentially promising applications for hardening obfuscated binaries as well. Instruction-level diversification (Cohen, 1993), opaque predicates (Majumdar and Thomborson, 2005), and control-flow flattening (Wang et al., 2000) are some examples of code obfuscation and anti-piracy techniques that are commonly applied by code-producers to prevent effective binary reverse-engineering.

For example, flattening adds a dispatcher to the program through which all control-flow transfers are rerouted. This makes it more difficult for adversaries to reverse-engineer the control-flows, but it also prevents context-insensitive CFI protections from securing them, since the flattening transforms individual CFG edges into *chains* of edges that must be permitted or rejected. Context-sensitivity is needed to reject the chain without rejecting the individual edges in the chain. The context-sensitivity of our approach therefore makes it well-suited to such obfuscations.

6.1.2 Shared Libraries

Our experiments report results for CCFG policies enforced on user-level applications and their dedicated libraries, but not on system shared libraries. Securing system shared libraries can be accomplished similarly, but if the library continues to be shared, its policy must permit all the semantic features of all the applications that import it. This can introduce unavoidable false negatives for the individual applications that share it. We therefore recommend that consumers who prioritize security should avoid shared versions of the system libraries in security-critical applications, so that control-flow trimming can specialize even the system library code to the application's specific usage requirements.

6.1.3 Concurrency, Non-determinism, and Non-control Data Attacks

Our IRM implementation stores contextual information in thread-local machine registers for safety and efficiency. This immunizes it against context pollution due to concurrency. However, it also means that it cannot block attacks that have no effect upon any thread's control-flow, such as non-control data attacks in which one thread corrupts another thread's data without affecting its own control-flows or those of the victim thread. Such attacks are beyond the scope of all CFI-based defenses (Abadi et al., 2009).

6.2 Dissertation Summary

While much of the prior literature on binary software hardening has focused on elimination of *artifacts*—software functionalities unanticipated by the code's original developers (e.g., bugs)—Chapters 2 and 3 propose an approach for safely and automatically removing features intended by developers but undesired by individual consumers. This is crucial in the context of security-sensitive organizations which tend to use commercial-off-the-shelf (COTS) binaries as they come with better support and reliability. A prototype implementation of our solution indicates that the approach is feasible without requiring any formal specification from consumer or source-code from developers.

The availability of a consumer-side test suite that exercises desired software features is one of the primary prerequisites for our solution. Such a test suite is usually available from consumer side in order to assess whether the software meets their requirements and is compatible with their existing infrastructure. This test suite is crucial and can highly affect the performance of our approach. For example, binary trimming accuracy suffers from a small test suite that cannot be expanded easily and does not exercise some of the consumer-desired functionalities. Such difficulties can arise in highly interactive software, such as GUI-based applications, where the test suite suffers significant incompleteness. Future research should therefore investigate more sophisticated machine learning algorithms and models in order to infer more accurate policies for such programs in the presence of testing incompleteness.

Experimental evaluations for Intel x86-64 architecture demonstrate that the overhead of this approach is not significant, and that it can be applied to real-world software on large scales. Our prototype take advantage of SSE registers whenever they are free and available, and thereby minimizes memory access overheads. Applying this approach to other programs that extensively use SSE registers, or providing a solution for other architectures that support only small register pools, may considerably affect the performance overhead, and is left for future research. It was shown that binary control-flow trimming can remove unknown vulnerabilities, and pushes the bar higher for an attacker to create a malicious payload by disallowing gadget chaining using contextual CFI, thus significantly reducing attack surface of the program.

Chapter 4 presents ConFIRM, the first evaluation methodology and microbenchmarking suite that is designed to measure applicability, compatibility, and performance characteristics relevant to control-flow security hardening evaluation. ConFIRM comprises of 24 CFI-relevant code features and coding idioms. Experimental evaluations shows that state-of-the-art CFI solutions are compatible with only about 53% of these tests which are widely found in deployed COTS software products and production software systems that are frequently targeted by cybercriminals. Compatibility and security limitations related to multithreading, custom memory management, and various forms of runtime code generation are identified as presenting some of the greatest barriers to adoption. Future research should prioritize filling these gaps between CFI theory and practice in order to compatibly support and secure larger, more complex software products.

REFERENCES

Abadi, M., M. Budiu, Ú. Erlingsson, and J. Ligatti (2005). Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pp. 340–353.

Abadi, M., M. Budiu, Ú. Erlingsson, and J. Ligatti (2009). Control-flow integrity principles, implementations, and applications. *ACM Transaction on Information and System Security (TISSEC) 13*(1).

Abbasi, A., T. Holz, E. Zambon, and S. Etalle (2017). ECFI: Asynchronous control flow integrity for programmable logic controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, pp. 437–448.

Abera, T., N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik (2016). C-FLAT: Control-flow attestation for embedded systems software. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pp. 743–754.

Adepu, S., F. Brasser, L. Garcia, M. Rodler, L. Davi, A.-R. Sadeghi, and S. Zonouz (2018). Control behavior integrity for distributed cyber-physical systems. *CoRR abs/1812.08310.*

Akritidis, P., C. Cadar, C. Raiciu, M. Costa, and M. Castro (2008). Preventing memory error exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security & Privacy (S&P)*, pp. 263–277.

Al-Qudsi, M. (2017). Will AMD's Ryzen finally bring SHA extensions to Intel's CPUs? *NeoSmart Technologies*.

Almaghairbe, R. and M. Roper (2017). Separating passing and failing test executions by clustering anomalies. *Software Quality Journal 25*(3), 803–840.

Anderson, B., D. Quist, J. Neil, C. Storlie, and T. Lane (2011). Graph-based malware detection using dynamic analysis. *Journal in Computer Virology 7*(4), 247–258.

Apache (2019). Apache benchmark. http://httpd.apache.org/docs/current/programs/ab.html.

Apple (2013). Sunspider 1.0 JavaScript benchmark suite. https://webkit.org/perf/sunspider/sunspider.html.

Balakrishnan, G., R. Gruian, T. Reps, and T. Teitelbaum (2005). CodeSurfer/x86. In *Proceedings of the 14th International Conference on Compiler Construction (CC)*, pp. 250–254.

Bauman, E., Z. Lin, and K. W. Hamlen (2018). Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium (NDSS)*.

Berger, E. D. and B. G. Zorn (2006). DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 158–168.

Bhatkar, S., D. C. DuVarney, and R. Sekar (2003). Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*.

Bittau, A., A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh (2014). Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security & Privacy (S&P)*, pp. 227–242.

Bletsch, T., X. Jiang, and V. Freeh (2011). Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, pp. 353–362.

Bletsch, T., X. Jiang, V. W. Freeh, and Z. Liang (2011). Jump-oriented programming: A new class of code-reuse attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pp. 30–40.

Bounov, D., R. G. Kici, and S. Lerner (2016). Protecting C++ dynamic dispatch through VTable interleaving. In *Proceedings of the 23rd Annual Network & Distributed System Security Symposium (NDSS)*.

Bowring, J. F., J. M. Rehg, and M. J. Harrold (2004). Active learning for automatic classification of software behavior. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 195–205.

Brewster, T. (2016). Everything we know about nso group: The professional spies who hacked iphones with a single text.

Bruening, D. L. (2004). *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph. D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

Burguera, I., U. Zurutuza, and S. Nadjm-Tehrani (2011). Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, pp. 15–26.

Burow, N., S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz (2017). Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys 50*(1), 16:1–16:33.

Burow, N., D. McKee, S. A. Carr, and M. Payer (2018). CFIXX: Object type integrity for C++. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium (NDSS).*

Carlini, N., A. Barresi, M. Payer, D. Wagner, and T. R. Gross (2015). Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, pp. 161–176.

Carlini, N. and D. Wagner (2014). ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 385–399.

Chanet, D., B. D. Sutter, B. D. Bus, L. V. Put, and K. D. Bosschere (2005). System-wide compaction and specialization of the Linux kernel. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 95–104.

Cheng, Y., Z. Zhou, Y. Miao, X. Ding, and H. R. Deng (2014). ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21st Annual Network & Distributed System Security Symposium (NDSS).*

Cohen, F. B. (1993). Operating system protection through program evolution. *Computer Security 12*(6), 565–584.

Coker, R. (2016). Disk performance benchmark tool – Bonnie. https://www.coker.com.au/bonnie++.

Cowan, C., C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang (1998). StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pp. 63–77.

Crane, S. J., S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz (2015). It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications and Security (CCS)*, pp. 243–255.

Criswell, J., N. Dautenhahn, and V. Adve (2014). KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 292–307.

Crofford, C. and D. McKee (2017, March). Ransomeware families use NSIS installers to avoid detection, analysis. *McAfee Labs*.

Davi, L., R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi (2012). MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS).*

Davi, L., M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin (2015). HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*.

Davi, L., C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose (2015). Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS)*.

de Melo, A. C. (2009). Performance counters on Linux. In *Linux Plumbers Conference*.

Delamore, B. and R. K. Ko (2015). A global, empirical analysis of the Shellshock vulnerability in web applications. In *Proceedings of the 1st IEEE International Workshop on Trustworthy Software Systems (TrustSoft)*.

Ding, R., C. Qian, C. Song, W. Harris, T. Kim, and W. Lee (2017). Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Security Symposium*.

Donnelly, S. (2018). Soft target: The top 10 vulnerabilities used by cybercriminals. Technical Report CTA-2018-0327, Recorded Future.

Driscoll, E. and T. Johnson (2016). Lean and efficient software: Whole-program optimization of executables. Technical report, GrammaTech.

Enck, W., P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth (2010). TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 393–407.

Erlingsson, Ú., M. Abadi, M. Vrable, M. Budiu, and G. C. Necula (2006). XFI: Software guards for system address spaces. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 75–88.

Evans, I., S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglu-Douskos, M. Rinard, and H. Okhravi (2015). Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P)*, pp. 781–796.

Evans, I., F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos (2015). Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pp. 901–913.

Feng, Y., S. Anand, I. Dillig, and A. Aiken (2014). Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pp. 576–587.

Gawlik, R. and T. Holz (2014). Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, pp. 396–405.

Gawlik, R., B. Kollenda, P. Koppe, B. Garmany, and T. Holz (2016). Enabling client-side crash-resistance to overcome diversification and information hiding. In *Proceedings of the 23rd Annual Network & Distributed System Security Symposium (NDSS)*.

Ge, X., W. Cui, and T. Jaeger (2017). 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)*, pp. 585–598.

Ge, X., N. Talele, M. Payer, and T. Jaeger (2016). Fine-grained control-flow integrity for kernel software. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 179–194.

Ghaffarinia, M. and K. W. Hamlen (2019). Binary control-flow trimming. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1009–1022.

Göktaş, E., E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 417–432.

Google (2013). Octane JavaScript benchmark suite. https://developers.google.com/octane.

Gu, Y., Q. Zhao, Y. Zhang, and Z. Lin (2017). PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 173–184.

Gu, Z., B. Saltaformaggio, X. Zhang, and D. Xu (2014). FACE-CHANGE: Application-driven dynamic kernel view switching in a virtual machine. pp. 491–502.

Hamlen, K. W., G. Morrisett, and F. B. Schneider (2006). Computability classes for enforcement mechanisms. *ACM Transaction on Programming Languages and Systems (TOPLAS) 28*(1), 175–205.

Haran, M., A. Karr, A. Orso, A. Porter, and A. Sanil (2005). Applying classification techniques to remotely-collected program execution data. In *Proceedings of the 10th European Software Engineering Conference (ESEC)*, pp. 146–155.

He, H., S. K. Debray, and G. R. Andrews (2007). The revenge of the overlay: Automatic compaction of OS kernel code via on-demand code loading. In *Proceedings of the 7th ACM/IEEE International Conference on Embedded Software (EMSOFT)*, pp. 75–83.

Heo, K., W. Lee, P. Pashakhanloo, and M. Naik (2018). Effective program debloating via reinforcement learning. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 380–394.

Homescu, A., S. Neisius, P. Larsen, S. Brunthaler, and M. Franz (2013). Profile-guided automated software diversity. In *Proceedings of the 11th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–11.

Homescu, A., M. Stewart, P. Larsen, S. Brunthaler, and M. Franz (2012). Microgadgets: Size does matter in Turing-complete return-oriented programming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, pp. 64–76.

Hu, H., C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee (2018). Enforcing unique code target property for control-flow integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, pp. 1470–1486.

Intel (2017, June). Control-flow enforcement technology preview, revision 2.0. Technical Report 334525-002, Intel Corporation.

Intel (2019, April). *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Chapter 2.6.3: Intel Microarchitecture Code Name Nehalem: Execution Engine. Intel Corporation.

Jang, D., Z. Tatlock, and S. Lerner (2014). SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS).*

Jones, N. D., C. K. Gomard, and P. Sestoft (1993). *Partial Evaluation and Automatic Program Generation.* Prentice Hall International.

Kemerlis, V. P., G. Portokalidis, and A. D. Keromytis (2012). kGuard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium*, pp. 459–474.

Kolbitsch, C., P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang (2009). Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium*, pp. 351–366.

Konkel, F. (2017). The Pentagon's bug bounty program should be expanded to bases, DOD official says. *Defense One*.

Krebs on Security (2019). What we can learn from the capital one hack. http://www.krebsonsecurity.com.

Kurmus, A., S. Dechand, and R. Kapitza (2014). Quantifiable run-time kernel attack surface reduction. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 212–234.

Kurmus, A., A. Sorniotti, and R. Kapitza (2011). Attack surface reduction for commodity OS kernels: Trimmed garden plants may attract less bugs. In *Proceedings of the 4th European Workshop on System Security (EUROSEC)*.

Kurmus, A., R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, and D. Lohmann (2013). Attack surface metrics and automated compile-time OS kernel tailoring. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*.

Kuznetsov, V., L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song (2014). Code-pointer integrity. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 147–163.

Kwon, D., J. Seo, S. Baek, G. Kim, S. Ahn, and Y. Paek (2018). VM-CFI: Control-flow integrity for virtual machine kernel using Intel PT. In *Proceedings of the 18th International Conference on Computational Science and Its Applications (ICCSA)*, pp. 127–137.

Larsen, P., A. Homescu, S. Brunthaler, and M. Franz (2014). SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security & Privacy (S&P)*, pp. 276–291.

Lee, C.-T., Z.-W. Rong, and J.-M. Lin (2003). Linux kernel customization for embedded systems by using call graph approach. In *Proceedings of the 6th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 689–692.

Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 26th ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 190–200.

Majumdar, A. and C. Thomborson (2005). Securing mobile agents control flow using opaque predicates. In *Proceedings of the 9th International Conference on Knowledge-based Intelligent Information and Engineering Systems (KES)*, pp. 1065–1071.

Malecha, G., A. Gehani, and N. Shankar (2015). Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*, pp. 1504–1511.

Mashtizadeh, A. J., A. Bittau, D. Boneh, and D. Mazières (2015). CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pp. 941–951.

McCamant, S. and G. Morrisett (2006). Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*.

Microsoft (2013). Lite-Brite Benchmark. https://testdrive-archive.azurewebsites.net/Performance/LiteBrite.

Mishra, S. and M. Polychronakis (2018). Shredder: Breaking exploits through API specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pp. 1–16.

Mohan, V., P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz (2015). Opaque control-flow integrity. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS)*.

Moser, A., C. Kruegel, and E. Kirda (2007). Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, pp. 421–430.

Mozilla (2013). Kraken 1.1 JavaScript benchmark suite. http://krakenbenchmark.mozilla.org.

Mulliner, C. and M. Neugschwandtner (2015). Breaking payloads with runtime code stripping and image freezing. Black Hat USA.

Muntean, P., M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert (2018). τCFI: Type-assisted control flow integrity for x86-64 binaries. In *Proceedings of the 21st Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pp. 423–444.

Nandi, A., A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya (2016). Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 215–224.

Niu, B. and G. Tan (2013). Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pp. 199–210.

Niu, B. and G. Tan (2014a). Modular control-flow integrity. In *Proceedings of the 35th ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 577–587.

Niu, B. and G. Tan (2014b). RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pp. 1317–1328.

Niu, B. and G. Tan (2015). Per-input control-flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pp. 914–926.

Novark, G. and E. D. Berger (2010). DieHarder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computing and Communications Security (CCS).*

Office of Inspector General (2018). Evaluation of DHS' information security program for FY 2017. Technical Report OIG-18-56, Department of Homeland Security (DHS).

Pappas, V., M. Polychronakis, and A. D. Keromytis (2013). Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, pp. 447–462.

Park, Y., D. S. Reeves, and M. Stamp (2013). Deriving common malware behavior through graph clustering. *Computers & Security 39*, 419–430.

Payer, M., A. Barresi, and T. R. Gross (2015). Fine-grained control-flow integrity through binary hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 144–164.

Pewny, J. and T. Holz (2013). Control-flow restrictor: Compiler-based CFI for iOS. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, pp. 309–318.

Postmark (2013). Email delivery for web apps. https://postmarkapp.com.

Pozo, R. and B. Miller (2016). SciMark 2. http://math.nist.gov/scimark2.

Prakash, A., X. Hu, and H. Yin (2015). vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS).*

Qian, C., H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee (2019). RAZOR: A framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Security Symposium*, pp. 1733–1750.

Quach, A., A. Prakash, and L.-K. Yan (2018). Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*, pp. 869–886.

Raymond, E. S. (2003). *The Art of Unix Programming*, pp. 313. Addison-Wesley.

RightWare (2019). Basemark web 3.0. https://web.basemark.com.

Rodola, G. (2018). pyftpdlib. https://github.com/giampaolo/pyftpdlib.

Roemer, R., E. Buchanan, H. Shacham, and S. Savage (2012). Return-oriented programming: Systems, languages, and applications. *ACM Transaction on Information and System Security (TISSEC) 15*(1).

Salwan, J. (2018). ROPgadget tool. https://github.com/JonathanSalwan/ROPgadget. Retrieved 5/6/2018.

Sarbinowski, P., V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos (2016). VTPin: Practical vtable hijacking protection for binaries. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, pp. 448–459.

Schneider, F. B. (2000). Enforceable security policies. *ACM Transaction on Information and System Security (TISSEC) 3*(1), 30–50.

Schuster, F., T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz (2015). Counterfeit object-oriented programming. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P)*, pp. 745–762.

Schwartz, E. J., T. Avgerinos, and D. Brumley (2011). Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium.*

Seibert, J., H. Okhravi, and E. Söderström (2014). Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pp. 54–65.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pp. 552–561.

Shacham, H., M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh (2004). On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pp. 298–307.

Snow, K. Z., F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi (2013). Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, pp. 574–588.

Solar Designer (1997). “return-to-libc” attack. *Bugtraq, Aug.*

Srinivasan, V. and T. Reps (2015a). Partial evaluation of machine code. In *Proceedings of the 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 860–879.

Srinivasan, V. and T. Reps (2015b). Synthesis of machine code from semantics. In *Proceedings of the 36th ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 596–607.

Strackx, R., Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter (2009). Breaking the memory secrecy assumption. In *Proceedings of the 2nd European Workshop on System Security (EUROSEC)*, pp. 1–8.

Tang, J. (2015). Exploring Control Flow Guard in Windows 10. Technical report, Trend Micro Threat Solution Team.

Tartler, R., A. Kurmus, B. Heinloth, V. Rothberg, A. Ruprecht, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann (2012). Automatic OS kernel TCB reduction by leveraging compile-time configurability. In *Proceedings of the 8th Conference on Hot Topics in System Dependability (HotDep)*.

The Wine Committee (2020). Wine. http://www.winehq.org.

Tice, C. (2012). Improving function pointer security for virtual method dispatches. In *GNU Cauldron Workshop*.

Tice, C., T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike (2014). Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 941–955.

van der Kouwe, E., G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida (2019). SoK: Benchmarking flaws in systems security. In *Proceedings of the 4th IEEE Eurpean Symposium on Security and Privacy (EuroS&P)*.

van der Veen, V., D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida (2015). Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pp. 927–940.

van der Veen, V., E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida (2016). A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P)*.

Vaughan-Nichols, S. J. (2014, September). Shellshock: Better 'bash' patches now available. *ZDNet*. https://www.zdnet.com/article/shellshock-better-bash-patches-now-available.

Wahbe, R., S. Lucco, T. E. Anderson, and S. L. Graham (1993). Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 203–216.

Walden, J., J. Stuckman, and R. Scandariato (2014). Predicting vulnerable components: Software metrics vs text mining. In *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 23–33.

Wang, C., J. Hill, J. Knight, and J. Davidson (2000). Software tamper resistance: Obstructing static analysis of programs. Technical report, U. Virginia, Charlottesville.

Wang, M., H. Yin, A. V. Bhaskar, P. Su, and D. Feng (2015). Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, pp. 331–340.

Wang, S., P. Wang, and D. Wu (2015). Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium*, pp. 627–642.

Wang, W., X. Xu, and K. W. Hamlen (2017). Object flow integrity. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pp. 1909–1924.

Wang, Z. and X. Jiang (2010). HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, pp. 380–395.

Wartell, R., V. Mohan, K. W. Hamlen, and Z. Lin (2012a). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pp. 157–168.

Wartell, R., V. Mohan, K. W. Hamlen, and Z. Lin (2012b). Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pp. 299–308.

Wartell, R., Y. Zhou, K. W. Hamlen, and M. Kantarcioglu (2014). Shingled graph disassembly: Finding the undecidable path. In *Proceedings of the 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pp. 273–285.

Wheeler, D. A. (2015). Shellshock. In *Learning from Disaster*. https://dwheeler.com/essays/shellshock.html.

Whittaker, Z. (2019). Two years after wannacry, a million computers remain at risk. http://techcrunch.com/2019/05/12/wannacry-two-years-on/.

Wilander, J., N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen (2011). RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, pp. 41–50.

Xu, X., M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin (2019). CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *Proceedings of the 28th USENIX Security Symposium*, pp. 1805–1821.

Yee, B., D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar (2009). Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security & Privacy (S&P)*, pp. 79–93.

Zhang, C., S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song (2016). VTrust: Regaining trust on virtual calls. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS).*

Zhang, C., C. Song, K. Z. Chen, Z. Chen, and D. Song (2015). VTint: Protecting virtual function tables' integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS).*

Zhang, C., T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zo (2013). Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, pp. 559–573.

Zhang, J., R. Hou, J. Fan, K. Liu, L. Zhang, and S. A. McKee (2017). RAGuard: A hardware based mechanism for backward-edge control-flow integrity. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, pp. 27–34.

Zhang, J., B. Qi, Z. Qin, and G. Qu (2019). HCIC: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things Journal 6*(1), 458–471.

Zhang, M. and R. Sekar (2013). Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, pp. 337–352.

Zimmermann, T., N. Nagappan, and L. Williams (2010). Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, pp. 421–428.

BIOGRAPHICAL SKETCH

Masoud Ghaffarinia was born in Gorgran, a northern city in Iran. As he lost his father in his early life, he was mostly influenced by his brothers, sister and her mother. His enthusiasm for studying computers started a long time ago. When he was eleven, he had a trip to his brother's university dormitory and he was fascinated with a simple Pac-Man game written by his brother's roommate. He was amazed how one is able to interpret one's thoughts in terms of machine-readable code to build up an intended functionality. After that, he became more interested in computers and programming, as he found programming very similar to mathematics as far as the thought process is concerned.

Programming, logic and discrete mathematics, as topics of interest, were all major incentives to choose Information Technology as his major in undergraduate studies. After entering Shiraz University of Technology, he followed his interest in programming and it became his hobby. His enthusiasm for coding, along with his habit of writing programs in online contests, enabled him to become a member of the university programming team in ACM contests.

After finishing his BS, he found graduate studies as a way in which he could gain knowledge more deeply in a field. Information Security, as a crucial area on account of its high critical issues, persuaded him to choose it as his major in his master study. Admitted to Amirkabir University of Technology (Tehran Polytechnic), he gained a real insight into this field.

A major milestone in his life was when he married his beloved wife, came to United States and started his PhD study at The University of Texas at Dallas, all in January 2015. He is blessed that he was supervised by Kevin Hamlen, one of the best and well-known researchers in the field of language-based security. At UT Dallas, he was trained to tackle software security problems scientifically. He had two internships at Cisco and Google during his study. After completing his PhD, Masoud will start his new career at Google.

CURRICULUM VITAE

Masoud Ghaffarinia

March 23, 2020

Contact Information:

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: `ghaffarinia.masoud@utdallas.edu`

Educational History:

B.S., Information Technology Engineering, Shiraz University of Technology, 2012
M.S., Information Technology Enginneering, Amirkabir University of Technology, 2014
Ph.D., Computer Science, University of Texas at Dallas, 2020

AUTOMATED BINARY SOFTWARE ATTACK SURFACE REDUCTION
Ph.D. Dissertation
Computer Science Department, University of Texas at Dallas
Advisors: Dr. Kevin W. Hamlen

Employment History:

Research Assistant, The University of Texas at Dallas, January 2015 – present
Software Engineer Intern, Google, San Francisco, May 2019 – August 2019
Software Engineer Intern, Cisco, Palo Alto, June 2017 – August 2017

Publications:

Masoud Ghaffarinia and Kevin W. Hamlen. **Binary Control-Flow Trimming**. In Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS), November 2019.
Xiaoyang Xu, **Masoud Ghaffarinia**, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. **ConFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software**. In Proceedings of the 28th USENIX Security Symposium, August 2019.
Masoud Ghaffarinia and Kevin W. Hamlen. **Binary Software Complexity Reduction: From Artifact to Feature-Removal**. In Proceedings of the 1st Workshop on Forming an Ecosystem Around Software Transformation (FEAST), October 2016.