NATIVE SOFTWARE SECURITY HARDENING IN THE REAL WORLD: COMPATIBILITY, MODULARITY, EXPRESSIVENESS, AND PERFORMANCE

by

Xiaoyang Xu

APPROVED BY SUPERVISORY COMMITTEE:

____
Dr. Kevin W. Hamlen, Chair

____
Dr. Bhavani M. Thuraisingham

____
Dr. Latifur Khan

____
Dr. Shuang Hao

Copyright © 2020

Xiaoyang Xu

All rights reserved

In memory of my dad.

NATIVE SOFTWARE SECURITY HARDENING IN THE REAL WORLD:
COMPATIBILITY, MODULARITY, EXPRESSIVENESS, AND PERFORMANCE

by

XIAOYANG XU, BE

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

Firstly, I would like to express the deepest appreciation to my advisor, Dr. Kevin Hamlen, for his unyielding guidance and thoughtful support. I would not have started my PhD if he had never shown me the world of cybersecurity six years ago in his classes. This dissertation could not have been completed without Dr. Hamlen serving as the best mentor to me.

Special thanks should be given to my friend and research partner, Wenhao Wang, who accompanied me for countless days and nights, staring at computer screens that were full of assembly code and breakpoints. I would also like to thank my colleagues, Masoud Ghaffarinia, Jun Duan, Benjamin Ferrell, and all who have inspired and worked with me.

I would like to extend my sincere thanks to my doctoral committee members, Dr. Bhavani Thuraisingham, Dr. Latifur Khan, and Dr. Shuang Hao, for their valuable advice, time, and efforts throughout the writing of this dissertation, and Rhonda Walls, who is always available whenever I need help.

Finally, I'm extremely grateful to my mom and my fiancée, whose unconditional support and love encouraged me to explore new directions in life and seek my own destiny.

The research reported in 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 Defense Advanced Research Projects Agency (DARPA) under award FA8750-19-C-0006, the Air Force Office of Scientific Research (AFOSR) under Young Investigator Program (YIP) awards FA9550-14-1-0119 and FA9550-14-1-0173, the National Science Foundation (NSF) under CAREER awards #1513704, #1834215 and #1054629, and NSF Industry-University Collaborative Research Center (I/UCRC) awards from Raytheon Company and Lockheed Martin. All opinions, recommendations, and conclusions expressed are those of the author and not necessarily of the ONR, DARPA, AFOSR, NSF, Raytheon, or Lockheed-Martin.

March 2020

NATIVE SOFTWARE SECURITY HARDENING IN THE REAL WORLD:
COMPATIBILITY, MODULARITY, EXPRESSIVENESS, AND PERFORMANCE

Xiaoyang Xu, PhD
The University of Texas at Dallas, 2020

Supervising Professor: Dr. Kevin W. Hamlen, Chair

This dissertation presents a series of new technologies that significantly bridge the gap between theory and practice of software hijacking defenses based on *control-flow integrity* (CFI) and *in-lined reference monitors* (IRMs). CFI has emerged over the past 15 years as one of the strongest known defenses against *code-reuse attacks*, which are among the top threats to modern software ecosystems. Such attacks wrest control of critical software systems away from lawful users into the hands of adversaries by reusing or repurposing legitimate code blocks for malicious purposes. CFI offers provably strong protections against code-reuse attacks by confining vulnerable software to a strict security policy that constrains its flow of control to paths chosen in advance by developers and legitimate users.

Research over the past decade has increased the power and performance of CFI defenses; however, effectively applying many of the strongest CFI algorithms to large, production-level software products have remained difficult and challenging. To expose the root causes of these difficulties, this dissertation presents a new evaluation methodology and microbenchmarking suite, ConFIRM, that is designed to measure applicability, compatibility, and performance characteristics relevant to CFI algorithm evaluation. It provides a set of 20 tests of various CFI-relevant code features and coding idioms (e.g., event-driven callbacks and exceptions),

which are widely found in commodity COTS software products and constitute the greatest barriers to more widespread CFI adoption.

To overcome a significant class of fundamental challenges identified by ConFIRM, the dissertation then presents *object flow integrity* (OFI), which is the first source-agnostic CFI system that augments CFI protections with secure, first-class support for binary object exchange across inter-module trust boundaries. A prototype implementation for Microsoft Component Object Model (COM) demonstrates that OFI scales to component-based, event-driven consumer software with low overheads of under 1%. The approach is demonstrated in practice through an interface-driven approach that is the first to secure full COTS, GUI-driven Windows products with CFI without needing the application source code.

Finally, the IRM technology underlying CFI is shown to be effective in web domains for enforcing safety policies by injecting runtime security guards into binary web scripts. In particular, a method of detecting and interrupting unauthorized, browser-based cryptomining is proposed, based on semantic signature-matching. The approach addresses a new wave of cryptojacking attacks, including XSS-assisted, web gadget-exploiting, counterfeit mining. Evaluation shows that the approach is more robust than current static code analysis defenses, which are susceptible to code obfuscation attacks.

TABLE OF CONTENTS

ACKNOWLEDGMENTS v

ABSTRACT vi

LIST OF FIGURES xi

LIST OF TABLES xii

LIST OF LISTINGS xiii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 EVALUATING COMPATIBILITY AND RELEVANCE OF CONTROL-FLOW INTEGRITY PROTECTIONS FOR MODERN SOFTWARE 8

2.1 Introduction 9

2.2 Background 12

2.3 Compatibility Metrics 15

2.3.1 Indirect Branches 16

2.3.2 Other Metrics 22

2.4 Implementation 29

2.5 Evaluation 32

2.5.1 Evaluation of CFI Solutions 32

2.5.2 Evaluation Trends 37

2.5.3 Performance Evaluation Correlation 40

2.6 Conclusion 41

CHAPTER 3 OBJECT FLOW INTEGRITY 44

3.1 Introduction 45

3.2 Background 49

3.2.1 Inter-module Object Flows 49

3.2.2 CODE-COOP Attacks 51

3.3 Design 53

3.3.1 Object Proxying 53

3.3.2 Type-based Contracts 57

3.3.3 Trust Model 61

3.4 Implementation 64
3.4.1 Architecture 64
3.4.2 Dispatcher Implementation 65
3.4.3 Automated Mediator Synthesis 72
3.5 Evaluation 76
3.5.1 Transparency 77
3.5.2 Performance Overheads 79
3.5.3 Security Evaluation 82
3.5.4 Scalability 84
3.6 Conclusion 85
CHAPTER 4 TOWARDS INTERFACE-DRIVEN COTS BINARY HARDENING . 86
4.1 Introduction 86
4.2 Attack Example 88
4.3 Technical Approach 92
4.4 Case Study 94
4.4.1 Object-oriented Design 94
4.4.2 API Surface 96
4.4.3 Object Exchanges 96
4.4.4 Callbacks 100
4.5 Future work 100
4.6 Conclusion 102
CHAPTER 5 SEISMIC: SECURE IN-LINED SCRIPT MONITORS FOR INTERRUPTING CRYPTOJACKS 103
5.1 Introduction 104
5.2 Background 107
5.2.1 Monero 107
5.2.2 WebAssembly 107
5.3 Ecosystem of Browser-based Cryptocurrency Mining 108
5.4 Counterfeit Mining Attacks 110

5.5 Detection . . . 113
5.5.1 Current Methods . . . 114
5.5.2 Semantic Signature-matching . . . 114
5.5.3 SEISMIC In-lined Reference Monitoring . . . 118
5.6 Evaluation . . . 123
5.6.1 Runtime Overhead . . . 124
5.6.2 Robustness . . . 125
5.7 Conclusion . . . 126
CHAPTER 6 RELATED WORK . . . 127
6.1 Prior CFI Evaluations . . . 127
6.2 CFI Surveys . . . 129
6.3 SFI and CFI . . . 129
6.4 VTable Protection . . . 131
6.5 COOP Attacks . . . 133
6.6 Immutable Modules . . . 133
6.7 Component-based Software Engineering . . . 134
6.8 Cryptocurrencies . . . 135
6.8.1 Cross-Site Scripting . . . 135
6.9 Related Web Script Defenses . . . 136
6.10 Semantic Malware Detection and Obfuscation . . . 137
CHAPTER 7 CONCLUSION . . . 138
7.1 Dissertation Summary . . . 138
7.2 Future Work . . . 139
REFERENCES . . . 141
BIOGRAPHICAL SKETCH . . . 155
CURRICULUM VITAE

LIST OF FIGURES

2.1 Source code compiled to indirect call 18
3.1 Cross-module OFI control-flows 55
3.2 Proxy object binary representation 57
3.3 A type system for expressing CFI obligations as OFI contracts 58
3.4 Mediator enforcement of OFI contracts 60
3.5 REINS system architecture 64
3.6 Automated mediator synthesis 73
3.7 OFI runtime overhead 80
4.1 Object binary representation 91
4.2 Proxy object binary representation 93
5.1 Browser-based mining workflow 109
5.2 Reflected (left) and stored (right) counterfeit mining attacks 112
5.3 Antivirus detection of CryptoNight before and after function renaming 115
5.4 Semantic profiles for mining vs. non-mining Wasm apps 118
5.5 SEISMIC transformation of Wasm binaries 119

LIST OF TABLES

2.1 ConFIRM compatibility metrics . . . 17
2.2 Tested results for CFI solutions on ConFIRM . . . 33
2.3 Overall compatibility of CFI solutions . . . 39
2.4 Correlation between SPEC CPU and ConFIRM performance . . . 42
3.1 Interactive COM applications used in experimental evaluation . . . 78
3.2 Micro-benchmark overheads . . . 81
3.3 Attack simulation results . . . 82
3.4 Browser experimental results . . . 83
4.1 Interoperating COM modules used in case study . . . 95
4.2 APIs with object exchanges . . . 97
4.3 COM interfaces . . . 98
4.4 Methods with object exchanges . . . 99
4.5 APIs with callback pointers . . . 101
5.1 Security-related features of popular miners . . . 109
5.2 Top 30 Opcodes Used as Features to Distinguish Mining and Non-mining . . . 116
5.3 Execution trace average profiles . . . 117
5.4 Mining overhead . . . 124
5.5 SVM stratified 10-fold cross validation . . . 125

LIST OF LISTINGS

3.1 Code that opens a file-save dialog box . . . 50
3.2 CODE-COOP attack vulnerability . . . 51
3.3 Vault Dispatch implementation (abbreviated) . . . 66
3.4 Virtual Vault Dispatch implementation (abbreviated) . . . 67
3.5 Bouncer implementation (abbreviated) . . . 68
3.6 BouncerDown implementation (abbreviated) . . . 69
3.7 Virtual Bouncer-down implementation (abbreviated) . . . 70
3.8 Synthesized vaulter implementation . . . 74
3.9 Reflective template (abbreviated) . . . 75
3.10 Mediator synthesis via template recursion . . . 76
4.1 Code that registers a running application Windows Image Acquisition (WIA) event notification . . . 89
4.2 Function call in assembly . . . 90
5.1 Embedded miner HTML code . . . 111
5.2 JavaScript gadget . . . 112
5.3 C++ source code for compilation to Wasm . . . 120
5.4 Original Wasm compiled from C++ . . . 120
5.5 Instrumented Wasm . . . 122
5.6 SEISMIC JavaScript code . . . 123

CHAPTER 1

INTRODUCTION

Software is increasingly complex. Google Chrome runs on 6.7 million lines of code; Microsoft Windows operating system has roughly 50 million lines of code; and a typical new-model vehicle comes with 100 million lines of code.[1] This is due to the fact that modern software is designed to support more platforms and various features to meet users' needs. Unfortunately, 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 higher possibility for programmer error, malicious code introduction, and unforeseen component interactions.

Cyberattacks continue to dominate headlines. Multiple companies revealed data breach in 2019, including Capital One (Capital One, 2019), Marriott Hotels (Fruhlinger, 2020) and Facebook (Moore, 2019). A bug in iOS Facetime discovered in 2019 allowed malicious iPhone users to eavesdrop merely by calling the victim's iPhone, even if the victim does not accept the call (Mayo, 2019). In May 2019, the city of Baltimore, Maryland was attacked by hackers who froze thousands of city computers and demanded bitcoins as ransom, and this attack cost the city $18 million (Duncan, 2019). In Summer 2019, the entire nation of Bulgaria got hit and over 5 million Bulgarians had their personal data stolen by hackers from the country's tax revenue office (Santora, 2019).

Moreover, the most ubiquitous and prevalent form of software is native code, which remains one of the primary targets for malicious software attacks. It is harder to secure *commercial off-the-shelf* (COTS) and legacy binaries in use today, which do not have source or debug information available, or for which consumers will not accept a significant performance degradation purely for the sake of improved security. Therefore, it is very common

[1] https://informationisbeautiful.net/visualizations/million-lines-of-code/

for software users to possess known but not fully trusted native code, or unknown binaries that they are lured to run. Even security-sensitive software organizations, such as military agencies, face a difficult challenge when it comes to selecting secure software, since the most up-to-date, feature-filled, and well-tested software tends to be commercial products whose developers prioritize sales over security.

Recently, technologies for secure binary analysis and transformation have emerged that aim to safely filter untrustworthy code or statically transform it into safe code. Among such technologies, binary instrumentation via *in-lined reference monitors* (IRMs) has been firmly established as a powerful and versatile technique for enforcing fine-grained security policies, without requiring access to source code. IRM implementations operate by inserting code for security checks into the untrusted software, thereby constraining its behavior to a security policy. Unlike executing untrusted software within a sandboxing virtual machine (VM), IRMs are light-weight and do not require kernel modifications or administrative privileges to safely run the new self-monitoring code. This allows IRMs to be deployed in environments where such controls are unavailable. Another advantage of IRMs is that they can enforce specialized, application-specific policies or perform platform-specific optimizations, making them more flexible and offering lower overhead than traditional OS- or VM-level execution monitoring.

IRM frameworks must consider that a monitored application may attempt to bypass the inserted checks by jumping over them. Therefore, they must impose restrictions on control-flow. However, modern control-flow hijacking attacks can subvert control-flows of vulnerable programs by exploiting memory corruptions and redirecting control arbitrarily. Over the past two decades, operating system and hardware developers have proposed two primary defenses against traditional memory corruption exploits, namely *data execution prevention* (DEP) (Andersen, 2004) and *address space layout randomization* (ASLR) (PaX Team, 2003). DEP successfully stops a program from executing injected code. However, it has limitations:

Firstly, just-in-time (JIT) compilation intentionally violates DEP's strategy, since JIT code requires memory pages that are both writable and executable. Moreover, even with DEP enabled, follow-on *code-reuse attacks* (Bletsch et al., 2011) (e.g., "return-to-libc" attacks) can circumvent it without any code injection. In response, defenders proposed ASLR, which breaks code reuse attacks by randomizing code layout to thwart its abuse. Unfortunately, DEP and ASLR are imperfect antidotes. They motivated attackers to tailor even more elaborate attacks, including *return-oriented programming* (ROP) (Roemer et al., 2012) and *jump-oriented programming* (JOP) (Bletsch et al., 2011), which locate, stitch, and execute short instruction sequences (*gadgets*) of benign code to implement malicious payloads and achieve arbitrary code execution.

More recently, *control-flow integrity* (CFI) (Abadi et al., 2005) has emerged as a more comprehensive and principled defense against this malicious code-reuse. To protect untrusted code from software hijacking, CFI imposes security policies that constrain the targets of control-flow transfers. The policy that a CFI implementation enforces is a *control-flow graph* (CFG), which whitelists the set of legitimate transfer destinations. A CFI framework typically consists of two components: the first component analyzes the untrusted software statically or dynamically and constructs a CFG; the other component then instruments the untrusted software with in-lined runtime guards to enforce this CFG.

A software CFI methodology can be either a source-aware, source-to-source approach or a source-agnostic, binary-to-binary transformation. Source-aware CFI solutions typically leverage source-level information and generate CFI-enforcing object code via a compiler. This allows CFI algorithms to produce more precise policies, and can often perform more optimization to achieve better performance. Such CFI frameworks 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). Reliance on available source code has the potential compatibility problem of reducing deployment flexibility, since the vast majority of COTS software (or library) is closed-source to consumers, due to intellectual property concerns and constraints imposed by developer business models. On the other hand, source-agnostic CFI solutions instrument and harden already-compiled binary code without the aid of source code to achieve this flexibility. Examples 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). However, such CFI implementations are still facing some challenging problems. For example, since perfect disassembly is known to be undecidable in general (Wartell et al., 2014), lacking source-level control-flow information can result in less strict CFG, and thus more permissive control-flow policies.

Inspired by the original CFI published in 2005, there has been productive new research devoted to protecting software from more elaborate attacks, deriving richer policies, and providing better performance. These new frameworks are generally evaluated and compared in terms of security and performance. Security is usually assessed using the RIPE test suite (Wilander et al., 2011) or with manually crafted proof-of-concept attacks such as COOP (Schuster et al., 2015). Performance overhead is commonly evaluated in terms of the CPU benchmark suites (e.g., SPEC). Particularly, among 54 surveyed CFI algorithms and implementations published in tier-1 scientific venues between 2005–2019, 66% evaluate performance overheads by applying SPEC CPU benchmarking programs.

While this attention to performance and security has stimulated rapid gains in the ability of CFI solutions to efficiently enforce powerful, precise security policies, fewer works have aimed at 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. In particular, 88% of the surveyed CFI solutions report evaluations on 3 or fewer large, independent applications. Moreover, such compatibility issues can have dangerous security ramifications if they prevent protection of software needed in mission-critical contexts, or if the protection policies must be weakened in order to achieve compatibility. For example, to avoid incompatibilities related to C/C++ pointer arithmetic, 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.

Understanding these compatibility limitations, including their impacts on real-world software performance and security, requires a new suite of CFI benchmarks with substantially different characteristics than CPU benchmarks typically used to assess compiler or hardware performance. In particular, CFI compatibility 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 CPU benchmarks (e.g., SPEC CPU), which are designed to exercise CPU computational units using only simple control-flow graphs, or by utility suites (e.g., GNU Corelibs) 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 these compatibility and applicability limitations of modern CFI solutions, and to identify the coding idioms and features that constitute the greatest barriers to more widespread CFI adoption, Chapter 2 presents ConFIRM (CONtrol-Flow Integrity Relevance Metrics), a new 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.

ConFIRM consists of 24 tests designed specifically to examine compatibility characteristics relevant to control-flow security hardening evaluation. Each test is designed to exhibit one or more code features or coding idioms with high compatibility impact found in a large number of commodity software products. Reevaluation of 12 CFI algorithms using ConFIRM shows that state-of-the-art CFI solutions are compatible with only about half of the CFI-relevant code features and coding idioms needed to protect large, production software systems that are frequently targeted by cybercriminals. In addition, using ConFIRM for microbenchmarking reveals performance characteristics not captured by SPEC benchmarks.

Among the coding idioms and features present in Chapter 2, CFI historically suffered difficulty of hardening COTS software that contains immutable system modules with large, object-oriented APIs—which are particularly common in component-based, event-driven consumer software. Modifying and transforming such native software is challenging in contests where surrounding software environment includes closed-source, unmodifiable, and possibly obfuscated binary components, such as system libraries and OS kernels.

In response to this difficulty, Chapter 3 of this dissertation presents *object flow integrity* (OFI), which extends CFI with secure, first-class support for immutable, trusted modules with object-oriented APIs. OFI augments untrusted software with safely exchanging binary objects across inter-module trust boundaries without varying trusted module code, by ensuring that trusted callee modules never receive writable code pointers from untrusted, CFI-protected callers. To achieve this without breaking intricate object exchange protocols, OFI implementation centers around the idea of *proxy objects* that are actually IRMs that wrap and mediate access to the methods of the objects they proxy. Prior to the introduction of OFI enhancements, no CFI algorithm successfully preserved and secured the full functionality of Windows Notepad—one of the most ubiquitous consumer software products available. A prototype implementation for Microsoft Component Object Model (COM) showcases that OFI is scalable to complex COTS software (e.g. Mozilla Firefox) that has

large interfaces on the order of tens of thousands of methods, and exhibits low overheads of under 1%.

Next, Chapter 4 reports experience results of using OFI to secure application-level software modules without the need to harden all other trusted modules in the environment with exactly the same protection strategy or policies. The experimental results show that, coupled with OFI and CFI, the approach can effectively thwart elaborated code-reuse attacks by completely mediating the interfaces between trusted and untrusted modules.

In addition, IRMs can enforce safety policies by injecting runtime security guards not only into native code but also in web scripts. Chapter 5 introduces SEcure In-lined Script Monitors for Interrupting Cryptojacks (SEISMIC). SEISMIC is a novel semantic-based signature-matching approach that automatically modifies incoming WebAssembly (Wasm) binary programs so that they self-profile themselves as they execute, to detect unauthorized cryptomining activity. When cryptomining is detected, the instrumented script warns and prompts the user to explicitly opt-out or opt-in. Opting out halts the script, whereas opting in continues the script without further profiling (allowing it to execute henceforth at full speed). An implementation of SEISMIC offers a browser-agnostic deployment strategy that is applicable to average end-user systems without specialized hardware or operating systems.

The rest of this dissertation is laid out as follows. Chapter 2 explores the compatibility and applicability limitations of modern CFI solutions, and then presents the first CFI-relevant evaluation methodology and microbenchmarking suite: ConFIRM (Xu et al., 2019). Chapter 3 demonstrates the OFI (Wang et al., 2017) framework and implementation, and a detailed case study (Xu et al., 2018) of OFI is reported in Chapter 4. Chapter 5 presents a semantic-based in-line script monitoring system, SEISMIC (Wang et al., 2018), which instruments Wasm binaries with mining detectors. Finally, Chapter 6 discusses related works and Chapter 7 concludes.

CHAPTER 2

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

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. CFI protects native software from hijacking by inserting guard code that restricts program execution to a set of legitimate control-flow targets at runtime.

Although CFI has become a mainstay of protecting certain classes of software from code-reuse attacks, and continues to be improved by ongoing research, its ability to preserve intended program functionalities (*semantic transparency*) of diverse, mainstream software products have been under-studied in the literature. This is in part because although CFI solutions are evaluated in terms of performance and security, 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.

In this chapter, we propose ConFIRM (CONtrol-Flow Integrity Relevance Metrics), which is 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.

---

[1]This chapter contains material previously published as: 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*, pp. 1805–1821, August 2019.

The rest of this chapter is laid out as follows. Section 2.1 begins with an introductory overview of the CFI compatibility problem and ConFIRM's evaluation metrics. Then, Section 2.2 reports a summary of technical CFI attack and defense details important for understanding the evaluation approach. Section 2.3 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 2.4 describes an implementation of the resulting benchmarks. Section 2.5 reports our evaluation of CFI solutions using ConFIRM and discusses significant findings. Finally, Section 2.6 concludes.

2.1 Introduction

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 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, to avoid incompatibilities related to C/C++ pointer arithmetic, 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 benchmarks 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 corelibs) 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 benchmarks designed to exhibit code features most relevant to CFI evaluation. Our design of ConFIRM is based

on over 25 years of collective experience building and evaluating CFI systems for a variety of architectures, including Linux, Windows, Intel x86/x64, and ARM32 in both academia and industry. Each benchmark 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).

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, 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 9 publicly available CFI implementations published in the open literature. The results show substantial performance differences and trade-offs not revealed by prior CPU-based benchmarking. For example, tested CFI implementations exhibit a median overhead of over 70% to secure returns, in contrast with average overheads of about 3% reported in the prior literature for CPU benchmarks; and a new *cross-thread stack-smashing* attack defeats all tested CFI defenses.

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 important 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.

2.2 Background

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 native code transformation). Source-aware solutions typically benefit from source-level information to derive more precise policies, and can often perform more aggressive 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 was 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. 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.[2] 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

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

(Office of Inspector General, 2018). None of the top 10 vulnerabilities exploited by cybercriminals in 2017 target Linux software (Donnelly, 2018).

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 benchmarking 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.

2.3 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 2.1 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.

2.3.1 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, memory-reads, or both. Such transfers tend to be of high interest to attackers, since computed destinations have more potential to be manipulated. CFI solutions 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.

Figure 2.3.1 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.

2.3.1.1 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.

Table 2.1. 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 family software, including MS Excel, MS PowerPoint, MS PowerPoint, etc.

Source code	Assembly code
1 void foo { return; } 2 void bar { return; } 3 void main { 4 void (* fptr) (); 5 int n = input(); 6 if (n) 7 fptr = foo; 8 else 9 fptr = bar; 10 fptr(); 11 }	1 ... 2 call _input 3 test eax, eax 4 jnz offset_7 5 mov ecx, offset_bar 6 cmovnz ecx, edx 7 call ecx 8 ...

Figure 2.1. Source code compiled to indirect call

2.3.1.2 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.

2.3.1.3 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.

2.3.1.4 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.

2.3.1.5 Tail Calls

Modern C/C++ compilers can optimize tail-calls by replacing them with jumps. Table 2.1 lists relevant options for mainstream compilers. 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.

2.3.1.6 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.

2.3.1.7 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.

2.3.1.8 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.

2.3.2 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.

2.3.2.1 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 §2.3.1), 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.

2.3.2.2 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.

2.3.2.3 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.

2.3.2.4 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.

2.3.2.5 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.

2.3.2.6 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 2.1), 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.

2.3.2.7 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.

2.4 Implementation

To facilitate easier evaluation of the compatibility considerations outlined in Section 2.3 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 2.3 (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 2.3, 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 is exercised, and tests determine whether function calls to symbols that are exported by the 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 an intensive 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 and setjmp/-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 the `AddVectoredExceptionHandler()` API.

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 2.3.

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

2.5 Evaluation

2.5.1 Evaluation of CFI Solutions

To examine ConFIRM's effect on real CFI solutions, 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 2.2 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 2.2. 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[⊞]	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[⊞]	✓	⚠	✓	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
veh[⊞]	⚠	⚠	⚠	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
convention	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
multithreading	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠
tls_callback[⊞,$]	N/A	N/A	N/A	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
pic	✓	✓	✓	⚠	⚠	✓	✓	✓	✓	✓	✓	✓	✓
mem	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	✗	✗	✗	✓	✗
jit	⚠	⚠	⚠	✗	✗	⚠	⚠	⚠	✗	✗	✗	⚠	✗
unpacking[$]	N/A	N/A	N/A	✗	✗	N/A	N/A	N/A	N/A	N/A	N/A	✗	✗
api_hook[⊞]	⚠	⚠	⚠	✗	✗	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

⊞: 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 when executed. The solution 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 2.2, 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 2.2 reports evaluation of Microsoft's MCFG solution, which is integrated into the MSVC compiler. MCFG provides security checks for calls via function pointers, 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 the related features, but does not protect vectored exception handlers. MCFG's most significant shortcoming is its weak protection of return addresses. In addition, MCFG generates guard code at call sites at compile time only. Therefore, code that links to immutable modules or modules compiled with a differ-

ent protection scheme remains potentially insecure. This results in failures against callback corruption attacks and CODE-COOP attacks.

Columns 5–6 of Table 2.2 report compatibility testing results for Reins and OFI, which are both source-free binary rewriting 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 both 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, however.

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 2.2, 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 2.2. 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.

2.5.2 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 is unclear whether its hardware shadow stack will be compatible with software idioms that exhibit unmatched call-return pairs (see §2.3).

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 2.3 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%.

Table 2.3. 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.

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.

2.5.3 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 rele-

vant to CFI. High correlations therefore indicate to what degree SPEC benchmarks exercise code features whose performance are affected by CFI controls.

Table 2.4 reports the results, in which correlations between each SPEC CPU benchmark and ConFIRM median values are computed as Pearson correlation coefficients:

$$r_{xy} = \frac{(\sum_{i=1}^{n} x_i \times y_i) - (n \times \bar{x} \times \bar{y})}{(n-1) \times s_x \times s_y} \tag{2.1}$$

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.

2.6 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.

Table 2.4. 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**

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, performance analysis indicates that 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 3

OBJECT FLOW INTEGRITY[1]

Chapter 2 demonstrates some notable gaps between CFI theory and practice in the current state-of-the-art. Several of the code features and coding idioms in this gap are identified as presenting significant barriers to adoption, and can defeat all the tested CFI defenses (e.g. the cross-thread stack-smashing attack). One particularly large class of barriers involves the difficulty of protecting software that contains immutable system modules with large, object-oriented APIs—which are particularly common in component-based, event-driven consumer software.

To extend both source-aware and source-free CFI and SFI technologies to this large class of previously unsupported software, this chapter presents Object Flow Integrity (OFI), which augments CFI and SFI protections with secure, first-class support for binary object exchange across inter-module trust boundaries. It also helps to protect these inter-module object exchanges against confused deputy-assisted vtable corruption and counterfeit object-oriented programming attacks.

In addition, a prototype implementation for Microsoft Component Object Model (COM) demonstrates that OFI is scalable to large interfaces on the order of tens of thousands of methods, and exhibits low overheads of under 1% for some common-case applications. Significant elements of the implementation are synthesized automatically through a principled design inspired by type-based contracts.

The rest of Chapter 3 is structured as follows. Section 3.1 begins with a high-level overview of the existing CFI algorithms and a new form of confused deputy attack. Section 3.2 then presents a detailed examination of such attack and how it manages to evade

[1]This chapter contains material previously published as: Wenhao Wang, Xiaoyang Xu, and Kevin W. Hamlen. "Object Flow Integrity." In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pp. 1909–1924, November 2017.

incomplete CFI protections applied to source-free, component-based software. Section 3.3 demonstrates OFI's approach to addressing these dangers. Our prototype implementation and its evaluation are presented in Sections 3.4 and 3.5, respectively. Section 3.6 concludes the chapter.

3.1 Introduction

Control-flow integrity (CFI) (Abadi et al., 2009) and *software fault isolation* (SFI) (Wahbe et al., 1993) secure software against control-flow hijacking attacks by confining its flows to a whitelist of permissible control-flow edges. The approach has proven successful against some of the most dangerous, cutting-edge attack classes, including *return-oriented programming* (ROP) (Roemer et al., 2012) and other *code-reuse attacks* (CRAs) (Bletsch et al., 2011). Attacks in these families typically exploit dataflow vulnerabilities (e.g., buffer overflows) to corrupt code pointers and thereby redirect control to attacker-chosen program subroutines. By validating each impending control-flow target at runtime before it is reached, CFI guards can often thwart these hijackings.

CFI and SFI frameworks work by statically instrumenting control-flow transfer instructions in vulnerable software with extra *guard code* that validates each computed jump destination at runtime. The instrumentation can be performed at compile-time (e.g., Akritidis et al. (2008); Abadi et al. (2009); Bletsch et al. (2011); Niu and Tan (2013, 2014b,a); Tice et al. (2014); Jang et al. (2014); Mashtizadeh et al. (2015); Niu and Tan (2015); Zhang et al. (2016); Bounov et al. (2016); Tang (2015)) or on sourceless binaries (e.g., Wartell et al. (2012b); Zhang et al. (2013); Zhang and Sekar (2013); Wang et al. (2015); Zhang et al. (2015); Payer et al. (2015); Mohan et al. (2015); van der Veen et al. (2016)). This facility to harden source-free binary software is important for securing software in-flight—allowing third parties to secure dynamically procured binary software on-demand in a way that is

transparent to code producers and consumers—and for securing the large quantity of software that is closed-source, or that incorporates software components (e.g., binary libraries) whose source code is unavailable to code consumers.

While the past decade has witnessed rapid progress toward more powerful, higher performance, and more flexible CFI enforcement strategies, there still remain large classes of consumer software to which these technologies are extremely difficult to apply using existing methods. Such limitations often stem from many source-aware CFI algorithms' need for full source code for the entire software ecosystem (e.g., even for the OS kernel, device drivers, and complete runtime system) in order to properly analyze application control-flows, or the difficulty of analyzing complex flows common to certain well-entrenched consumer software paradigms, such as GUI-interactive, event-driven, and component-based software applications. For example, although CFI has been applied successfully to some large applications, in our experience no CFI/SFI algorithm published in the literature to date (see §6) successfully preserves and secures the full functionality of Windows Notepad—one of the most ubiquitous consumer software products available.

The central problem is a lack of first-class support for architectures in which immutable, trusted software components have huge *object-oriented interfaces*. Programs like Notepad interact with users by displaying windows, monitoring mouse events, and sending printer commands. At the binary level, this is achieved by calling runtime system API methods that expect binary objects as input. The app-provided binary object contains a virtual method table (*vtable*), whose members are subsequently called by the runtime system to notify the app of window, mouse, and printer events. The call sites that target untrusted code, and that CFI algorithms must instrument, are therefore *not exclusively located within the untrusted app code*—many are within trusted system libraries that cannot be modified (or sometimes even examined) by the instrumentation process, since they are part of the protected runtime system.

Most CFI algorithms demand write-access to all system software components that may contain unguarded, computed jumps—including the OS, all dynamically loaded libraries, and all future updates to them—in order to ensure safety. In component-driven settings, where modules are dynamically procured on-demand via a distributed network or cloud, this is often impractical. Unfortunately, such settings comprise >98% of the world's software market,[2] including many mission-critical infrastructures that incorporate consumer software components.

One approach for coping with this pervasive problem has been to secure objects passed to uninstrumented modules at call sites within the instrumented modules, before the trusted module receives them (e.g., Tice et al., 2014). But this approach fails when trusted modules retain persistent references to the object, or when their code executes concurrently with untrusted module code. In these cases, verifying the object at the point of exchange does not prevent the untrusted module from subsequently modifying the vtable pointer to which the trusted module's reference points (e.g., as part of a data corruption attack). We refer to such attacks as *COnfused DEputy-assisted Counterfeit Object-Oriented Programming* (CODE-COOP) attacks, since they turn recipients of counterfeit objects (Schuster et al., 2015) into confused deputies (Hardy, 1988) who unwittingly invoke policy-prohibited code on behalf of callers.

Faced with such difficulties, many CFI systems conservatively resort to disallowing untrusted module accesses to trusted, object-oriented APIs to ensure safety. This confines such approaches to architectures with few trusted object-oriented system APIs (e.g., Linux), applications that make little or no use of such APIs (e.g., benchmark or command-line utilities), or platforms where the majority of the OS can be rewritten (e.g., ChromeOS (Tice et al., 2014)). The majority of present-day software architectures that fall outside these restrictive parameters have remained unsupported or receive only incomplete CFI security.

[2] https://www.netmarketshare.com

To bridge this longstanding gap, we introduce *object flow integrity* (OFI)—a systematic methodology for imbuing CFI and SFI systems with first-class support for immutable, trusted modules with object-oriented APIs. OFI facilitates safe, transparent flow of binary objects across trust boundaries in multi-module processes, without any modification to trusted module code. To maintain the deployment flexibility of prior CFI/SFI approaches, OFI assumes no access to untrusted application or trusted system source code; we assume only that trusted *interfaces* are documented (e.g., via public C++ header files or IDL specifications).

Our prototype implementation showcases OFI's versatility and scalability by targeting the largest, most widely deployed object-oriented system API on the consumer software market—Microsoft *Component Object Model* (COM) (Gray et al., 1998). Most Windows applications rely upon COM to display dialog boxes (e.g., save- and load-file dialogs), create interactive widgets (e.g., ActiveX controls), or dynamically discover needed system services. To handle these requests in a generalized, architecture-independent manner, COM implements an elaborate system of dynamic, shared module loading; distributed, inter-process communication; and service querying facilities—all fronted by a vast, language-independent, object-oriented programming interface. Consequently, COM-reliant applications (which constitute a majority of consumer software today) have remained significantly beyond the reach of CFI/SFI defenses prior to OFI.

To keep our scope tractable, this chapter does not attempt to address all research challenges faced by the significant body of CFI literature. In particular, we do not explicitly address the challenges of optimizing the performance of the *underlying* CFI enforcement mechanism, deriving suitable control-flow policies for CFI mechanisms to enforce (cf., Schuster et al., 2015), or obtaining accurate native code disassemblies without source code (cf., Wartell et al., 2014). Our goal is to enhance existing CFI/SFI systems with support for a much larger class of target application programs and architectures without exacerbating any of these challenges, which are the focuses of related works.

In summary, our contributions are as follows:

- We introduce a general methodology for safely exchanging binary objects across inter-module trust boundaries in CFI/SFI-protected programs without varying trusted module code.

- A prototype implementation for Microsoft COM demonstrates that the approach is feasible for large, complex, object-oriented APIs on the order of tens of thousands of methods.

- A significant portion of the implementation is shown to be synthesizable automatically through a novel approach to reflective C++ programming.

- Experimental evaluation indicates that OFI imposes negligible performance overhead for some common-case, real-world applications.

3.2 Background

3.2.1 Inter-module Object Flows

To motivate OFI's design, Listing 3.1 presents typical C++ code for creating a standard file-open dialog box on a COM-based OS, such as Windows. The untrusted application code first creates a shared object $\mathtt{o}_1$ (line 1), where $\langle \mathit{clsid} \rangle$ and $\langle \mathit{iid}_1 \rangle$ are global numeric identifiers for the system's `FileOpenDialog` class and `IFileOpenDialog` interface of that class, respectively. Method `Show` is then invoked to display the dialog (line 2).

While executing `Show`, the trusted system module separately manipulates object $\mathtt{o}_1$, including calling its `QueryInterface` method to obtain a new interface $\mathtt{o}_2$ for the object, and invoking its methods (lines 3–5). Once the user has finished interacting with the dialog and it closes, the untrusted module calls $\mathtt{o}_1$'s `GetResult` method to obtain an `IShellItem` interface $\mathtt{o}_3$ whose `GetDisplayName` method discloses the user's file selection (lines 6–7). Finally, the untrusted module releases the shared objects (lines 8–9).

```
    Untrusted Module
1   CoCreateInstance(⟨clsid⟩, ..., ⟨iid1⟩, &o1);
2   o1→Show(...);                                    Trusted Module
3                                                    o1→QueryInterface(⟨iid2⟩, &o2);
4                                                    o2→GetOptions(...);
5                                                    o2→Release();
6   o1→GetResult(&o3);
7   o3→GetDisplayName(...);
8   o3→Release();
9   o1→Release();
```

Listing 3.1. Code that opens a file-save dialog box

Safely supporting this interaction is highly problematic for CFI frameworks. All method calls in Listing 3.1 target *non-exported functions* located in trusted system libraries. The function entry points are only divulged to untrusted modules at runtime within vtables of shared object data structures produced by trusted modules. By default, most CFI policies block such control-flows as indistinguishable from control-flow hijacking attacks.

If one whitelists these edges in the control-flow policy graph to permit them, a significant new problem emerges: Each method call implicitly passes an object reference (the *this* pointer) as its first argument. A compromised, untrusted module can therefore pass a counterfeit object to the trusted callee, thereby deputizing it to commit control-flow violations when it invokes the object's counterfeit method pointers.

One apparent solution is to validate these object references on the untrusted application side at the time they are passed, but this introduces a TOCTOU vulnerability: Since shared COM objects are often dynamically allocated in writable memory, a compromised or malicious application can potentially modify the object's vtable pointer or its contents after passing a reference to it to a trusted module. Trusted modules must therefore re-validate

```
1 LPCTSTR lpFileName = TEXT("dnscmmc.dll");
2 HMODULE hModule;
3 IUnknown *o1;
4 HRESULT(WINAPI *lpGCO)(REFCLSID, REFIID, LPVOID*);

6 hModule = LoadLibrary(lpFileName);
7 (FARPROC&) lpGCO = GetProcAddress(hModule, "DllGetClassObject");
8 lpGCO(⟨clsid⟩, ⟨iid1⟩, (LPVOID*) &o1);

10 // ... code containing a data corruption vulnerability ...

12 IUnknown *o2;
13 o1→QueryInterface(⟨iid2⟩, (LPVOID*) &o2);
```

Listing 3.2. CODE-COOP attack vulnerability

all code pointers at time-of-use to ensure safety, but this breaks CFI's deployment model because it necessitates rewriting all the system libraries.

3.2.2 CODE-COOP Attacks

Listing 3.2 demonstrates the danger with a common Windows COM programming idiom that is vulnerable to CODE-COOP attack even with CFI protections enabled for all application-provided modules. Lines 6–8 dynamically load a COM library (e.g., dnscmmc.dll) and invoke its `DllGetClassObject` function to obtain an object reference `o1`. Line 13 later obtains a new interface `o2` to the object.

A data corruption vulnerability (e.g., buffer overwrite) in line 10 can potentially allow an attacker to replace `o1`'s vtable with a counterfeit one. CFI protections guarantee that line 13 nevertheless targets a valid `QueryInterface` implementation, but if the process address

space contains any system COM library that has not undergone CFI instrumentation, the attacker can redirect line 13 to an unguarded `QueryInterface`. Since all `QueryInterface` implementations internally call other methods on `o1` (e.g., `AddRef`), the attacker can corrupt those to redirect control arbitrarily.

To demonstrate this, we compiled and executed Listing 3.2 on Windows 10 (Enterprise 1511, build 10586.545) with Microsoft Control Flow Guard (MCFG) (Tang, 2015) enabled, and nevertheless achieved arbitrary code execution. MCFG is a Visual Studio addition that compiles CFI guard code into indirect call sites, including line 13. The guards constrain the sites to a whitelist of destinations. Most Windows 10 system libraries are compiled with MCFG enabled so that their call sites are likewise protected, but many are not. We counted 329 unprotected system libraries on a clean install of Windows 10—many of them in the form of legacy libraries required for backward compatibility. (For example, some have binary formats that predate COFF, and are therefore incompatible with MCFG.) These include dnscmmc.dll (the DNS Client Management Console), which Listing 3.2 exploits. If an attacker can contrive to load any of them (e.g., through dll injection or by corrupting variable lpFileName in line 6), CODE-COOP attacks become threats. Since COM services obtain libraries dynamically and remotely on-demand, replacement of all 329 of the libraries we found with CFI-protected versions is not an antidote—universal adoption of MCFG across all software vendors and all module versions would be required.

Moreover, even universal adoption of MCFG is insufficient because MCFG cannot protect returns in component-based applications, which are the basis of many code-reuse attacks (e.g., ROP). Stronger CFI systems that do protect returns must likewise universally modify all binary components or suffer the same vulnerability. We consider the existence of at least some uninstrumented modules to be a practical inevitability in most deployment contexts; hence, we propose an alternative approach that augments arbitrary existing CFI approaches to safely tolerate such modules without demanding write-access to system code.

3.3 Design

3.3.1 Object Proxying

OFI solves this problem by ensuring that trusted callee modules (i.e., potential deputies) never receive writable code pointers from untrusted, CFI-protected callers. Achieving this without breaking intricate object exchange protocols and without demanding full source code requires careful design. Our solution centers around the idea of *proxy objects*. Each time an object flows across an inter-module trust boundary, OFI delivers a substitute proxy object to the callee. There are two kinds of proxies in OFI:

- *Floor proxy objects* $\lfloor o \rfloor$ are delivered to trusted callees when an untrusted caller attempts to pass them an object o. (Floor objects are so-named because higher-trust tenants see them when "looking down" toward low-trust objects).

- *Ceiling proxy objects* $\lceil o \rceil$ are delivered to untrusted callees when a trusted caller attempts to pass them an object o. (Low-trust tenants see them when "looking up" toward high-trust objects.)

Functions $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ are inverses, so $\lfloor \lceil o \rceil \rfloor = \lceil \lfloor o \rfloor \rceil = o$. Thus, if one tenant passes an object to another, who then passes it back, the original tenant receives back the original object, making the proxying transparent to both parties.

At a high level, proxy objects are *in-lined reference monitors* (IRMs) (Schneider, 2000) that wrap and mediate access to the methods of the objects they proxy. When called, their methods must (1) enforce control-flow and dataflow guards that detect and prevent impending CFI violations, and (2) seamlessly purvey the same services as the object they proxy (whenever this does not constitute an integrity violation). These requirements are known as IRM *soundness* and *transparency* (Hamlen et al., 2006; Ligatti et al., 2009) in the

literature. The soundness property enforced by a proxy object can be formalized as a type-based contract derivable from the method's type signature, as detailed in §3.3.2; transparency is achieved by the proxy's reversion to the original object's programming whenever the contract is satisfied.

When applying OFI to binary code without source code, it is not clear where to inject guard code that introduces these proxy objects. All of the calls in Listing 3.1 take the form of computed jump instructions at the binary level, whose destinations cannot generally be statically predicted. Injecting guard code that accommodates every possible proxy scenario at every computed jump instruction in the program would introduce unacceptable performance overhead.

To avoid this, OFI adopts a lazy, recursive approach to object proxying: At object creation points, OFI substitutes the created objects with proxy objects whose methods are *mediators* that enforce CFI guards before falling through to the proxied object's original programming. The mediators recursively introduce a new layer of proxying for any potentially insecure objects being passed as arguments. Thus, proxying occurs dynamically, on-demand, as each method is called by the various principals and with various object arguments. For example, OFI transforms line 6 so that $\lceil \mathsf{o}_1 \rceil \rightarrow \mathsf{GetResult}$ points to mediator method `GetResult_vaulter`, whose implementation calls $\mathsf{o}_1 \rightarrow \mathsf{GetResult}$ with *this* pointer equal to $\lfloor \lceil \mathsf{o}_1 \rceil \rfloor = \mathsf{o}_1$. When control returns to the mediator, it replaces out-argument o_3 with $\lceil \mathsf{o}_3 \rceil$ and then returns to the untrusted caller. We refer to proxy methods that mediate low-to-high calls followed by high-to-low returns as *vaulters*, and those that mediate high-to-low calls followed by low-to-high returns as *bouncers*.

CODE-COOP attacks that attempt to deputize object recipients by corrupting proxy vtables are thwarted by storing proxy objects entirely within read-only memory. This is possible because proxy objects need no writable data; modern object exchange protocols like COM and CORBA require object recipients to access any data via accessor methods (e.g., to accommodate distributed storage), while the object's creator may access in-memory data fields

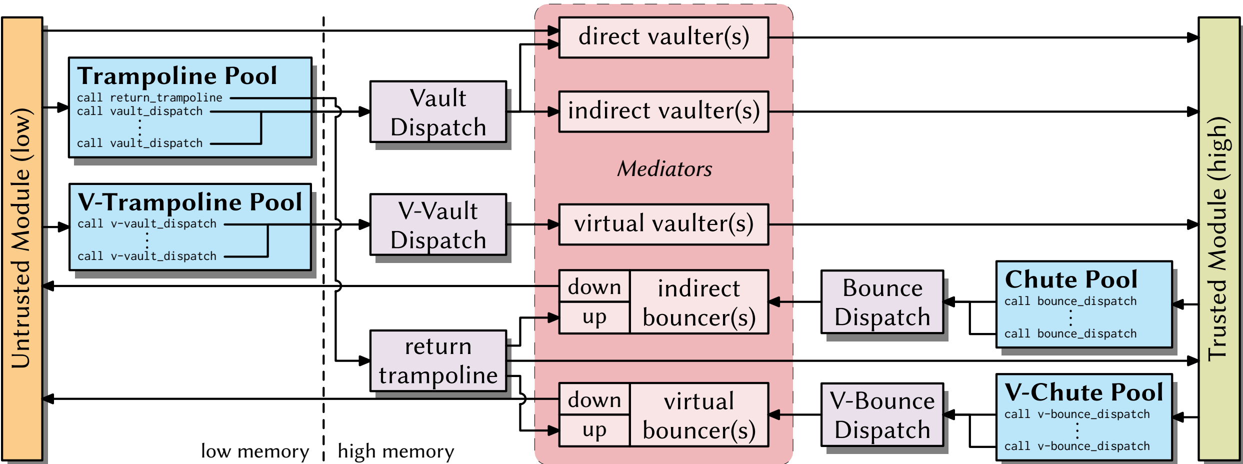

Figure 3.1. Cross-module OFI control-flows

directly. Thus, OFI proxies consist only of a fixed vtable and no data. Moreover, to avoid overhead associated with dynamically allocating them, our design assigns all proxy objects *the same* vtable. This allows the entire proxy object pool to be efficiently implemented as a single, read-only physical page of memory (possibly allocated to multiple virtual pages) filled with the shared vtable's address. Each such vtable pointer constitutes a complete proxy object, ready to be used as a fresh proxy during mediation. The vtable methods all call a central dispatcher method that consults the call stack to determine which proxy object and virtual method is the desired destination, and invokes the appropriate mediator implementation.

Figure 3.1 illustrates the resulting control-flows. When an untrusted module attempts to call a method of a shared object, the code pointer it dereferences points into a *v-trampoline pool* consisting of direct call instructions that all target OFI's *v-vault dispatch* subroutine. The dispatcher pops the return address pushed by the v-trampoline pool to determine the index of the method being called, and consults the stack's *this* pointer to determine the object. Based on this information, it selects and tail-calls the appropriate *virtual vaulter* mediator. The vaulter proxies any in-arguments, calls the trusted module's implementation of the method, then proxies any out-arguments, and returns to the caller.

In the reverse direction, trusted modules call into a *chute pool* that targets OFI's *bounce dispatch* subroutine, which dispatches control to a *virtual bouncer*. To safely accommodate the return of the untrusted callee to the trusted caller (which constitutes a control-flow edge from untrusted code to a non-exported trusted address, which many CFI policies prohibit), the bouncer replaces the return address with the address of a special *return trampoline* that safely returns control to the "up" half of the bouncer implementation.

This approach generalizes to direct untrusted-to-trusted calls and indirect (non-virtual) untrusted-to-trusted calls, which are both represented atop Figure 3.1. Direct calls are statically identifiable by (both source-aware and source-free) CFI, and are therefore statically replaced with a direct call to a corresponding *direct vaulter* implementation. Indirect, inter-module calls dereference code pointers returned by the system's dynamic linking API (e.g., `dlsym()` or `GetProcAddress()` on Posix-based or Windows-based OSes, respectively). OFI redirects these to trampoline pool entries that dispatch appropriate *indirect vaulters*. (Dynamic linking can also return pointers to statically linked functions, in which case the dispatcher targets a direct vaulter.)

Another benefit of this proxy object representation strategy is its natural accommodation of subclassing relationships. Callees with formal parameters of type C_0 may receive actual arguments of any subtype $C_1 <: C_0$; likewise, callers expecting return values or out-arguments of type C_0 may receive objects of any subtype $C_1 <: C_0$. It is therefore essential that proxy objects obey a corresponding subtyping relation that satisfies

$$C_1 <: C_0 \Longrightarrow \left(\lfloor C_1 \rfloor <: \lfloor C_0 \rfloor\right) \wedge \left(\lceil C_1 \rceil <: \lceil C_0 \rceil\right) \tag{3.1}$$

in order to preserve computations that depend on subtyping.

At the binary level, object vtables support inheritance as illustrated in Figure 3.2—ordering method pointers from most to least abstract class allows code expecting a more abstract class to transparently access the prefix of the vtable that is shared among all its

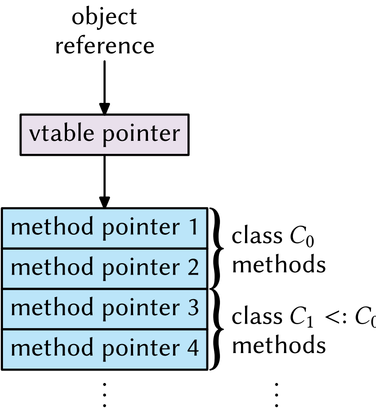

Figure 3.2. Proxy object binary representation

subclasses. Instantiating all proxy objects with a shared, fixed vtable therefore allows all proxy objects to transparently subtype all other proxy objects (since their vtables are identical). This avoids introducing and consulting potentially complex runtime typing information for each object, which would lead to additional overhead related to protecting that information from malicious tampering.

3.3.2 Type-based Contracts

In order to reliably synthesize and interpose its mediation logic into all trust boundary-crossing method calls, OFI must base its mediation on a description of each interface that links the communicating modules. Since interfaces are collections of method type signatures, OFI therefore enforces a *type-based contract* (Findler and Felleisen, 2002) between caller and callee. That is, each trusted interface method's type signature encodes a set of contractual obligations on code pointers that must be enforced by OFI to ensure CFI-compliant operation. This type-theoretic foundation is essential for scalably automating OFI for large interfaces.

Figure 3.3 defines OFI contracts as a core subset of the type system used by major interface description languages, such as MIDL and CORBA IDL (Exton et al., 1997), for component communication. Interface methods have types $\tau \rightarrow_{cc} \tau'$, which denote functions

$$
\begin{aligned}
\tau : \mathcal{U} ::= {} & \bot && \text{(security-irrelevant byte)} \\
& \mid \tau_1 \times \tau_2 && \text{(structures)} \\
& \mid \tau^s && \text{(arrays)} \\
& \mid \tau_1 + \tau_2 && \text{(unions)} \\
& \mid C && \text{(shared object classes)} \\
& \mid \tau \rightarrow_{cc} \tau' && \text{(functions)} \\
& \mid \texttt{[}dir\texttt{]}\tau * && \text{(pointers)} \\
& \mid \Sigma_{(v:\tau)} f && \text{(dependent pairs)} \\
& \mid \mu t.\tau \mid t && \text{(recursive datatypes)}
\end{aligned}
$$

(singleton types)

$$
\begin{aligned}
s ::= {} & n \mid \texttt{ZT} \; \textit{(zero-terminated)} && \text{(array sizes)} \\
n \in {} & \mathbb{N} && \text{(numeric constants)} \\
f : {} & \mathbb{N} \rightarrow \mathcal{U} && \text{(type dependencies)} \\
dir ::= {} & \texttt{in} \mid \texttt{out} \mid \texttt{inout} && \text{(argument directions)} \\
cc ::= {} & \texttt{callee_pop} \mid \texttt{caller_pop} && \text{(calling conventions)}
\end{aligned}
$$

Figure 3.3. A type system for expressing CFI obligations as OFI contracts

from an argument list of type τ to a return value of type τ'. Calling convention annotation cc is used by OFI to preserve and secure the call stack during calls. Classes, structures, and function argument lists are encoded as tuples $\tau_1 \times \tau_2 \times \cdots \times \tau_n$, which denote structures having n fields of types $\tau_1, \ldots, \tau_n$, respectively. For convenience, named classes are here written as named types C (in lieu of writing out their usually large, recursive type signatures). Static-length arrays and zero-terminated strings have repetition types τ^n and $\tau^{\texttt{ZT}}$, respectively. Pointer arguments whose referents are provided by callers (resp. callees) have type $\texttt{[in]}\tau*$ (resp. $\texttt{[out]}\tau*$). Those with a caller-supplied referent that is replaced by the callee before returning use bidirectional annotation `[inout]`. Self- or mutually-referential types are denoted by $\mu t.\tau$, where τ is a type that uses type variable t for recursive reference.

For example, Listing 3.1's `GetResult` method has type

$$
\texttt{GetResult} : (\texttt{[in]}C_{\text{IFD}}* \times \texttt{[out]}C_{\text{ISI}}*) \rightarrow_{\texttt{callee_pop}} \bot^4 \tag{3.2}
$$

where C_{IFD} and C_{ISI} are the types of the `IFileDialog` and `IShellItem` interfaces. This type reveals that a correct vaulter for `GetResult` must replace the first stack argument (i.e., the *this* pointer) with a floor proxy of type $\lfloor C_{\text{IFD}} \rfloor$ before invoking the trusted callee, and then replace the second stack argument with a ceiling proxy of type $\lceil C_{\text{ISI}} \rceil$ before returning to the untrusted caller.

In addition to the usual types found in C, we found that we needed *dependent pair types* $\Sigma_{(v:\tau)} f$ in order to express many API method contracts. Values with such types consist of a field v of some numeric type τ, followed by a second field of type $f(v)$. Function f derives the type of the second field from value v. For example, the contract of `QueryInterface` is expressible as:

$$
\begin{aligned}
\texttt{QueryInterface} : \texttt{[in]}C_{\text{IFD}}* \times \\
\Sigma_{(iid:\bot^{16})}(iid = \langle iid_1 \rangle \Rightarrow \texttt{[out]}C_1* \\
\mid iid = \langle iid_2 \rangle \Rightarrow \texttt{[out]}C_2* \mid \cdots) \rightarrow_{\texttt{callee_pop}} \bot^4
\end{aligned}
\tag{3.3}
$$

This type indicates that the second stack argument is a 16-byte (128-bit) integer that identifies the type of the third stack argument. If the former equals $\langle iid_1 \rangle$, then the latter has type $\texttt{[out]}C_1*$, etc.

There is a fairly natural translation from interface specifications expressed in C/C++ IDLs, such as SAL, to this type system. Products ($\times$), repetition (τ^s), sums ($+$), classes (C), functions ($\rightarrow$), pointers ($*$), and datatype recursion (μ) are expressed in C++ datatype definitions as structures, arrays, unions, shared classes, function pointers/references, and type self-reference (or mutual self-reference), respectively. SAL annotations additionally specify argument directions and array bounds dependencies. Special dependencies involving class and interface identifiers, such as those in `QueryInterface`'s contract, can be gleaned from the system-maintained list of registered classes and interfaces.

OFI contract types are then automatically translated into effective procedures for enforcing the contracts they denote (i.e., mediator implementations). Figure 3.4 details the

$$
\begin{aligned}
\mathcal{E}_x[\![\bot]\!]\, d\, p &= \{\} \\
\mathcal{E}_x[\![\tau_1 \times \tau_2]\!]\, d\, p &= \mathcal{E}_x[\![\tau_1]\!]\, d\, p; \quad \mathcal{E}_x[\![\tau_2]\!]\, d\,(p + |\tau_1|) \\
\mathcal{E}_x[\![\tau^n]\!]\, d\, p &= \Big(n > 0 \Rightarrow (\mathcal{E}_x[\![\tau]\!]\, d\, p; \quad \mathcal{E}_x[\![\tau^{n-1}]\!]\, d\,(p + |\tau|))\Big) \\
\mathcal{E}_x[\![\tau^{\texttt{ZT}}]\!]\, d\, p &= \Big(*p \neq 0 \Rightarrow (\mathcal{E}_x[\![\tau]\!]\, d\, p; \quad \mathcal{E}_x[\![\tau^{\texttt{ZT}}]\!]\, d\,(p + |\tau|))\Big) \\
\mathcal{E}_x[\![\tau_1 + \tau_2]\!]\, d\, p &= \mathcal{E}_x[\![\tau_1]\!]\, d\, p;\, \mathcal{E}_x[\![\tau_2]\!]\, d\, p \\
\mathcal{E}_x[\![\tau \rightarrow_{cc} \tau']\!]\, d\, p &= {}_1 \text{ copy } \tau \text{ from caller to callee;} \\
&\quad {}_2\ \mathcal{E}_x[\![\tau]\!]\ (\texttt{in})\ (\&\text{callee_frame}); \\
&\quad {}_3\ r := \text{call } p; \\
&\quad {}_4\ \mathcal{E}_{x^{-1}}[\![\tau']\!]\ (\texttt{out})\ (\&r); \\
&\quad {}_5\ \mathcal{E}_{x^{-1}}[\![\tau]\!]\ (\texttt{out})\ (\&\text{caller_frame}); \\
&\quad {}_6 \text{ pop } \tau \text{ from } \mathit{opposite}(cc) \text{ stack;} \\
&\quad {}_7 \text{ return } r \\
\mathcal{E}_x[\![\texttt{[}\mathit{dir}\texttt{]}\,\tau *]\!]\, d\, p &= \Big((\mathit{dir} \in \{d, \texttt{inout}\} \wedge *p \neq 0) \Rightarrow \\
&\text{match } \tau \text{ with } (_ \rightarrow _) \Rightarrow *p := \&(\mathcal{E}_{x^{-1}}[\![\tau]\!]\,(\texttt{in})\,(*p)) \\
&\qquad \mid\ C \Rightarrow *p := x(*p) \\
&\qquad \mid\ _ \Rightarrow \mathcal{E}_x[\![\tau]\!]\, d\,(*p)\Big) \\
\mathcal{E}_x[\![\Sigma_{(v:\tau)} f]\!]\, d\, p &= \mathcal{E}_x[\![\tau]\!]\, d\, p; \quad \mathcal{E}_x[\![f(*p)]\!]\, d\,(p + |\tau|) \\
\mathcal{E}_x[\![\mu t.\tau]\!]\, d\, p &= \mathcal{E}_x[\![\tau[\mu t.\tau/t]]\!]\, d\, p
\end{aligned}
$$

Figure 3.4. Mediator enforcement of OFI contracts

translation algorithm in the style of a denotational semantics[3] where $\mathcal{E}_x[\![\tau]\!]\, d\, p$ yields a procedure for enforcing the contract denoted by type τ with proxying function $x \in \{\lfloor\cdot\rfloor, \lceil\cdot\rceil\}$ in call-direction $d \in \{\texttt{in}, \texttt{out}\}$ on the bytes at address p.

For example, valuation function $\mathcal{E}_{\lfloor\cdot\rfloor}[\![\tau_{\texttt{GetResult}}]\!](\texttt{in})(\&\texttt{GetResult})$ yields the implementation of `GetResult_vaulter`, where $\tau_{\texttt{GetResult}}$ is the type in equation 3.2. The implementation first copies caller stack frame τ to a secure callee-owned stack (line 1). It then enforces the in-contract for τ (line 2), which replaces the argument of type C_{IFD} with a proxy of type $\lfloor C_{\text{IFD}} \rfloor$, before invoking `GetResult` (line 3). Upon return, the out-contracts for return type τ' and frame τ are enforced (lines 4–5). In this case, return type $\tau' = \bot$[4] is security-irrelevant,

[3]Here, notation $|\tau|$ denotes the size of data having type τ.

but the out-contract for τ demands replacing stack object C_{ISI} with proxy $\lceil C_{\mathrm{ISI}} \rceil$. Finally, the frame of the participant (*viz.*, caller or callee) that did not already clean its stack is popped (line 6), and control returns to the caller (line 7). (The first and last steps are required because OFI separates untrusted and trusted stacks for memory safety, temporarily duplicating the shared frame.)

Each contract enforcement (lines 2, 4, and 5) entails recursively parsing the binary datatypes of Figure 3.3 and substituting code pointers with pointers to mediators that enforce the proper contracts. Structure, array, and union contracts are enforced by recursively enforcing the contracts of their member types. Function pointer contracts are enforced by lazily replacing them with mediator pointers, shared class contracts are enforced by proxying, and other pointer contracts are enforced by eagerly dereferencing the pointer and enforcing the pointee's contract. Dependent pairs are enforced by resolving the dependency to obtain the appropriate contract for the next datum. Finally, recursive types are enforced as a loop that lazily unrolls the type equi-recursively (Crary et al., 1999).

An OFI implementation can enforce the contract implied by a trusted interface by implementing mediator algorithm $\mathcal{E}_{\lfloor\cdot\rfloor}[\![\tau \rightarrow_{cc} \tau']\!](\mathtt{in})$ for each method signature $\tau \rightarrow_{cc} \tau'$ in the interface. Such mediators are vaulter implementations. Some rules in Figure 3.4 invert proxy function x, prompting the enforcement to also implement bouncer mediators of the form $\mathcal{E}_{\lceil\cdot\rceil}[\![\tau \rightarrow_{cc} \tau']\!]$. These mediate callbacks, such as those commonly used in event-driven programming. Bouncers also mediate methods by which trusted modules initiate unsolicited contact with untrusted modules, such as those that load untrusted libraries and invoke their initializers.

3.3.3 Trust Model

OFI's attacker model assumes that original, untrusted modules may be completely malicious, containing arbitrary native code, but that they have been transformed by CFI/SFI into code

compliant with the control-flow policy. The transformed code monitors and constrains all security-relevant API calls and their arguments as long as control-flow stays within the sandbox (cf., Abadi et al. (2009); Wartell et al. (2012b)). Malicious apps must therefore first escape the control-flow sandbox before they can abuse system APIs to do damage. OFI blocks escape attempts that abuse call sites in immutable modules that depend on objects or code pointers supplied by instrumented modules. It thereby extends whatever policy is enforced by the underlying CFI/SFI mechanism to those call sites. In order to defeat CODE-COOP attacks, the underlying CFI/SFI must therefore enforce a COOP-aware policy (Schuster et al., 2015) for OFI to extend (see §6.5).

Control-flow policies consist of a (possibly dynamic) graph of whitelisted control-flow edges that is consulted and enforced by CFI/SFI guard code before each control-flow transfer from untrusted modules (but not before those from trusted modules). OFI requires that this graph omit edges directly from low- to high-trust modules; such edges must be replaced with edges into OFI's trampoline pools, to afford OFI complete mediation of such transfers.

A facility for read-only, static data is required for OFI to maintain tamper-proof proxy objects. This can be achieved by leveraging CFI/SFI to restrict untrusted access to the system's virtual memory API—untrusted modules must not be permitted to enable write-access to OFI-owned data or code pages.

To prevent untrusted modules from directly tampering with trusted modules' data, some form of memory isolation is required. SFI achieves this by sandboxing all memory-writes by untrusted modules (e.g., Wahbe et al. (1993); McCamant and Morrisett (2006)). CFI leverages control-flow guards to enforce atomic blocks that guard memory-writes (e.g., Kuznetsov et al. (2014); Erlingsson et al. (2006); Nagarakatte et al. (2010)).

Data fields of shared objects are conservatively treated as private; non-owners must access shared object data via accessor methods. This is standard for interfaces that support computing contexts where object locations cannot be predicted statically (e.g., in a distributed computations), including all COM interfaces. This affords the accessor methods

an opportunity to dynamically fetch or synchronize requested data fields when they are not available locally.

Our design of OFI is carefully arranged to require almost no persistent, writable data of its own, eliminating the need to protect such data within address spaces shared by OFI with malicious modules. In multithreaded processes, OFI therefore conservatively stores its temporary data in CPU registers or other secured, thread-local storage spaces. There are three exceptions:

Dynamic CFGs. If the control-flow policy is dynamic (e.g., new edges become whitelisted during dynamic linking), then OFI requires a safe place to store the evolving policy graph. This is typically covered by the underlying SFI/CFI's self-integrity enforcement mechanisms.

Object Inverses. A small hash table associating objects with their proxies is required, in order to compute inverses $\lfloor\lceil\cdot\rceil\rfloor$ and $\lceil\lfloor\cdot\rfloor\rceil$. This can be confined to dedicated memory pages, admitting the use of efficient, OS-level memory protections. For example, on Windows desktop OSes we allocate a shared memory mapping to which a separate *memory-manager process* has write access, but to which the untrusted process has read-only access. OFI modules residing in untrusted processes can then use lightweight RPC to write to the hash table. CFI protections prevent untrusted modules from accessing the RPC API to perform counterfeit writes.

Reference Counts. To prevent double-free attacks, in which an untrusted module improperly frees objects held by trusted modules, object proxies maintain reference counts independent from the objects they proxy. When the proxy is first created, OFI increments the proxied object's reference count by one. Thereafter, acquires and releases of the proxy are not reflected to the proxied object; they affect only the proxy object's reference count. When the proxy's reference count reaches zero, it decreases the proxied object's reference count by one and frees itself. Proxy object reference counters are stored within the secure hash table entries (see above) to prevent tampering.

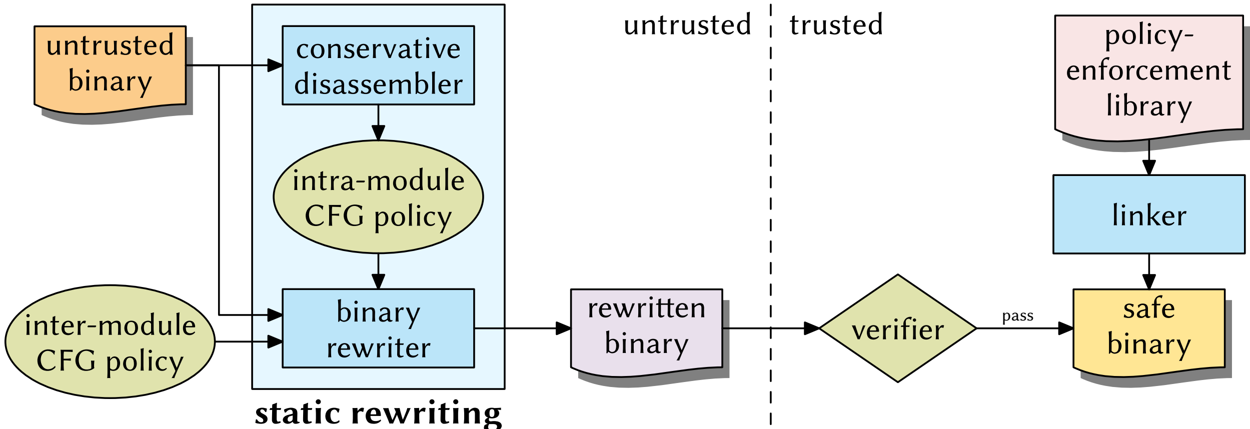

Figure 3.5. REINS system architecture

3.4 Implementation

3.4.1 Architecture

Our prototype implementation of OFI extends the REINS system (Wartell et al., 2012b). We chose REINS because it realizes fully source-free SFI+CFI (including no reliance on symbol files), and it supports Windows platforms. This affords an aggressive evaluation of OFI's design in austere contexts that lack the benefits of source code and that must support extensive, complex object-oriented APIs, such as COM. Prior to the introduction of OFI enhancements, REINS could not support COM-dependent features of any target application; triggering such features induced its CFI protections to prematurely abort the application with a security violation.

Figure 3.5 depicts the system architecture. Untrusted native code binaries are first disassembled to obtain a conservative control-flow graph (CFG) policy. The policy dictates that only the control-flow paths statically uncovered and analyzed by the disassembly process are permissible. A binary rewriting module then injects guard code at all control-flow transfer sites to constrain all flows to the CFG.

OFI is agnostic to the particular guard code used to realize SFI/CFI, so we here assume merely that the underlying SFI/CFI implementation protects each control-flow transfer in-

struction with arbitrary (sound) code pointer validation or sanitization logic (see §6). (Reins employs SFI-style *chunking* and *masking* (McCamant and Morrisett, 2006) for efficient sandboxing of intra-module flows, followed by CFI-style whitelisting of inter-module flows. This could be replaced with more precise but less efficient CFI-only logic without affecting OFI.) A separate verifier module independently validates control-flow safety of the secured binary code. This shifts the large, unvalidated rewriting implementation out of the trusted computing base.

Aside from adjusting the control-flow policy to incorporate OFI mediation, OFI extensions inhabit only the *policy enforcement library* portion of the architecture; no change to the disassembly, rewriting, verification, or linking stages was required. This indicates that OFI can be implemented in a modular fashion that does not significantly affect the underlying SFI/CFI system's internals.

The enhancements to the policy enforcement library introduce the inter-module control-flow paths depicted in Figure 3.1. Their implementations are detailed below.

3.4.2 Dispatcher Implementation

3.4.2.1 Vault Dispatch

OFI's Vault Dispatch subroutine directs control from a non-virtual trampoline to a corresponding vaulter. Listing 3.3 sketches its implementation. The index of the calling trampoline is first computed from the return address passed by the trampoline to the dispatcher (lines 2–5). Reins allocates exactly one trampoline in the pool for each non-virtual, trusted callee permitted as a jump destination by the policy. The index therefore unambiguously determines the correct vaulter for the desired callee (line 6). No CFI guards are needed here because CFI guard code in-lined into the untrusted call site has already constrained the flow to a permissible trampoline entry. Finally, the dispatcher tail-calls the selected vaulter (line 9).

```
1  void VaultDispatch() {
2    __asm pop eax
3    PROLOGUE // create secure stack frame
4    __asm mov ret_addr, eax
5    index = (trampoline_pool_base − ret_addr) / TRAMPOLINE_SIZE;
6    vaulter_addr = get_vaulter(index);
7    __asm mov eax, vaulter_addr
8    EPILOGUE // pop secure stack frame
9    __asm jmp eax
10 }
```

Listing 3.3. Vault Dispatch implementation (abbreviated)

The implementation therefore enforces the control-flow policy in four steps: (1) CFI guard code at the call site ensures that the call may only target trampolines assigned to permissible trusted callees. (2) The dispatcher implementation exclusively calls the vaulter that mediates the CFI-validated callee. (3) The vaulter implementation enforces the callee's OFI contract and exclusively calls the callee it guards. (4) The trusted callee never receives caller-writable object vtables; it only receives immutable proxy objects whose methods re-validate call destinations at time-of-callback. This secures the trusted callee against attacks that try to corrupt or replace the underlying object's vtable.

3.4.2.2 V-Vault Dispatch

Dispatching virtual calls is similar but requires more steps. Listing 3.4 sketches its implementation. In this case the caller-provided *this* pointer is retrieved along with the trampoline index (lines 3 and 6). Since the destination is a vaulter, valid *this* pointers are always ceiling

```
 1 void VVaultDispatch() {
 2   __asm pop ecx
 3   __asm mov eax, [esp+4]
 4   PROLOGUE // create secure stack frame
 5   __asm mov ret_addr, ecx
 6   __asm mov ceiling_proxy_object, eax
 7   index = (vtrampoline_pool_base − ret_addr) / TRAMPOLINE_SIZE;
 8   trusted_object = floor(ceiling_proxy_object);
 9   if (!trusted_object) security_violation();
10   v_vaulter = get_v_vaulter(ceiling_proxy_object, index);
11   __asm mov eax, trusted_object
12   __asm mov [ebp+8], eax
13   __asm mov eax, v_vaulter
14   EPILOGUE // pop secure stack frame
15   __asm jmp eax
16 }
```

Listing 3.4. Virtual Vault Dispatch implementation (abbreviated)

proxy objects. OFI applies the floor mapping ($\lfloor\cdot\rfloor$) to recover a reference to the trusted function it proxies (line 8). If this fails, a counterfeit object is detected, so OFI aborts with a security violation (line 9). Otherwise the correct vaulter is computed from the ceiling proxy and the index (line 10), the callee's *this* pointer is replaced with the proxied object (lines 11–12), and the vaulter is tail-called (line 15).

3.4.2.3 Bounce Dispatch

Dispatching non-virtual flows from trusted to untrusted modules is analogous to the vault dispatching procedure (Listing 3.3), except that the indexing is into the chute pool rather

```
1  void Bouncer() {
2    PROLOGUE // create untrusted callee stack frame

4    // switch to new fiber for down part of bouncer
5    childinfo[0] = &parent_stack;
6    childinfo[1] = argsize;
7    childinfo[2] = untrusted_callee_addr;
8    childfiber = CreateFiber(0, BouncerDown, childinfo);
9    SwitchToFiber(childfiber);

11   // up part of the bouncer: return from untrusted callee
12   DeleteFiber(childfiber);
13   r = TlsGetValue(tlsindex);
14   enforce_ret_contract(r); // run E_⌊·⌋⟦τ′⟧out (see Fig. 3.4)
15   enforce_out_contract(); // run E_⌊·⌋⟦τ⟧out (see Fig. 3.4)

17   // clean stack and return to trusted caller
18   __asm mov eax, r
19   __asm mov ecx, argsize
20   EPILOGUE // pop secure stack frame
21   __asm pop edx
22   __asm add esp, ecx
23   __asm push edx
24   __asm ret
25 }
```

Listing 3.5. Bouncer implementation (abbreviated)

```
1  void BouncerDown() {
2    // initialize callee stack
3    __asm sub esp, childinfo[1]
4    __asm mov esi, childinfo[0]
5    __asm mov edi, esp
6    __asm rep movs byte ptr [edi], byte ptr [esi]
7    __asm push offset return_trampoline

9    enforce_in_contract(); // run  E⌈·⌉⟦τ⟧(in)  (see Fig. 3.4)

11   __asm mov eax, childinfo[2]
12   CFI_VALIDATE(eax)
13   __asm jmp eax
14 }
```

Listing 3.6. BouncerDown implementation (abbreviated)

than the trampoline pool, and the dispatcher targets bouncers rather than vaulters. The callee-provided return address is also replaced with the address of OFI's return trampoline, so that it can mediate the return.

The bouncer implementation(s) invoked by the dispatcher (see Listing 3.5) first switch to a fresh, callee-writable stack (lines 2–9), to prevent the untrusted callee from corrupting trusted caller-owned stack frames before it returns. SFI memory guards prevent the callee from writing into the protected, caller-owned stack. OFI contracts carry sufficient information to implement this stack-switching transparently. For example, the contracts reveal the size of the topmost (shared) activation frame and the calling convention, allowing that frame to be temporarily replicated on both stacks.

```
1  void VBouncerDown() {
2    // initialize callee stack
3    __asm sub esp, childinfo[1]
4    __asm mov esi, childinfo[0]
5    __asm mov edi, esp
6    __asm rep movs byte ptr [edi], byte ptr [esi]
7    __asm push offset return_trampoline

9    enforce_in_contract(); // run ℰ⌈·⌉⟦τ⟧in (see Fig. 3.4)

11   // get virtual function address through ''this'' argument
12   __asm mov eax, [esp+4]
13   __asm mov eax, [eax]
14   __asm mov eax, [eax+childinfo[2]]
15   CFI_VALIDATE(eax)
16   __asm jmp eax
17 }
```

Listing 3.7. Virtual Bouncer-down implementation (abbreviated)

To facilitate efficient stack-switching, we leverage the Windows *Fibers* API (Duffy, 2008). In the trusted-to-untrusted direction, we first create a child fiber. The fiber's stack is arranged so that its return address targets the return trampoline, and the "down" part (see Listing 3.6) of the bouncer implementation (lines 1–14) is the child fiber's start address.

The "down" implementation copies the arguments to the new stack (lines 3–6) and then enforces the relevant typing contract on in-arguments (line 9) as described in §3.3.2, before

falling through to the untrusted callee (lines 11–13). Crucially, the underlying object's method pointer is re-validated at time-of-call (line 12), to thwart CODE-COOP attacks.

On return, the return trampoline switches back to the parent fiber, which invokes the "up" half of the bouncer (lines 12–24). This enforces the typing contracts for return values and out-arguments (lines 14–15) as described in §3.3.2 before returning to the trusted caller.

3.4.2.4 V-Bounce Dispatch

Dispatching virtual calls from trusted to untrusted modules is analogous to the bouncer dispatching procedure (Listing 3.7), except that the child is passed a vtable index rather than a callee entry point address. An extra step is therefore required within the "down" implementation to recover the correct callee method address from the "this" pointer's vtable (lines 12–14). Again, the result is re-validated at time-of-call (line 15) to block CODE-COOP attacks.

3.4.2.5 Return Trampoline

Whenever the trusted caller goes through a bouncer to an untrusted callee, the bouncer creates a new stack in which the return address targets OFI's return trampoline. CFI guards for inter-module return instructions must therefore permit flows to the return trampoline in place of the validated return address. For example, if the underlying CFI system enforces return-flows via a shadow stack, it must validate the return address on the shadow stack as usual, but then allow returning to the return trampoline instead. The return trampoline flows to the "up" half of the bouncer mediator, which returns to the CFI-validated return address stored on the shadow stack. This is the only piece of OFI's implementation that requires explicit cooperation from the underlying CFI implementation.

3.4.3 Automated Mediator Synthesis

When trusted interfaces are specified in a machine-readable format, mediator implementations for them can be automatically synthesized from callee type signatures (see §3.3.2). Such automation becomes a practical necessity when interfaces comprise thousands of methods or more.

Unfortunately, the only machine-readable specifications of many real-world APIs are as C++ header files, which can be quite complex due to the power of C's preprocessor language, compiler-specific pragmas, and compiler-predefined macros. For example, the `Windows.h` header, which documents the Windows API, defines millions of symbols and macros spanning hundreds of files, and is not fully interpretable by any tool other than Microsoft Visual C++ in our experience. The best tools for parsing them are the C++ compilers intended to consume them.

We therefore innovated a strategy of conscripting C++ compilers to interpret interface-documenting header files for us, using the resulting information to automatically synthesize mediation library code. Our strategy achieves static reflective programming for C++ without modifying the compiler, language, or header files. Specifically, our synthesis tool is a C++ program that `#include`s interface headers, and then reflects over itself to inspect function prototypes, structures, and their types. To achieve reflection on structures (which is not supported by C++17 (Chochlík and Naumann, 2016)) the program reads its own symbol file in a multi-pass compilation.

Figure 3.6 illustrates the synthesis process. The interface header, list of exported functions (dumped from the trusted library's export table), and synthesizer source code are first compiled to produce a debug symbol file (e.g., PDB file). Our Reflector tool parses the symbol file to produce C++ templates that facilitate first-class access to the static types of all constituent structure and class members. By including the resulting templates into a second compilation pass, the program reflects upon itself and synthesizes the source code for

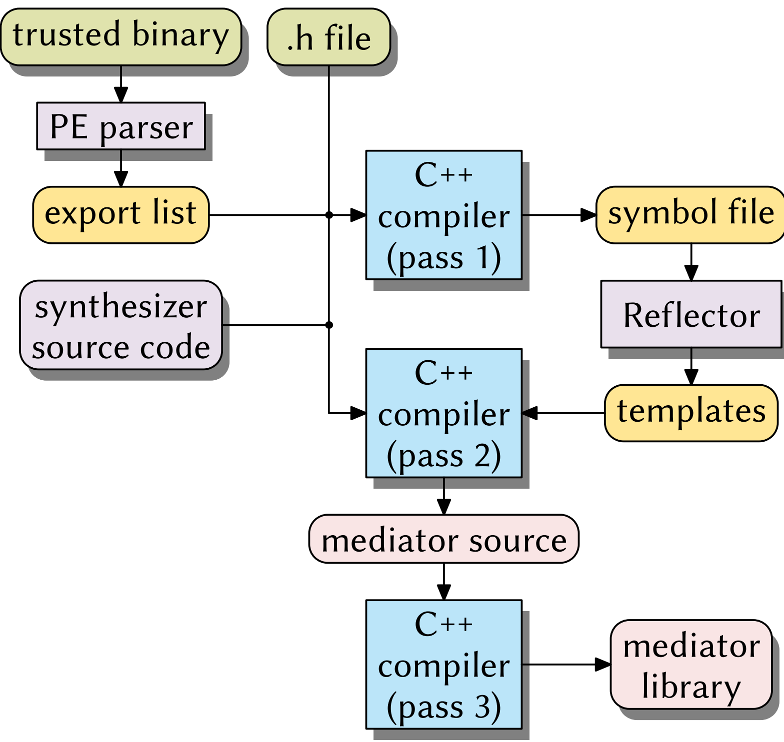

Figure 3.6. Automated mediator synthesis

appropriate mediation code (*viz.*, vaulters and bouncers). A third compilation pass applied to this synthesized mediation code yields the final mediation library.

As an example, Listing 3.8 shows an automatically synthesized vaulter implementation for the `TrySubmitThreadpoolCallback` Windows API function. In this case, the synthesizer has discovered that the trusted callee treats the top stack argument as a code pointer to an untrusted callee expecting 8 bytes of stack arguments (line 2). In addition, stack offset 8 holds a pointer to a structure which, if non-null (line 5), contains two more code pointers to untrusted callees expecting 8 bytes of stack arguments each (lines 6–7). Finally, since

```
1 void TrySubmitThreadpoolCallback_vaulter(char x) {
2   fix_pointer(&x, 8);
3   { _TP_CALLBCK_ENVIRON_V3* x =
4         *((_TP_CALLBACK_ENVIRON_V3**) (&x + 8));
5     if (x) {
6       fix_pointer(&x→CleanupGroupCancelCallback, 8);
7       fix_pointer(&x→FinalizationCallback, 8);
8     }
9   }
10  EPILOGUE // pop stack frame
11  __asm jmp TrySubmitThreadpoolCallback
12 }
```

Listing 3.8. Synthesized vaulter implementation

no out-arguments or return values need sanitization, the vaulter safely tail-calls the trusted callee for more efficient dispatch (line 11).

The typing information necessary to synthesize this implementation is exposed by our Reflect tool as a template of the form shown in Listing 3.9. The template introduces `Reflect<`τ`>::specialize` as a general mechanism for specializing polymorphic template functions to the particular field types of any desired structure type τ. Specifically, lines 7–10 declare a function parameter f whose arguments are specialized to the field types of τ. When called, `Reflect<`τ`>::specialize`(o, f) therefore calls f with a series of pointer arguments specialized to the types and locations of object o's fields. (Reference o is used only for pointer arithmetic, so need not be an actual object instance.)

The specialized polymorphic function can then iterate over its type parameters using SFINAE programming idioms (Vandevoorde and Josuttis, 2002). For example, Listing 3.10 uses the template to generate OFI mediator code to secure a security-relevant structure argument

```
1 typedef _TP_CALLBACK_ENVIRON_V3 typ1162;

3 template<> struct Reflect<typ1162> {
4   template <typename RetTyp>
5   static inline auto specialize(
6     typ11623 *obj,
7     auto(f)(decltype(typ1162::Version)*, . . . ,
8             decltype(typ1162::CleanupGroupCancelCallback)*,
9             decltype(typ1162::FinalizationCallback)*, . . .
10           )→RetTyp
11  )→RetTyp
12  {
13    return f(
14      &(obj→Version), . . . ,
15      &(obj→CleanupGroupCancelCallback),
16      &(obj→FinalizationCallback), . . .
17    );
18  }
19 }
```

Listing 3.9. Reflective template (abbreviated)

to an API function. Lines 1–2 first prototype a generic recursive template function that will recurse over all fields of an arbitrary structure. Lines 4–5 define the base case of zero fields. Lines 7–11 implement the particular case of enforcing the contract for a field of type `FARPROC` (i.e., generic function pointer field). (This is just one representative case; the full implemen-

```
1 template <typename... FieldTypes>
2 void enforce_contract(FieldTypes...);

4 template <>
5 void enforce_contract() {}

7 template <typename... FieldTypes>
8 void enforce_contract(FARPROC *field1, FieldTypes... rest) {
9   // Generate C code to enforce FARPROC contract here...
10  enforce_contract(rest);
11 }

13 Reflect<typ1162>::specialize<void>((typ1162*)0, enforce_contract);
```

Listing 3.10. Mediator synthesis via template recursion

tation has cases for all the types in Figure 3.3.) Code in line 9 treats argument `field1` as an index into the object layout where the field resides at runtime on the stack or heap.

Line 13 demonstrates specializing the generic template to a particular class type. The `Reflect` template in Listing 3.9 is applied to specialize the generic `enforce_contract` template. This allows mediation code for tens of thousands of API methods to be automatically synthesized from just a few hundred lines of hand-written template code, keeping OFI's trusted computing base relatively small and manageable.

3.5 Evaluation

Performance evaluation of OFI on CPU benchmarks (e.g., SPEC CPU2006) exhibits no measurable overhead because CPU benchmarks do not typically access object-oriented system

APIs within loops, which is where OFI introduces overhead. To evaluate the effectiveness of OFI, we therefore tested our prototype with the set of binaries listed in Table 3.1. The test binaries were chosen to be small and simple enough to be amenable to fully automated binary reverse engineering and instrumentation (whose efficacy is orthogonal to OFI), yet reliant upon large, complex system APIs representative of typical consumer software (and therefore an appropriate test of our approach's practical feasibility). All experiments detailed below were performed on an Intel Xeon E5645 workstation with 24 GB RAM running 64-bit Windows 7. We have no source code for any of the test binaries.

Column 2 reports a count of the total number of libraries loaded (statically and dynamically) by each test program, and column 3 reports a count of all methods exported by those libraries. On average, each program loads 12 libraries that export about 7,500 trusted methods. Taking these statistics into consideration, although the test binaries are small-to-moderate in size, the trusted interfaces that must be supported to accommodate them are large. In total, we need to mediate the interfaces of 54 trusted system libraries that collectively expose 18,059 trusted methods, many of which have challenging method signatures involving code pointers, recursive types, class subtyping, dependent types, and object (or object-like) data structures.

3.5.1 Transparency

Without OFI extensions, none of the test programs ran correctly after CFI instrumentation. All COM-dependent operations—including dialog boxes, certain menus, and in some cases even application start-up—failed with a control-flow violation.

After adding OFI to the instrumentation, we manually tested all program features systematically. All features we tested exhibited full functionality. While we cannot ensure that such testing is exhaustive, we consider it similar to the level of quality assurance to which such applications are typically subjected prior to release.

Table 3.1. Interactive COM applications used in experimental evaluation

Binary Program	# DLLs	# Interface Funcs	File Size			Code Segment Size			Rewriting Times (s)
			Old (KB)	**New** (KB)	**Increase** (%)	**Old** (KB)	**New** (KB)	**Increase** (%)	
calc	17	11,755	758	1,263	67	330	514	56	19.00
cmd	8	7,321	296	521	76	139	225	62	5.85
explorer	27	15,324	2,555	3,611	41	701	1,056	51	29.60
magnify	16	14,073	615	751	22	91	136	50	4.74
MCFG_Exploit	7	2,074	11	34	209	4	23	475	0.89
minesweeper	8	6,560	117	153	31	16	35	119	1.08
notepad	14	10,441	176	248	41	43	72	67	2.21
osk	19	13,662	631	849	35	147	218	48	7.10
powershell	9	6,318	442	489	11	36	47	31	2.32
solitaire	8	6,379	56	109	95	24	54	125	1.44
WinRAR	17	7,536	1,374	2,928	113	1,008	1,554	54	70.92
wmplayer	9	6,997	161	186	16	12	25	108	0.84
median	12	7,429	369	505	41%	67	104	59%	3.53s

3.5.2 Performance Overheads

Rewriting Time and Space Overheads. Table 3.1 reports the percentage increase of the file size and code segment, as well as the time taken by OFI to rewrite each binary. Our prototype rewrites about 60KB of code per second on average. A rewritten binary increases in size by about 41%. Code segment sizes increase by about 59%. The large percentage increases exhibited by the MCFG_Exploit experiment (209% and 475%, respectively) are artifacts of the exceptionally small size of that program. (It is the synthetic MCFG exploit test reported in Section 3.2.2.)

Runtime Performance. Figure 3.7 reports OFI runtime overheads of the programs in Table 3.1. Since almost all object exchanges occur during application startup and in response to user events (e.g., mouse clicks), we created macros that open, manipulate, and close each test program as rapidly as possible. By running such a simulation in a loop for 1000 iterations, we obtain an average running time. We measure the runtime overhead imposed by OFI as the ratio of time spent within the OFI modules to the total runtime.

The median overhead is 0.34%; and no program has overhead larger than 2.00%, except for MCFG_Exploit—our proof-of-concept CODE-COOP exploit implementation. Its size is small, and its only runtime operation initializes a COM object, which involves OFI mediation, resulting in abnormally high percentage overheads. The remaining tests are common consumer apps. Of these, the calculator program returns the worst overhead of 1.82%. This is due to the fact that switching the calculator's mode between standard, scientific, programmer, and statistics requires frequent OFI mediation. Each such switch reconstructs the GUI via 3,500 method calls that involve shared code and/or object pointers, and thus OFI mediation. Nevertheless, we consider the 1.82% overhead to be modest and unnoticeable by users. All other test programs have runtime overheads below 1%, and the calculator's <2% worst-case overhead only occurs on mode changes.

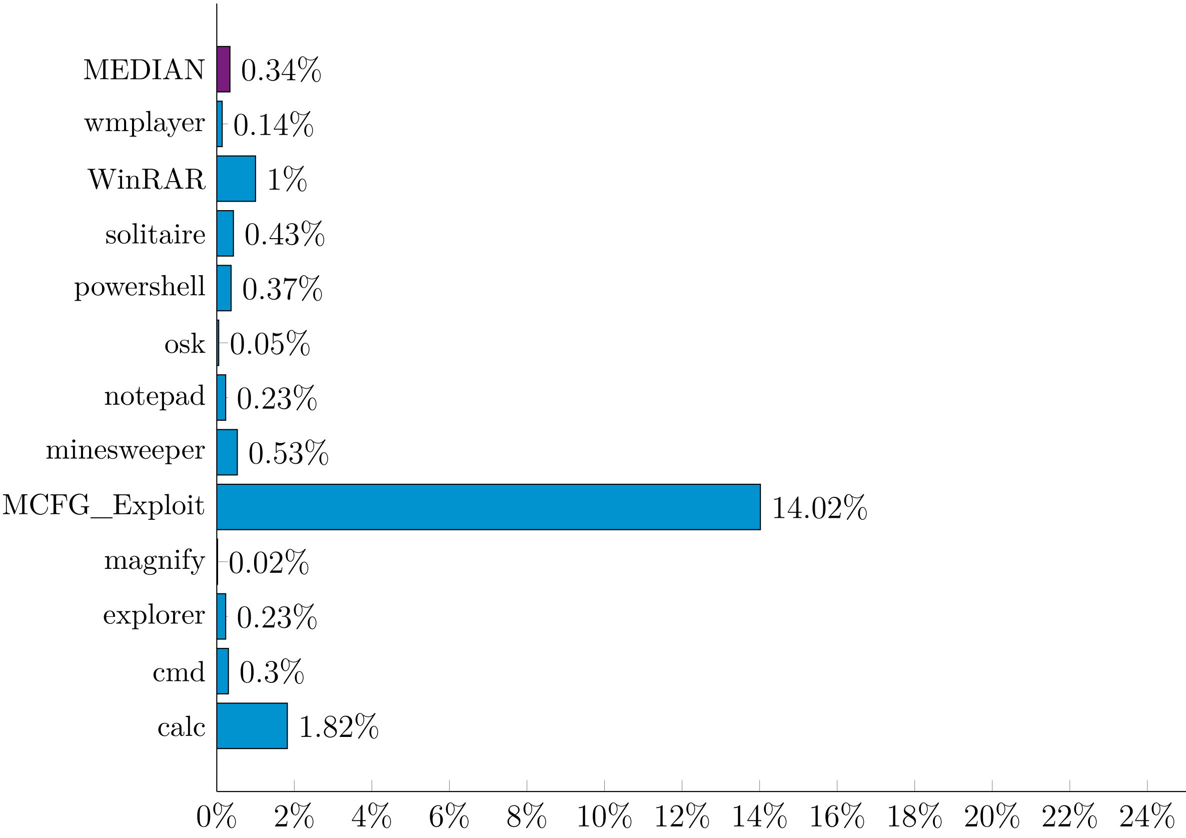

Figure 3.7. OFI runtime overhead

The performance overheads reported in Figure 3.7 attempt to measure semi-realistic usage scenarios for user-interactive applications, which tend to be the ones that use COM the most. However, to derive a worst-case performance bound for OFI, we also created a set of micro-benchmarks. Each implements a non-interactive program that creates, manipulates, and destroys COM objects in a tight loop. Technical details for each benchmark are provided in Table 3.2. Although not realistic, these tests can measure the extreme worst-case scenario that a program constantly crosses the trust boundary without performing any other computation, triggering OFI mediation continuously.

Micro-benchmarking yielded a median overhead of 32.44%, with a maximum of just over 50%. We know of no realistic application that would exhibit these overheads in practice, but

Table 3.2. Micro-benchmark overheads

No.	Description	Interfaces	Functions	Overhead
#1	(1) creates object, (2) destroys object	IUnknown	::Release()	50.67%
#2	(1) creates object, (2) raises and lowers ref count, (3) destroys object	IUnknown	::AddRef(), Release()	41.30%
#3	(1) creates open dialog obj, (2) adds controls, (3) destroys object	IFileOpenDialog IFileDialogCustomize	::QueryInterface(), Release() ::AddPushButton(), AddMenu(), AddText(), AddControlItem(), Release(),	41.69%
#4	(1) creates open dialog obj, (2) binds file to shell object, (3) retrieves file path, (4) destroys all objects	IFileOpenDialog IShellItem	::SetFileName(), Show(), GetResult(), Release() ::GetDisplayName(), Release()	32.22%
#5	(1) creates open dialog obj, (2) binds files to array, (3) binds elements to shell objects, (4) retrieves the file paths, (5) destroys all objects	IFileOpenDialog IShellItemArray IShellItem	::GetOptions(), SetOptions(), SetFileName(), Show(), GetResults(), Release() ::GetCount(), GetItemAt(), Release() ::GetDisplayName(), Release()	32.44%
#6	(1) creates open dialog obj, (2) binds file to shell object, (3) creates save dialog object, (4) sets save-as default, (5) binds saved file to new shell object, (6) retrives path of new shell object, (7) destroys all objects	IFileOpenDialog IShellItem IFileSaveDialog	::SetFileName(), Show(), GetResult(), Release() ::GetDisplayName(), Release() ::SetSaveAsItem(), SetFileName(), Show(), GetResult(), Release()	31.76%
#7	(1) creates save dialog object, (2) binds saved file to shell object, (3) retrieves the file path, (4) creates shell link object, (5) sets path as link target, (6) saves link in persist storage, (7) destroys all objects	IFileSaveDialog IShellLink IPersistFile	::SetFileName(), Show(), GetResult(), Release() ::SetPath(), SetDescription(), QueryInterface(), Release() ::Save(), Release()	31. 69%

Table 3.3. Attack simulation results

Binary Program	# Attacks	Security Aborts Within Callee	After Return	Within OFI
calc	5	1	4	0
MCFG_Exploit	1	0	0	1
notepad	5	0	5	0
powershell	5	1	4	0
WinRAR	5	3	2	0

they reveal the overhead of instrumentation relative to the non-instrumented inter-module control-flow paths.

3.5.3 Security Evaluation

To assess OFI's response to attacks, we launched synthetic vtable corruption and COOP attacks against some programs rewritten by our prototype. We simulate COOP attacks by temporarily modifying the v-vault dispatcher to occasionally choose the wrong vaulter. This simulates a malicious caller who crafts a counterfeit object whose vtable pointer identifies a structurally similar (e.g., similarly typed) vtable but not the correct one.

Table 3.3 reports the attack simulation results. Each program in column 1 is exposed to 5 attacks. In each case, the attack quickly results in a security abort and premature termination; no control-flow policy violations were observed. Among the 20 attacks, the callee aborted in 5 cases (column 3), and the caller aborted after return in 15 cases (column 4).

Most of the security aborts take the form of SFI memory access rejections (e.g., when an untrusted caller attempts to write to an SFI-protected, callee-owned object). This is because OFI ensures that even if an incorrect vaulter is chosen, control still flows to a vaulter that enforces the contract demanded by its callee, and therefore the callee does not receive any policy-violating objects or code pointers. The callee might nevertheless receive incorrect

Table 3.4. Browser experimental results

Binary Program	**# DLLs**	**# Interface Funcs**	**File Size**			**Code Segment Size**			**Rewriting Times** (s)
			Old (KB)	**New** (KB)	**Increase** (%)	**Old** (KB)	**New** (KB)	**Increase** (%)	
firefox.exe	1	1,393	376	522	39	80	146	82	15.36
browsercomps.dll	8	7,611	43	100	133	28	57	103	5.15
freebl3.dll	3	4,166	329	688	109	236	359	52	41.77
lgpllibs.dll	2	3,359	50	110	120	36	59	64	6.14
mozglue.dll	3	3,374	104	238	129	84	134	59	15.33
msvcp120.dll	2	3,359	429	848	98	392	418	7	126.21
nss3.dll	5	4,422	1,662	3,972	139	1,360	2,310	70	242.72
nssdbm3.dll	2	3,359	84	216	157	72	131	83	14.27
nssckbi.dll	2	3,359	386	469	22	40	82	105	9.29
sandboxbroker.dll	5	5,193	198	381	92	100	182	82	20.06
softokn3.dll	2	3,359	137	339	147	112	202	81	20.67
xul.dll	36	12,657	51,251	104,116	103	31,184	52,865	70	5,662.68
median	3	3,367	264	425	115%	92	164	75%	17.71s

(but not policy-violating) arguments, such as data pointers into inaccessible memory. In such cases, the callee safely aborts with a memory access violation. Other times the callee runs correctly but returns data or code pointers not expected by the caller, whereupon CFI or SFI protections on the caller side intervene.

The MCFG_Exploit attack (see Section 3.2.2) is detected within the OFI vaulter code when OFI identifies the counterfeit vtable.

3.5.4 Scalability

To exhibit OFI's scalability, we applied our prototype to Mozilla Firefox (version 48.0.1) for Windows, which is larger and more complex than our other test applications in Table 3.1. Like many large software products, Firefox is heavily multi-module—most of its functionalities are implemented in whole or part within application-level DLLs that ship along with the main executable. Applying OFI merely to firefox.exe hence does not provide much security. We therefore treated all modules in Table 3.4 as untrusted for this experiment. Similar to Table 3.1, column 2 in Table 3.4 counts the number of trusted libraries imported by each module, and column 3 counts the methods exported by each library. The other columns in Table 3.4 report file size increase, code segment size increase, and the time that OFI took to rewrite each module. On average, file sizes increase by about 115%, and code segments by about 75%.

One problem that we encountered was that Firefox's Just-In-Time (JIT) JavaScript compiler performs runtime code generation, which our Reins prototype does not yet support. (It conservatively denies execution access to writable memory.) Future work should overcome this by incorporating a CFI-supporting JIT compiler, such as RockJIT (Niu and Tan, 2014b). As a temporary workaround, for this experiment we installed a vectored exception handler that catches and redirects control-flows to/from runtime-generated code through OFI. This is potentially unsafe (because the runtime-generated code remains uninstrumented by CFI) and slow (because exception handling introduces high overhead), but allowed us to test

preservation of Firefox's functionalities in the presence of OFI. All browser functionalities we tested exhibited full operation after OFI instrumentation.

To estimate the performance impact of OFI on the application, we conducted the same evaluation methodology as reported in Section 3.5.2, but subtracted out the overhead of the extra context-switches introduced by the exception handler. This yields an estimated overhead of about 0.84%.

3.6 Conclusion

OFI is the first work to extend CFI security protections to the significant realm of mainstream software in which one or more object-exchanging modules are immune to instrumentation. It does so by mediating object exchanges across inter-module trust boundaries with the introduction of tamper-proof proxy objects. The mediation strategy is source-agnostic, making it applicable to both source-aware and source-free CFI approaches. A type-theoretic basis for the mediation algorithm allows for automatic synthesis of OFI mediation code from interface description languages.

A prototype implementation of OFI for Microsoft COM indicates that the approach is feasible without access to source code, and scales to large interfaces that employ callbacks, event-driven programming, interface inheritance, datatype recursion, and dependent typing. Experimental evaluation shows that OFI exhibits low overheads of under 1% for some real-world consumer software applications.

CHAPTER 4

TOWARDS INTERFACE-DRIVEN COTS BINARY HARDENING[1]

Chapter 3 introduces OFI, which extends CFI to scale to consumer software with large, widely deployed object-oriented interfaces. Based on the general binary hardening algorithm presented in Chapter 3, this chapter presents a detailed case study that applies the algorithm to a production-level, event-driven, Windows COTS application.

The remainder of the chapter proceeds as follows: Section 4.1 reviews why CFI suffers difficulty of hardening interface-driven cots software. Next, Section 4.2 demonstrates how applying previously published CFI hardening to application code without applying the same hardening to interoperating system modules results in exploitable critical vulnerabilities. Section 4.3 summarizes our interface-driven approach for closing such vulnerabilities without modifying system modules, followed by a detailed case-study in Section 4.4. Section 4.5 discusses future work directions, and Section 4.6 concludes.

4.1 Introduction

Hardening binary software applications against low-level exploits (e.g., control-flow hijacking and code reuse attacks (Sadeghi et al., 2015; Crane et al., 2015)) is widely recognized as an important step in defending software ecosystems. Software Fault Isolation (SFI) (Wahbe et al., 1993) and Control-Flow Integrity (CFI) (Abadi et al., 2005) are two important examples of such hardening. Implementation approaches include XFI (Erlingsson et al., 2006), PittSFIeld (McCamant and Morrisett, 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) and

[1]This chapter contains material previously published as: Xiaoyang Xu, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. "Towards Interface-Driven COTS Binary Hardening." In *Proceedings of the 3rd Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pp. 1909–1924, November 2017.

OCFI (Mohan et al., 2015). However, most hardening techniques in the literature assume that interoperating software components are all hardened in the same way, using the same code transformation algorithm. For example, XFI's binary transformation entails instrumenting all reachable control-flow transfer instructions in all modules with guard code that checks for XFI-added security labels at jump destinations. This uniformity of enforcement is a prerequisite assumption of XFI's proof of safety (Abadi et al., 2009).

VTable protections, which include source-aware (Tice, 2012; Jang et al., 2014; Zhang et al., 2016; Bounov et al., 2016; Kuznetsov et al., 2014; Crane et al., 2015, 2013; Haller et al., 2015) and source-free (Gawlik and Holz, 2014; Zhang et al., 2015; Prakash et al., 2015) approaches for preventing or detecting vtable corruption at control-flow operations involving vtable method pointers, likewise typically require that all call sites where such pointers might be dereferenced must be uniformly instrumented with common guard code in order to be effective. If some pointers flow to call sites located within other modules compiled with a different pointer protection mechanism, control-flow security cannot be guaranteed.

Unfortunately, a large number of mission-critical software environments include diverse, interoperating components that are not all secured in exactly the same way. For example, the user interfaces of many critical infrastructure applications are implemented atop Microsoft Windows OSes, which purvey essential services to binary applications via closed-source, binary system libraries. These libraries are difficult to modify for a variety of reasons: some are digitally signed, others are aggressively optimized in ways that frustrate accurate disassembly even by the best reverse-engineering tools, and some are loaded dynamically (e.g., from cloud services) as applications execute and discover they need particular services. Similarly, many event-driven Linux applications are implemented atop toolkits such as GTK+[2], which dynamically serve user interface widgets and supporting library code on-demand, and which

[2]https://www.gtk.org

therefore may have been separately compiled with a diverse variety of different protection strategies.

Although recompiling the universe of all software components with some uniform protection scheme is obviously one option for coping with this problem, doing so is unrealistic for many operating contexts. This motivates the development of a more modular methodology for hardening application code that relies on services implemented with diverse protections, but without the need to modify or even disassemble interoperating binary modules on which the application relies.

4.2 Attack Example

CFI and SFI binary hardening algorithms typically work by instrumenting all indirect jump sites in the software with guard code that blocks jumps to illegal destinations at runtime. This prevents many forms of control-flow hijacking, including many code-reuse attacks. However, when the enforcement cannot retrofit all modules, jumps in unmodified modules may remain unguarded, or guarded by a different and possibly inconsistent safety mechanism. This becomes problematic when interoperating modules exchange code pointers— a common practice of object-oriented software that shares objects. In such cases, the disparate guard code can fail to enforce the protection scheme expected by cross-module callees.

One approach to this problem is to secure the objects passed to uninstrumented modules at call sites within the instrumented modules (e.g., Tice et al., 2014). But this approach fails when trusted modules retain persistent references to the object, or when their code executes concurrently with untrusted module code. In these cases, verifying the object at the point of exchange does not prevent the untrusted module from subsequently modifying the vtable pointer to which the trusted module's reference points (e.g., as part of a data corruption attack). These *COnfused DEputy-assisted Counterfeit Object-Oriented Programming*

```
   Untrusted Module
1  CoCreateInstance(⟨clsid⟩, ..., ⟨iid1⟩, &o1);
2  o1→RegisterEventCallbackInterface(..., o2, ...);
                                            Trusted Module
3                                           o2→AddRef();
```

Listing 4.1. Code that registers a running application Windows Image Acquisition (WIA) event notification

(CODE-COOP) attacks (Wang et al., 2017) deputize the receiving module (Hardy, 1988) into violating the control-flow policy by passing them counterfeit objects (Schuster et al., 2015).

Before a detailed walkthrough of a CODE-COOP attack, we first show how objects are typically exchanged between modules with object-oriented interfaces. Listing 4.1 provides a code snippet dissassembled from a Microsoft Paint binary. For this example, we assume that the Paint application code is untrusted, whereas the system DLLs it loads are trusted. The application code first creates a shared object o_1 (line 1), where $\langle clsid \rangle$ and $\langle iid_1 \rangle$ are numeric identifiers for the desired system class and its `IWiaDevMgr` interface, respectively. Method `RegisterEventCallbackInterface` is then invoked to register a running application Windows Image Acquisition (WIA) event notification (line 2). This method takes argument o_2, which is a pointer to the `IWiaEventCallback` interface that the WIA system uses to send the event notification.

While executing `RegisterEventCallbackInterface`, the trusted system module calls object o_2's `Addref` method (line 3), which increments the reference count for the object. Listing 4.2 exhibits the code at the assembly level. The object is first moved to register `EAX` (line 1), and its method table is moved to register `ECX` (line 2). Then all arguments are pushed onto the stack (line 4), including the object (line 6). In the end, the corresponding method is called by indexing the method table (line 7).

```
1  MOV EAX, ⟨object⟩
2  MOV ECX, DWORD PTR DS:[EAX]
3  ...
4  PUSH ⟨arguments⟩
5  ...
6  PUSH EAX
7  CALL DWORD PTR DS:[ECX + ⟨index⟩]
```

Listing 4.2. Function call in assembly

Our attacker model assumes that untrusted modules might be completely malicious, containing arbitrary native code, but that they have been transformed by a CFI algorithm into code compliant with the control-flow policy. Unfortunately, the code snippet in Listing 4.1 is vulnerable to CODE-COOP attack even with CFI protections enabled for the untrusted module. Such protections prevent the function call on line 2 from violating the control-flow policy, but line 3 is not protected in the same way because it resides in an unmodifiable system library. Argument o_2 passed into the trusted module can therefore potentially be corrupted to escape the CFI sandbox.

An object reference o_2 cannot be simply treated as a function pointer (e.g., for a signature check) because the reference points to an object containing a vtable pointer, as illustrated in Figure 4.1. The vtable stores many method pointers. Some of these methods create and return more objects containing new vtables and method pointers when called, creating a complex web of interconnected code pointer exchanges. Since dynamically generated vtables frequently reside in untrusted, writable memory, a data corruption vulnerability (e.g., buffer overwrite) can potentially replace the vtable of o_2 with a counterfeit one. This malicious replacement can happen after the function signature check (e.g., if the application is multithreaded or the callee retains a persistent reference to the object).

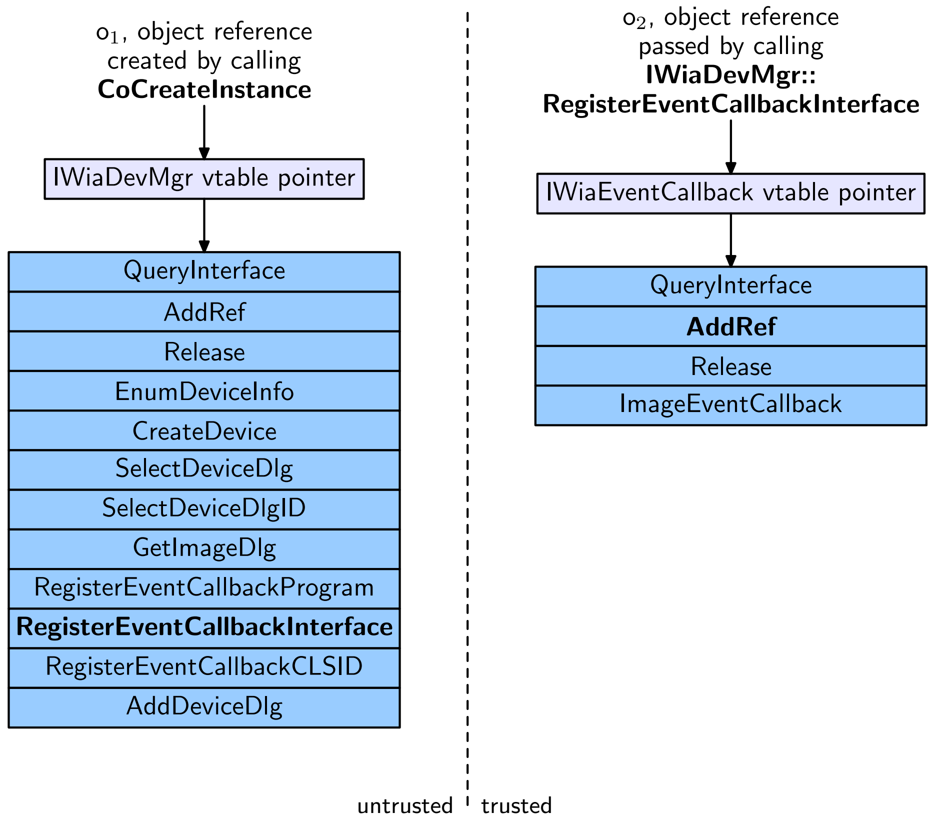

Figure 4.1. Object binary representation

Thus, the counterfeit vtable can reroute object o_2's method `Addref` call to any location specified by the attacker (line 3). The policy mismatch occurs because the destination of the `Addref` call is computed from an untrusted code pointer, but the call site is located in a trusted, unmodifiable system library and cannot be instrumented directly with guard code.

Cross-module control-flow hijacks are recognized as a significant class of code-reuse attacks in practice. For example, they have been leveraged to hijack Chrome from within Google Native Client by exploiting differences between the CFI policies enforced by different interoperating browser modules (Obes and Schuh, 2012). Prior work has advanced compiler-side solutions that require recompiling all modules to the same protection strategy (Niu and Tan, 2014a), while OFI (Wang et al., 2017) is currently the only proposed binary solution.

Our work is the first to admit and harmonize differing protection strategies through automated binary interface synthesis. The next section proposes a modular, source-free approach to this that avoids directly modifying any trusted modules.

4.3 Technical Approach

Our proposed solution instruments untrusted application binary code in such a way that trusted callee modules (i.e., potential victim deputies) never receive writable code pointers from untrusted, CFI-protected callers. Placing the entire object in read-only memory is infeasible because objects typically contain writable data adjacent to the vtable pointer, which cannot easily be moved without breaking the application. We therefore instead automatically substitute shared objects with read-only proxy objects when they flow across an inter-module trust boundary. All proxy objects and their vtables inhabit read-only memory so CODE-COOP attacks cannot corrupt proxy vtables.

Instrumented modules retain direct references to the original object, allowing them to write to data fields, but uninstrumented object recipients receive a read-only proxy. This works because modern binary-level object exchange protocols, such as Component Object Model (COM), enforce an abstraction layer that requires object recipients to access data indirectly via accessor methods. (This allows shared objects to be located on remote machines during RPC.) As illustrated in Figure 4.2, our proxy objects' methods therefore wrap the methods of the underlying object to enforce control-flow guards that intervene whenever object recipients attempt to call one of the object's methods.

For each object-oriented API imported by an untrusted module, we write a wrapper in which every shared object argument is replaced by a proxy object. Thus, when a trusted module attempts to call a method of an object, it actually calls a wrapper method of the proxy object. Control then flows to a dispatch subroutine. The dispatcher pops the return address to determine the index of the method being called, and consults the stack's *this*

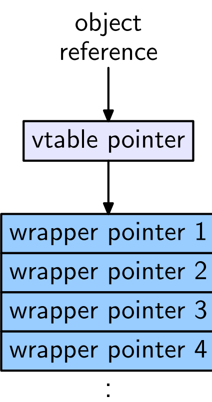

Figure 4.2. Proxy object binary representation

pointer to identify the object. Based on this information, it selects and tail-calls a mediator that wraps and secures the original method according to its type signature. If original method involves object arguments, the mediator replaces them with corresponding proxies. Finally, the mediator passes the control to the original method.

The wrappers must also sometimes introduce new proxy objects in the reverse direction (i.e., during trusted-to-untrusted cross-module calls and returns) in order to secure methods that return new objects or interfaces. For example, if a trusted callee returns an object whose methods accept objects as arguments, the untrusted caller instead receives a proxy object whose wrappers substitute objects arguments with proxies before passing control back to the trusted module.

To assure complete mediation of these interfaces (which are often large and complex), our approach is conservative: The CFI policy is defined to block all cross-module control-flow edges except the ones implemented by the mediators. Inadvertent omission of an API from the mediator library therefore provokes a security abort at runtime. In practice, the mediator code is synthesized automatically from the interface descriptions (e.g., C header or IDL files), so that all documented interface members are automatically included.

Our approach defends against the attack shown in Section 4.2. After hardening the code in Listing 4.1, the shared object $\mathtt{o}_1$ is replaced by its proxy. Original method `Register-`

`EventCallbackInterface` instead invokes a wrapper method of the proxy object of $\mathtt{o}_1$. This wrapper method reroutes the control to the mediator of `RegisterEventCallbackInterface`. The mediator finds that object $\mathtt{o}_2$ is passed from the untrusted module to the trusted module. Then the proxy object of object $\mathtt{o}_2$ is generated and handed to the trusted module. Hardware write-protections prevent the proxy object's vtable from being corrupted. Therefore, even without modifying the trusted module, the `AddRef` call is guaranteed to target a permitted destination.

4.4 Case Study

To demonstrate how our approach can harden closed-source, binary software against CODE-COOP attacks, and to exhibit some of the challenges, we next discuss our experience hardening a simple but representative Windows application: Microsoft Paint.

4.4.1 Object-oriented Design

Paint is a simple desktop application that has been included with all versions of Windows. Like most commercial software, it does not access system kernel services directly; rather, its design extends system-provided classes to construct objects that inherit kernel-accessing functionalities from their base methods. On Windows, such applications typically draw their base classes from the Microsoft Foundation Class (MFC) Library—a shared C++ library designed for event-driven software development. A large percentage of all Windows software is built atop MFC, but this design presents great challenges for traditional CFI because of the complex object exchanges it engenders at the binary level. Surveys of the prior CFI literature (cf., Wang et al., 2017) exhibit no examples prior to OFI where CFI was successfully evaluated against an MFC-based product without opening CODE-COOP vulnerabilities.

Since MFC is extremely tightly coupled to the applications with which it links, our approach treats both Paint and MFC as untrusted, application-level modules and leaves the

Table 4.1. Interoperating COM modules used in case study

Module	File Size			Code Segment Size			Rewriting Times (s)
	Old (KB)	**New** (KB)	**Increase** (%)	**Old** (KB)	**New** (KB)	**Increase** (%)	
mspaint.exe	6228	7094	14	557	886	59	104.78
mfc42u.dll	1137	2583	127	1025	1480	44	203.91

others as trusted. To do so, we applied our automated binary retrofitting (built atop the OFI framework) to the Paint (mspaint.exe) and MFC (mfc42u.dll) binary libraries, and placed the retrofitted MFC in the retrofitted Paint application's load path, thereby overriding the system-level MFC.

Table 4.1 reports the percentage increase of the file size and code segments, as well as the time taken to rewrite each module. After instrumenting, we manually tested all program features of Paint systematically. All features we tested exhibited full functionality. We measure the runtime overhead imposed by our approach as the ratio of time spent within the wrapper modules to the total runtime. Paint has an overhead of 0.38%.

4.4.2 API Surface

Table 4.2 lists all the system APIs with object arguments that Paint and MFC called during our experiments. There are 22 APIs from 4 different trusted modules. Column 3 reports the type of object argument in each API. An OUT-object argument (e.g., in `CoCreateInstance`) is usually an interface pointer returned from a trusted module. An IN-object argument (e.g., in `CoLockObjectExternal`) is usually an interface pointer that an untrusted module passes to a trusted module. More complex APIs can have IN-object and OUT-object arguments together. For example, API `SHCreateShellItem` passes an `IShellFolder` interface pointer to shell32.dll and receives an address of a pointer to a `IShellItem` interface after the API returns.

4.4.3 Object Exchanges

Table 4.3 reports the interfaces mediated by our guard code when running Paint and MFC. Column 2 reports the number of virtual methods (including inherited methods if any) in the vtable of each interface, and column 3 reports the trusted module to which the interface belongs. Overall, we mediate 34 interfaces and 510 methods from 8 trusted modules. Among

Table 4.2. APIs with object exchanges

API	DLL	Object Type
CoCreateInstance	ole32	OUT
CoDisconnectObject	ole32	IN
CoGetClassObject	ole32	OUT
CoLockObjectExternal	ole32	IN
CoRegisterMessageFilter	ole32	IN & OUT
CreateFileMoniker	ole32	OUT
CreateStreamOnHGlobal	ole32	OUT
DoDragDrop	ole32	IN
GdipLoadImageFromStream	gdiplus	IN
GdipSaveImageToStream	gdiplus	IN
GetRunningObjectTable	ole32	OUT
OleCreateLinkFromData	ole32	IN & OUT
OleGetClipboard	ole32	OUT
OleSetClipboard	ole32	IN
OleIsCurrentClipboard	ole32	IN
OleIsRunning	ole32	IN
OleRun	ole32	IN
RegisterDragDrop	ole32	IN
SafeArrayPutElement	olaeut32	IN
SHBindToParent	shell32	OUT
SHCreateShellItem	shell32	IN & OUT
SHGetDesktopFolder	shell32	OUT

the interfaces and methods in Table 4.3, Table 4.4 reports the methods that have object arguments. An object argument can also be an OUT-object or an IN-object, similar to the APIs reported in Table 4.2.

As discussed in Section 4.3, for each API and virtual method, we synthesized a mediator in which OFI recursively substitutes both types of object arguments with appropriate proxy objects immediately before the cross-module call and immediately after the cross-module return.

Table 4.3. COM interfaces

Interface	# Methods	DLL
IAccPropServices	12	oleacc
IDataObject	12	ole32
IEnumWIA_DEV_INFO	8	ole32
IMessageFilter	6	ole32
IMarshal	9	ole32
IMoniker	23	ole32
IOleClientSite	9	ole32
IPropertyStore	8	propsys
IRunningObjectTable	10	ole32
IShellFolder	13	shell32
IShellItem	8	shell32
IShellItem2	21	shell32
IStream	14	ole32
IUIApplication	6	uiribbon
IUICollection	10	uiribbon
IUICommandHandler	5	uiribbon
IUIFramework	12	uiribbon
IUIImage	4	uiribbon
IUIImageFromBitmap	4	uiribbon
IUIRibbon	6	uiribbon
IUISimplePropertySet	4	uiribbon
IUnknown	3	uiribbon
IWiaDevMgr	12	wiaservc
IWiaEventCallback	4	ole32
IWICBitmapDecoder	14	windowscodecs
IWICBitmapEncoder	13	windowscodecs
IWICBitmapFrameDecode	11	windowscodecs
IWICBitmapFrameEncode	14	windowscodecs
IWICImagingFactory	28	windowscodecs
IWICMetadataBlockReader	7	windowscodecs
IWICMetadataBlockWriter	12	windowscodecs
IWICStream	18	windowscodecs
IXMLDOMDocument	82	msxml6
IXMLDOMDocument2	88	msxml6

Table 4.4. Methods with object exchanges

Interface	Method(s)	Object Type
IRunningObjectTable	IRunningObjectTable::Register	IN
	IRunningObjectTable::GetObject	IN & OUT
IShellFolder	IShellFolder::EnumObjects	OUT
IShellItem2	IShellItem2::QueryInterface	OUT
	IShellItem2::GetPropertyStore	OUT
IStream	IStream::QueryInterface	OUT
IUIApplication	IUIApplication::OnViewChanged	IN
	IUIApplication::OnCreateUICommand	OUT
	IUIApplication::OnDestroyUICommand	IN
IUICollection	IUICollection::Add	IN
	IUICollection::GetItem	OUT
	IUICollection::Insert	IN
	IUICollection::Replace	IN
IUICommandHandler	IUICommandHandler::Execute	IN
	IUICommandHandler::UpdateProperty	IN
IUIFramework	IUIFramework::QueryInterface	OUT
	IUIFramework::Initialize	IN
IUIImageFromBitmap	IUIImageFromBitmap::QueryInterface	OUT
	IUIImageFromBitmap::CreateImage	OUT
IUIRibbon	IUIRibbon::LoadSettingsFromStream	IN
	IUIRibbon::SaveSettingsToStream	IN
IUISimplePropertySet	IUISimplePropertySet::QueryInterface	OUT
IUnknown	IUnknown::QueryInterface	OUT
IWiaDevMgr	IWiaDevMgr::RegisterEventCallbackInterface	IN & OUT
IWiaEventCallback	IWiaEventCallback::QueryInterface	OUT
IWICBitmapDecoder	IWICBitmapDecoder::QueryInterface	OUT
	IWICBitmapDecoder::GetFrame	OUT
IWICBitmapEncoder	IWICBitmapEncoder::Initialize	IN
	IWICBitmapEncoder::CreateNewFrame	IN & OUT
IWICBitmapFrameEncode	IWICBitmapFrameEncode::QueryInterface	OUT
	IWICBitmapFrameEncode::Initialize	IN
	IWICBitmapFrameEncode::WriteSource	IN
IWICImagingFactory	IWICImagingFactory::CreateDecoderFromFilename	OUT
	IWICImagingFactory::CreateEncoder	OUT
	IWICImagingFactory::CreateStream	OUT
IWICMetadataBlockReader	IWICMetadataBlockReader::GetReaderByIndex	OUT
IWICMetadataBlockWriter	IWICMetadataBlockWriter::InitializeFromBlockReader	IN
IWICStream	IWICStream::InitializeFromIStream	IN
	IWICStream::InitializeFromIStreamRegion	OUT
IXMLDOMDocument	IXMLDOMDocument::QueryInterface	OUT
	IXMLDOMDocument::save	IN

4.4.4 Callbacks

Table 4.5 reports the APIs that have code pointers (*callbacks*) as arguments. Such an argument can be a direct code pointer (e.g., in `CallWindowProc`), a pointer to an array of callbacks (e.g., in `initterm`), or a pointer to a structure that has a callback in one or more of its fields (e.g., in `RegisterClass`). Paint and MFC import 14 such APIs from 5 trusted modules. We implemented a mediator for each of these APIs in which code pointer validation or sanitization secures the code pointer exchange against hijacking attacks.

4.5 Future work

Although our approach successfully secures inter-module object exchanges in the presence of unmodifiable (e.g., system) modules, unmodified modules can still potentially contain other security weaknesses that might leave retrofitted applications vulnerable to attack. For example, if a trusted module retains a persistent reference to an object, but stores that reference in an unsafe location (e.g., memory that the retrofitting mechanism considers untrusted and application-writable), then a malicious module could replace the reference with a counterfeit object to implement a CODE-COOP attack despite our defense.

Our current prototype mitigates such vulnerabilities by leveraging software fault isolation (SFI) to isolate module data and stack segments from cross-module writes. However, this approach cannot support modules that need direct access to each other's memory (e.g., when trusted modules store object references into writable buffers provided by untrusted modules).

An important line of future research therefore entails the development of binary-level code analyses and tools that can discover the memory safety policies implicitly expected and enforced by interoperating modules with differing protection schemes. Future work should use such analyses to derive appropriate memory and control-flow safety policies for application-level retrofitting algorithms to enforce in order to ensure safety in the presence of unmodifiable libraries that have differing security expectations and requirements.

Table 4.5. APIs with callback pointers

API	**DLL**
__beginthread	msvcrt
__beginthreadex	msvcrt
__initterm	msvcrt
__onexit	msvcrt
CallWindowProc	user32
ChooseColor	comdlg32
ChooseFont	comdlg32
DialogBoxIndirectParam	user32
DialogBoxParam	user32
CreateDialogIndirectParam	user32
CreateDialogParam	user32
EnumFonts	user32
EnumFontFamilies	gdi32
EnumFontFamiliesEx	gdi32
EnumObjects	gdi32
EventRegister	advapi32
GetOpenFileName	comdlg32
GetSaveFileName	comdlg32
PrintDlg	comdlg32
RegisterClass	user32
RegisterClassEx	user32
SendMessageCallback	user32
SetAbortProc	gdi32
SetProp	user32
SetWindowLong	user32
SetWindowsHookEx	user32

4.6 Conclusion

We have presented a modular approach that hardens application-level software without the need to modify interoperating modules on which the application replies. Our interface-driven approach presented in this paper mediates object exchanges across inter-module trust boundaries with proxy objects, and therefore modules that obeys their interface specifications get protected when they call proxy object methods. We showed that coupled with OFI and CFI, the approach can effectively thwart CODE-COOP attacks by completely mediating the interfaces between trusted and untrusted modules.

CHAPTER 5

SEISMIC: SECURE IN-LINED SCRIPT MONITORS FOR INTERRUPTING CRYPTOJACKS[1]

As discussed in Chapters 3–4, OFI centers around the idea of proxy objects that are *in-lined reference monitors* (IRMs) that wrap and mediate access to the methods of the objects they proxy to enforce costumer-specific security policies. IRMs protect software by automatically instrumenting untrusted programs with guard code that monitors security-relevant program operations.

This chapter presents a novel IRM, SEcure In-lined Script Monitors for Interrupting Cryptojacks (SEISMIC), that detects and interrupts unauthorized, browser-based cryptomining, based on semantic signature-matching. The approach addresses a new wave of cryptojacking attacks, including XSS-assisted, web gadget-exploiting counterfeit mining. Evaluation shows that the approach is more robust than current static code analysis defenses, which are susceptible to code obfuscation attacks. SEISMIC offers a browser-agnostic deployment strategy that is applicable to average end-user systems without specialized hardware or operating systems.

The remainder of this chapter is arranged as follows: Section 5.1 begins with an introductory overview of cryptojacking attacks and cryptomining in WebAssembly. Next, Sections 5.2 and 5.3 summarize technologies of rising importance in web cryptomining, and survey the cryptomining ecosystem, respectively. Section 5.4 presents a new cryptojacking attack that details how adversaries can bypass current security protections in this ecosystem to abuse end-user computing resources and illicitly mine cryptocurrencies. Section 5.5 introduces our

[1]This chapter contains material previously published as: Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. "SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks." In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS)*, pp. 122–142, September 2018.

defense strategy based on semantic signature-detection and in-lined reference monitoring, and Section 5.6 evaluates its effectiveness. Finally, Section 5.7 concludes the chapter.

5.1 Introduction

Cryptojacking—the unauthorized use of victim computing resources to mine and exfiltrate cryptocurrencies—has recently emerged as one of the fastest growing new web cybersecurity threats. Network-based cryptojacking attacks increased 600% in 2017, with manufacturing and financial services as the top two targeted industries, according to IBM X-Force (McMillen, 2017). Adguard reported a 31% surge in cryptojacking attacks in November 2017 alone (Meshkov, 2017). The Smominru botnet is estimated to be earning its owners about $8,500 each week via unauthorized Monero[2] mining, or an estimated $2.8–3.6 million total as of January 2018 (Kafeine, 2018).

The relatively recent escalation of cryptojacking threats can be traced to several converging trends, including the emergence of new mining-facilitating technologies that make cryptojacking easier to realize, next-generation cryptocurrencies that are easier to mine and offer greater anonymity to criminals, and the rising value of cryptocurrencies (Lau, 2017). Among the chiefs of these new technologies is WebAssembly (Wasm),[3] a new bytecode language for web browsers that affords faster and more efficient computation than previous web scripting languages, such as JavaScript (JS). By implementing cryptomining algorithms in Wasm, legitimate miners can make more efficient use of client computing resources to generate greater revenue, and attackers can covertly establish illicit mining operations on browsers around the world with only average hardware and computing resources, thereby achieving the mass deployment scales needed to make cryptojacking profitable. For this reason, a majority of in-browser coin miners currently use Wasm (Neumann and Toro, 2018).

[2]https://cointelegraph.com/news/monero

[3]http://webassembly.org

Unfortunately, this availability of transparent cryptomining deployment models is blurring distinctions between legitimate, legal cryptomining and illegitimate, illegal cryptojacking. For example, in 2015, New Jersey settled a lengthy lawsuit against cryptomining company Tidbit, in which they alleged that Tidbit's browser-based Bitcoin mining software (which was marketed to websites as a revenue-generation alternative to ads) constituted "access to computers ... without the computer owners' knowledge or consent" (OAG, New Jersey, 2015). The definition and mechanism of such consent has therefore become a central issue in protecting users against cryptojacking attacks. For example, numerous top-visited web sites, including Showtime (Liao, 2017), YouTube (Goodin, 2018), and The Pirate Bay (Hruska, 2017), have come under fire within 2017–2018 for alleged cryptojacking attacks against their visitors. In each case, cryptocurrency-generation activities deemed consensual by site owners were not deemed consensual by users.

In order to provide end-users an enhanced capability to detect and consent to (or opt-out of) browser-based cryptomining activities, this chapter investigates the feasibility of *semantic signature-matching* for robustly detecting the execution of browser-based cryptomining scripts implemented in Wasm. We find that top Wasm cryptominers exhibit recognizable computation signatures that differ substantially from other Wasm scripts, such as games. To leverage this distinction for consent purposes, we propose and implement SEcure In-lined Script Monitors for Interrupting Cryptojacks (SEISMIC). SEISMIC automatically modifies incoming Wasm binary programs so that they self-profile as they execute, detecting the echos of cryptomining activity. When cryptomining is detected, the instrumented script warns the user and prompts her to explicitly opt-out or opt-in. Opting out halts the script, whereas opting in continues the script without further profiling (allowing it to execute henceforth at full speed).

This semantic signature-matching approach is argued to be more robust than syntactic signature-matchers, such as *n*-gram detectors, which merely inspect untrusted scripts syntactically in an effort to identify those that might cryptomine when executed. Semantic

approaches ignore program syntax in favor of monitoring program behavior, thereby evading many code obfuscation attacks that defeat static binary program analyses.

Instrumenting untrusted web scripts at the Wasm level also has the advantage of offering a browser-agnostic solution that generalizes across different Wasm virtual machine implementations. SEISMIC can therefore potentially be deployed as an in-browser plug-in, a proxy service, or a firewall-level script rewriter. Additional experiments on CPU-level instruction traces show that semantic signature-matching can also be effective for detection of non-Wasm cryptomining implementations, but only if suitable low-level instruction tracing facilities become more widely available on commercial processors.

To summarize, this chapter makes the following contributions:

- We conduct an empirical analysis of the ecosystem of in-browser cryptocurrency mining and identify key security-relevant components, including Wasm.

- We introduce a new proof-of-concept attack that can hijack mining scripts and abuse client computing resources to gain cryptocurrency illicitly.

- We develop a novel Wasm in-line script monitoring system, SEISMIC, which instruments Wasm binaries with mining sensors. SEISMIC allows users to monitor and consent to cryptomining activities with acceptable overhead.

- We apply SEISMIC on five real-world mining Wasm scripts (four families) and seven non-mining scripts. Our results show that mining and non-mining computations exhibit significantly different behavioral patterns. We also develop a classification approach and achieve $\geq 98\%$ accuracy to detect cryptomining activities.

5.2 Background

5.2.1 Monero

Monero (XMR) is a privacy-focused cryptocurrency launched in April 2014. The confidentiality and untraceability of its transactions make Monero particularly popular on darknet markets. Monero's mining process is egalitarian, affording both benign webmasters and malicious hackers new funding avenues.

The core of Monero involves the CryptoNight proof-of-work hash algorithm based on the CryptoNote protocol (van Saberhagen, 2013). CryptoNight makes mining equally efficient on CPU and GPU, and restricts mining on ASIC. This property makes Monero mining particularly feasible on browsers. A majority of current browser-based cryptocurrency miners target CryptoNight, and miner web script development has become an emerging business model. Page publishers embed these miners into their content as an alternative or supplement to ad revenue.

5.2.2 WebAssembly

Wasm (Haas et al., 2017) is a new bytecode scripting language that is now supported by all major browsers (DeMocker, 2017). It runs in a sandbox after bytecode verification, where it aims to execute nearly as fast as native machine code.

Wasm complements and runs alongside JS. JS loads Wasm scripts, whereupon the two languages share memory and call each other's functions. Wasm is typically compiled from high-level languages (e.g., C, C++, or Rust). The most popular toolchain is Emscripten,[4] which compiles C/C++ to a combination of Wasm, JS glue code, and HTML. The JS glue code loads and runs the Wasm module.

[4]http://kripken.github.io/emscripten-site

Browsers can achieve near-native speeds for Wasm because it is designed to facilitate fast fetching, decoding, JIT-compilation, and optimization of Wasm bytecode instructions relative to JS. Wasm does not require reoptimization or garbage collection. These performance advantages make Wasm attractive for computationally intensive tasks, leading most browser-based cryptocurrency miners to use Wasm.

5.3 Ecosystem of Browser-based Cryptocurrency Mining

Although cryptomining is technically possible on nearly any browser with scripting support, efficient and profitable mining with today's browsers requires large-scale deployment across many CPUs. Webmasters offering services that attract sufficient numbers of visitors are therefore beginning to adopt cryptomining as an alternative or supplement to online ads as a source of revenue. This has spawned a secondary business model of cryptomining web software development, which markets mining implementations and services to webmasters.

Thus, although mining occurs on visitors' browsers, miner developers and page publishers play driving roles in the business model. As more miner developers release mining libraries and more page publishers adopt them, a browser-based cryptocurrency mining ecosystem forms. To better understand the ecosystem, we here illustrate technical details of browser-based mining.

Page publishers first register accounts with miner developers. Registration grants the publisher an asymmetric key pair. Publishers then download miner code from the miner developer and customize it to fit their published pages, including adding their public keys. The miner developer uses the public key to attribute mining contributions and deliver payouts to page publishers.

Figure 5.1 illustrates the resulting workflow. After publishers embed the customized miner into their pages, it is served to client visitors and executes in their browsers. The HTML file first loads into the client browser, causing the mining bar to trigger supporting

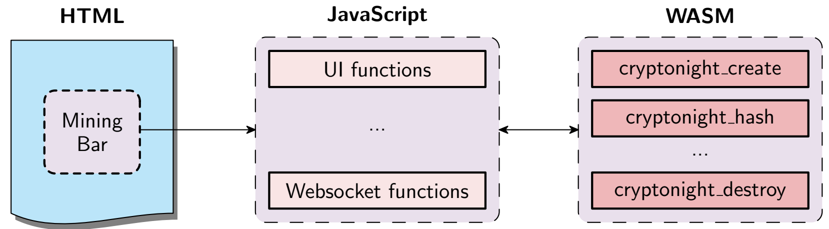

Figure 5.1. Browser-based mining workflow

Table 5.1. Security-related features of popular miners

	Wasm	**Domain Whitelisting**	**Opt-In**	**CPU Throttle**
Adless	✓	✗	✗	✓
Coinhive	✓	✗	✓	✓
CoinImp	✓	✗	✗	✓
Crypto-Loot	✓	✗	✗	✓
JSECoin	✓	✓	✗	✓
WebMinePool	✓	✗	✗	✓

JS modules, which share functionalities with Wasm modules. The Wasm code conducts computationally intensive tasks (e.g., `cryptonight_hash`), whereas UI and I/O interactions (e.g., Websocket communications) are implemented in JS. The code framework is typically created and maintained by miner developers.

Table 5.1 summarizes security-related features of top web miner products:

- *Wasm:* Most miners use Wasm for performance. For example, Coinhive mines Monera via Wasm, and has about 65% of the speed of a native miner.[5]

- *Domain Whitelisting:* To help deter malicious mining, some miner developers offer domain name whitelisting to webmasters. If miner developers receive mining contributions from unlisted domains, they can withhold payouts.

[5]https://coinhive.com

- *Opt-In Tokens:* To support ad blockers and antivirus vendors, some miner products generate opt-in tokens for browsers. Mining can only start after an explicit opt-in from the browser user. The opt-in token is only valid for the current browser session and domain.

- *CPU Throttling:* Using all the client's computing power tends to draw complaints from visitors. Miner developers therefore advise page publishers to use only a fraction of each visitor's available computing power for mining. Webmasters can configure this fraction.

5.4 Counterfeit Mining Attacks

To underscore the dangers posed by many browser-based mining architectures, and to motivate our defense, we next demonstrate how the ecosystem described in §5.3 can be compromised through *counterfeit mining*—a new cryptojacking attack wherein third-party adversaries hijack mining scripts to work on their behalf rather than for page publishers or page recipients.

Our threat model for this attack assumes that miner developers, page publishers, and page recipients are all non-malicious and comply with all rules of the cryptomining ecosystem in §5.3, and that mining scripts can have an unlimited variety of syntactic implementations. Specifically, we assume that miner developers and webmasters agree on a fair payout rate, publishers notify visitors that pages contain miners, and mining only proceeds with visitor consent. Despite this compliance, we demonstrate that malicious third-parties can compromise the ecosystem by abusing the miner software, insecure web page elements, and client computing resources to mine coins for themselves illegitimately.

To understand the attack procedure, we first illustrate how publishers embed miners into their web pages. Listing 5.1 shows the HTML code publishers must typically add. Line 1

```
1 <script src="https://authedmine.com/lib/simple-ui.min.js" async>
2 </script>
3 <div class="coinhive-miner"
4     style="width:256px;height:310px"
5     data_key="YOUR_SITE_KEY">
6     <em>Loading...</em>
7 </div>
```

Listing 5.1. Embedded miner HTML code

imports the JS library maintained by miner developer. Line 4 specifies the dimensions of the miner rendered on the page. Line 5 identifies the publisher to the miner developer. To receive revenue, publishers must register accounts with miner developers, whereupon each publisher receives a unique data key. This allows miner developers to dispatch payroll to the correct publishers.

Our attack is predicated on two main observations about modern web pages: First, cross-site scripting (XSS) vulnerabilities are widely recognized as a significant and pervasive problem across a large percentage of all web sites (WhiteHat Security, 2017; Gupta and Gupta, 2017). Thus, we realistically assume that some mining pages contain XSS vulnerabilities. Second, although some XSS mitigations can block injection of executable scripts, they are frequently unsuccessful at preventing all injections of non-scripts (e.g., pure HTML). Our attack therefore performs purely HTML XSS injection to hijack miners via web gadgets (Lekies et al., 2017)—a relatively new technique whereby existing, non-injected script code is misused to implement web attacks.

Examining the JS library called in line 1 reveals several potentially abusable gadgets, including the one shown in Listing 5.2. This code fragment selects all `div` elements of class `.coinhive-miner` on the page, and renders a miner within each. Unfortunately, line 1 is

```
1 var elements = document.querySelectorAll('.coinhive-miner');
2 for (var i = 0; i < elements.length; i++) {
3     new Miner(elements[i])
4 }
```

Listing 5.2. JavaScript gadget

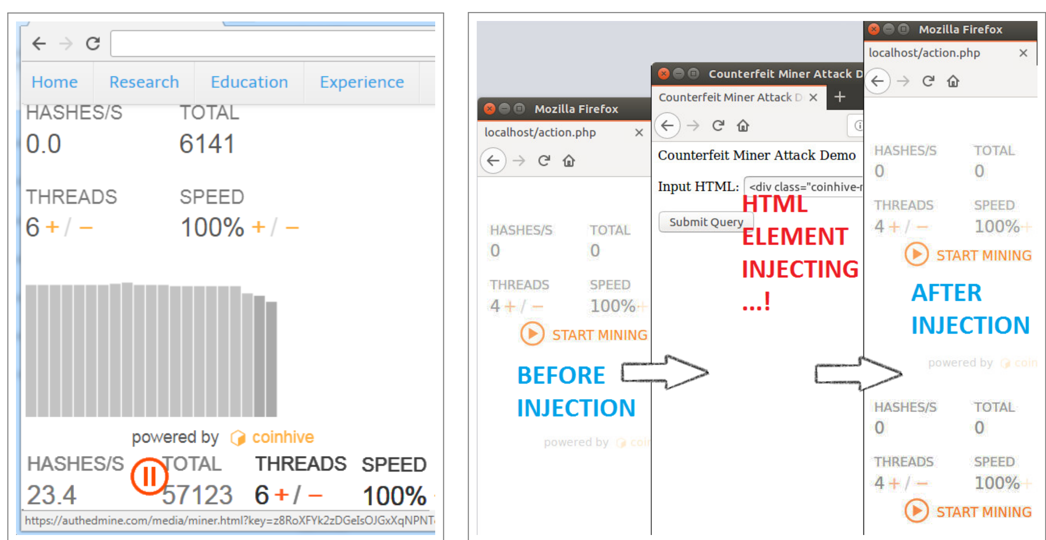

Figure 5.2. Reflected (left) and stored (right) counterfeit mining attacks

exploitable because it cannot distinguish publisher-provided `div` elements from maliciously injected ones. This allows an adversary to maliciously inject a `div` element of that class but with a different data key, causing the recipient to mine coins for the attacker instead of the publisher. We emphasize that in this attack, the exploited gadget is within the miner software, not within the publisher's page. Therefore *all web pages that load the miner* are potentially vulnerable, creating a relatively broad surface for criminals to attack.

To verify our counterfeit miner attack, we deploy two proof-of-concept attacks. Since the attacks begin with XSS exploits, we give two demonstrations: one using a reflected XSS vulnerability and one with a stored XSS vulnerability. The reflected XSS attack crafts a URL link containing the injected HTML code, where the injected code is a `div` element

similar to Listing 5.1. After enticing visitors to click the URL link (e.g., via phishing), the visitor's browser loads and executes the counterfeit miner. The left of Figure 5.2 shows a snapshot of the infected page, in which the counterfeit miner is visible at the bottom.

The stored XSS attack involves a page that reads its content from a database, to which visitors can add insufficiently sanitized HTML elements. In this scenario, injecting the malicious miner HTML code into the database causes the counterfeit miner to permanently inhabit the victim page. The right of Figure 5.2 illustrates the attack procedure. The three screenshots show sequential phases of the attack.

Counterfeit mining attacks illustrate some of the complexities of the cryptomining consent problem. In this case, asking users to consent to mining in general on affected web pages does not distinguish between the multiple miners on the compromised pages, some of which are working for the page publisher and others for a malicious adversary. The next section therefore proposes an automated, client-side consent mechanism based on in-lined reference monitoring that is per-script and is page- and miner-agnostic. This allows users to detect and potentially block cryptomining activities of individual scripts on a page, rather than merely the page as a whole.

5.5 Detection

In light of the dangers posed by counterfeit and other cryptomining attacks, this section proposes a robust defense strategy that empowers page recipients with a more powerful detection and consent mechanism. Since cryptojacking attacks ultimately target client computing resources, we adopt a strictly client-side defense architecture; supplementary publisher- and miner developer-side mitigations are outside our scope.

Section 5.5.1 begins with a survey of current static approaches and their limitations. Section 5.5.2 then proposes a more dynamic strategy that employs semantic signature detec-

tion, and presents experimental evidence of its potential effectiveness. Finally, Section 5.5.3 presents technical details of our defense implementation.

5.5.1 Current Methods

Antivirus engines detect browser mining primarily via script file signature databases. The most popular Wasm implementation of the CryptoNight hashing algorithm (van Saberhagen, 2013) is flagged by at least 21 engines. A few of these (e.g., McAfee) go a step further and detect cryptomining implementations based on function names or other recognized keywords and code file structures.

Unfortunately, these static approaches are easily defeated by code obfuscations. For example, merely changing the function names in the CryptoNight Wasm binary bypasses all antivirus engines used on VirusTotal. Figure 5.3 shows detection results for the original vs. obfuscated CryptoNight binary.

Web browsers also have some detection mechanisms in the form of plugins or extensions, but these have similar limitations. The No Coin (Keramidas, 2017) Chrome extension enforces a URL blacklist, which prevents miners from contacting their proxies. However, criminals can bypass this by setting up new proxies not on the blacklist. MinerBlock[6] statically inspects scripts for code features indicative of mining. For instance, it detects CoinHive miners by searching for functions named `isRunning` and `stop`, and variables named `_siteKey`, `_newSiteKey`, and `_address`. These static analyses are likewise defeated by simple code obfuscations.

5.5.2 Semantic Signature-matching

A common limitation of the aforementioned detection approaches is their reliance on syntactic features (*viz.*, file bytes and URL names) that are easily obfuscated by attackers. We

[6]https://github.com/xd4rker/MinerBlock

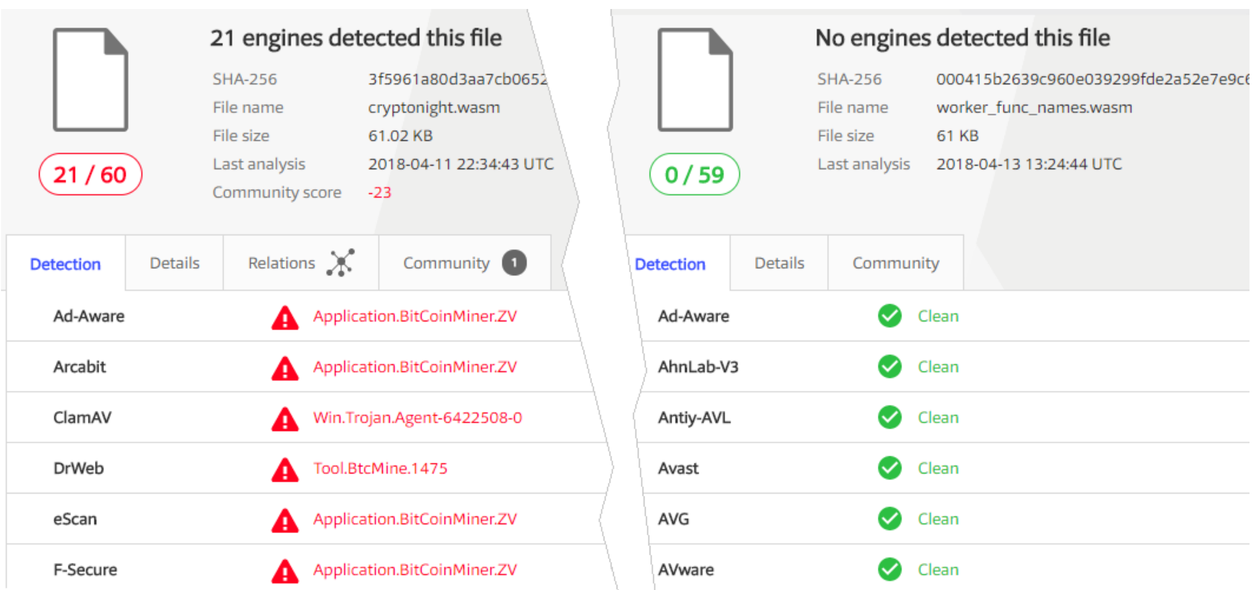

Figure 5.3. Antivirus detection of CryptoNight before and after function renaming

therefore focus on detection via semantic code features that are less easy to obfuscate because they are fundamental to the miner's computational purpose. Our proposed solution monitors Wasm scripts as they execute to derive a statistical model of known mining and non-mining behavior. Profiling reveals a distribution of Wasm instructions executed, which we use at runtime to distinguish mining from non-mining activity.

Using Intel Processor Tracing (PT), we first generated native code instruction counts for Wasm web apps. We recorded native instruction counts for 1-second computation slices on Firefox, for web apps drawn from: 500 pages randomly selected from Alexa top 50K, 500 video pages from YouTube, 100 Wasm embedded game or graphic pages, and 102 browser mining pages. Detailed results are presented in Table 5.2. The traces reveal that crypto-

Table 5.2. Top 30 Opcodes Used as Features to Distinguish Mining and Non-mining

Rank	Opcode	Description
1st	SUB	subtract.
2nd	CMOVS	conditional move if sign (negative).
3rd	UNPCKHPS†	unpacks and interleaves the two high-order values from two single-precision floating-point operands.
4th	DIVSD‡	divide scalar double-precision floating-point values.
5th	SETB	set byte if below.
6th	MOVQ⋆	move quadword.
7th	MAXPS†	return maximum packed single-precision floating-point values.
8th	CMOVNLE	conditional move if not above or equal.
9th	COMVLE	conditional move if less or equal.
10th	PSUBUSW⋆	subtract packed unsigned word integers with unsigned saturation.
11th	CMOVNL	conditional move if not less.
12th	UNPCKLPS†	unpacks and interleaves the two low-order values from two single-precision floating-point operands.
13th	ROUNDSD†	round scalar double precision floating-point values.
14th	CMPPS†	compare packed single-precision floating-point values.
15th	MOVLHPS†	move two packed single-precision floating-point values from the low quadword of an XMM register to the high quadword of another XMM register.
16th	LOCK	lock bus.
17th	CMOVB	conditional move if below.
18th	SETBE	set byte if below or equal.
19th	SETNZ	set byte if not zero.
20th	ROL	rotate left.
21st	MUL	multiply (unsigned).
22nd	SETNLE	set byte if not less or equal.
23rd	CVTTSD2SI‡	convert with truncation scalar double-precision floating-point values to scalar doubleword integers.
24th	MOVMSKPS†	extract sign mask from four packed single-precision floating-point values.
25th	CMOVZ	conditional move if zero.
26th	TEST	logical compare.
27th	CMOVNZ	conditional move if not zero.
28th	ROUNDSS†	round scalar single precision floating-point values.
29th	STMXCSR†	save `mxcsr` register state.
30th	CMOVNB	conditional move if not below or equal.

⋆ MMX Instruction. † SSE Instruction. ‡ SSE2 Instruction.

Table 5.3. Execution trace average profiles

	i32.add	**i32.and**	**i32.shl**	**i32.shr_u**	**i32.xor**
A-Star	86.78	4.71	5.52	0.44	2.54
Asteroids	89.67	4.33	5.10	0.44	0.42
Basic4GL	75.78	8.43	13.75	1.78	0.27
Bullet(1000)	84.42	3.55	11.30	0.20	0.51
CoinHive	19.90	17.90	22.60	17.00	22.60
CoinHive_v0	20.20	17.50	22.70	17.00	22.70
CreaturePack	54.70	0.52	44.27	0.21	0.40
FunkyKarts	77.89	8.68	12.28	0.44	0.71
HushMiner	62.53	6.45	17.87	6.23	6.93
NFWebMiner	28.00	15.80	20.40	15.30	20.40
Tanks	61.90	12.29	22.27	2.02	1.51
YAZECMiner	57.99	4.37	30.75	3.26	3.63

mining Wasm scripts rely much more upon packed arithmetic instructions from the MMX, SSE, and SSE2 instruction sets of CISC processors than do other Wasm scripts, like games.

Although PT is useful for identifying semantic features of possible interest, it is not a good basis for implementing detection on average client browsers since PT facilities are not yet widely available on average consumer hardware and OSes. We therefore manually identified the top five Wasm bytecode instructions that JIT-compile to the packed arithmetic native code instructions identified by the PT experiments. These five instructions are the column labels of Table 5.3.

We next profiled these top-five Wasm instructions at the Wasm bytecode level by instrumenting Wasm binary scripts with self-profiling code. We profiled four mining apps plus one variant, and seven non-mining apps. The non-mining apps are mostly games (which is the other most popular use of Wasm), and the rest are graphical benchmarks. For each app, we executed and interacted with them for approximately 500 real-time seconds to create each profile instance. For each app with configurable parameters, we varied them over their entire range of values to cover all possible cases.

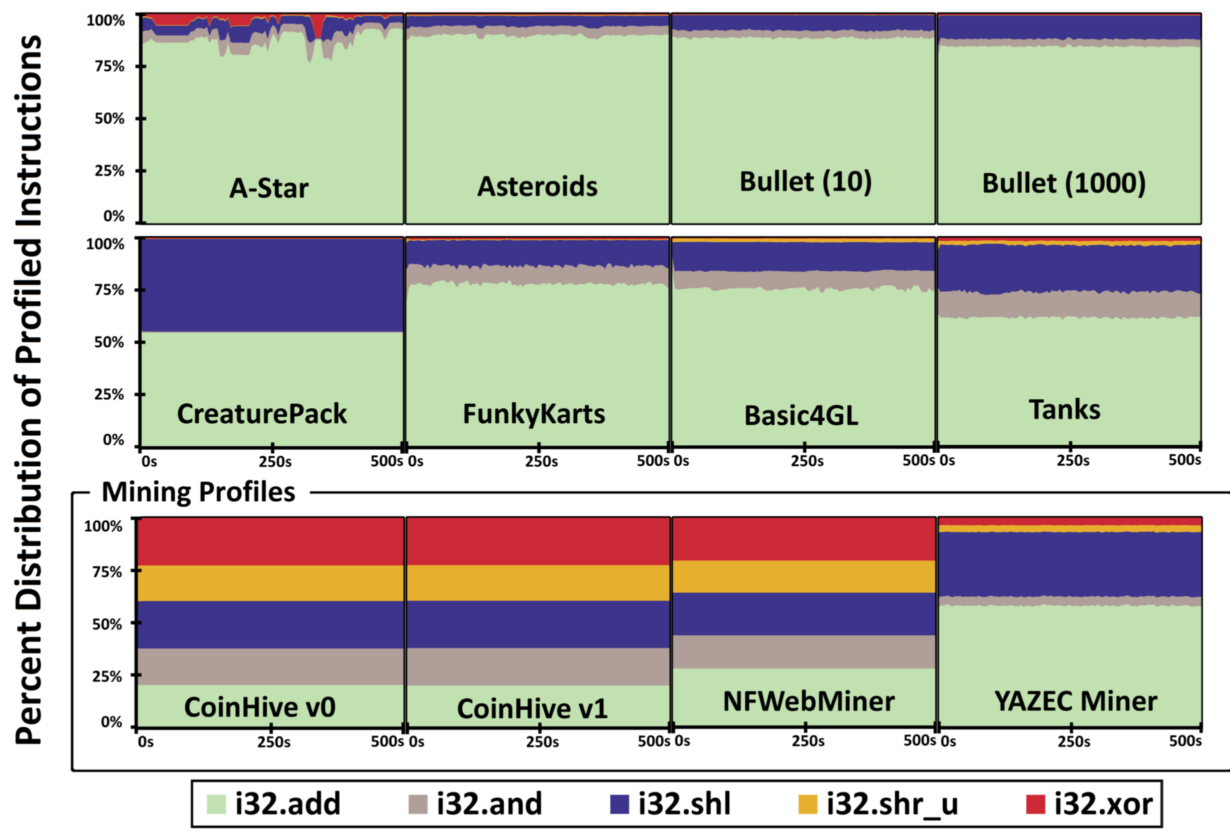

Figure 5.4. Semantic profiles for mining vs. non-mining Wasm apps

Figure 5.4 displays the resulting distributions. There is a clear and distinct stratification for the two CoinHive variants and NFWebMiner, which are based on CryptoNight. YAZEC (Yet Another ZEC) Miner uses a different algorithm, and therefore exhibits slightly different but still distinctive profile. Table 5.3 displays an average across the 100 distributions for all of the profiled applications.

5.5.3 SEISMIC In-lined Reference Monitoring

Our profiling experiments indicate that Wasm cryptomining can potentially be detected by semantic signature-matching of Wasm bytecode instruction counts. To implement such a detection mechanism that is deployable on end-user browsers, our solution adopts an *in-lined reference monitor* (IRM) (Schneider, 2000; Erlingsson and Schneider, 1999) approach. IRMs

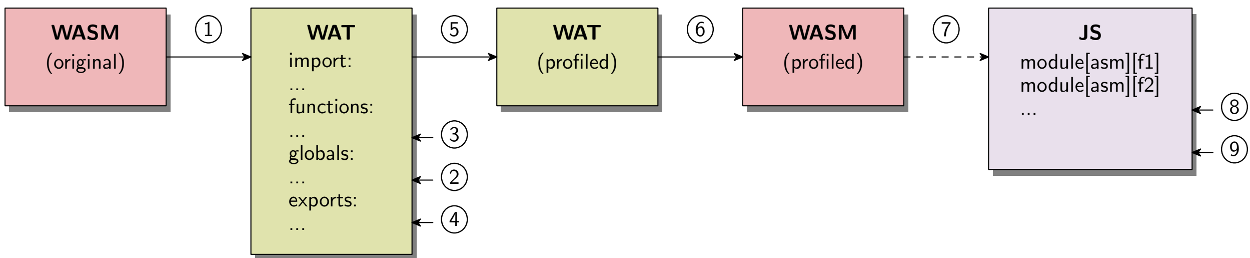

Figure 5.5. SEISMIC transformation of Wasm binaries

automatically instrument untrusted programs (e.g., web scripts) with guard code that monitors security-relevant program operations. The code transformation yields a new program that self-enforces a desired policy, yet preserves policy-compliant behaviors of the original code. In browsing contexts, IRM formalisms have been leveraged to secure other scripting languages, such as JS and Flash (cf., Phung et al., 2015), but not yet Wasm. In this scenario, our goal is to design and implement an IRM system that automatically transforms incoming Wasm binaries to dynamically compute their own semantic features and match them to a given semantic signature.

Wasm scripts are expressed in binary or human-readable textual form. Each can be translated to the other using the Wasm Binary Toolkit (WABT). Typically scripts are distributed in binary form for size purposes, but either form is accepted by Wasm VMs. The programs are composed of *sections*, which are each lists of section-specific content. Our automated transformation modifies the following three Wasm section types:

- Functions: a list of all functions and their code bodies
- Globals: a list of variables visible to all functions sharing a thread
- Exports: a list of functions callable from JS

Figure 5.5 shows a high-level view of our Wasm instrumentation workflow. We here explain a workflow for a single Wasm binary file, but our procedure generalizes to pages

```
1 int pythag(int a, int b) { return a * a + b * b; }
```

Listing 5.3. C++ source code for compilation to Wasm

```
1 (module (table 0 anyfunc) (memory $0 1)
2  (export "memory" (memory $0))
3  (export "pythag" (func $pythag))
4  (func $pythag (; 0 ;) (param $0 i32) (param $1 i32) (result i32)
5   (i32.add (i32.mul (get_local $1) (get_local $1))
6            (i32.mul (get_local $0) (get_local $0)))))
```

Listing 5.4. Original Wasm compiled from C++

with multiple binaries. As a running example, Listing 5.3 contains a small C++ function that computes the sum of the squares of its two inputs. Compiling it yields the Wasm bytecode in Listing 5.4.

Our prototype implementation of SEISMIC first parses the untrusted binary to a simplified abstract syntax tree (AST) similar to the one in Listing 5.4 using `wasm2wat` from WABT with the `-fold-exprs` flag (①). It next injects a fresh global variable of type i64 (64-bit integer) into the globals section for each Wasm instruction opcode to be profiled (②). The JS-Wasm interface currently does not support the transfer of 64-bit integers, so to allow JS code to read these counters, 32-bit accessor functions `getInstLo` and `getInstHi` are added (③). An additional `reset` function that resets all the profile counters to zero is also added, to allow the security monitor to separately profile different time slices of execution. All three functions are added to the binary's exports (④).

The transformation algorithm next scans the bodies of all Wasm functions in the script and in-lines counter-increment instructions immediately after each instruction to be profiled (⑤). Our prototype currently takes the brute-force approach of in-lining the counter-

increment guard code for each profiled instruction, but optimizations that improve efficiency by speculatively increasing counters by large quantities in anticipation of an uninterruptable series of signature-relevant operations are obviously possible.

The modified Wasm text file is now ready to be translated to binary form, which we perform by passing it to `wat2wasm` from WABT (⑥). At this point, we redirect the JS code that loads the Wasm binary to load the new one (⑦). This can be done either by simply using the same name as the old file (i.e., overwriting it) or by modifying the load path for the Wasm file in JS to point to the new one.

Listing 5.5 shows the results of this process when profiling Wasm instructions `i32.add` and `i32.mul`. Lines 4–6 export the IRM helper functions defined in lines 15–20. Lines 21 and 22 define global counter variables to profile `i32.add` and `i32.mul` instructions, respectively. The two `i32.mul` instructions are instrumented on lines 10 and 12, and the single `i32.add` instruction is instrumented on line 9.

SEISMIC's instrumentation procedure anticipates an attack model in which script authors and their scripts might be completely malicious, and adversaries might know all details of SEISMIC's implementation. For example, adversaries might craft Wasm binaries that anticipate the instrumentation procedure and attempt to defeat it. We therefore designed our instrumentation in accordance with secure IRM design principles established in the literature (Schneider, 2000; Hamlen et al., 2006; Ligatti et al., 2009). In particular, the Wasm bytecode language does not include unrestricted computed jump instructions, allowing our transformation to implement uncircumventable basic blocks that pair profiling code with the instructions they profile. Moreover, Wasm is type-safe (Haas et al., 2017), affording the implementation of incorruptible state variables that track the profiling information. Type-safety ensures that malicious Wasm authors cannot use pointer arithmetic or untyped references to corrupt the IRM's profiling variables (cf., Sridhar and Hamlen (2010b,a)). These language properties are the basis for justifying other Wasm security features, such as control-flow integrity (WebAssembly Community Group, 2018).

```
1 (module (table 0 anyfunc) (memory $0 1)
2  (export "memory" (memory $0))
3  (export "pythag" (func $pythag))
4  (export "_getAddsLo" (func $_getAddsLo))
5  ...
6  (export "_reset" (func $_reset))

8  (func $pythag (; 0 ;) (param $0 i32) (param $1 i32) (result i32)
9   (i32.add (set_global 0 (i64.add (get_global 0) (i64.const 1)))
10   (i32.mul (set_global 1 (i64.add (get_global 1) (i64.const 1)))
11    (get_local $1) (get_local $1))
12   (i32.mul (set_global 1 (i64.add (get_global 1) (i64.const 1)))
13    (get_local $0) (get_local $0))))

15 (func $_getAddsLo (; 1 ;) (result i32)
16   (return (i32.wrap/i64 (get_global 0))))
17 ...
18 (func $_reset (; 5 ;)
19   (set_global 0 (i64.const 0))
20   (set_global 1 (i64.const 0)))
21 (global (;0;) (mut i64) (i64.const 0))
22 (global (;1;) (mut i64) (i64.const 0)))
```

Listing 5.5. Instrumented Wasm

```
1 function wasmProfiler() {
2  if (Module["asm"] != null && typeof _reset === "function") {
3   console.log(_getAddsHi() * 2^32 + _getAddsLo() + "␣adds");
4   console.log(_getMulsHi() * 2^32 + _getMulsLo() + "␣multiplies");
5   _reset();
6 } else { console.log("Wasm␣not␣loaded␣yet"); }
7  setTimeout(wasmProfiler, 5000);
8 }
9 wasmProfiler();
10 ...
11 Module["asm"] = asm;
12 var _getAddsLo = Module["_getAddsLo"] = function() {
13   return Module["asm"]["_getAddsLo"].apply(null, arguments) };
14 ...
```

Listing 5.6. SEISMIC JavaScript code

To start the enforcement, Listing 5.6 instantiates a JS timer that first executes at page-load and checks whether Wasm code has been loaded and compiled (⑧). If so, all Wasm instruction counters are queried, reset, and logged to the console. The timer profiles another slice of computation time every 5000 milliseconds. This affords detection of scripts that mine periodically but not continuously.

5.6 Evaluation

To evaluate our approach, we instrumented and profiled the web apps listed in Table 5.3. The majority of Wasm code we profiled was identifiable as having been compiled with Emscripten, an LLVM-based JS compiler that yields a JS-Wasm pair of files for inclusion on web pages. The JS file contains an aliased list of exported functions, where we insert our new entries

Table 5.4. Mining overhead

	Vanilla	**Profiled**
CoinHive v1	36 hash/s	18 hash/s
CoinHive v0	40 hash/s	19 hash/s
NFWebMiner	38 hash/s	16 hash/s
HushMiner	1.6 sol/s	0.8 sol/s
YAZECMiner	1.8 sol/s	0.9 sol/s

for the counters (⑨). The remaining Wasm programs we profiled have a similar structure to the output of Emscripten, so they can be modified in a similar manner.

We profiled every instruction used in the CoinHive worker Wasm, which is a variant of the CryptoNight hashing algorithm, and determined the top five bytecode instructions used: `i32.add`, `i32.and`, `i32.shl`, `i32.shr_u`, and `i32.xor`. Normalized counts of how many times these instructions execute constitute feature vectors for our approach.

5.6.1 Runtime Overhead

Table 5.4 reports runtime overheads for instrumented binaries. The data was obtained by running each miner in original and instrumented form over 100 trials, and averaging the results. CoinHive and NFWebMiner were set to execute with 4 threads and their units are in hashes per second. HushMiner and Yet Another ZEC Miner are single-threaded and display units in solutions per second. In general, the miners we tested incurred a runtime overhead of roughly 100%. We deem this acceptable because once mining is explicitly allowed by the user, execution can switch back to the faster original code.

Non-mining code overhead must be calculated in a different way, since most are interactive and non-terminating (e.g., games). We therefore measured overhead for these programs by monitoring their frames-per-second. In all cases they remained at a constant 60 frames-per-second once all assets had loaded. Overall, no behavioral differences in instrumented scripts

Table 5.5. SVM stratified 10-fold cross validation

Miner	Fold	Precision	Recall	F_1	Fold	Precision	Recall	F_1
N	1	1.00	0.99	0.99	2	1.00	1.00	1.00
Y		0.96	1.00	0.98		1.00	1.00	1.00
N	3	1.00	1.00	1.00	4	1.00	1.00	1.00
Y		1.00	1.00	1.00		1.00	1.00	1.00
N	5	1.00	1.00	1.00	6	1.00	0.99	0.99
Y		1.00	1.00	1.00		0.96	1.00	0.98
N	7	1.00	1.00	1.00	8	1.00	1.00	1.00
Y		1.00	1.00	1.00		1.00	1.00	1.00
N	9	1.00	1.00	1.00	10	1.00	1.00	1.00
Y		1.00	1.00	1.00		1.00	1.00	1.00

were observable during the experiments (except when mining scripts were interrupted to obtain user consent). This is expected since guard code in-lined by SEISMIC is implemented to be transparent to the rest of the script's computation.

5.6.2 Robustness

Our approach conceptualizes mining detection as a binary classification problem, where mining and non-mining are the two classes. Features are normalized vectors of the counts of the top five used Wasm instructions. For model selection, we choose Support Vector Machine (SVM) with linear kernel function. We set penalty parameter C to 10, since it is an unbalanced problem (there are far fewer mining instances than non-mining instances). To evaluate this approach, we use stratified 10-fold cross validation on 1900 instances, which consist of 500 miners and 1400 non-miners.

The results shown in Table 5.5 are promising. All mining activities are identified correctly, and the overall accuracy (F_1 score) is 98% or above in all cases. SEISMIC monitoring exhibits negligible false positive rate due to our strict threhhold for detection. Visitors can also manually exclude non-mining pages if our system exhibits a false positive, though the cross-validation results indicate such misclassifications are rare.

5.7 Conclusion

SEISMIC offers a semantic-based cryptojacking detection mechanism for Wasm scripts that is more robust than current static detection defenses employed by antivirus products and browser plugins. By automatically instrumenting untrusted Wasm binaries in-flight with self-profiling code, SEISMIC-modified scripts dynamically detect mining computations and offer users explicit opportunities to consent. Page-publishers can respond to lack of consent through a JS interface, affording them opportunities to introduce ads or withdraw page content from unconsenting users. Experimental evaluation indicates that self-profiling overhead is unobservable for non-mining scripts, such as games (and is eliminated for miners once consent is granted). Robustness evaluation via cross-validation shows that the approach is highly accurate, exhibiting very few misclassifications.

CHAPTER 6

RELATED WORK

6.1 Prior CFI Evaluations

We surveyed 54 CFI algorithms and implementations published between 2005–2019 to prepare ConFIRM, over half of which were published in 2015–2019. Of these, 66% evaluate performance overhead based on SPEC CPU benchmarks. Examples include 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 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), VTPin (Sarbinowski et al., 2016), VTrust (Zhang et al., 2016), TypeArmor (van der Veen et al., 2016), PittyPat (Ding et al., 2017), RAGuard (Zhang et al., 2017), GRIFFIN (Ge et al., 2017), OFI (Wang et al., 2017), PT-CFI (Gu et al., 2017), HCIC (Zhang et al., 2018) 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, 2019; Postmark, 2013; Pozo and Miller, 2016; de Melo, 2009).

Several of the more recent 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 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 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.

6.2 CFI Surveys

There has been one prior survey of CFI performance, precision, and security, published in 2016 (Burow et al., 2017). It surveys 30 previously published CFI frameworks, with qualitative and quantitative comparisons of their technical approaches and overheads as reported in each original publication. Five of the approaches are additionally reevaluated on SPEC CPU benchmarks.

In contrast, ConFIRM establishes a foundation for evaluating *compatibility* and *relevance* of various CFI algorithms to modern software products, and highlights important security and performance impacts that arise from incompatibility limitations facing the state-of-the-art solutions.

6.3 SFI and CFI

SFI was originally conceived as a means of sandboxing untrusted software modules via software guards to a subset of a shared address space (Wahbe et al., 1993). CFI refined this idea to enforce more specific control-flow graphs (CFGs) (Abadi et al., 2005, 2009). Later work merged the two approaches for more efficient enforcement (Erlingsson et al., 2006; McCamant and Morrisett, 2006; Akritidis et al., 2008; Yee et al., 2009; Niu and Tan, 2013; Wartell et al., 2012b,a), so that today distinctions between SFI and CFI are blurred. We therefore here refer to CFI in a broad sense that includes both lines of research.

With the rise of return-oriented programming and code-reuse attacks (cf., Sadeghi et al. (2015); Crane et al. (2015)), the impact of CFI research has increased in recent years. In addition to securing user-level application software against such threats, it has also been applied to harden smartphones (Davi et al., 2012; Pewny and Holz, 2013; Miguel et al., 2009), embedded systems (Abera et al., 2016), hypervisors (Wang and Jiang, 2010), and operating system kernels (Kemerlis et al., 2012; Criswell et al., 2014; Ge et al., 2016). CFI-

enforcing hardware is also being investigated (Ge et al., 2017; Gu et al., 2017; de Clercq et al., 2016; Nick et al., 2016; Xia et al., 2012; Yuan et al., 2015; Davi et al., 2015, 2014).

Software CFI methodologies can be broadly partitioned into compiler-side *source-aware* approaches and binary-only *source-free* approaches. Source-aware CFI leverages information from source code to generate CFI-enforcing object code via a compiler. 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), and MCFG (Tang, 2015). The availability of source code affords these approaches much greater efficiency and precision than source-free alternatives. For example, source code analysis typically reveals a much more precise CFG for CFI to enforce, and compilers enjoy opportunities to arrange data structure and code layouts to optimize CFI guard code.

MCFI highlights the need for better multi-module CFI enforcement algorithms and tools. To address this problem in source-aware settings, it introduces a modular, separate-compilation approach integrated into the LLVM compiler. However, this requires all modules to be recompiled with an MCFI-equipped compiler; environments where some modules are immutable, are dynamically procured in binary form, or are closed-source, are not supported.

In general, reliance on source code has the potential disadvantage of reducing deployment flexibility. Much of the world's software is closed-source, with source-level information unlikely to be disclosed to consumers due to intellectual property concerns and constraints imposed by developer business models. Software whose sources are available frequently link to or otherwise rely upon binary modules (e.g., libraries) whose sources are not available, requiring approaches for dealing with those source-free components. Finally, software distribution models that deliver binary code on-demand (e.g., as plugins, mobile apps, or hotpatches) usually lack readily available source code with which to implement additional third-party or consumer-side CFI protections.

Concerns over this inflexibility have therefore motivated source-free CFI approaches that transform and harden already-compiled binary code without the aid of source code. Examples 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) and OCFI (Mohan et al., 2015). Source-free approaches face some difficult challenges, including the problem of effectively disassembling arbitrary native code binaries (Wartell et al., 2014), and severe restrictions on which code and data structures they can safely transform without breaking the target program's functionality. Poorer performance than source-aware solutions typically results (Burow et al., 2017). They also tend to enforce more permissive control-flow policies, since they lack source-level control-flow semantics with which to craft a tighter policy (Schuster et al., 2015). This has led to successful attacks against these coarse-grained policies (e.g., Wollgast et al. (2016); Göktas et al. (2014); Davi et al. (2014); Conti et al. (2015)).

In contrast to this usual dichotomy, OFI is source-agnostic—it can extend any of the source-aware or source-free approaches listed above to enhance the security and compatibility of objects that flow between CFI-protected software modules and those lacking CFI protections. It does, however, require documentation of the API that links the interacting modules, as described in §3.3.2.

6.4 VTable Protection

VTable protections prevent or detect vtable corruption at or before control-flow operations that depend on vtable method pointers. Like CFI, there are both source-aware and source-free approaches:

On the source-aware side, GNU VTV (Tice, 2012), SafeDispatch (Jang et al., 2014), and VTrust (Zhang et al., 2016) statically analyze source code class hierarchies to generate CFI-style guards that restrict all virtual method call sites to destinations that implement

matching callees (according to C++ dynamic dispatch semantics). OVT-IVT (Bounov et al., 2016) improves performance by reorganizing vtables to permit quick validation as a simple bounds check. CPI (Kuznetsov et al., 2014) heuristically derives a set of sensitive pointers, and guards their integrity to prevent control-flow hijacking. CPS (Kuznetsov et al., 2014) optimizes CPI to improve overheads for programs with many virtual functions by instrumenting only code pointers, but at the expense of less security for vtable pointers exploited by confused deputy attacks. Readactor++ (Crane et al., 2015) extends vtables into execute-only memory, where their layouts are randomized and laced with booby trap entries (Crane et al., 2013) to counter brute-force attacks. Shrinkwrap (Haller et al., 2015) refines VTV for tighter object inheritance precision.

On the source-free side, T-VIP (Gawlik and Holz, 2014) instruments virtual call sites with guard code that verifies that the vtable is in read-only memory and that the indexed virtual method is a valid virtual method pointer. VTint (Zhang et al., 2015) additionally assigns them IDs that are dynamically checked at call sites. This ensures that instrumented virtual calls always index a valid vtable (though it cannot ensure that the indexed vtable is the precise one demanded by the original source code semantics). VfGuard (Prakash et al., 2015) goes further and infers C++ class hierarchies and call graphs from native code through a suite of decompilation techniques. This yields a more precise, source-approximating CFI policy that can be enforced through static or dynamic binary instrumentation.

OFI differs from these approaches by focusing on protecting software modules that cannot be instrumented (e.g., because they cannot be modified, they have defenses that reject modification, or dynamic loading prevents them from being statically identified). Such immutability renders the vtable protections above inapplicable, since they must instrument all call sites where corrupted vtables might be dereferenced in order to be effective.

6.5 COOP Attacks

COOP (Schuster et al., 2015) is a dangerous new attack paradigm that substitutes vtable pointers or vtable method pointers with structurally similar but counterfeit ones to hijack control-flows of victim programs. In the context of CFI-protected software, such attacks effectively hijack software without violating the CFI-enforced control-flow policy. They achieve this by traversing control-flow edges that are permitted by the policy but that were never intended to be traversed by the original program semantics. They therefore exploit limitations in the defender's ability to derive suitable policies for CFI to enforce—especially in source-free contexts.

OFI does not directly defend against COOP attacks because it does not suggest better policies for CFI to enforce. Rather, it extends defenses that do work against COOP to be effective in contexts where not all call sites can be instrumented with guard code. For example, WIT (Akritidis et al., 2008) can block COOP attacks in WIT-instrumented code, but not if the code links to uninstrumented modules to which it passes objects. In that context, a COOP attacker can flow counterfeit objects to unguarded call sites in the uninstrumented modules. Lacking guards, these sites traverse the prohibited edge prescribed by the object, resulting in policy violations.

When coupled with a CFI defense enforcing a suitably semantics-aware policy, OFI addresses this CODE-COOP attack. By completely mediating the interface between guarded and unguarded modules that share objects, it shields uninstrumented modules from counterfeit objects. OFI is the first defensive work to focus on this attack class.

6.6 Immutable Modules

We are not the first to identify immutable modules as a challenge for CFI. For example, source-aware CFI instrumentation of Chrome on ChromeOS identified two third-party libraries for which source code was not available, and that interact with instrumented modules

through object-oriented interfaces (Tice et al., 2014). Forward CFI's solution to this *mixed code* problem validates object references at call sites within instrumented modules. But this is insecure if the uninstrumented recipients retain persistent references to the shared objects, or if they execute concurrently with untrusted (instrumented) code. In both cases, the untrusted code may later corrupt the shared vtable pointers without calling them, leaving the uninstrumented module in possession of a corrupt, never-validated vtable.

In general, all prior source-aware and source-free CFI and vtable protection research must instrument all interoperating modules in order to thwart control-flow hijacking attacks. OFI is the first solution that accommodates immutable modules. In deployment contexts where the OS cannot be included in the instrumentation process, such modules can be extremely prevalent—potentially including most or all of the system libraries, plus an ongoing stream of incoming upgrades, patches, and extensions to them. OFI seeks to open such environments to CFI assistance.

6.7 Component-based Software Engineering

Microsoft COM (Gray et al., 1998) is presently the dominant industry standard for *component-based software engineering* (McIlroy, 1968) of native code modules in consumer software markets. Its many facets include Object Linking and Embedding (OLE), ActiveX, COM+, Distributed COM (DCOM), DirectX, User-Mode Driver Framework (UMDF), and the Windows Runtime (WinRT). Microsoft .NET applications typically access Windows OS services via the .NET COM Interop, which wraps COM. This prevalence makes COM an appropriate (but challenging) test of OFI's real-world applicability.

Another primary competing standard is OMG's Common Object Request Broker Architecture (CORBA) (Vinoski, 1997). CORBA resembles COM but enforces additional layers of abstraction, including an Object Request Broker (ORB) that has the option of supplying different representations of shared objects to communicating modules. OFI is therefore

potentially easier to realize for CORBA than for COM, since it can take the role of a CFI-enforcing ORB. Interfaces that communicate between CORBA and COM have also been developed (Pawar et al., 2013).

6.8 Cryptocurrencies

Researchers have conducted a variety of systematic analyses of cryptocurrencies and discussed open research challenges (Bonneau et al., 2015). A comprehensive study of Bitcoin mining malware has shown that botnets generate additional revenue through mining (Huang et al., 2014). MineGuard (Tahir et al., 2017) utilizes hardware performance counters to generate signatures of cryptocurrency mining, which are then used to detect mining activities. Other research has focused on the payment part of cryptocurrencies. For example, EZC (Androulaki et al., 2014) was proposed to hide the transaction amounts and address balances. Double-spending attacks threaten fast payments in Bitcoin (Karame et al., 2012). Bitcoin timestamp reliability has been improved to counter various attacks (Szalachowski, 2018). Through analysis of Bitcoin transactions of CryptoLocker, prior studies revealed the financial infrastructure of ransomware (Liao et al., 2016) and reported its economic impact (Conti et al., 2018). In contrast, in-browser cryptomining, such as Monero, is less studied in the scholarly literature. In this work, we conducted the first analysis to study Wasm-based cryptomining, and developed new approaches to detect mining activities.

6.8.1 Cross-Site Scripting

Our counterfeit mining attack (§5.4) leverages cross-site scripting (XSS). The attacks and defenses of XSS have been an ongoing cat-and-mouse game for years. One straightforward defense is to validate and sanitize input on the server side, but this places a heavy burden on web developers for code correctness. XSS-GUARD (Bisht and Venkatakrishnan, 2008)

utilizes taint-tracking technology to centralize validation and sanitization on the server-side. Blueprint (Louw and Venkatakrishnan, 2009), Noncespaces (Gundy and Chen, 2012), DSI (Nadji et al., 2014), and CSP (Stamm et al., 2010) adopt the notion of client-side HTML security policies (Weinberger et al., 2011) to defend XSS. Large-scale studies have also been undertaken to examine the prevalence of DOM-based XSS vulnerabilities (Lekies et al., 2013) and the security history of the Web's client side (Stock et al., 2017), concluding that client-side XSS stagnates at a high level. To remedy the shortcomings of string-based comparison methods, taint-aware XSS filtering has been proposed to thwart DOM-based XSS (Stock et al., 2014). DOMPurify (Heiderich and Späth, 2017) is an open-source library designed to sanitize HTML strings and document objects from DOM-based XSS attacks. Recently, attacks leveraging script gadgets have been discovered that circumvent all currently existing XSS mitigations (Lekies et al., 2017). We showed that in-browser crypomining is susceptible to such gadget-powered XSS attacks to hijack Wasm mining scripts.

Although our SEISMIC defense detects and warns users about cryptomining activities introduced through XSS, XSS can still potentially confuse users into responding inappropriately to the warnings. For example, attackers can potentially leverage XSS to obfuscate the provenance of cryptomining scripts, causing users to misattribute them to legitimate page publishers. This longstanding attribution problem is a continuing subject of ongoing study (cf., Rowe, 2015).

6.9 Related Web Script Defenses

A cluster of research on defense mechanisms is also related to our work. ObliviAd (Backes et al., 2012) is an online behavioral advertising system that aims to protect visitors' privacy. MadTracer (Li et al., 2012) leverages decision tree models to detect malicious web advertisements. JStill (Xu et al., 2013) compares the information from both static analysis and runtime inspection to detect and prevent obfuscated malicious JS code. Analysis of access

control mechanisms in the browser has observed that although CSP is a clean solution in terms of access control, XS-search attacks can use timing side-channels to exfiltrate data from even prestigious services, such as Gmail and Bing (Gelernter and Herzberg, 2015). Blacklist services provided by browsers to thwart malicious URLs have been shown to be similarly limited (Virvilis et al., 2015). BridgeScope (Yang et al., 2017) was proposed to precisely and scalably find JS bridge vulnerabilities. Commix (Stasinopoulos et al., 2018) automates the detection and exploitation of command injection vulnerabilities in web applications. Our system is orthogonal to these prior defense mechanisms, in that it profiles Wasm execution and helps users detect unauthorized in-browser mining of cryptocurrencies.

6.10 Semantic Malware Detection and Obfuscation

Our semantic signature-matching approach to cryptomining detection is motivated by the widespread belief that it is more difficult for adversaries to obfuscate semantic features than syntactic ones (cf., Christodorescu et al. (2005); Kinder et al. (2005)). Prior work has demonstrated that semantic features can nevertheless be obfuscated with sufficient effort, at the cost of reduced performance (e.g., Moser et al. (2007); Wu et al. (2010)). While such semantic obfuscations could potentially evade our SEISMIC monitors, we conjecture that the performance penalty of doing so could make obfuscated cryptojacking significantly less profitable for attackers. Future work should investigate this conjecture once semantically obfuscated cryptojacking attacks appear and can be studied.

CHAPTER 7

CONCLUSION

7.1 Dissertation Summary

Hardening native software applications against various kinds of software hijacking attacks has been recognized as one of the most important steps in defending software ecosystems. However, compatibility and applicability limitations constitute the greatest barriers to more widespread state-of-the-art software protection frameworks adoption. This dissertation presents a detailed analysis of compatibility, modularity, expressiveness and performance for hardening native software security in the real world.

Chapter 2 presents ConFIRM, which is a novel evaluation methodology and benchmarking suite designed specifically for measuring compatibility and applicability characteristics relevant to control-flow 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. Reevaluation of twelve major CFI frameworks using ConFIRM reveals that state-of-the-art CFI solutions are compatible with only about half of the CFI-relevant code features and coding idioms needed to protect large production software systems that are frequently targeted by cybercriminals.

With understanding the compatibility limitations discussed in Chapter 2, in order to augment CFI protections to scale to more architectures and larger software products, Chapter 3 presents OFI, which is the first work to extend CFI security protections to the significant realm of mainstream software that contains immutable system modules with large, object-oriented APIs. A prototype implementation of OFI for Microsoft COM indicates that the approach is feasible for large, complex, object-oriented APIs on the order of tens of thousands of methods. A case study detailed in Chapter 4 indicates that OFI imposes negligible performance overhead for some common-case, real-world applications and effectively protects native software from control-flow hijacking attacks.

Chapter 5 then presents SEISMIC, a novel semantic-based method of detecting and interrupting unauthorized, browser-based cryptomining. SEISMIC is a Wasm in-line script monitoring system that allows users to monitor and consent to cryptomining activities with acceptable overhead.

Finally, in Chapter 6 we discuss associated work in the area.

To summarize, we consider these works to be a significant contribution to the field, because prior CFI evaluations primarily prioritize expressiveness and performance over compatibility and modularity of a CFI solution. In contrast to the prior approaches, the focus of this dissertation has been on discovering CFI compatibility issues and extending CFI to a large class of software products that are event-driven and component-based.

7.2 Future Work

At the time of publication, ConFIRM consists of 20 compatibility metrics that are identified by spending many hundreds of man-hours on applying CFI algorithms to large software products. Although ConFIRM is already a rigorous baseline for CFI solution evaluation, it might be incomplete. Future research should consider extending this list to cover more code features and coding idioms that must be preserved, in order to augment a wider domain of applications with CFI enforcement.

As discussed in Chapter 2, the reevaluation of CFI solutions using ConFIRM reveals a substantial gap between CFI theory and practice. For instance, our cross-thread stack-smashing attack is able to break every CFI sandbox that we tested, and all the publicly available CFI frameworks fail to completely protect COTS software products that rely on runtime code generation (e.g., Google Chrome, Microsoft Office, and others). Operating system and hardware developers should consider exploring novel techniques to bridge this gap.

Since we have shown that software-only CFI frameworks (e.g., MCFG, LLVM-CFI, etc.) suffer from compatibility problems, the proposal for Intel CET (Intel, 2017) has drawn attention of both industry and academia to hardware-assisted solutions. But it is unclear whether CET is able to surmount the challenging compatibility issues before its official release. Future work should therefore consider testing the functionalities of the CET-hardened applications for transparency and correctness.

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 Transactions 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 and Security (CCS)*.

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 and Privacy (S&P)*, pp. 263–277.

Andersen, S. (2004). Part 3: Memory protection technologies. In V. Abella (Ed.), *Changes in Functionality in Windows XP Service Pack 2*. Microsoft TechNet. http://technet.microsoft.com/en-us/library/bb457155.aspx.

Androulaki, E., G. Karame, and S. Capkun (2014). Hiding transaction amounts and balances in Bitcoin. In *Proceedings of the 7th ACM International Conference on Trust and Trustworthy Computing (TRUST)*, pp. 161–178.

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

Backes, M., A. Kate, and M. Maffei (2012). ObliviAd: Provably secure and practical online behavioral advertising. In *Proceedings of the 33th IEEE Symposium on Security and Privacy (S&P)*, pp. 257–271.

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

Bisht, P. and V. Venkatakrishnan (2008). XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 23–43.

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.

Bonneau, J., A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten (2015). SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, pp. 104–121.

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

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

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

Capital One (2019, September). Information on the Capital One cyber incident. https://www.capitalone.com/facts2019/.

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 Conference on Security (USENIX)*, pp. 161–176.

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 Network and Distributed System Security Symposium (NDSS)*.

Chochlík, M. and A. Naumann (2016). Static reflection (revision 4). C++ Standards Committee Paper P0194R0.

Christodorescu, M., S. Jha, S. A. Seshia, D. Song, and R. E. Bryant (2005). Semantics-aware malware detection. In *Proceedings of the 26th IEEE Symposium on Security & Privacy (S&P)*, pp. 32–46.

Coker, R. (2016). Disk Performance Benchmark Tool - Bonnie. https://www.coker.com.au/bonnie++.

Conti, M., S. J. Crane, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi (2015). Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications and Security (CCS)*, pp. 952–963.

Conti, M., A. Gangwal, and S. Ruj (2018). On the economic significance of ransomware campaigns: A Bitcoin transactions perspective. arXiv:1804.01341.

Crane, S. J., P. Larsen, S. Brunthaler, and M. Franz (2013). Booby trapping software. In *Proceedings of the 2013 on New Security Paradigms Workshop (NSPW)*, pp. 95–106.

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.

Crary, K., R. Harper, and S. Puri (1999). What is a recursive module? In *Proceedings of the 20th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 50–63.

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 Network and 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 52th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6.

Davi, L., P. Koeberl, and A.-R. Sadeghi (2014). Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the 51th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6.

Davi, L., A.-R. Sadeghi, D. Lehmann, and F. Monrose (2014). Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 401–416.

de Clercq, R., R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. D. Bosschere, B. Preneel, B. D. Sutter, and I. Verbauwhede (2016). SOFIA: Software and control flow integrity architecture. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1172–1177.

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

DeMocker, J. (2017, November). WebAssembly support now shipping in all major browsers. *Mozilla Blog*.

Ding, R., C. Qian, C. Song, B. Harris, T. Kim, and W. Lee (2017). Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Security Symposium*, pp. 131–148.

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

Duffy, J. (2008). *Concurrent Programming on Windows*. Addison-Wesley.

Duncan, I. (2019, May). Baltimore estimates cost of ransomware attack at $18.2 million as government begins to restore email accounts. https://www.baltimoresun.com/maryland/baltimore-city/bs-md-ci-ransomware-email-20190529-story.html.

Erlingsson, Ú., M. Abadi, M. Vrable, M. Budiu, and G. C. Necula (2006). XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 75–88.

Erlingsson, Ú. and F. B. Schneider (1999). SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pp. 87–95.

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.

Exton, C., D. Watkins, and D. Thompson (1997). Comparisons between CORBA IDL & COM/DCOM MIDL: Interfaces for distributed computing. In *Proceedings of the 25th Technology of Object-Oriented Languages and Systems Conference (TOOLS)*, pp. 15–32.

Findler, R. B. and M. Felleisen (2002). Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 48–59.

Fruhlinger, J. (2020, February). Marriott data breach faq: How did it happen and what was the impact? https://www.csoonline.com/article/3441220/marriott-data-breach-faq-how-did-it-happen-and-what-was-the-impact.html.

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.

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.

Gelernter, N. and A. Herzberg (2015). Cross-site search attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pp. 1394–1405.

Göktas, E., E. Athanasopoulos, H. Bos, and G. Portokalidis (2014). Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 575–589.

Goodin, D. (2018, January). Now even YouTube serves ads with CPU-draining cryptocurrency miners. *Ars Technica*.

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

Gray, D. N., J. Hotchkiss, S. LaForge, A. Shalit, and T. Weinberg (1998). Modern languages and Microsoft's component object model. *Communications of the ACM (CACM) 41*(5), 55–65.

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.

Gundy, M. V. and H. Chen (2012). Noncespaces: Using randomization to defeat cross-site scripting attacks. *Computers & Security 31*(4), 612–628.

Gupta, S. and B. Gupta (2017). Cross-site scripting (XSS) attacks and defense mechanisms: Classification and state-of-the-art. *International Journal of System Assurance Engineering Management 8*(1), 512–530.

Haas, A., A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien (2017). Bringing the Web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 185–200.

Haller, I., E. Göktas, E. Athanasopoulos, G. Portokalidis, and H. Bos (2015). ShrinkWrap: VTable protection without loose ends. In *Proceedings of the 31th Annual Computer Security Applications Conference (ACSAC)*, pp. 341–350.

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

Hardy, N. (1988). The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review 22*(4), 36–38.

Heiderich, M. and C. Späth (2017). DOMPurify: Client-side protection against XSS and markup injection. In *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS)*, pp. 116–134.

Hruska, J. (2017, September). Browser-based mining malware found on Pirate Bay, other sites. *ExtremeTech*.

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.

Huang, D. Y., H. Dharmdasani, S. Meiklejohn, V. Dave, C. Grier, D. Mccoy, S. Savage, N. Weaver, A. C. Snoeren, and K. Levchenko (2014). Botcoin: Monetizing stolen cycles. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*.

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

Kafeine (2018, January). Smominru Monero mining botnet making millions for operators. *ProofPoint Threat Insight*.

Karame, G., E. Androulaki, and S. Capkun (2012). Double-spending fast payments in Bitcoin. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pp. 906–917.

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.

Keramidas, R. (2017, September). Stop coin mining in the browser with No Coin. https://ker.af/stop-coin-mining-in-the-browser-with-no-coin.

Kinder, J., S. Katzenbeisser, C. Schallhart, and H. Veith (2005). Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 174–187.

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

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.

Lau, H. (2017, December). Browser-based cryptocurrency mining makes unexpected return from the dead. *Sympantec Threat Intelligence*.

Lekies, S., K. Kotowicz, S. Groß, E. V. Nava, and M. Johns (2017). Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pp. 1709–1723.

Lekies, S., B. Stock, and M. Johns (2013). 25 million flows later: Large-scale detection of DOM-based XSS. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pp. 1193–1204.

Li, Z., K. Zhang, Y. Xie, F. Yu, and X. Wang (2012). Knowing your enemy: Understanding and detecting malicious web advertising. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pp. 906–917.

Liao, K., Z. Zhao, A. Doupé, and G.-J. Ahn (2016). Behind closed doors: Measurement and analysis of CryptoLocker ransoms in Bitcoin. In *Proceedings of the 11th APWG Symposium on Electronic Crime Research (eCrime)*, pp. 1–13.

Liao, S. (2017, September). Showtime websites secretly mined user CPU for cryptocurrency. *The Verge*.

Ligatti, J., L. Bauer, and D. Walker (2009). Run-time enforcement of nonsafety policies. *ACM Transactions on Information and Systems Security (TISSEC) 12*(3).

Louw, M. T. and V. N. Venkatakrishnan (2009). Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*, pp. 331–346.

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.

Mayo, B. (2019, January). Major iPhone FaceTime bug lets you hear the audio of the person you are calling ... before they pick up. *9to5Mac*. https://9to5mac.com/2019/01/28/facetime-bug-hear-audio.

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

McIlroy, M. (1968). Mass produced software components. In P. Naur and B. Randell (Eds.), *Proceedings of the NATO Software Engineering Conference*, pp. 138–156.

McMillen, D. (2017, September). Network attacks containing cryptocurrency CPU mining tools grow sixfold. *IBM X-Force SecurityIntelligence*.

Meshkov, A. (2017, November). Cryptojacking surges in popularity growing by 31% over the past month. *AdGuard Research*.

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

Miguel, C., C. Manuel, M. Jean-Philippe, P. Marcus, A. Periklis, D. Austin, B. Paul, and B. Richard (2009). Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pp. 45–58.

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

Moore, M. (2019, December). Facebook data breach sees millions of user personal details leaked online. https://www.techradar.com/news/millions-of-facebook-user-phone-numbers-leaked-online.

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.

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 International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pp. 423–444.

Nadji, Y., P. Saxena, , and D. Song (2014). Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*.

Nagarakatte, S., J. Zhao, M. M. K. Martin, and S. Zdancewic (2010). CETS: Compiler-enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management (ISMM)*, pp. 31–40.

Neumann, R. and A. Toro (2018, April). In-browser mining: Coinhive and WebAssembly. *Forcepoint Security Labs*. https://blogs.forcepoint.com/security-labs/browser-mining-coinhive-and-webassembly.

Nick, C., C. George, A. Elias, and I. Sotiris (2016). HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 38–49.

Niu, B. and G. Tan (2013). Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 21st 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 SIGPLAN 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 23rd 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.

OAG, New Jersey (2015, May). New Jersey Division of Consumer Affairs obtains settlement with developer of Bitcoin-mining software found to have accessed New Jersey computers without users' knowledge or consent. Office of the Attorney General, Department of Law & Public Safety, State of New Jersey.

Obes, J. L. and J. Schuh (2012, May). A tale of two pwnies (part 1). Chromium Blog. https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html.

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

Pawar, M., R. Patel, and N. Chaudhari (2013). Interoperability between .Net framework and Python in component way. *International Journal of Computer Science Issues (IJCSI) 10*(1), 165–170.

PaX Team (2003). Pax address space layout randomization (aslr). https://pax.grsecurity.net/docs/aslr.txt.

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.

Phung, P. H., M. Monshizadeh, M. Sridhar, K. W. Hamlen, and V. Venkatakrishnan (2015). Between worlds: Securing mixed JavaScript/ActionScript multi-party web content. *IEEE Transactions on Dependable and Secure Computing (TDSC) 12*(4), 443–457.

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)*.

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

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

Rowe, N. C. (2015). The attribution of cyber warfare. In J. A. Green (Ed.), *Cyber Warfare: A multidisciplinary Analysis*, Routledge Studies in Conflict, Security and Technology. Routledge.

Sadeghi, A.-R., L. Davi, and P. Larsen (2015). Securing legacy software against real-world code-reuse exploits: Utopia, alchemy, or possible future? In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pp. 55–61.

Santora, M. (2019, July). 5 million bulgarians have their personal data stolen in hack. https://www.nytimes.com/2019/07/17/world/europe/bulgaria-hack-cyberattack.html.

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

Schneider, F. B. (2000). Enforceable security policies. *ACM Transactions on Information and Systems 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 and Privacy (S&P)*, pp. 745–762.

Sridhar, M. and K. W. Hamlen (2010a). ActionScript in-lined reference monitoring in Prolog. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pp. 149–151.

Sridhar, M. and K. W. Hamlen (2010b). Model-checking in-lined reference monitors. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pp. 312–327.

Stamm, S., B. Sterne, and G. Markham (2010). Reining in the Web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pp. 921–930.

Stasinopoulos, A., C. Ntantogian, and C. Xenakis (2018). Commix: Automating evaluation and exploitation of command injection vulnerabilities in web applications. *International Journal of Information Security*, 1–24.

Stock, B., M. Johns, M. Steffens, and M. Backes (2017). How the Web tangled itself: Uncovering the history of client-side Web (in)security. In *Proceedings of the 26th USENIX Security Symposium*, pp. 971–987.

Stock, B., S. Lekies, T. Mueller, P. Spiegel, and M. Johns (2014). Precise client-side protection against DOM-based cross-site scripting. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 655–670.

Szalachowski, P. (2018). Towards more reliable Bitcoin timestamps. arXiv:1803.09028.

Tahir, R., M. Huzaifa, A. Das, M. Ahmad, C. Gunter, F. Zaffar, M. Caesar, and N. Borisov (2017). Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pp. 287–310.

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

The Wine Committee (2019). Wine 4.0. 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 and Privacy (S&P)*, pp. 934–953.

van Saberhagen, N. (2013, October). CryptoNote v 2.0. Technical report, CryptoNote Technology.

Vandevoorde, D. and N. M. Josuttis (2002). *C++ Templates: The Complete Guide*. Addison-Wesley.

Vinoski, S. (1997). CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine 35*(2), 46–55.

Virvilis, N., A. Mylonas, N. Tsalis, and D. Gritzalis (2015). Security busters: Web browser security vs. suspicious sites. *Computers & Security 52*, 90–105.

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, 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, W., B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao (2018). SEISMIC: Secure in-lined script monitors for interrupting cryptojacks. In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS)*, pp. 122–142.

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.

WebAssembly Community Group (2018). Security. http://webassembly.org/docs/security.

Weinberger, J., A. Barth, and D. Song (2011). Towards client-side HTML security policies. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security (HotSec)*, pp. 8–8.

WhiteHat Security (2017). Application security statistics report, vol. 12.

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.

Wollgast, P., R. Gawlik, B. Garmany, B. Kollenda, and T. Holz (2016). Automated multi-architectural discovery of cfi-resistant code gadgets. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS)*, pp. 602–620.

Wu, Z., S. Gianvecchio, M. Xie, and H. Wang (2010). Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pp. 536–546.

Xia, Y., Y. Liu, H. Chen, and B. Zang (2012). CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)*, pp. 1–12.

Xu, W., F. Zhang, and S. Zhu (2013). JStill: Mostly static detection of obfuscated malicious JavaScript code. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 117–128.

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

Xu, X., W. Wang, K. W. Hamlen, and Z. Lin (2018). Towards interface-driven COTS binary hardening. In *Proceedings of the 3rd Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pp. 20–26.

Yang, G., A. Mendoza, J. Zhang, and G. Gu (2017). Precisely and scalably vetting JavaScript bridge in Android hybrid apps. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pp. 143–166.

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 and Privacy (S&P)*, pp. 79–93.

Yuan, P., Q. Zeng, and X. Ding (2015). Hardware-assisted finegrained code-reuse attack detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pp. 66–85.

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 and 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 (2018). HCIC: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things Journal*, 1–1.

Zhang, M. and R. Sekar (2013). Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Conference on Security (USENIX)*, 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

Xiaoyang Xu was born in October 1988, in Jinan, China, where he spent his childhood playing Nintendo Game Boy games with his friends, piano with his mother, and toys when he was alone. As a huge fan of LEGO, Xiaoyang loved to take things apart and reassemble them, from toys to electronics. This introduced him to the world of programming later in his life and led him to pursue a Bachelor of Computer Science from Shandong University.

After interning at IBM Research in Beijing for six months, in 2012, Xiaoyang decided to pursue a master's degree in Computer Science at The University of Texas at Dallas. During his master's program, Xiaoyang took Advanced Programming Language and Language-Based Security from Dr. Kevin W. Hamlen. This showed him an even more fascinating world and proved to be a turning-point for him – he moved to the doctoral program in 2014. Under Dr. Hamlen's supervision, Xiaoyang's research was mainly focusing on native software hardening.

After completing his PhD, Xiaoyang will start his career at Google.

CURRICULUM VITAE

Xiaoyang Xu

February 24, 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: `xiaoyang.xu@utdallas.edu`

Educational History:

B.E., Computer Science and Technologh, Shandong University, 2011
Ph.D., Computer Science, The University of Texas at Dallas, 2020

Employment History:

Research Intern, IBM Research in Beijing, June 2011 – December 2011
Research Assistant, The University of Texas at Dallas, September 2014 – May 2020

Publications:

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.* Santa Clara, CA. August 2019. *[acceptance rate: 16%]*

Xiaoyang Xu, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. **Towards Interface-Driven COTS Binary Hardening**, In *Proceedings of the 3rd Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*. pp. 20-26, October 2018.

Wenhao Wang, Benjamin Ferrell, **Xiaoyang Xu**, Kevin W. Hamlen, and Shuang Hao. **SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks**. In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS)*, pp. 122-142, September 2018. *[acceptance rate: 20%]*

Wenhao Wang, **Xiaoyang Xu**, and Kevin W Hamlen. **Object Flow Integrity**. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pp. 1909-1924, November 2017. *[acceptance rate: 18%]*