Enhancing Software Security through Code Diversification Verification, Control-flow Restriction, and Automatic Compartmentalization

Jae-Won Jang

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Jin-Hee Cho
Ryan Gerdes
Freek Verbeek
Haining Wang

July 16th, 2024
Blacksburg, Virginia

Keywords: Formal Verification, Control-Hijacking Prevention, Compartmentalization, Memory Safety, Security
Copyright 2024, Jae-Won Jang

Enhancing Software Security through Code Diversification Verification, Control-flow Restriction, and Automatic Compartmentalization

Jae-Won Jang

(ABSTRACT)

In today's digital age, computer systems are prime targets for adversaries due to the vast amounts of sensitive information stored digitally. This ongoing cat-and-mouse game between programmers and adversaries forces security researchers to continually develop novel security measures. Widely adopted schemes like NX bits have safeguarded systems against traditional memory exploits such as buffer overflows, but new threats like code-reuse attacks quickly bypass these defenses. Code-reuse attacks exploit existing code sequences, known as gadgets, without injecting new malicious code, making them challenging to counter. Additionally, input-based vulnerabilities pose significant risks by exploiting external inputs to trigger malicious paths. Languages like C and C++ are often considered unsafe due to their tendency to cause issues like buffer overflows and use-after-free errors. Addressing these complex vulnerabilities requires extensive research and a holistic approach.

This dissertation initially introduces a methodology for verifying the functional equivalence between an original binary and its diversified version. The Verification of Diversified Binary (**VDB**) algorithm is employed to determine whether the two binaries—the original and the diversified—maintain functional equivalence. Code diversification techniques modify the binary compilation process to produce functionally equivalent yet different binaries from the same source code. Most code diversification techniques focus on analyzing non-functional properties, such as whether the technique improves security. The objective of this contribution is to enable the use of untrusted diversification techniques in essential applications. Our evaluation demonstrates that the **VDB** algorithm can verify the functional equivalence of 85,315 functions within binaries from the GNU Coreutils 8.31 benchmark suite.

Next, this dissertation proposes a binary-level tool that modifies binaries to protect against control-flow hijacking attacks. Traditional approaches to guard against ROP attacks either introduce significant overhead, require hardware support, or need intimate knowledge of the binary, such as source code. In contrast, this contribution does not rely on source code nor the latest hardware technology (e.g., Intel Control-flow Enforcement Technology). Instead, we show that we can precisely restrict control flow transfers from transferring to non-intended paths even without these features. To that end, this contribution proposes a novel control-flow integrity policy based on a deny list called Control-flow Restriction (**CFR**). **CFR** determines which control flow transfers are allowed in the binary without requiring source code. Our implementation and evaluation of **CFR** show that it achieves this goal with an

average runtime performance overhead for commercial off-the-shelf (COTS) binaries in the range of 5.5% to 14.3%. In contrast, a state-of-the-art binary-level solution such as BinCFI has an average overhead of 61.5%.

Additionally, this dissertation explores leveraging the latest hardware security primitives to compartmentalize sensitive data. Specifically, we use a tagged memory architecture introduced by ARM called the Memory Tagging Extension (MTE), which assigns a metadata tag to a memory location that is associated with pointers referencing that memory location. Although promising, ARM MTE suffers from predictable tag allocation on stack data, vulnerable plain-text metadata tags, and lack of fine-grained memory access control. Therefore, this contribution introduces Shroud to enhance data security through compartmentalization using MTE and protect MTE's tagged pointers' vulnerability through encryption. Evaluation of Shroud demonstrates its security effectiveness against non-control-data attacks like Heartbleed and Data-Oriented Programming, with performance evaluations showing an average overhead of 4.2% on lighttpd and 2% on UnixBench. Finally, the NPB benchmark measured Shroud's overhead, showing an average runtime overhead of 2.57%.

The vulnerabilities highlighted by exploits like Heartbleed capitalize on external inputs, underscoring the need for enhanced input-driven security measures. Therefore, this dissertation describes a method to improve upon the limitations of traditional compartmentalization techniques. This contribution introduces an Input-Based Compartmentalization System (IBCS), a comprehensive toolchain that utilizes user input to identify data for memory protection automatically. Based on user inputs, IBCS employs hybrid taint analysis to generate sensitive code paths and further analyze each tainted data using novel assembly analyses to identify and enforce selective targets. Evaluations of IBCS demonstrate its security effectiveness through adversarial analysis and report an average overhead of 3% on Nginx.

Finally, this dissertation concludes by revisiting the problem of implementing a classical technique known as Software Fault Isolation (SFI) on an x86-64 architecture. Prior works attempting to implement SFI on an x86-64 architecture have suffered from supporting a limited number of sandboxes, high context-switch overhead, and requiring extensive modifications to the toolchain, jeopardizing maintainability and introducing compatibility issues due to the need for specific hardware. This dissertation describes x86-based Fault Isolation (XFI), an efficient SFI scheme implemented on an x86-64 architecture with minimal modifications needed to the toolchain, while reducing complexity in enforcing SFI policies with low performance (22.48% average) and binary size overheads (2.65% average). XFI initializes the sandbox environment for the rewritten binary and, depending on the instructions, enforces data-access and control-flow policies to ensure safe execution. XFI provides the security benefits of a classical SFI scheme and offers additional protection against several classes of side-channel attacks, which can be further extended to enhance its protection capabilities.

Enhancing Software Security through Code Diversification Verification, Control-flow Restriction, and Automatic Compartmentalization

Jae-Won Jang

(GENERAL AUDIENCE ABSTRACT)

In today's digital age, cyber attackers frequently target computer systems due to the vast amounts of sensitive information they store. As a result, security researchers must constantly develop new protective measures. Traditional defenses like NX bits have been effective against memory exploits, but new threats like code-reuse attacks, which leverage existing code without introducing new malicious code, present new challenges. Additionally, vulnerabilities in languages like C and C++ further complicate security efforts. Addressing these issues requires extensive research and a comprehensive approach.

This dissertation introduces several innovative techniques to enhance computer security. First, it presents a method to verify that a diversified program is functionally equivalent to its original version, ensuring that security modifications do not alter its intended functions. Next, it proposes a technique to prevent control-flow hijacking attacks without requiring source code or advanced hardware. Then, the dissertation explores leveraging advanced hardware, such as ARM's Memory Tagging Extension, to protect sensitive data, demonstrating robust security against attacks like Heartbleed. Recognizing that adversaries often use external inputs to exploit vulnerabilities, this dissertation introduces Input-Based Compartmentalization to automatically protect memory based on user input. Finally, an efficient implementation of a well-known security technique called Software Fault Isolation on x86-64 architecture ensures safe execution with low overhead. These advancements collectively enhance the robustness of computer systems against modern cyber threats.

Dedication

To my parents Deoksoon Kim and Stanton Wortham.

Acknowledgments

First and foremost, I would like to thank my advisor, mentor, and friend, Professor Binoy Ravindran, for giving me a second opportunity to pursue a Ph.D. and for providing honest guidance throughout my journey. During times when I felt lost, he encouraged me to speak candidly with him so he could offer sincere feedback. His timely interventions ensured that I continued to move forward. His positive demeanor, balanced with a touch of strictness when necessary, was perfect for helping me grow as a researcher and ultimately gave me the push I needed to cross the finish line. I am deeply grateful for his unwavering support.

Furthermore, I want to thank my committee members: Professor Jin-Hee Cho, Professor Ryan Gerdes, Professor Freek Verbeek, and Professor Haining Wang, for providing valuable insights and feedback on my dissertation. I deeply appreciated the detailed feedback from all members regarding my presentation and writing, which taught me invaluable lessons not only for my Ph.D. but also for my future career.

Additionally, I would like to thank my former advisor, Professor Swaroop Ghosh, for allowing me to start my career as a researcher, and Professor Xiaoguang Wang, for teaching me how to be a system security researcher and for providing a unique perspective on my work. Lastly, thank you, Professor Brian Smith, for allowing me to discuss my concerns when I needed it the most.

I am deeply grateful to my SSRG friends and colleagues, especially (in alphabetical order) Josh Bockenek, Mincheol Sung, and Sengming Yeoh, for their invaluable companionship and support during my Ph.D. journey. I also appreciate the time spent with friends outside the lab, such as Jongwoon Kim, Myojoong Kim, and Hanjun Park. Lastly, thank you to my longtime friends Anirudh Iyengar and Deivis Largo for their enduring support.

I want to thank Yi-Ting Chen, who has been extremely patient when I was stressed and provided unmatched kindness and support when needed. She is more than I could hope for, and I am thankful for her companionship and everything that has transpired thus far.

Finally, I would like to express my deepest gratitude to my family: my mom, Deoksoon Kim; my dad, Stanton Wortham; my uncle, Chun Kim; my aunt, Soyoung Ahn; and my sister, Dalis Kim. My mom's willingness to take the huge risk of immigrating to the USA and becoming a professor gave me a goal to pursue in life. My dad always took time off his busy schedule to offer me unwavering encouragement and solutions when I needed them most. My uncle and aunt were always there when I needed them. Lastly, thank you to my sister for your care. Their support allowed me to pursue this opportunity without obstacles.

This work is supported in part by the US Office of Naval Research (ONR) under grants N00014-17-1-2297, N00014-18-1-2022, and N00014-16-1-2818; the US Naval Surface Warfare Center Dahlgren Division/Naval Engineering Education Consortium (NEEC) under grants N00174-20-1-0009 and N00174-16-C-0018; and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4028 and DARPA under Agreement No. HR.00112090028.

Contents

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

1.1 Motivation 3

1.1.1 Code Diversification 3

1.1.2 Preventing Control-flow Hijacking 3

1.1.3 Challenges of Utilizing ARM MTE 4

1.1.4 Automatically Compartmentalizing Sensitive Data 5

1.1.5 Efficient x86-based Software Fault Isolation 5

1.1.6 Amalgamation 6

1.2 Summary of Research Contributions 7

1.2.1 **VDB**: Verification of Code Diversification Techniques 7

1.2.2 **CFR**: Control-flow Restriction 9

1.2.3 **Shroud**: Automatic Tagged Memory Compartmentalization 10

1.2.4 **IBCS**: Input-Based Compartmentalization System 11

1.2.5 **XFI**: x86-based Fault Isolation 11

1.3 Dissertation Organization 12

2 Background **13**

2.1 Data-Based Attacks 13

2.1.1 Control-Data Attacks 13

2.1.2 Non-Control-Data Attacks 14

2.2 Software Security Primitives 14

2.2.1 Security Policy Enforcement 14

2.2.2 Code Diversification 15
2.2.3 Code Obfuscation 15
2.2.4 Software Fault Isolation (SFI) 15
2.3 Hardware Security Primitives 16
2.3.1 Tagged Memory Architecture 16
2.3.2 ARM Memory Tagging Extension (MTE) 16
2.3.3 Intel MPK 17
2.4 Disassembly 17
2.4.1 Linear Sweep 18
2.4.2 Recursive Traversal 18
2.4.3 Assembly Instructions 18
2.5 Formal Verification 19
2.5.1 Model Checking 19
2.5.2 Symbolic Execution 20
2.6 Compartmentalization Spaces 20
2.7 Taint Analysis 21

3 Related Work **22**
3.1 Static and Dynamic Analysis 22
3.2 Equivalence Checking Using Symbolic Execution 23
3.3 Low-level Formal Verification 23
3.4 Code-Reuse Attacks 24
3.4.1 Return-oriented Programming (ROP) 24
3.4.2 Advanced Variant of Code-reuse Attacks 24
3.5 Protection Schemes for Code-Reuse Attacks 26
3.6 Non-Control-Data Attacks 27
3.6.1 Data-Oriented Programming (DOP) 27
3.7 Protection Schemes for Non-Control-Data Attacks 28
3.8 Compartmentalization 29

3.9 Software Fault Isolation (SFI) 32

4 VDB: Verification of Code Diversification Techniques 35

4.1 Overview of Methodology 35

4.1.1 Disassembly 37

4.1.2 Control-flow Graph Construction 37

4.1.3 Local Variable Normalization 38

4.1.4 CFG Comparison 38

4.1.5 Output 39

4.2 Divergence-sensitive Stuttering Bisimulation 39

4.3 Transition Relation 40

4.4 State Comparison Function 41

5 VDB Algorithm 43

5.1 Symbolic Execution 43

5.2 $\texttt{rsp}_0$ Substitution 45

5.3 Stuttering Bisimulation Check 46

6 VDB Evaluation 48

6.1 Setup Environment 48

6.2 Soundness Evaluation 48

6.3 Discussion 50

6.4 Summary 51

7 Control-flow Restriction (CFR) 52

7.1 Control-flow Restriction and CFI Relation 53

7.1.1 Control-flow Policy 53

7.1.2 Control-flow Integrity 53

7.1.3 Control-flow Restriction 55

7.2 Formal Definitions 57

7.2.1 $\mathbf{CFR}_0$ Policy . . . 57
7.2.2 $\mathbf{CFR}_1$ Policy . . . 57
7.2.3 $\mathbf{CFR}_2$ Policy . . . 58
7.3 Motivating Example . . . 58
7.3.1 Attack Vectors . . . 59
7.3.2 CFR and CFI Comparisons . . . 59
7.4 CFR Design Overview . . . 60
7.4.1 Deny List Generation . . . 60
7.4.2 Byte Packing . . . 61
7.4.3 Disassembly . . . 61
7.4.4 Binary Rewriting . . . 61

8 CFR Evaluation 63
8.1 Setup Environment . . . 63
8.2 Runtime Overhead Evaluation . . . 64
8.3 Security Evaluation . . . 68
8.3.1 Policy Evaluation . . . 68
8.3.2 False Gadgets . . . 69
8.3.3 Adversarial Analysis . . . 70
8.4 Summary . . . 72

9 Shroud: Automatic Tagged Memory Compartmentalization 73
9.1 Motivating Example . . . 74
9.2 Shroud Design Overview . . . 77
9.2.1 Software Abstraction of Shroud . . . 77

10 Shroud Implementation 79
10.1 Static Taint Analysis . . . 79
10.2 Dynamic Taint Analysis . . . 80
10.3 Compartmentalization Analysis . . . 81

10.4 Encryption . . . 82

11 SHROUD Evaluation **84**

11.1 Setup Environment . . . 84

11.2 Runtime Overhead Evaluation . . . 85

11.3 Security Evaluation . . . 87

11.3.1 Threat Model and Assumptions . . . 88

11.3.2 Adversarial Analysis . . . 88

11.3.3 Sensitive Data Analysis . . . 89

11.4 Discussion . . . 91

11.5 Summary . . . 93

12 IBCS: Input-Based Compartmentalization System **94**

12.1 Motivating Example . . . 95

12.2 Challenges . . . 96

12.3 IBCS Design Overview . . . 98

13 IBCS Implementation **100**

13.1 Identification Phase . . . 100

13.1.1 Dynamic Taint Analysis . . . 100

13.1.2 Static Taint Analysis . . . 101

13.1.3 Static Variable Offset Analysis . . . 101

13.1.4 Static Pointer Offset Tree Analysis . . . 102

13.2 Enforcement Phase . . . 105

13.2.1 Assembly Lexical Analysis . . . 105

13.2.2 Fine-grained Compartment Creation . . . 107

13.2.3 Fine-grained Compartment Relocation . . . 108

13.2.4 Static Verification . . . 109

13.2.5 Runtime Enforcement . . . 109

14 IBCS Evaluation **110**
14.1 Runtime Overhead Evaluation . . . 110
14.2 Security Evaluation . . . 112
14.3 Discussion . . . 114
14.4 Summary . . . 117

15 XFI: x86-based Fault Isolation **118**
15.1 Challenges . . . 119
15.2 **XFI** Design Overview . . . 120
15.3 Design Considerations for **XFI** . . . 121

16 XFI Implementation **123**
16.1 **XFI** Preparation . . . 123
16.1.1 **XFI** Sandbox Scheme . . . 123
16.1.2 Compiler Modification . . . 124
16.1.3 Section Offset Analysis . . . 126
16.1.4 Sandbox Creation . . . 126
16.2 XFI Policy Enforcement . . . 127
16.2.1 Data-access Enforcement Policy . . . 127
16.2.2 Control-flow Enforcement Policy . . . 129

17 XFI Evaluation **134**
17.1 Runtime Overhead Evaluation . . . 134
17.2 Code Size Overhead Evaluation . . . 137
17.3 Security Evaluation . . . 139
17.4 Discussion . . . 140
17.5 Summary . . . 142

18 Conclusions and Future Work **143**
18.1 Future Work . . . 146

18.1.1 Binary-level Data-flow Generation 146
18.1.2 Data-flow Restriction 147
18.1.3 Scalable Pointer Offset Tree Analysis 148
18.1.4 Heap Compartmentalization of **IBCS** 149
18.1.5 Supporting Different Types of Binaries for **IBCS** and **XFI** 151

Bibliography **153**

List of Figures

2.1 Overview of MTE enforcing memory safety 16

2.2 Model checking overview [11] 20

4.1 Overview of verifying functional equivalence 35

4.2 Example 36

5.1 Symbolic execution example 44

6.1 The coverage rate per diversification techniques over all GNU `Coreutils` . . 50

7.1 Security and runtime overhead comparison 53

7.2 Motivating example: (a) code listing that is vulnerable to control-flow hijacking attack; (b) call graph; (c) extracted CFG of code listing from `angr` [201]; (d) intended/non-intended execution paths 58

7.3 Binary hardened through **CFR** 59

7.4 Design overview: (a) deny list generation; (b) binary rewriting and **CFR** enforcement during runtime 60

8.1 COTS applications evaluation. The percent difference formula used in this Figure is ((`modified` - `original`) / `original`) × 100 66

8.2 SPEC2006/2017 benchmarks evaluation. BinCFI results are from the Figure provided in [240], and **CFR** results are from our machine. BinCFI does not apply to the SPEC2017 benchmark. The percent difference formula used in this Figure is ((`modified` - `original`) / `original`) × 100 67

8.3 ROP Emporium adversarial analysis 71

9.1 An overview of how **Shroud** safeguards unsafe data to protect against non-control-data attacks 77

12.1 Hybrid Taint Analysis Call Graph 97

12.2 **IBCS** Design Overview 98

12.3 IBCS Toolchain 99

13.1 AsmST of the `movl $0, (%rax)` instruction 106

14.1 Visual Representation of IBCS Compartmentalization 113

15.1 XFI Design Overview 120
15.2 XFI Toolchain 121

18.1 Overview of DFG construction 146
18.2 Overview of DFR 148
18.3 IBCS Heap Compartmentalization Overview 150

List of Tables

1.1 State-of-the-art Code Diversification Techniques 8

3.1 Code-reuse Attacks Protection Schemes 25
3.2 Comparison of Shroud with State-of-the-Art Systems 28
3.3 Comparison of IBCS with State-of-the-Art Systems 30

6.1 The evaluation of our methodology on `GNU Coreutils 8.31` 49

8.1 SPEC2006/2017 benchmarks evaluations 64
8.2 Commercial Off-The-Shelf (COTS) applications evaluations 65
8.3 Commercial Off-The-Shelf (COTS) applications' test behaviors and false gadget counts 65
8.4 Security evaluation on ROP Emporium challenges 68

11.1 `httpd` relocation analysis evaluations 85
11.2 UnixBench benchmark evaluations 86
11.3 SNU_NPB (Class A) benchmark evaluations 86
11.4 lighttpd ApacheBench evaluations 87
11.5 Shroud analyses evaluations 89

14.1 Nginx Performance Evaluations 111
14.2 GNU `Coreutils` Rewriting Evaluations 112

17.1 Runtime Performance Overhead of Original and XFI Instrumented Binaries . 135
17.2 `tiny` web server Performance Evaluations 136
17.3 Code Size Overhead of Original and XFI Instrumented Binaries 137
17.4 Binary Size Comparison of Original and XFI Instrumented Binaries 138

18.1 Summary of Contributions in this Dissertation 144

Chapter 1

Introduction

Memory safety errors, such as out-of-bounds reads/writes, continue to pose significant risks in computer systems, primarily due to the prevalent use of memory-unsafe languages like C and C++ [152]. These languages provide developers with unrestricted access to underlying memory, which increases the likelihood of spatial (out-of-bounds dereference) and temporal (use after free) violations.

A binary, which results from compiling source code into machine code, represents a deterministic process. If the source code remains unchanged, the compiled binary will be identical across instances. This characteristic implies that any memory safety bugs within the binary can be consistently exploited, making security a critical aspect of software engineering.

Code-reuse attacks, such as Return-Oriented Programming (ROP), leverage existing code in memory to exploit vulnerabilities like buffer overflows, altering the intended behavior of a binary. These attacks involve assembling harmless code sequences into malicious ”gadgets”, each ending with an indirect control-flow transfer instruction (e.g., `callq rax, jmpq rax, retq`). Although individually harmless, these gadgets can be maliciously sequenced together. The sophistication of attacks that chain these gadgets highlights the complexity of modern exploits [27, 54, 80, 119, 178].

To counter such threats, state-of-the-art code diversification techniques alter the compilation process to generate varied binaries from the same source code, thereby complicating the reuse of exploits across different instances [97]. Although effective, these techniques require prior trust as significant system components, like compilers, must be altered before deployment.

The principle of least privilege, a security concept from the early 1970s, advocates limiting process operations to only what is necessary [185]. Software compartmentalization, which enforces this principle, decomposes software into isolated units with restricted capabilities, mitigating memory vulnerabilities and preventing privilege escalation, thereby enhancing overall system robustness [107, 121, 140, 169].

However, effectively implementing compartmentalization is challenging. Sensitive data are often protected by coarse-grained memory mechanisms, which are unsuitable for securing small data segments. Moreover, identifying which components to isolate often results in significant performance trade-offs and increases complexity in managing access [211].

First and foremost, this dissertation proves the correctness of code diversification techniques designed to protect against code-reuse attacks. It begins by presenting a formal definition of functional equivalence to support this claim. Following this, a novel security mechanism based on binary rewriting is proposed to further guard against such attacks.

Additionally, the dissertation introduces a secure methodology that utilizes an architectural feature to compartmentalize potentially vulnerable stack data, coupled with a software abstraction that provides developers with fine-grained control over memory access. To fully automate the process of compartmentalization, we first detail a comprehensive toolchain that automatically analyzes user input to identify sensitive data requiring protection. This toolchain incorporates innovative assembly code analyses and rewriting techniques to modify assembly code, ensuring its correctness and enhancing security.

Finally, to explore the limitations of our proposed fine-grained compartmentalization techniques (e.g., global variables, static data, control-flow enforcement), we investigate a low-overhead, coarse-grained compartmentalization approach in the form of a software fault isolation mechanism. This mechanism enforces data-access and control-flow policies to ensure process-level protection from adversaries.

1.1 Motivation

1.1.1 Code Diversification

There have been a plethora of proposed code diversification techniques presented in the literature [97, 124]. Many of these techniques are based on either recompilation from source code or rewriting at the machine-code level. Examples include `nop` insertion [95, 108], stack layout randomization [73], and relocation of basic blocks, functions, and instructions [117].

The `nop` insertion technique diversifies the binary by pseudo-randomly inserting `nop` instructions *in between* machine instructions. Stack layout randomization increases the size of the stack frame to a random size. Such change affects both the stack layout and the location of such modified function. Lastly, the relocation of different binary parts (basic blocks, functions, and instructions) prevents adversaries from consistently locating critical information. These diversification techniques disrupt the assumptions of adversaries and significantly lower the success rate of exploits developed for the vanilla (i.e., original) binary when applied to the diversified binary. These techniques are generally evaluated in terms of factors like increased entropy, performance overhead, and similar characteristics.

None of the existing techniques [76, 136, 144, 175] are evaluated on whether the diversification preserves functional equivalence, which reduces trust in deploying such techniques in production settings. Upon diversification, no proof or theorem shows that the diversified binary is functionally equivalent to the vanilla binary. A key challenge is that strictly speaking, they are *not* functionally equivalent. Various registers and memory locations may contain different values in both worlds. However, the binaries should be *similar* for *relevant* state parts, such as the registers storing in- and output.

Therefore, we argue the need for a method to prove the correctness.

1.1.2 Preventing Control-flow Hijacking

Control-flow hijacking prevention techniques consist of different mechanisms to allow or disallow *control flow transfers* of a binary. Traditional control-flow hijacking prevention approaches either introduce significant overhead, require hardware support, or require intimate knowledge of the binary, such as source code. For instance, using dynamic binary instrumentation (DBI) [35, 52, 53] can implement a control-flow hijacking prevention technique. However, DBI introduces significant runtime overhead. This is because DBI first needs to analyze the behavior of a binary, then an arbitrary code needs to be augmented *during* execution. Intel Pin [143] and the Valgrind [161] are some examples of DBI tools.

New hardware technologies such as Intel Control-flow Enforcement Technology (CET) [102] have helped to delegate security tasks to hardware to incur lower overhead than complete software solutions. Intel CET leverages a new instruction called `ENDBRANCH` to prevent

indirect branches from jumping to unintended locations. This `ENDBRANCH` instruction marks valid indirect `call/jmp` targets in the input program. However, legacy systems without such latest hardware remain vulnerable, and newer hardware is still subject to exploitation as many bypasses exist [58, 199].

Although the source code provides access to a plethora of information that is not easily obtainable with a binary [99, 112, 121], a complete binary level solution to prevent control-flow hijacking is still crucial today. There are few advantages of using the source code. First, a compiler using the source code can generate an accurate CFG. Second, using the source code makes it easier to remove/replace elements before the binary compilation to eliminate any potential threats (e.g., replace all indirect transfer instructions to direct transfer instructions). However, it is unreasonable to assume the source code will be available for any binaries.

Therefore, it is necessary to explore ways to prevent control-flow hijacking at a binary level.

1.1.3 Challenges of Utilizing ARM MTE

Tagged memory architectures, which assign metadata tags to all data in memory for self-identification, are crucial for enforcing security policies that help detect memory safety violations [70, 222, 237]. The Memory Tagging Extension (MTE) introduced by ARM in version v8.5-A exemplifies this approach by assigning tags to memory regions and corresponding pointers. If a pointer with an incorrect tag attempts access, MTE triggers a violation, signaling a discrepancy. Currently, MTE is primarily utilized in conjunction with hardware-assisted tools like AddressSanitizer [196] and MemTagSanitizer [141] to detect, rather than prevent, memory overflows and bugs.

There are significant challenges in leveraging MTE for direct security enhancements. The 4-bit tag length used by MTE provides only a 7% security barrier (1/16 chance of tag prediction), with tags stored as plaintext on pointers, thus posing serious security risks [165]. Additionally, MTE's default strategy for stack tagging does not truly randomize tags but sequentially assigns them (`0x01addr`, `0x02addr`, etc.), rendering the system vulnerable to inference attacks and limiting its effectiveness in securing data against sophisticated exploits [170, 248].

Moreover, the automation of tag management in MTE is lacking; in particular, heap tagging requires manual intervention[1]. Users are responsible for identifying and tagging sensitive data. This represents a significant limitation for applications that could benefit from enhanced tag entropy by transitioning sensitive data from stack to heap to avoid ”increment-by-one” strategy [39].

[1] https://github.com/google/sanitizers/wiki/Stack-instrumentation-with-ARM-Memory-\Tagging-Extension-(MTE)

These issues underscore the need for more robust tagging methodologies and the automation of security mechanisms to fully exploit the potential of memory tagging in real-world scenarios.

1.1.4 Automatically Compartmentalizing Sensitive Data

Recent advancements in CPU technology have introduced hardware features that significantly enhance compartmentalization efficiency. Notable developments include ARM's Pointer Authentication Code (PAC) [132] and Memory Tagging Extensions (MTE) [148], Intel's Secure Guard Extensions (SGX) [135] and Memory Protection Key (MPK), as well as the Capability Hardware Enhanced RISC Instructions (CHERI) [222]. These technologies aim to provide robust security frameworks by delegating complex tasks to hardware and enhancing access controls and memory safety.

Despite these technological strides, many current compartmentalization techniques still depend on developer annotations to define compartment boundaries [41, 86, 100, 211, 214], thereby increasing the Trusted Computing Base (TCB) and placing significant burdens on developers to balance security with performance. Some systems have automated this process, but their application is typically restricted to embedded systems that rely on specific architectural features [111, 179]. Automating the identification of compartmentalization targets without extensive annotations remains a formidable challenge.

Furthermore, while identifying compartmentalization targets is challenging, automating the enforcement of these compartment boundaries in both coarse-grained and fine-grained fashions is an even greater challenge. Most existing efforts, though effective, are reliant on the specific architecture they were designed for. This dependency underscores the urgent need for an architecture-agnostic approach that can both automate and effectively enforce compartmentalization, ensuring robust security across various computing platforms.

This highlights the importance of developing versatile, architecture-independent methods for compartmentalization that can automate the identification of necessary targets, provide boundary enforcement, and enhance security across diverse platforms without sacrificing performance.

1.1.5 Efficient x86-based Software Fault Isolation

Software Fault Isolation (SFI) is a classical software security technique that isolates a domain within a system to confine the impact of exploitation and protect other domains within the same system [215]. The technique involves modifying the original binary code to instrument necessary checks for any unsafe instructions (e.g., control-flow transfers and memory accesses). Originally, this idea was efficiently implemented on the well-aligned and simplistic nature of RISC architecture but became difficult to realize on more complex CISC

architectures [72, 193, 234].

Therefore, many researchers have begun delegating tasks to the latest hardware [190, 209, 211, 247] to enforce SFI policies more efficiently. These technologies enable new possibilities for researchers to explore different implementation approaches for SFI. However, in scenarios where such hardware features may not be available, it is still worthwhile to revisit the problem of software-based techniques. Existing techniques suffer from high performance overhead due to context switching, high complexity, and invasive modifications needed for deployment.

This provides an opportunity to explore a potentially efficient implementation of an SFI scheme on the CISC architecture that can be deployed without relying on hardware features.

1.1.6 Amalgamation

The examination of code diversification, control-flow hijacking prevention, and the challenges of utilizing ARM's Memory Tagging Extension (MTE) collectively underscores the complexity and diversity of software security challenges. Each method, while potent in its own domain, presents limitations that can be effectively addressed through a comprehensive and integrated security strategy.

Code diversification disrupts attack predictability but often fails to preserve functional equivalence, crucial for system integrity and reliability in operational environments. Control-flow hijacking prevention, such as Intel's CET, provides robust defenses against unauthorized code execution paths but typically requires source code and advanced hardware, limiting its applicability across legacy systems and various architectures. Additionally, ARM MTE enhances memory safety with tag-based access controls but faces issues like predictable tagging and a lack of automation in tag management, reducing its effectiveness against sophisticated inference attacks.

Moreover, the need for compartmentalization techniques that do not heavily rely on developer annotations underscores the challenges of achieving fine-grained security without imposing significant performance overheads or development complexities. This necessity is particularly critical in environments where security must be ensured without compromising operational efficiency or scalability. Although fine-grained security can protect against certain attack classes, it should not be the sole solution, highlighting the importance of a more coarse-grained defense that efficiently protects against various attack classes.

This dissertation advocates for a multi-layered approach combining code diversification, advanced control-flow integrity, enhanced memory tagging, and comprehensive compartmentalization and software-fault isolation strategies. By addressing the unique challenges posed by different security mechanisms, we aim to create a robust, effective security framework adaptable to evolving threats across various platforms. This integrated approach is essential for developing security solutions that provide comprehensive protection while maintaining system performance and usability.

1.2 Summary of Research Contributions

This dissertation defines a formal definition of different code diversification techniques and proposes a novel mechanism with a formal definition to defend against code-reuse attacks.

- **Formally Proving the Correctness of Code Diversification:** We introduce a formal definition of functional equivalence that is resilient to code diversification. Additionally, we present a scalable methodology to establish equivalence between the original (vanilla) and diversified binaries.
- **Binary-Level Control-Flow Deny List Policy:** Our novel security policy restricts indirect control-transfer (ICT) instructions to legitimate targets only, operating without any reliance on source code information.
- **Enhanced Memory Tagging Safeguard Mechanism:** We propose a novel scheme that leverages ARM's Memory Tagging Extension by automatically identifying and compartmentalizing vulnerable stack data. This approach utilizes hybrid taint analysis, one-time-pad encryption, and a dedicated library, providing robust defense against various non-control data attacks with manageable performance overhead.
- **Automatic Software Compartmentalization:** We have developed a comprehensive toolchain that uses hybrid taint analysis and novel assembly rewriting techniques to automatically identify and compartmentalize sensitive data based on user input. This enhances security while maintaining minimal performance impact.
- **Low-overhead Software Fault Isolation:** We have implemented a process-level, low-overhead software fault isolation mechanism that rewrites assembly code to enforce data-access and control-flow policies, ensuring the process does not access any untrusted components. This mechanism complements our fine-grained compartmentalization technique to enhance the protection of an application.

1.2.1 VDB: Verification of Code Diversification Techniques

This contribution presents a formal definition of binary equivalence resilient to diversification. The definition is based on stuttering bisimulation [11], showing that the binaries share a large class of temporal properties. Bisimulation is a relation between the transition systems, where one can simulate the other transition system's behavior (transitions) and vice versa. Stuttering bisimulation addresses internal steps—actions that do not produce observable behavior. Essentially, it permits two transition systems to be considered equivalent, irrespective of the number of internal steps required to reach the subsequent state.

Furthermore, we introduce a methodology to verify the functional equivalence between the original and diversified binaries. This approach involves disassembly, symbolic execution,

Table 1.1: State-of-the-art Code Diversification Techniques

Diversification Technique	**Reference**
Instruction Reordering	[73, 94, 104, 113, 168]
Basic Block Reordering	[43, 73, 117]
Stack Layout Randomization	[17, 73, 78, 130]
`nop` Insertion	[43, 95, 97, 105, 108, 124, 217]
Function Reordering	[18, 78, 117]
Static Binary Rewriting	[93, 94, 117, 219]

establishing stack-pointer-related invariants, and mapping memory regions in both the original and diversified versions. The final step of this methodology determines whether the previously defined formal criteria are met or identifies a counterexample.

We encapsulate our methodology in a tool and evaluate it using three off-the-shelf diversification tools, which collectively cover all the techniques listed in Table 1.1: 1) `nop` insertion [95], 2) Compiler-assisted Code Randomization (CCR) [117], and 3) stack shuffling [73].

The `nop` insertion method places `nop` instructions in front of targeted instructions [95]. CCR [117] employs a compiler-rewriter process that transforms inserted metadata into security primitives. Initially, the modified compiler inserts metadata into the binary, which is then rewritten using instruction and basic block reordering techniques. The stack shuffling technique [73] diversifies the stack layout for each binary.

There exists a body of work that discusses various metric systems to quantify the effectiveness of code diversification techniques [91, 96, 206]. These studies often utilize a set of software metrics known as Halstead metrics, introduced in 1977 to measure the complexity of software programs by analyzing the operators and operands in the source code [71, 89]. Halstead metrics include measures such as program length, vocabulary size, and several others.

However, the quantification of the effectiveness of diversification techniques is orthogonal to the problem addressed in this contribution. While these metric systems are crucial for developers of diversification techniques to provide quantifiable evidence of their effectiveness, our focus is on *verification*. Specifically, we aim to ensure that the original binary and the diversified binary are functionally equivalent using formal verification concepts. Consequently, the diversification techniques selected in Table 1.1 were chosen based on their ability to diversify the binary, rather than on how their effectiveness is quantified.

Lastly, an algorithm named Verification of Diversified Binary (VDB) is introduced to verify whether an original binary and its diversified counterpart meet the defined equivalence criteria. The aim of this contribution is to enable the use of untrusted diversification techniques in critical operational settings. We implemented the proposed methodology on three advanced diversification techniques applied to the GNU `Coreutils` package. The results demonstrate that our method can successfully prove the functional equivalence of 85,315

functions in the examined binaries.

The closest related work to this that of Gao et al. [76] proposes a technique based on backtracking to find semantical differences in binaries with different register allocation and basic block reordering. However, they do not deal with instructions relocation and stack-layout randomization. Moreover, they do not provide a formal definition of their soundness criterion and therefore do not show what class of properties is preserved between the vanilla and the diversified binaries.

1.2.2 CFR: Control-flow Restriction

This contribution revisits the binary-only CFI problem and proposes a CFI policy called *Control-Flow Restriction* (**CFR**) to guard against control-flow hijacking attacks. **CFR** secures the binary's control flow from an *attacker's perspective*, preventing malicious ICTs. Therefore, CFR only needs to search the possible payload gadgets from the binary, making it less reliant on a statically generated CFG. For example, in an attack that bypasses the most precise CFG [28, 68], not relying on the CFG will allow us to generate the necessary deny list to prevent the attack. Specifically, **CFR** first searches the binary for a list of attractive control-flow hijacking targets (e.g., gadgets) like an adversary and assembles these targets into a *deny* list. **CFR** patches the ICT instructions with a runtime check to *restrict* the attacker's arsenal based on the deny list. If the **CFR**-patched binary's execution fails the **CFR** policies, it terminates the execution. **CFR** does not require source code or special hardware support. A deny list can contain *any* potentially malicious address. We chose *ROP gadgets* in our work as they are a primary weapon for code-reuse attacks. Other control-flow hijacking targets could also be added to this deny list.

To properly evaluate the security effectiveness of **CFR**, we perform an adversarial analysis of CFR on ROP Emporium[1]. ROP Emporium is a collection of vulnerable binaries designed to teach ROP through a series of different challenges. Initially, we perform an ROP attack on each of these challenges to "capture the flag" (CTF) by only using a vulnerable binary. Subsequently, we apply our tool to *rewrite* the vulnerable binary and then perform the same attack to demonstrate that the attack is no longer possible.

We apply CFR on commercial off-the-shelf (COTS) applications (e.g., `xterm, gimp,` and `git`) and SPEC2006/2017 benchmarks. To summarize, **CFR** incurs an average runtime overhead of 5.5% – 14.3% for COTS applications, 40.8% – 74.8% for SPEC2006 benchmark, and 53.1% – 97% for SPEC2017 benchmark. Few applications incur a high runtime overhead even with a simple `NOP` instrumentation, such as `453/511.povray` applications in SPEC2006/2017 benchmarks. The main reason for this is because `e9patch` does not rewrite the binary by `inlining` the instrumentation code. Instead, `e9patch` inserts a jump to trampoline functions that contain an instrumentation code. After the instrumentation code completes, it

[1] https://ropemporium.com/

returns to the original location; this is the trade-off of not requiring CFG to perform binary rewriting. Furthermore, by lowering the policy level of **CFR** (e.g., $\mathbf{CFR}_0$) and selectively whitelisting necessary addresses, **CFR** can achieve a lot better performance.

1.2.3 Shroud: Automatic Tagged Memory Compartmentalization

This contribution propose a mechanism to leverage the ARM MTE to compartmentalize sensitive data. Shroud leverages ARM's Memory Tagging Extension (MTE) to enhance data security through compartmentalization. It begins with taint analysis—default static, optionally hybrid—to automatically identify potentially unsafe stack data. Static analyses such as LLVM StackSafetyAnalysis[2] identify unsafe stack data at compile time, while `libdft` [110], a dynamic analysis framework, tracks data flow at byte-granularity. We enhance static analysis by integrating it with SVF-based static taint analysis [207], which refines the identification of unsafe data by reducing false positives. Optionally, dynamic taint analysis further refines this by focusing on runtime "hot spots" to decrease overapproximation.

Following taint analysis, Shroud compartmentalizes identified unsafe data by relocating it to separate memory regions, thus disrupting traditional stack-based buffer vulnerabilities and enhancing the security effectiveness without significantly impacting performance. This relocation procedure prepares data for MTE heap tagging, allowing for finer-grained access control. The process culminates with one-time-pad (OTP) encryption using ARM's TRNG feature [182] to secure the MTE-tagged pointers against tag sniffing attacks and manual tag matching [248]. The system is complemented by libShroud, a software abstraction that provides various APIs to facilitate the protection of sensitive data using MTE.

We developed a Shroud prototype on LLVM [125] and conducted several evaluations to assess its performance and security effectiveness. Initial tests without taint analysis on the `httpd` application measured the overhead of relocating potentially unsafe stack objects, showing an average overhead of 0.55%. Further performance evaluations on Memcached+SASL and lighttpd reported average overheads of 3.5% and 1.7%, respectively, while UnixBench and the NPB benchmark recorded overheads of 11.45% and 2.11%.

To evaluate the security effectiveness of Shroud against prevalent common vulnerabilities and exposures (CVEs) like CVE 2006-5815 and CVE 2014-0160, we performed the adversarial analysis of Shroud using `min-dop` and `min-heartbleed`. These two programs are open-source demonstration examples that replicate similar CVEs' attack behavior for educational purposes, which fits our needs.

[2] https://clang.llvm.org/docs/SafeStack.html

1.2.4 IBCS: Input-Based Compartmentalization System

This contribution addresses the limitations of traditional compartmentalization methods that rely heavily on annotations and developer input to identify security targets. Recognizing that most external exploits exploit user inputs, we propose a novel approach that utilizes these inputs directly to define compartmentalization targets. By shifting away from static analysis alone, which often leads to complexity and overapproximation, our method aims to balance the granularity of security measures more effectively, minimizing unnecessary revisits of known execution paths from initialization to `main` and thereby enhancing enforcement efficiency with manageable overhead.

We introduce the Input-Based Compartmentalization System (IBCS), designed to dynamically adapt to varying user inputs while ensuring memory safety. IBCS generates a dynamic graph from arbitrary user inputs, which, when integrated with static analysis, helps explore previously uncharted execution paths. This integration is crucial for the precise identification of vulnerable data targets, which are then automatically compartmentalized through our toolchain. The system's ability to adjust its analysis based on ongoing user input reduces underapproximation in enforcing compartmentalization, filling gaps that static methods alone may miss. Our system innovates with a variety of novel static analyses techniques related to assembly code rewriting that broadly apply across different software architectures. By implementing these techniques, IBCS is capable of dynamically adapting its security mechanisms in response to real-time input changes, thereby maintaining stringent security measures across a range of applications. We validate the practicality and adaptability of our approach by applying IBCS to various software scales—from a `tiny` web server to extensive applications like GNU `Coreutils` and Nginx.

The effectiveness of IBCS is further demonstrated through rigorous adversarial analysis. These evaluations not only showcase the system's ability to identify and secure vulnerable targets but also highlight its broader applicability and robustness in defending against existing and potential security threats. Through these tests, IBCS has proven to reduce the gap in enforcement underapproximation significantly, confirming its utility in enhancing software security against a variety of attack vectors.

1.2.5 XFI: x86-based Fault Isolation

This contribution revisits the concept of Software Fault Isolation (SFI), a software sandboxing technique that isolates one or more processes from each other to ensure the system as a whole is not compromised if one process is exploited. SFI is a form of compartmentalization known for its significant overhead, posing a major challenge in balancing security and performance. The overhead of SFI mechanisms largely stems from factors such as the need to rewrite binaries to insert instrumentation checks, ensuring that all data accesses remain within designated regions, and verifying control-flow transfers to prevent jumps to untrusted

locations.

Several works have explored the design of SFI systems, including Google Native Client (NaCl) [234]. However, NaCl has several limitations, such as the requirement to use x86 segmented memory to simplify security checks, high average overhead, code bloat due to padding, and the need for a heavily altered compiler toolchain, which complicates adoption and development. Recently, a work named Lightweight Fault Isolation (LFI) [233] was presented. LFI is a process-level SFI system that supports the isolation of loads, stores, and indirect transfers with low overhead. It improves upon NaCl's limitations by leveraging assembly rewriting and architecture-specific optimizations on ARM64 to implement a scalable sandboxing system.

We present a system called x86-based Fault Isolation (**XFI**), which leverages techniques used in LFI to implement SFI on an x86 process. **XFI** ensures basic memory loads, stores, and indirect transfers with low performance overhead without requiring complex toolchain modifications or code bloat. This is achieved by rewriting assembly code, avoiding modifications to the compiler's backend, and sandboxing a process using simple assembly instructions, eliminating the need for expensive instrumentation checks to ensure accesses are within trusted components. We implemented the **XFI** toolchain using the musl C library, a lightweight implementation of the standard C library [1], and a rewriter that analyzes the object file of the original assembly code to insert the necessary sandboxing mechanisms. We evaluate the effectiveness of **XFI** by rewriting many GNU `Coreutils` applications of varying sizes.

1.3 Dissertation Organization

This dissertation is organized as follows. After this introductory chapter, Chapter 2 provides the necessary background information to understand the foundational concepts discussed. Chapter 3 reviews related work in the fields addressed by this dissertation. Chapters 4, 5, and 6 present **VDB** and offer an in-depth examination of the methods used to verify functional equivalence between vanilla and diversified binaries. A CFI policy based on a deny list, termed **CFR**, is detailed in Chapters 7 and 8. The mechanism for compartmentalizing sensitive data using tagged memory architecture is explored in Chapters 9, 10, and 11. The development and implementation of an architecture-agnostic comprehensive toolchain, **IBCS**, which automatically identifies and enforces protection on vulnerable targets based on user input, are covered in Chapters 12, 13, and 14. A low-overhead SFI system, **XFI**, that enforces security policies for basic memory loads, stores, and indirect transfers on an x86-based system, is presented in Chapters 15, 16 and 17. Finally, the dissertation wraps up in Chapter 18.

Chapter 2

Background

This chapter of the dissertation provides necessary background information to understand the work presented. Firstly, we provide an explanation of different types of attacks based on data in Section 2.1. Next, in Section 2.2, we begin diving delve into various types of security techniques starting with software security primitives. Subsequently, we explore different hardware security primitives that exist today in Section 2.3 that can be used as a protection. Following this section, we provide information on the binary disassembly process in Section 2.4. In Section 2.5, we offer background regarding formal verification techniques. Then, we define a variety of compartmentalization spaces in Section 2.6. Finally, in Section 2.7, we describe the concept of taint analysis and its variants used in the literature.

2.1 Data-Based Attacks

2.1.1 Control-Data Attacks

A control-data attack modifies the data loaded into the processor's program counter at some point [29] to perform malicious actions. An example of such an attack is control-flow hijacking, where attackers aim to modify return addresses or function pointers to perform arbitrary execution [24]. The premise behind a control-data attack is to exploit known memory vulnerabilities (e.g., buffer overflow) to divert the control flow from its original execution.

Control-data attacks have existed for decades [235], and the concept has become more advanced, such as return-to-libc (ret2libc), which reuses functions in the libc library [197]. Subsequently, the attack by reusing the code became generalized to what is widely known as Return-Oriented Programming [178], which reuses pieces of code from the target program. Since then, advanced variants of ROP attacks have been proposed, mainly Jump-Oriented Programming [20] and Call-Oriented Programming [27]. JOP is a type of code-reuse attack that relies on a dispatcher gadget that ends with an indirect jump (e.g., `jmp %rbx`), whereas COP is a type that relies on special gadgets that all end in a `call` instruction.

2.1.2 Non-Control-Data Attacks

A non-control-data attack manipulates the program's non-control data (e.g., data without the transfer target address) without diverting the control flow to jeopardize spatial memory safety. Therefore, a non-control-data attack can perform malicious actions bypassing even the most precise control-flow hijacking prevention techniques as it adheres to the original path [28]. For example, Data-Oriented Programming (DOP) [98] is an advanced non-control-data attack that can express a Turing-complete exploit to read/write memory arbitrarily. Similarly, the Heartbleed exploit [64] leverages unsafe programming languages to leak sensitive data by arbitrarily overreading the memory buffer. Existing countermeasure techniques often use memory pages to compartmentalize sensitive data from the rest of the program [140, 211]. However, current page-based memory protection approaches are not suitable for memory protection of a few bytes.

2.2 Software Security Primitives

Software security is a cat-and-mouse game between researchers and adversaries. Adversaries continuously exploit new vulnerabilities, and researchers must propose new security primitives to counter these attacks. Consequently, there are two main types of security primitives deployed today: software-based and hardware-based techniques.

2.2.1 Security Policy Enforcement

Security policy enforcement is a technique that monitors a program to ensure it adheres to well-defined security properties [131]. In other words, enforcement-based software security primitives are defense mechanisms that impose a set of rules on the target program. If the target program deviates to an unknown execution path or attempts to perform unauthorized actions, the defense mechanism flags it as a potential attack. Therefore, enforcement-based defenses are proactive security measures and do not protect against zero-day attacks (i.e., attacks that are previously unknown).

Control-Flow Integrity (CFI) [3, 24] is a well-studied example of a policy enforcement technique. CFI is a security property that requires a program's execution path to follow a statically generated Control-Flow Graph (CFG). The CFG consists of control-flow transfer edges (either forward-edge or backward-edge), and this information can be used along with a runtime monitor to ensure the program is jumping only to intended locations.

2.2.2 Code Diversification

Code diversification, as the name implies, diversifies intrinsic code properties to create unique, diversified versions of the program [97]. All of these programs are functionally equivalent but semantically different. In other words, if an adversary manages to exploit one instance of a program, the same type of exploit will fail on another instance, significantly increasing the adversary's effort as they will need a unique attack model for each binary instance.

The motivation chapter (Chapter 1.1.1) provides various examples and details for many of these techniques. There are other types of code diversification techniques that revolve around using a binary. For instance, a binary can have diversified features while maintaining its original functionality through the binary rewriting approach. Furthermore, dynamic binary instrumentation (DBI) is a technique that *augments* arbitrary code to extend a binary's original functionality without modifying it.

2.2.3 Code Obfuscation

Code obfuscation techniques aim to scramble or obfuscate the code to create a binary instance that is difficult to reverse engineer [226]. In contrast to code diversification, an obfuscated binary is functionally and semantically the same as the input binary, but it creates a *syntactically* different binary from the original version.

For example, a function `foo()` in the original binary might transform into a function `f()`, or the text "`Hello World`" might transform into a byte pattern like "`\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x64`". These complex transformations make it challenging for adversaries to reverse engineer and extract useful information to construct an attack model. Furthermore, many obfuscation techniques arbitrarily change the control flow or the program's address space to confuse adversaries attempting to decipher the code.

2.2.4 Software Fault Isolation (SFI)

Sandboxing, derived from the principle of least privilege, is a type of defense mechanism aimed at deliberately isolating an application into a "box" to restrict its access to system resources. By doing so, even if the sandboxed application is exploited by adversaries, the adverse effects are confined within its box, thereby minimizing the damage inflicted on the overall system [82].

Software Fault Isolation (SFI) is an interchangeable term used in software to formally refer to sandboxing. SFI is a software-instrumentation technique used on an application to establish legalized access domains within a process [210]. Specifically, all SFI techniques first designate memory regions to be considered either as trusted or untrusted components. Then, they

rewrite the application, either by modifying the compiler or rewriting the assembly code, to insert the necessary instrumentation. This ensures that memory accesses (data-access policy) and control-flow transfers (control-flow policy) remain within the trusted components.

2.3 Hardware Security Primitives

In contrast to software security primitives, which are developed solely around leveraging software-based concepts (e.g., code, processes), hardware security primitives leverage a variety of hardware features. One widely known example is the non-executable (NX) bit [44], where a specified memory page becomes non-executable, preventing adversaries from executing malicious payloads, because the CPU will fault any code that gets executed from such a page.

2.3.1 Tagged Memory Architecture

Tagged memory architecture is a computer architecture that designates a *metadata tag* to a memory block. The user can then use this metadata tag to enforce various security mechanisms [46, 205, 222, 223, 225, 237]. For example, a memory-bound violation can be enforced by only allowing specific memory block access only if a metadata tag matches. Due to the continuous advancement of non-control-data attacks, researchers have proposed many defense mechanisms [29, 37, 39, 45, 46, 60, 121, 156, 157, 205, 222, 223, 225, 237, 242].

2.3.2 ARM Memory Tagging Extension (MTE)

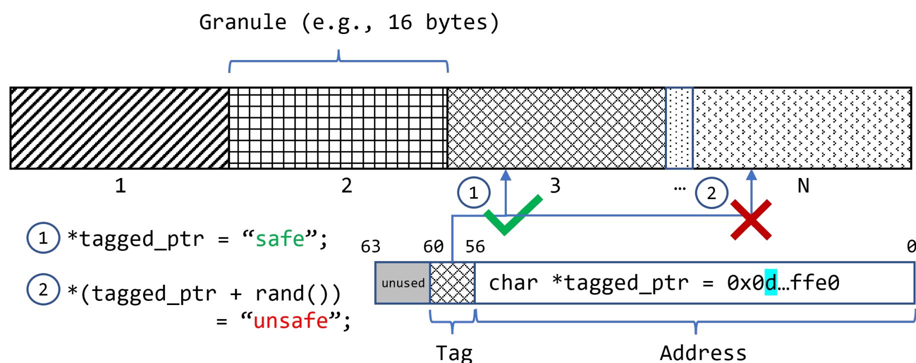


Figure 2.1: Overview of MTE enforcing memory safety

Recently, ARM unveiled new architectural support using tagged memory architecture called Memory Tagging Extension (MTE) with the release of ARM v8.5-A [75]. Figure 2.1 depicts

an overview of ARM MTE technology. MTE divides all memory blocks into small fix-sized blocks called *granules*. Each of these granules is associated with a 4-bit *allocation* tag (denoted with different tag values) in the physical address space.

These allocation tags reside in a dedicated RAM region reserved during CPU boot. A pointer subjected to be MTE-enforced will be associated with a 4-bit *logical* tag (highlighted `0x0d` in `char *tagged_ptr`) in the unused upper bits in the virtual address space. This is possible thanks to the Top-Byte-Ignore (TBI) feature that *ignores* the upper eight bits of a pointer. Whenever the tagged pointer attempts to access the memory, it first checks the memory granule's allocation tag against the logical tag of a tagged pointer. If they match, such access is valid. Otherwise, the memory access is invalid, and the execution halts.

2.3.3 Intel MPK

Intel's Memory Protection Keys (MPK) represent an x86 ISA extension that enables developers to control memory access permissions in a fine-grained and efficient manner. Traditionally, researchers limited access to sensitive memory regions by updating page table entries (PTEs) with different permissions for potentially malicious processes. However, this approach, which requires flushing translation lookaside buffer (TLB) entries, incurs non-negligible performance overheads. Additionally, modifying PTEs is considered coarse-grained due to fixed page sizes and lack of runtime control. Consequently, Intel MPK has become widely used for isolating untrusted components from trusted ones.

Intel MPK enhances applicability through a new user-mode register, the Protection Key Rights Register (`PKRU`), which allows flexible modification of permissions associated with individual memory page keys at runtime. The `PKRU` is a 32-bit register that specifies the access rights (read, write, or both) for the memory pages linked to each key. The instructions `WRPKRU` and `RDPKRU` enable writing to and reading from the `PKRU` register directly from user space, improving performance by obviating the need for system calls. Additionally, for PKU-enabled kernels, new system calls are introduced: `pkey_alloc` for dynamically allocating a protection key, `pkey_free` for releasing a previously allocated key, and `pkey_mprotect` to tag a page with a specified key.

2.4 Disassembly

Disassembly is an integral process of binary analysis, a code review that assesses intrinsic code properties of an unknown program. Binary disassembly takes a non-human machine language as an input, and the output is a human-readable mnemonics corresponding to the input. Many types of disassembly techniques exist in the literature.

2.4.1 Linear Sweep

The linear sweep algorithm sequentially disassembles a binary by linearly iterating through bytes of the binary [192]. In other words, it starts from the beginning of the code section, then continuously proceeds by decoding each byte it finds into an instruction until the end. Thus, the linear sweep algorithm is relatively straightforward.

However, a linear sweep algorithm can suffer from a disassembly desynchronization problem. This is a problem where a disassembler places data bytes into a location where instruction bytes should be due to misinterpretation. Furthermore, the linear sweep algorithm does not consider the control flow behavior of the program, which becomes problematic when handling control transfer instructions (e.g., `jmp`).

Many variants of the linear sweep algorithm exist in the literature, such as superset disassembly [13], which improves upon the original linear sweep algorithm. Superset disassembly is a linear sweep variant that does not require the knowledge of entry points, which is a requirement for the recursive traversal algorithm. Instead, it disassembles and reassembles each offset of each section in a program. By doing so, superset disassembly can handle the majority of indirect control-flow transfers correctly and provide the complete recovery of all legal instructions.

2.4.2 Recursive Traversal

In contrast to the linear sweep algorithm, the recursive traversal algorithm takes control flow behavior into account [192]. First, this algorithm performs a linear sweep algorithm from a known entry point (e.g., `main`). Then when the algorithm reaches control transfer instructions (e.g., `jmp` or `call`), it recursively follows the instruction and finds control flow successors. This allows the algorithm to circumvent unnecessary data bytes and only obtain the appropriate instructions.

However, the recursive traversal algorithm has its limitation. It is challenging to construct a complete control flow of indirect control transfer instructions (e.g., `jmp rax`) because the offset address is unknown until the runtime. Nevertheless, the recursive traversal algorithm is still a standard for the binary disassembly technique.

2.4.3 Assembly Instructions

Throughout this paper, we will use assembly instructions in AT&T syntax, which have the form of:

$$\texttt{opcode}\ \texttt{op}_1\texttt{,}\ \texttt{op}_2\texttt{,}\ \texttt{...}$$

In AT&T syntax, instructions are indicated by an `opcode` (e.g., `mov`) followed by operands ($\texttt{op}_x$), with the source operand preceding the destination operand. Additionally, operand sizes are explicitly denoted by a suffix appended to the opcode (e.g., `movb` for byte-size operations, represented as `b`). Operands may be constants (`$1`), registers (`%rax`), or memory addresses. Memory addressing in AT&T syntax uniquely involves the use of parentheses to denote direct memory access, incorporating optional elements such as displacement, base register, and index register, exemplified by `4(%rax)`, which accesses memory at an address derived from the base `%rax` plus a displacement of 4. A specific `rdgsbase` instruction is also employed to read the base address of a segment designated by the `GS` register into a general-purpose register.

2.5 Formal Verification

Formal verification is a way to prove or disprove the correctness of a system mathematically. First, a formal specification is necessary to outline how a system should or should not perform. Formal specification translates a non-mathematical description into a formal language that provides a concise description of high-level behavior. Afterward, engineers can use different formal verification methods to prove whether specific properties exist.

2.5.1 Model Checking

Model checking is an automatic formal verification technique using a system modeled by state-transition systems [11]. The model checking methodology takes an input, a user-created model of a system, and an algorithm that describes the properties that the system should satisfy. Afterward, model checking performs a systematic check of this property for all states. During the checking process, if the property is not satisfied, a counterexample is produced.

Figure 2.2 presents an overview of the model checking technique. First, model checking must have a requirement to be checked. Next, this requirement needs formalization because it cannot be vague; it must be precise and unambiguous. This formalization will then turn into a specification for model checking.

On the other hand, a system for model checking needs to be modeled in a form that the model checker can use with specifications to check (e.g., transition systems). Model checker takes as an input both formal specification and the system model, then check the model and provide three possible outcomes:

1. Satisfied: a given system model *satisfies* the formal specification.

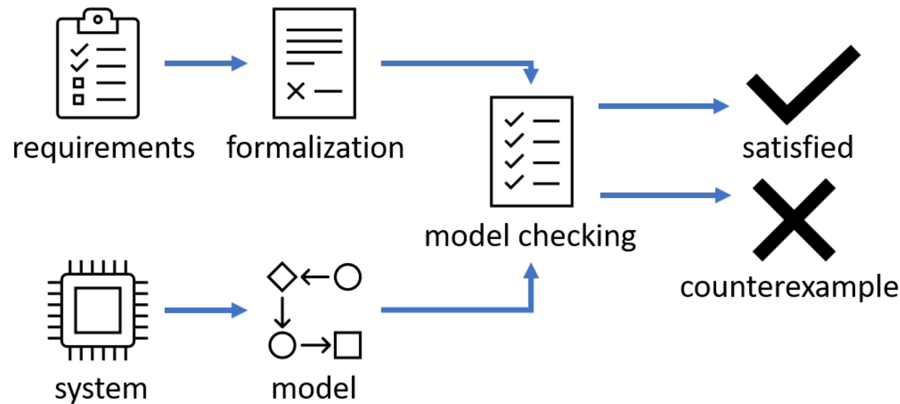


Figure 2.2: Model checking overview [11]

2. Counterexample: a given system model *violates* the provided specification but provides a counterexample.

3. Insufficient memory: segmentation fault, i.e., the system model is too big.

2.5.2 Symbolic Execution

Symbolic execution is a form of execution that uses symbols to represent arbitrary values [114]. The advantage of symbolic execution is that because inputs are symbolic value, this can represent all different inputs and explore multiple paths. Thus, symbolic execution is helpful to find potential bugs in software because testing a symbolic value generalizes the testing process as input represents precisely the set of executions whose concrete values satisfy the provided path condition.

However, although symbolic execution can explore different paths, it also has its limitation. The main limitation is a *path explosion problem* because as an input program becomes larger, the number of possible paths exponentially increases, which at a certain point becomes infeasible. Therefore, to prevent such a scenario from occurring, there needs to be a proper termination point.

2.6 Compartmentalization Spaces

Compartmentalization effectively decouples an application into different components, each isolated within its compartment [109, 186]. This isolation limits each compartment's access to resources within its boundaries, significantly reducing the attack surface by inherently providing isolation. An important metric for partitioning an application is the concept of

compartmentalization spaces [86].

In [86], the authors describe three distinct dimensions of compartmentalization: *data-centered*, isolating data accessed by the host code to prevent unauthorized interactions; *code-centered*, ensuring that different code components are segregated to maintain integrity and security; and *object-oriented*, which combines the data- and code-centered approaches for a comprehensive compartmentalization strategy that addresses both code and data concerns. Based on these definitions, we classify compartmentalization spaces as fine-grained or coarse-grained. Compartmentalization is considered *fine-grained* if each data element within an application is isolated from the others. Conversely, coarse-grained compartmentalization occurs when code and data, or only code components, are separated.

2.7 Taint Analysis

Taint analysis is a data-flow technique used to detect security vulnerabilities in software by tracking how data from untrusted sources influences application behavior. It identifies *taint sources* and monitors the propagation of tainted data (*taint propagation*) through the application to pinpoint vulnerable variables or memory locations (*taint sinks*). If tainted data is used in sensitive contexts without proper checks, it can lead to security vulnerabilities [10, 66, 232]. Taint analysis is categorized into three types: **Static taint analysis** analyzes code at compile-time to identify potential vulnerabilities by examining possible execution paths [232]. **Dynamic taint analysis** monitors a program's execution to trace data flow, providing a precise depiction of security risks, though it has reduced coverage [66, 110]. **Hybrid taint analysis** [244] combines static and dynamic approaches to balance their limitations, such as the overapproximations of static analysis and the coverage constraints of dynamic analysis.

Chapter 3

Related Work

Security and formal verification are diverse and well-active fields of research. In this chapter, we present several works that relate to our contributions.

Section 3.1 offers various types of analysis used for software similarity techniques. Following that, Section 3.2 discusses other works that perform equivalence checking using symbolic execution. Section 3.3 presents various works that formally verify low-level code (e.g., assembly language). Section 3.4 describes a powerful software attack known as code-reuse attacks, and Section 3.5 provides different protection schemes used to counter these attacks. Section 3.6 describes attacks revolving around non-control-data, and Section 3.7 provides various protection schemes for such attacks. Section 3.8 discusses existing compartmentalization techniques that are used to identify and enforce data. Lastly, Section 3.9 talks about existing work related to software fault isolation.

3.1 Static and Dynamic Analysis

Software similarity can be analyzed using either static or dynamic approaches. The static approach examines the code without execution [180], whereas the dynamic approach executes the functions of the target binaries with identical inputs and measures similarity dynamically [65]. The static analysis includes text-based, token-based, tree-based, graph-based, and metric-based techniques.

The text-based methods compare the text with minimal (or no) transformation on the source code. The token-based techniques [4, 63], on the contrary, transform source code into a sequence of lexical tokens that can produce an abstract representation of software. Tree-based techniques [106, 246] parse software into the tree data structure, such as Abstract Syntax Trees (ASTs), containing complete information about the software. Then, similar subtrees are searched with their respective comparison target with different tree matching techniques to obtain matching pairs.

The graph-based methods [30, 115] are identical to the tree-based capturing of the code structure but in CFG, which contains the semantics of the code with control or data flow information. Lastly, metric-based techniques involve breaking down source code into smaller comparable fragments. These fragments can then be quantitatively measured using its met-

rics [6, 40, 56] to compare the similarity between the software. The tree-based, graph-based, and statistical-based techniques are widely known as syntax-based approaches. These approaches convert the software into syntax-based pieces that can be semantically compared to measure the similarity. In contrast to the static analysis, which analyzes without executing a program, dynamic analysis runs functions of the target binaries with the same input and dynamically determines the similarity [65].

Our first contribution closely relates to the graph-based method [30, 115, 134, 137, 230]. Graph-based methods parse code into CFGs whose subtrees are searched using different graph matching techniques to obtain matching pairs. Lim and Nagarakatte [134] checked for equivalence using a graph-based methodology and symbolic execution. However, this work focused explicitly on cryptographic algorithms and did not deal with diversified binaries.

3.2 Equivalence Checking Using Symbolic Execution

Various research projects use symbolic execution to prove equivalence, such as `BinHunt` [76], `KLEE` [175], and many others [136, 172, 202]. `KLEE` checks, among others, code equivalence. However, this work is source-code based. The most similar work to ours is `BinHunt`. The authors propose a technique based on backtracking to find semantical differences in binaries with different register allocation and basic block reordering. However, they do not deal with instruction reordering and stack-frame shuffling. Moreover, they do not provide a formal definition of their soundness criterion and therefore do not show what class of properties is preserved between the vanilla and the diversified binaries.

The other papers are similar because their focus lies on different subjects, such as finding bugs or equivalence checking between cross-architecture. Lastly, `CoP` [144] is a code-obfuscation resilient work that searches for the semantical difference between the original and the suspicious binary. However, `CoP` approximates the similarity between the binaries and reports a score indicating the likelihood of reusing the original binary's components. This is different from our work as we check and prove the functional equivalence between the binaries.

3.3 Low-level Formal Verification

Formal verification of low-level code (e.g., assembly language) has been an active research field for decades [42, 231]. CompCert [126] is a formally verified compiler that guarantees that the safety properties proved on the source code hold for executable. However, to the best of our knowledge, CompCert cannot be used for binary diversification. Therefore, verifying low-level code is still crucial to build genuinely trustworthy software because a machine code that runs on a machine should also be verified.

The majority of low-level code is untyped instruction with registers or hexadecimal numbers,

unlike high-level code. Hence, it is challenging to analyze and verify the low-level code with high confidence since a lot of critical information is vaguely represented. Many proposals provide high-level semantics by enforcing a traditional basic-block structure [218] or applying software model checking (SMC) to provide higher abstraction [31].

A technique called decompilation into logic (DiL) has been proposed to verify low-level code [155]. This technique first takes a low-level code and a model of an instruction set architecture (ISA). Afterward, the DiL's decompiler extracts functions (defined in logic) to capture the low-level code's functional behavior. Decompilation results can prove the specific properties of these extracted functions. This is because decompilation results from the low-level code. Hence, any property proved in the decompilation result will directly apply to the low-level code.

3.4 Code-Reuse Attacks

3.4.1 Return-oriented Programming (ROP)

The first type of code-reuse attack was the `ret2libc` attack [197], which reused functions in the `libc` library. The underlying idea of this attack was to take advantage of programs that used `libc`. This attack's reliance on the `libc` library contents made it suppressible. However, soon after, Return-Oriented Programming (ROP) was unveiled [178], which hijacks control flow by reusing the pieces of the target program's code, known as *gadgets*.

Gadgets are sequences of instructions that end in an ICT instruction, and they perform a specific task (e.g., load, arithmetic, etc.). Gadgets can be short and long. However, the longer a gadget is, it becomes more difficult to use. Each gadget is harmless on its own. The composition of gadgets (i.e., chain of gadgets) allows an attacker to control the execution flow and launch code-reuse attacks.

3.4.2 Advanced Variant of Code-reuse Attacks

ROP attacks have advanced into various variations. Jump-Oriented Programming (JOP) [20, 33] uses gadget sequences that end with jump instructions. Call-Oriented Programming (COP) [79] utilizes indirect call instructions to chain gadgets. Just-in-Time (JIT) ROP attacks [204] leverage information disclosure to find gadgets during the application's runtime.

For example, if adversaries find a single leaked Application Programming Interface (API) function pointer, they can use this to gain information about the entire code page. Another attack, named Control-Flow Bending (CFB) [28], launches a non-control data attack. Counterfeit Object-Oriented Programming (COOP) [191] leverages C++ virtual functions as gadgets, using vTables to execute a malicious program injected by adversaries. Position-

Table 3.1: Code-reuse Attacks Protection Schemes

Study	Hardware(s) Dependencies	Input Type	Trusted Computing Base (TCB)	CFG Not Required
μCFI [99]	Required	Source Code	LLVM [125], μCFI Monitor, Intel PT [177]	✗
CFI-LB [112]	Required	Source Code	diStorm†, Intel PT	✗
GRIFFIN [77]	Required	Non-Stripped Binary	diStorm†, Kernel Module, Intel PT	✗
Hurdle [59]	Required	Stripped Binary	Dyninst [16], Z3 Solver [57], Intel CET [102], EHBR	✗
PathArmor [212]	Required	Stripped Binary	Dyninst, Kernel Module, LBR Registers	✗
Levee [121]	No	Source Code	LLVM	✗
CCFIR [238]	No	Stripped Binary	BitCover, BitRewrite, BitVerify	✗
BinCFI [240]	No	Stripped Binary	Objdump, Objcopy	✗
CFR (Our work)	No	Stripped Binary	Objdump, Binary Ninja*, Ropper‡, `e9patch` [62]	✓

$*$ = we do not trust *all* of Binary Ninja, only analyses concerning metadata information.
† = available at https://github.com/gdabah/distorm
‡ = available at https://github.com/sashs/Ropper

Independent Code Reuse (PiROP) [81] attacks use relative (e.g., nearby) gadgets to launch the attack.

3.5 Protection Schemes for Code-Reuse Attacks

Code-reuse attacks protection schemes have been an active research field. Table 3.1 provides an overview of different code-reuse attack protection schemes. The table presents whether external hardware is required to deploy a technique and the type of input needed. In addition, we list Trusted Computing Base (TCB) for each study. TCB, in this context, represents external tools that are trusted to function correctly. Lastly, we show whether a study *does not* require CFG, i.e., if the study's application *depends* on the CFG. This is important because generating a CFG reliably from a stripped binary (an executable that lacks debug information) is difficult. During the process of binary code transformation, the majority of source-level information disappears [34]. Therefore, an approach based on information from a CFG requires source code or hardware to be effective.

Control-Flow Integrity (CFI) Since the proposal of the CFI [3], there have been many works expanding its original approach. Newly proposed ideas differ in the type of input and the granularity of CFI policies. In a sense, granularity is to what extent CFI policy enforces (e.g., supported ICT instructions).

Many promising hardware technologies like Intel Processor Trace (PT) [177] and Control-flow Enforcement Technique (CET) [102] are available against code-reuse attacks. Many related works [59, 77, 99, 212, 236] incorporate these technologies to delegate tasks to hardware to protect against code-reuse attacks. However, using hardware also comes with a price. For example, Intel CET is only available in the latest processor, which means legacy hardware is not supported. Furthermore, bypasses exist for these technologies [58, 199], making them not infallible. Lastly, to use hardware, a user may need *root* access, which is a strong assumption.

Using the binary's source code allows access to the information to enforce a strong CFI policy using precise analysis. For example, Levee [121] identifies *all* code pointers (e.g., pointers that determine the target of ICT instructions) in a binary and then rewrites a binary after storing all sensitive pointers to the isolated memory region. Such a technique would not be possible without using source code. μCFI [99] is another source-code level technique that enforces a unique code target (UCT) policy, allowing only one allowed target for each invocation of an ICT.

However, requiring source code suffers from its limitations. For instance, the source code of a binary is not always available. Many of the legacy and commercial off-the-shelf (COTS) programs are nearly impossible to obtain the source code. Second, many source-code level works need a modification to a compiler, which would require root access to the system. Lastly, there remain issues regarding re-compilation for large-scale binaries and practicality. For example, if developers modify a binary source code, does this jeopardize the correctness of the techniques? Therefore, given these limitations, it is imperative to explore ways to improve the binary-level techniques.

The binary-level technique solves many limitations described above. For instance, using a binary improves the practicality of enforcing CFI policy because there is no need to obtain the source code. Earlier binary-level CFI techniques CCFIR [238] and BinCFI [240] highlighted such improvements by enforcing a relaxed CFI (RCFI) policy on many COTS applications. However, due to its loose policy, these works suffered from the overapproximation of invalid targets, which caused them to be potentially unsafe [27, 54, 79, 191].

Thus, later CFI works focus on implementing a stronger CFI policy such as context-sensitive CFI (CSCFI) policy on a binary because CSCFI enforces forward and backward ICT instructions. PathArmor [212] is a binary-level technique to enforce CSCFI policy on a binary using a kernel module and Last Branch Record (LBR) registers. GRIFFIN [77] is a hardware-assisted CSCFI technique. It uses recorded execution trace from Intel PT to perform online control-flow checks. Lastly, Hurdle [59] is a binary-level technique that incorporates Intel CET and Extended Branch History Registers (EHBR) to enforce each ICT instructions to match the obtained control flow history.

Randomization There is another type of protection scheme based on randomization. The randomization-based technique focuses on making the valuable information in a binary challenging to find by adding an unpredictability layer on top of the memory addresses. Recent code randomization techniques focus on randomizing different parts of the program [47, 88, 93, 95, 174, 224].

3.6 Non-Control-Data Attacks

Non-control-data attacks [36, 235] have begun to resurface as an area to exploit for adversaries. Chen et al. demonstrated how non-control data attacks are realistic threats to several real-world applications [36]. Subsequently, non-control data attacks have advanced, like DOP [98], where an attacker can express turing-complete exploits. Although non-control data attacks have been around for a while [235], the limited understanding and lack of protective mechanisms toward non-control data attacks have made this venue appealing to attackers.

3.6.1 Data-Oriented Programming (DOP)

Data-Oriented Programming (DOP) is an advanced non-control-data attack technique that expresses turing-complete computation by chaining *DOP gadgets* [98]. DOP gadgets are short instruction sequences that follow the format of *load* → *semantics* → *store* to simulate logical micro-operations (e.g., arithmetic, assignment, conditionals, etc.). With a successful DOP attack, the adversary can read/write memory arbitrarily.

3.7 Protection Schemes for Non-Control-Data Attacks

Table 3.2: Comparison of SHROUD with State-of-the-Art Systems

	Compartment Identification	Protection Level	Architecture	Granularity
HAKCS [148]	Manual (Annotation)	Multi-layered	ARM (PAC/MTE)	Coarse-grained
CHERI [225]	Manual (User-defined)	Single-layered	ARM & RISC-V (CHERI)	Coarse-grained / Fine-grained
HDFI [205]	Automatic (Static analysis)[e]	Single-layered	RISC-V	Coarse-grained
WIT [5]	Automatic (Static analysis)	Single-layered	Any	Fine-grained
MEMTAGSANITIZER	Automatic (Static analysis)	Single-layered	ARM (MTE)	Fine-grained
COLOR MY WORLD [133]	Automatic (Static analysis)	Multi-layered	ARM (MTE)	Fine-grained
SHROUD	Automatic (Hybrid analysis)	Multi-layered	ARM (MTE)	Fine-grained

Protective mechanisms against non-control data attacks have been an active research area in software [29, 39, 45, 121, 156] and hardware [205, 225]. ViK [39] is an object-ID validation approach to ensure temporal memory safety. However, ViK cannot prevent arbitrary memory read/write. Software defenses typically suffer from high-performance overhead and require significant changes. Therefore, delegating tasks to hardware, such as tagged memory architectures [70, 223, 225], has been considered promising due to their efficiency and reducing software's trusted computing base (TCB) size [165].

Tagged Memory Architecture Using a tag to mitigate memory safety violations is not a novel technique. Prior to the ARM MTE technology, CHERI [225], a capability hardware-enhanced RISC instructions, uses a memory tag to create an *unforgeable* fat pointer (i.e., a tag is used to ensure the tagged pointer's integrity). Capability in this context refers to the limit of the memory range that the CHERI pointer can access. In addition, CHERI suffers from the compatibility issue as it requires a pointer to be extended (e.g., 64-bit to 128 bits) and change in the processor architecture due to its capability model. Hardware-Assisted Data-Flow Isolation (HDFI) [205] virtually extends each memory unit with an additional tag associated with the memory units' physical address to protect the sensitive data. However, HDFI is a coarse-grained security mechanism because, in its current state, it performs a check during the read operation, which implies there remains a potential danger during the write operation. Lastly, HAKC uses ARM MTE and Pointer Authentication [132] to com-

partmentalize the components in the kernel [148]. However, HAKC does not automatically compartmentalize the data. Color My World [133] is a work that uses the Memory Tagging Extension (MTE) to protect unsafe data. This approach is the most similar to Shroud, but it leverages static analysis rather than the hybrid analysis used in Shroud, which could lead to more false positives.

Pointer Integrity and Cryptography To improve pointer integrity, *PointGuard* [45] proposed a protection mechanism that encrypts a pointer value in memory and decrypts it immediately before pointer dereferencing occurs. *WatchdogLite* [158] is a work that tracks the pointers' integrity by storing the base and bound of target pointers to inaccessible locations to adversaries and performing boundary checks when pointers are accessed. Since *PointGuard* and *WatchdogLite*, many other works have employed similar techniques, such as works that leverage the ARM Pointer Authentication (PA) feature [132]. ARM PA is an ARM's architectural feature that *authenticates pointers* by using cryptographic message authentication codes called *pointer authentication codes* (PAC) to check the integrity of critical pointers [181]. If the PAC were to be used along with the MTE, the PAC size needs to be reduced by eight bits (halved) to accommodate MTE as both share ARM's top-byte ignore feature. Furthermore, PAC cannot be used along with the MTE, depending on the ARM architecture profile.

3.8 Compartmentalization

Compartment Identification While several tools provide automatic enforcement, the majority [86, 135, 148, 222] still require developers to manually identify compartments. For example, SOAAP [86] offers developers performance metrics to help evaluate different compartmentalization strategies. However, SOAAP's annotation requirement is not intuitive and requires expert knowledge from developers. Similarly, HAKCS [148] facilitates kernel compartmentalization but also necessitates annotations to define cliques and data-ownership mechanisms.

Many tools [41, 100, 111] aim to automatically identify compartmentalization targets, though they tend towards semi-automatic rather than fully automatic processes. KSplit [100] isolates kernel modules into separate components, but ambiguous interface definition language (IDL) from static analysis may require developer intervention to resolve. The Embedded Compartmentalizer (EC) [111] provides an automatic toolchain that compartmentalizes firmware using a formally verified microkernel, yet still requires developer input to guide static analysis with program directives and type qualifiers. Similarly, ACES [41] uses a graph traversal policy to group partitions based on specific criteria.

SCALPEL [179] is fully comparable to **IBCS** in automatically identifying compartment targets. It employs static analysis to collect necessary data and integrates this with dynamic tracing to track functions and record runtime behavior. This combination generates quan-

Table 3.3: Comparison of **IBCS** with State-of-the-Art Systems

	Compartment Identification	Compartment Enforcement	Architecture (Feature(s) used)	Granularity
SCALPEL [179]	Automatic (Hybrid analysis)	Automatic (Tagged architecture)	RISC-V (PIPE)	Coarse-grained / Fine-grained
ACES [41]	Semi-automatic (User-defined)	Automatic (Instrumentation)	ARM (MPU)	Coarse-grained
EC [111]	Semi-automatic (User-defined)	Automatic (Instrumentation)	ARM (N/A)	Coarse-grained / Fine-grained
HAKCS [148]	Manual (Annotation)	Manual (API)	ARM (PAC/MTE)	Coarse-grained
KSPLIT [100]	Semi-automatic (Static analysis)	Manual (LVDs [`VMFUNC`])	Intel (VT-x)	Coarse-grained
CHERI [222]	Manual (User-defined)	Automatic (Capability-system)	ARM & RISC-V (CHERI)	Coarse-grained / Fine-grained
GLAMDRING [135]	Manual (Annotation)	Automatic (Instrumentation)	Intel (SGX)	Coarse-grained
ERIM [211]	Semi-automatic (`ptrace`-based monitor)	Manual (API)	Intel (MPK)	Fine-grained
CERBERUS [214]	Semi-automatic (`ptrace`-based monitor)	Manual (API)	Intel (MPK)	Fine-grained
SOAAP [86]	Manual (Annotation)	Manual (Capscicum [221])	Intel (N/A)	Coarse-grained / Fine-grained
IBCS (Our work)	Automatic (Hybrid analysis)	Automatic (Assembly Rewriting)	Intel (N/A)	Fine-grained

titative metrics that establish an optimized compartmentalization policy without manual input. However, SCALPEL specifically targets global variables, which are considered inherently risky.

The question of how to identify potentially dangerous data is an extremely important consideration when automatically enforcing protection mechanisms onto a system. In the context of fine-grained data protection mitigation contributions such as Shroud, the premise of identifying such dangerous data is based on the type-based static analysis from the work of [121]. This analysis attempts to understand the type and usage of a variable to determine whether it is safe (i.e., data involved in straightforward, predictable operations that can be ensured not to overwrite critical data or pointers) or unsafe (i.e., pointers that can be influenced by external inputs, potentially leading to unpredictable behavior).

Similarly, **IBCS** is inspired by the premise that external input can be considered a primary vector for introducing potential vulnerabilities into a system. Therefore, in this context, we first assume all input is potentially dangerous. By tracking the data flow, potentially dangerous data can be determined based on how such data is used throughout the system. This is the premise behind a classical technique called interprocedural data-flow analysis (IDFS) [12, 184]. However, considering all input (both based on a type-based rule and external input) may lead to an overapproximation, which requires using a well-studied technique called taint analysis to identify compartmentalization targets with higher accuracy.

Taint analysis is a well-justified technique for identifying potentially dangerous data because it effectively tracks the flow from the source of an external input through a program, marking such input from untrusted sources as "tainted" and monitoring its propagation [162, 183, 229]. This allows for precise identification of where and how untrusted data interacts with the system, thereby highlighting potential vulnerabilities. Taint analysis leverages IDFS to track the flow of tainted data across function boundaries, providing a comprehensive view of how untrusted data propagates through an entire program. This approach is particularly effective for identifying vulnerabilities related to input validation and sanitization, which are common attack vectors in real-world exploits.

In addition, LangSec [21, 188] is another approach that considers all input to be potentially dangerous, based on the premise that improper handling and invalid input parsing are common reasons why vulnerabilities occur. However, in contrast to taint analysis, LangSec advocates that the compiler should be responsible for properly parsing external inputs to ensure that only well-defined and predictable inputs are accepted. In other words, LangSec focuses on *input validation* through correct grammars and parsing within the compiler to prevent vulnerabilities. Although LangSec is conceptually different from the taint analysis technique, both approaches address the problem of identifying potentially dangerous data by initially considering all input to be potentially dangerous.

Compartment Enforcement There has been extensive work on enforcing runtime compartments achieved through software, hardware, or both. For instance, several studies [92, 116], including KSplit, utilize VM functions (`VMFUNC`) that allow the VT-x guest to change

extended page table (EPT) mappings. ARM also offers a feature called the Memory Protection Unit (MPU), which enables users to define regions of memory with specific access permissions. ACES leverages the MPU to create compartments based on functionality, thus separating distinct functionalities into different compartments. EC [111] uses a formally verified microkernel called the Embedded Compartmentalizer Kernel (ECK) and intra-kernel isolation in the ARM architecture to achieve compartmentalization.

Many works have also used Intel MPK as a compartment enforcement mechanism [211, 214]. Although the MPK itself does not guarantee code page safety, user-space instruction can be used to realize low-cost data compartmentalization. ERIM [211] utilizes a binary rewriting technique to prevent adversaries from exploiting unintended sensitive MPK instructions. Cerberus [214] can intercept MPK-related operations and protect the system by preventing adversaries' attempts at circumventing compartmentalization enforcement.

In recent years, several hardware primitives based on tagged memory architecture have been proposed and are gaining popularity. Notably, ARM's Memory Tagging Extension (MTE) and CHERI [222] are prominent examples. Tagged memory architecture assigns a tag to each memory block, which researchers can utilize to enforce various security mechanisms.

Specifically, HAKC employs ARM MTE to compartmentalize components within the kernel. Meanwhile, CHERI, a capability hardware-enhanced RISC instruction set, utilizes a memory tag to create an *unforgeable* fat pointer. This tag ensures the integrity of the pointer, where "capability" refers to the extent of memory that the CHERI pointer can access. However, CHERI faces compatibility issues as it necessitates extending pointers (e.g., from 64-bit to 128-bit) and altering the processor architecture to accommodate its capability model.

Another related work, SCALPEL, aligns closely with **IBCS**. It leverages the tagged architecture provided by RISC-V to implement word-sized metadata tags. These tags are associated with data words in the system and are used to denote privilege levels and enforce a variety of security policies. Consequently, SCALPEL is primarily targeted at embedded systems equipped with tagged architectures.

3.9 Software Fault Isolation (SFI)

XFI is inspired by Lightweight Fault Isolation (LFI) [233] by Yedidia, which builds upon the foundational work of Software Fault Isolation (SFI) by Wahbe et al. [215]. Isolating a potentially vulnerable program into its own address space effectively ensures that faults remain confined within that domain, thereby preventing impact on other system components. For instance, either a virtual machine [203, 245] or Remote Procedure Calls (RPC) [19] can be used to create fault isolation. However, achieving fault isolation through separate address spaces incurs significant overhead due to control transfers across boundaries and poses scalability concerns.

SFI innovates by *encapsulating* software to achieve fault isolation within a single address space. The goal, as stated by Wahbe et al. [215], is to provide a cost-effective and accessible technique for developers. SFI employs a transformation technique called "sandboxing," which isolates modules within their own *fault domain*—a logically separate segment of the address space. This approach prevents exploitation within one domain from affecting others.

The SFI system accomplishes this by reserving several registers to store necessary data, a practice followed by both **XFI** and LFI. Additionally, it partitions an application's virtual address space into *segments*, using a specialized upper bit known as the *segment identifier* to differentiate between applications. It is important to note that the "segments" referred to in the work of SFI are different from the concept of segmentation memory proposed in the x86-32 architecture (which we will discuss later). SFI also inserts essential instructions to conduct checks before each unsafe instruction by modifying the compiler (e.g., `gcc`). This technique is referred to as the Inline Reference Monitor (IRM) strategy in [210].

A key innovation of SFI was redirecting all potentially unsafe operations and performing necessary checks to isolate vulnerable processes using dedicated registers within a single address space in what they call it as *segment matching*. However, the original system incurred high overhead due to the requirement of two checks before every memory access and its reliance on the RISC architecture. This limitation prompted researchers to explore methods to reduce overhead or adapt the system to different architectures like CISC [67, 72, 147, 193, 234].

Implementing SFI in the CISC architecture presents challenges due to the variable size of x86 instructions, which range from one to fifteen bytes. This variability complicates ensuring control-flow transfers are aligned and do not jump into the middle of an instruction. PittSFIeld by McCamant and Morrisett [147] addressed this by inserting `nop` instructions to align branch targets, enhancing control-flow enforcement in x86 SFI policies. This approach was subsequently adopted by other projects, such as Google Native Client (NaCl) [193, 234], to strengthen their security measures.

Vx32 [72] leveraged the x86's segmentation facility to limit sandbox guest data accesses. Similarly, Google's NaCl [234] employed a unique sandboxing methodology utilizing segmentation hardware. Segmentation hardware deploys a memory management scheme known as segmentation memory, which divides the memory into segments aligned with segment selectors and segment descriptors to determine the access permissions of each segment, along with its base address and size.

Although both approaches applied SFI concepts to the x86 architecture, they were implemented on 32-bit hardware due to the requirement for segmented memory, which is absent in 64-bit hardware [116]. Sehr et al. ported NaCl to x86-64 [193]. Although this port supported 64-bit computing and a larger address space, it suffered from significant hardware differences that necessitated extensive modifications to the compiler toolchain and limited the enforcement scope to memory writes to mitigate performance overhead.

LFI [233] revisits the classical SFI problem, aiming to provide an efficient software-only SFI scheme for multiple processes within a single address space. Unlike previous SFI or fault isolation solutions (using virtual machines or containers), LFI does not support the CISC architecture but leverages the RISC architecture to simplify the SFI scheme without invasive compiler modifications. This is achieved through assembly rewriting and reducing the Trusted Computing Base (TCB) by proposing a static verifier that ensures program adherence to the proposed isolation invariants.

Chapter 4

VDB: Verification of Code Diversification Techniques

As the initial work of this dissertation, we explore a variety of security techniques in-depth to understand their concepts and implications. We propose a Verification of Diversified Binary (**VDB**) algorithm and present a methodology to verify the functional equivalence between vanilla and diversified binaries. Our approach involves disassembling both binaries to extract their respective assembly codes, constructing the Control-Flow Graph (CFG) from these codes, and normalizing local variables for CFG comparison. We then compare the CFGs by treating them as transition systems and checking for bisimilarity, ensuring a bisimulation (a binary relation between transition systems) exists between both binaries.

Section 4.1 provides an overview of the methodology's steps. Next, we introduce formal definitions of soundness. Section 4.2 presents a definition of divergence-sensitive stuttering bisimulation. Section 4.3 defines a transition relation, and finally, Section 4.4 defines a state comparison function. These foundational concepts are crucial for understanding our approach to verifying the functional equivalence between vanilla and diversified binaries.

4.1 Overview of Methodology

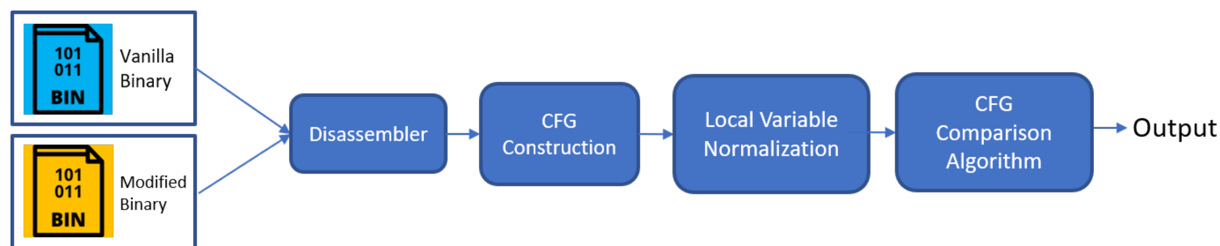


Figure 4.1: Overview of verifying functional equivalence

This section provides an overview of the input, intermediate steps, and output of our method. Figure 4.1 presents the overview of our methodology, and we use Figure 4.2 as a running example. Note that the C code in Figure 4.2a is shown for the sake of presentation only: we do not require it as an input to our method; we use this example to demonstrate how a diversification technique affects the original binary.

```
1   void *foo();
2   void *bar();
3
4   int main(int argc,
5   char **argv) {
6     int a = 0;
7     if (a == 0)
8     {
9       foo();
10    }
11    else
12    {
13      bar();
14    }
15    return 0;
16  }
17
18  void *foo() {
19    printf("Foo\n");
20  }
21
22  void *bar() {
23    int c = 17;
24  }
```

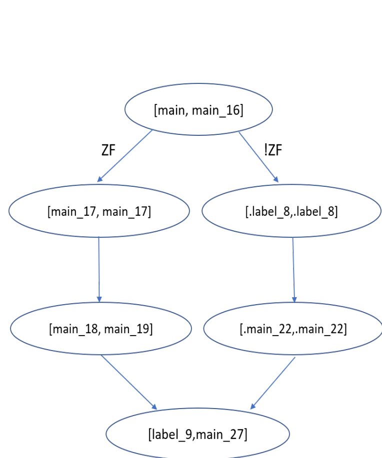
(a) Source code

```
1   main:
2     push rbp
3     mov  rbp, rsp
4     sub  rsp, 0x30
5     mov  dword ptr [rbp - 0x4], 0
6     mov  dword ptr [rbp - 0x8], edi
7     jne  .label_8
8     call foo
9     mov  qword ptr [rbp - 0x20], rax
10    jmp  .label_9
11  .label_8:
12    call bar
13    mov  qword ptr [rbp - 0x28], rax
14  .label_9:
15    xor  eax, eax
16    ...
17  foo:
18    ...
19    mov  dword ptr [rbp - 0xc], eax
20  bar:
21    push rbp
22    mov  rbp, rsp
23    mov  dword ptr [rbp - 0xc], 0x11
24    mov  rax, qword ptr [rbp - 0x8]
```

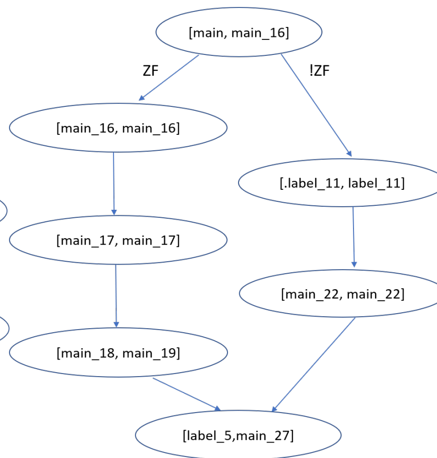
(b) Disassembled code

```
19  foo:
      ...
24    mov  dword ptr [rbp - 0x4], eax
      ...
11  .label_11:
      nop
12    call bar
13    mov  qword ptr [rbp - 0x28], rax
14  .label_5:
15    xor  eax, eax
      ...
1   main:
2     push rbp
3     mov  rbp, rsp
      mov  rbp, rbp
4     sub  rsp, 0x30
5     mov  dword ptr [rbp - 0x14], 0
6     mov  dword ptr [rbp - 0x04], edi
7     jne  .label_11
      lea  rdi, [rdi]
8     call foo
9     mov  qword ptr [rbp - 0x20], rax
10    jmp  .label_5
25  bar:
26    push rbp
27    mov  rbp, rsp
28    mov  dword ptr [rbp - 4], 0x11
      mov  rbp, rbp
29    mov  rax, qword ptr [rbp - 0x10]
```

(c) Disassembled diversified code



(d) Vanilla code CFG



(e) Diversified code CFG

```
Label relationship:
label_8 ↔ label_11
label_9 ↔ label_5

Local variable relationship
(Symbolic):
[rsp0 - 28, 4] ↔ {[rsp0 - 32, 4],
[rsp0 - 28, 4]}
[rsp0 - 16, 4] ↔ {[rsp0 - 12, 4]}
[rsp0 - 12 ,4] ↔ {[rsp0 - 32, 4],
[rsp0 - 28, 4]}

Successful check!
Relevant vanilla text sections: 3
Success: 3 text sections
```

(f) Framework output

Figure 4.2: Example

4.1.1 Disassembly

The methodology takes as input a vanilla and a diversified binary. The first step for our methodology is disassembly (see Figures 4.2b and 4.2c). Figure 4.2c shows an example of diversification, where function reordering reorders text sections (e.g., `main, foo, bar`). Furthermore, highlighted lines depict an example of a `nop` insertion technique where an instruction equivalent to `nop` instruction (e.g., `mov rbp, rbp` and `lea rdi, [rdi]`) is inserted throughout the code.

A key feature of disassembly that we require is *symbolization*. Symbolization replaces concrete addresses with symbolic values, i.e., labels. Binaries are usually compiled (or translated into machine-readable code) from high-level source code. After compilation, the assembler generates an object file (architecture-dependent) linked with different libraries to create a binary executable. During this process, a large portion of useful high-level information disappears. Specifically, relocation information disappears, which a linker uses to maintain the coherence between multiple libraries that the binary calls. Symbolization aims at recovering relocation-related information.

An example of symbolization is shown in Lines 11 and 14 of Figures 4.2b and 4.2c. The text section names of `main, foo`, and `bar` are preserved by compilation from the source code. The labels (`.label_8` and `.label_9`), however, are not present in the original binary. They are the result of symbolizing concrete addresses.

We use `Ramblr` for disassembly and symbolization [216]. `Ramblr` is a subset of an open-source disassembler framework `angr`. `angr` reliably disassembles the binary, and the authors of this work demonstrated its ability on a set of binaries from the DARPA cyber grand challenge (a computer security competition dealing with malicious binaries) [201].

4.1.2 Control-flow Graph Construction

The second step is the CFG construction. We extract CFGs from the binary using an off-the-shelf algorithm similar to `angr`'s CFGFast [201]. A CFG consists of nodes and edges. A node consists of a basic block (i.e., a list of instructions). Edges are labeled with flags. We construct one CFG per text section (for both vanilla and diversified binaries). The reason for this is because our methodology proves functional equivalence per text section rather than iterating through the complete CFG of an entire binary.

The vanilla and diversified binaries are supposed to be functionally equivalent, but as Figures 4.2d and 4.2e show, the diversified CFG is different. As previously mentioned, the process of `nop` insertion [95] inserted a `lea` instruction just before the `call foo` statement (between Lines 7–8 in Figure 4.2c). This causes the difference in CFGs, and it complicates the functional equivalence verification process, as we cannot only check whether the two CFGs are equal. Instead, we must check whether they are stuttering bisimilar.

4.1.3 Local Variable Normalization

Variables are storage locations that the program can manipulate. Local variables are referenced by an offset from special-purpose registers (the frame pointer `rbp` or the stack pointer `rsp`). Register `rbp` points to the top of the current stack frame and `rsp` points to the bottom.

After constructing CFGs, we normalize local variables to prepare for our comparison algorithm. Normalizing in this context means that local variable offsets become relative to the *initial* value of the stack ($\texttt{rsp}_0$) pointer. For example, in Figure 4.2b, the memory address $\texttt{rbp} - \texttt{0x4}$ is accessed at Line 5. Since the frame pointer is a copy of the stack pointer (Line 3), which has been decremented with 8, this memory address becomes $\texttt{rsp}_0 - \texttt{0x12}$ after normalization. Normalization allows us to compare memory locations with their shuffled counterparts.

4.1.4 CFG Comparison

Bisimulation is a relation between two transition systems, where one can simulate the others' (transitions) and vice versa. In particular, we use *divergence-sensitive stuttering bisimulation* [11, 22] for our work. To check whether two CFGs are stuttering bisimilar, we treat them as transition systems and check whether they are stuttering bisimulation-equivalent. In other words, we check if there exists a stuttering bisimulation between both transition systems.

The intuition behind a stuttering bisimulation deals with internal steps, steps that perform no visible behavior. Stuttering bisimulation allows two transition systems to be equivalent regardless of how many internal steps it takes to get to the next state. It mainly deals with basic blocks consisting solely of instructions that perform no state change, such as a basic block [`main_18, main_19`] in Figure 4.2e. Divergence sensitivity prevents a situation where one of the transition systems permanently executes internal steps, whereas the other does not.

A divergence-sensitive stuttering bisimulation is a strong notion of equivalence, as it preserves a large set of properties. This set of properties includes safety, liveness, and reachability properties. Formally, the set of properties preserved is computation tree logic (CTL) except the next operator, i.e., $\text{CTL}^*\backslash\mathbf{X}$ [11]. In other words, this means that any CTL^* property is preserved as long as it does not concern specifically the current branching decision.

4.1.5 Output

If our methodology does not identify a counterexample, it outputs a statement confirming that the binaries are functionally equivalent (see Figure 4.2f). However, if any potential issues are detected (e.g., limitations or discrepancies), our approach identifies the specific basic block responsible for the discrepancy. Additionally, our work provides supporting evidence for this claim by mapping the corresponding segments between the original and diversified binaries.

When a program becomes diversified by instruction or basic blocks reordering techniques, relocation information of such a program gets affected. For example, `.label_8` and `.label_11` from Figures 4.2b and 4.2c are semantically equivalent (Line 11 for both), but different in label names. The CFG comparison algorithm thus keeps track of a map relating labels in the vanilla world to labels in the diversified world. Similarly, stack shuffling may cause local variables to be put into different memory regions. A mapping is maintained that relates diversified memory regions to vanilla ones. For example, the 4-byte vanilla memory region $[\texttt{rsp}_0 - 16, 4]$ (accessed at Line 6 in Figure 4.2b) is mapped to the diversified memory region $[\texttt{rsp}_0 - 12, 4]$ (accessed at Line 6 in Figure 4.2c).

The rest of this chapter will define an equivalence relation over binaries. We will call a code diversification effort *sound* if the vanilla binary and the produced diversified binary are equivalent under that relation.

4.2 Divergence-sensitive Stuttering Bisimulation

The key idea is to establish a *divergence-sensitive stuttering bisimulation* over the transition systems modeling the two binaries [11]. Formally, a transition system TS is defined by a tuple $\langle S, \rightsquigarrow, I \rangle$. Here S is a set of states, $\rightsquigarrow$ of type $S \times S \mapsto \mathbb{B}$ is a transition relation, and I is a set of initial states.

Note that this definition omits either a labeling function or actions on edges. Typically, definitions of stuttering bisimulation are based on one of these two. However, due to stack-frame shuffling, neither of these is convenient. For example, let both the vanilla and diversified binaries have two labeling functions L_0 and L_1. Both translate concrete states to atomic propositions. Stuttering bisimulation then considers two states s and s' to be equal only if $L_0(s) = L_1(s')$.

Consider again function `main` in Figures 4.2b and 4.2c. The labeling functions should translate states to atomic propositions so that the value stored at the vanilla memory region $[\texttt{rsp}_0 - \texttt{16}, \texttt{4}]$ and the value stored at the diversified memory region $[\texttt{rsp}_0 - \texttt{12}, \texttt{4}]$ map to the same atomic proposition. The labeling function is thus hard to define, as it is dependent on the relation established between the two binaries. Instead, we will formalize a *state*

comparison function $\doteq$ of type $S_0 \times S_1 \mapsto \mathbb{B}$ and define stuttering bisimilarity relative to the given state comparison function.

Definition 4.1. Let TS_0 and TS_1 be two transition systems, and let $\doteq$ of type $S_0 \times S_1 \mapsto \mathbb{B}$ be a state comparison function. Binary relation $\mathcal{B}$ of type $S_0 \times S_1 \mapsto \mathbb{B}$ is a *divergence-sensitive stuttering bisimulation wrt.* $\doteq$, if and only if, for any states $s_0 \in S_0$ and $s_1 \in S_1$, such that $\mathcal{B}(s_0, s_1)$:

1. $s_0 \doteq s_1$
2. if $s_0 \rightsquigarrow_0 s_0'$ and $\neg\mathcal{B}(s_0', s_1)$, then there exists a finite path fragment $[s_1, t_0 \dots t_n, s_1']$ such that $\mathcal{B}(s_0, t_i)$ for all i and $\mathcal{B}(s_0', s_1')$
3. if $s_1 \rightsquigarrow_1 s_1'$ and $\neg\mathcal{B}(s_0, s_1')$, then there exists a finite path fragment $[s_0, t_0 \dots t_n, s_0']$ such that $\mathcal{B}(t_i, s_1)$ for all i and $\mathcal{B}(s_0', s_1')$
4. there exists an infinite path fragment $[s_0, t_0, t_1 \dots]$ such that $\mathcal{B}(t_i, s_1)$ for all i, if and only if, there exists an infinite path fragment $[s_1, u_0, u_1 \dots]$ such that $\mathcal{B}(s_0, u_j)$ for all j.

Two transition systems are divergence-sensitive stuttering bisimilar wrt. $\doteq$, notation $TS_0 \approx TS_1$, if and only if there exists a divergence-sensitive stuttering bisimulation $\mathcal{B}$ that relates all initial states, i.e., for all $s_0 \in I_0$, there exists some $s_1 \in I_1$ such that $R(s_0, s_1)$, and the other way around.

The soundness of diversification is expressed as a property over CFGs. We assume the existence of a function `cfg` that takes as input a binary and produces a CFG. Formally, a CFG consists of a tuple $\langle B, \rightarrow, e\rangle$, where B denotes a set of basic blocks, $\rightarrow$ of type $B \times B \mapsto \mathbb{B}$ denotes a transition relation, and e of type B denotes the entry point. The start address of a basic block b can be accessed via b.`addr`, the list of instructions via b.`instrs`.

4.3 Transition Relation

We consider the transition system corresponding to a given CFG. The transition system consists of concrete states that map registers, 64-bit byte-addressable memory, and flags to values. We use S_C to denote the set of concrete states, R to denote the set of registers, and A to denote the address space. Given a concrete state s, we use s.`rip` to denote the value stored in register `rip`, and similar for other registers and flags. Notation s.`mem`(a) returns given a 64-bit address a the byte-value stored in the memory at that address. Function run of type $B \times S_C \mapsto \{S_C\}$ takes as input a basic block and the current concrete state, then runs the list of instructions in the basic block. It returns the set of possible next concrete states. Since a basic block does not have loops, this function terminates.

Definition 4.2. Let $g = \langle B, \twoheadrightarrow, e \rangle$ be a CFG. The transition system of g, notation $\overline{g}$, is defined by $\langle S_C, \rightsquigarrow, I \rangle$. Here set of initial concrete states I is defined as follows:

$$I \stackrel{\text{def}}{=} \{s \mid s.\texttt{rip} = e.\texttt{addr}\}$$

and transition relation $\rightsquigarrow$ is constructed as follows:

$$\frac{s' \in \text{run}(b, s) \wedge b \twoheadrightarrow b' \wedge s.\texttt{rip} = b.\texttt{addr} \wedge s'.\texttt{rip} = b'.\texttt{addr}}{s \rightsquigarrow s'}$$

In words, the transition system $\overline{g}$ starts at states whose instruction pointer `rip` is equal to that of the first instruction of the basic block that is the CFGs' entry point. It then moves from state to state by executing entire basic blocks. The transition system is thus at the same granularity as the CFG.

4.4 State Comparison Function

Definition 4.1 depends on a state comparison function. The stronger this comparison function, the stronger the equivalence. As illustrated, if $s_0 \doteq s_1$ is true for any state, then any two transition systems are bisimilar. We thus define a comparison function for concrete states as strong as possible. Ideally, we want to compare all state parts, i.e., all registers, memory, and flags. In practice, we consider only the set of *relevant* registers. For instance, the instruction pointer (`rip`) is irrelevant: in the two worlds, it will differ since the executed instructions have different addresses due to, e.g., `nop` insertion. The frame- and stack pointers are irrelevant since stack frame shuffling can enlarge the stack frame. Which registers are relevant may depend on the current state. This is modeled with a function $\mathcal{R}$ that returns relevant registers (e.g., all registers except the irrelevant ones) given the current state. In practice, we ignore flags: their impact on execution is covered by proving that the same branching decisions are made. Finally, we need to map diversified memory addresses to their vanilla counterparts. This is modeled with a mapping $\mathcal{M}$.

Definition 4.3. Let $\mathcal{R}$ of type $S_C \mapsto \{R\}$ be a function that returns a set of relevant registers given a concrete state. Let $\mathcal{M}$ of type $A \mapsto A$ be a mapping from diversified memory addresses to vanilla memory addresses. The *concrete state comparison function* of M, notation $\stackrel{\text{c}}{=}_{\langle \mathcal{R}, \mathcal{M} \rangle}$, is defined as follows:

$$s_0 \stackrel{\text{c}}{=}_{\langle \mathcal{R}, \mathcal{M} \rangle} s_1 \stackrel{\text{def}}{=} \begin{cases} & \forall \texttt{r} \in \mathcal{R}(s_0) \cdot s_0.\texttt{r} = s_1.\texttt{r} \\ \wedge & \forall a \in A \cdot s_0.\texttt{mem}(a) = s_1.\texttt{mem}(\mathcal{M}(a)) \end{cases}$$

In words, two concrete states are considered equal if all relevant registers are equal and all memory in both worlds is the same after mapping diversified addresses to vanilla addresses. For example, in Figure. 4.2, the values stored at vanilla address $\texttt{rsp}_0 - \texttt{16}$ and diversified address $\texttt{rsp}_0 - \texttt{12}$ will be compared.

Definition 4.4. Let B be a binary and let D be a diversification function that takes as input a binary and produces a diversified binary. Diversification D is sound for binary B, if and only if, there exists a function $\mathcal{R}$ and mapping $\mathcal{M}$ such that the transition systems of the CFGs are divergence-sensitive stuttering bisimilar wrt. the concrete state comparison function of $\mathcal{R}$ and $\mathcal{M}$.

$$\text{sound}(D, B) \stackrel{\text{def}}{=} \exists \mathcal{R}, \mathcal{M} \cdot \overline{\text{cfg}(B)} \approx \overline{\text{cfg}(D(B))} \text{ wrt. } \stackrel{\text{c}}{=}_{\langle \mathcal{R}, \mathcal{M} \rangle}$$

Chapter 5

VDB Algorithm

To check soundness (see Definition 4.4), a witness must be found for function $\mathcal{R}$ and mapping $\mathcal{M}$. This chapter presents an algorithm called Verification of Diversified Binary (VDB) to find these witnesses.

Section 5.1 produces the function $\mathcal{R}$ and mapping $\mathcal{M}$ by running symbolic execution on each basic block in the two CFGs. Section 5.2 explains how we express all local variables in terms of the initial value of the stack pointer $\texttt{rsp}_0$. Lastly, in Section 5.3, we check for stuttering bisimulation on the symbolized CFGs.

5.1 Symbolic Execution

The purpose of symbolic execution is to express the semantics of each basic block in a way that is independent of the actual instructions. Figure 5.1 depicts an example of symbolic execution where we show a basic block and its symbolic output. The symbolic output consists of assignments of symbolic expressions to state parts (registers, memory flags). Symbolic expressions consist of, among others, immediate values, reading from state parts, and common bit-vector operations. These operations include taking bit subsets, concatenation, logical operators, casting operators, floating-point operators, and signed and unsigned arithmetic. For example, after the execution of the basic block in Figure 5.1, register `rax` (`eax` is the lower 32 bits of `rax`) has become 0, and the sign flag is set by a signed integer comparison of the lower 32 bits of the `rdi` and `rsi` registers. All values are relative to the initial state of the basic block. For example, the instruction at Line 8 uses the frame pointer `rbp`. However, since that value at that line is equal to the initial stack pointer minus 8, the instruction results in a write to the symbolic memory region $[\texttt{rsp} - \texttt{0xc}, 4]$.

Figure 5.1b presents that symbolic execution produces a result largely agnostic of diversification. We modified the basic block with two features found in typical diversification tools for the sake of presentation. At Line 5 in Figure 5.1a, a `lea` instruction has been inserted that performs no state change: effectively a `nop`. Instead of directly writing 0 to memory with one instruction, we use two instructions (Lines 6 and 7). First, `xor` instruction writes zero to register `eax` (which denotes the lower 32 bits of register `rax`), and then `mov` is used to do the memory write. Symbolization does not reflect these modifications since they do not influence the semantics of the basic block. The basic block without the manual modifications would

```
1  main:
2    push rbp
3    mov  rbp, rsp
4    sub  rsp, 0x30
5    lea  rdi, [rdi]
6    xor  eax, eax
7    mov  dword ptr [rbp-0x14], eax
8    mov  dword ptr [rbp-0x4] , edi
9    cmp  dword ptr [rbp-0x4] , esi
10   jne  .label_22
```

(a) Diversified `main` basic block

$$
\begin{array}{lcl}
\texttt{rsp} & := & \texttt{rsp} - 0\text{x}38 \\
\texttt{rbp} & := & \texttt{rsp} - 0\text{x}8 \\
\texttt{rax} & := & 0 \\
[\texttt{rsp} - \texttt{0x8}, 8] & := & \texttt{rbp} \\
[\texttt{rsp} - \texttt{0x1c}, 4] & := & 0 \\
[\texttt{rsp} - \texttt{0xc}, 4] & := & \langle 31, 0\rangle(\texttt{rdi}) \\
\texttt{ZF} & := & \langle 31, 0\rangle(\texttt{rdi}) = \langle 31, 0\rangle(\texttt{rsi}) \\
\texttt{CF} & := & \langle 31, 0\rangle(\texttt{rdi}) < \langle 31, 0\rangle(\texttt{rsi}) \\
\texttt{SF} & := & \langle 31, 0\rangle(\texttt{rdi}) <_s \langle 31, 0\rangle(\texttt{rsi})
\end{array}
$$

(b) Symbolic execution output

Figure 5.1: Symbolic execution example

have produced the same symbolic output. Finally, the example demonstrates that when comparing a vanilla and a diversified basic block, only the registers modified in the vanilla context are deemed relevant. The register `rax` would not be included in the symbolic output for an unmodified basic block. However, any state changes made in the vanilla context are similarly applied in the diversified context.

Formally, a symbolic state σ consists of registers, memory, and flags (we use s and σ for resp. concrete and symbolic states). For each modified register $\texttt{r}$, notation $\sigma.\texttt{r}$ returns a symbolic expression. The set of modified registers is returned by $\sigma.\texttt{regs}$. The memory is modeled by assigning symbolic expressions to symbolic memory regions. Consider the example in Figure 5.1b again. The notation $\sigma.[\texttt{rsp} - \texttt{0x8}, 8]$ returns the symbolic expression `rbp`. The set of modified memory regions is denoted by $\sigma.\texttt{mems}$.

The algorithm will compare symbolic states instead of concrete ones. This requires us to formulate a *symbolic* state comparison function, the symbolic counterpart to its concrete version (see Definition 4.3).

Definition 5.1. Let $\mathcal{N}$ be a mapping from diversified symbolic memory regions to vanilla symbolic memory regions. The *symbolic state comparison function* of $\mathcal{N}$, notation $\stackrel{s}{=}_{\mathcal{N}}$, is defined as follows:

$$
\sigma_0 \stackrel{s}{=}_{\mathcal{N}} \sigma_1 \stackrel{\text{def}}{=} \begin{cases} & \forall \texttt{r} \in \sigma_0.\texttt{regs} \cdot \sigma_0.\texttt{r} = \sigma_1.\texttt{r} \\ \wedge & \forall \texttt{r} \in \sigma_0.\texttt{mems} \cdot \sigma_0.\texttt{r} = \sigma_1.\mathcal{N}(\texttt{r}) \end{cases}
$$

Mapping $\mathcal{N}$ is defined over symbolic memory regions and symbolic expressions. All symbolic values can be concretized given a concrete state and the memory regions. Function γ takes as input a symbolic mapping $\mathcal{N}$ and produces a concrete mapping $\mathcal{M} = \gamma(\mathcal{N})$. For example, if diversified memory region $[\texttt{rsp} - \texttt{8}, 8]$ is mapped by $\mathcal{N}$ to vanilla region $[\texttt{rsp} - \texttt{16}, 8]$, then $\mathcal{M}$ will relate 8 individual concrete addresses based on the values of the concrete stack pointers.

Definition 5.2. Symbolic execution is sound, if and only if, for any basic blocks b_0 and b_1, assumption

$$\text{se}(b_0) \stackrel{s}{=}_{\mathcal{N}} \text{se}(b_1)$$

implies:

$$\forall s_0, s_1 \in S_C \;\cdot\; s_0.\texttt{rip} = b_0.\texttt{addr} \;\wedge\; s_1.\texttt{rip} = b_1.\texttt{addr} \Longrightarrow \text{run}(b_0, s_0) \stackrel{c}{=}_{\langle \mathcal{R}, \mathcal{M} \rangle} \text{run}(b_1, s_1)$$

where

$$\begin{array}{lcl} \mathcal{R}(s_0) & = & \text{se}(b_0).\texttt{regs} \\ \mathcal{M} & = & \gamma(\mathcal{N}) \end{array}$$

In other words, comparing two symbolic states should suffice to show a successful comparison of all the concrete states they represent. The set of relevant registers $\mathcal{R}$ is the set of all modified registers in the vanilla world. The concrete memory map $\mathcal{M}$ is obtained by concretizing the symbolic memory region map.

5.2 $\texttt{rsp}_0$ Substitution

Local variables are addressed relative to either the stack pointer `rsp` or the frame pointer `rbp`. The values in these registers are not static, i.e., they change during the function's execution. This change complicates formulating a static address mapping $\mathcal{M}$. Consider Lines 5 and 9 of Figure 4.2b, which deal with addresses $[\texttt{rbp} - \texttt{0x4}]$ and $[\texttt{rbp} - \texttt{0x20}]$. In between these lines, the value of register `rbp` may have changed so that these addresses are equal. To formulate a static mapping $\mathcal{M}$, we thus make all local variables relative to the initial value of the stack pointer $\texttt{rsp}_0$. This is achieved by propagating two substitutions through the symbolized CFG. Initially, these two substitutions are $\texttt{rsp} \coloneqq \texttt{rsp}_0$ and $\texttt{rbp} \coloneqq \texttt{rbp}_0$. Thus, for the entry block, any occurrence of the stack pointer is simply replaced by $\texttt{rsp}_0$. The substitutions are updated if the current basic block updates either the stack- or frame pointer. In the example, after the first basic block, the current substitution is $\texttt{rsp} \coloneqq \texttt{rsp}_0 - \texttt{0x38}$ and $\texttt{rbp} \coloneqq \texttt{rsp}_0 - \texttt{0x8}$.

In this manner, substitutions are propagated through the control flow graph (CFG). When a visited node is encountered (such as a loop or converging paths in the CFG), we check that the current substitution aligns with the one already applied to the visited node. If this condition is met each time a visited node is encountered, then the current substitutions establish an invariant.

5.3 Stuttering Bisimulation Check

Algorithm 1 Check Between Vanilla and Diversified CFGs

```
1:  function CHECK(b_0, b_1)
2:      if b_0 and b_1 are unvisited then
3:          mark b_0 and b_1 as visited
4:          σ_0 = se(b_0)
5:          σ_1 = se(b_1)
6:          UPDATE (N, σ_0, σ_1)
7:          if σ_0 =^s_N σ_1 ∧ eq_branching(b_0, b_1) then
8:              for each (b'_0, b'_1) ∈ get_children(b_0, b_1) do
9:                  CHECK (b'_0, b'_1)
10:             end for
11:         else if is_skip(b_1) then
12:             mark b_1 as visited
13:             CHECK (b_0, b'_1) with b_1 ⇀ b'_1
14:         else
15:             mark current b_1 text section as counterexample
16:             return False
17:         end if
18:     else
19:         return True
20:     end if
21: end function
```

Algorithm 1 presents a procedure **CHECK** that compares two CFGs. It takes as input the current basic blocks – initially starting with the entry points – and traverses the CFGs simultaneously. It provides as output a Boolean indicating the existence of a stuttering bisimulation (Line 19). Before we explain the algorithm in more detail, we introduce some definitions.

Definition 5.3. Let b_0 and b_1, be two basic blocks (vanilla and diversified respectively). Branching for basic blocks is equivalent, if and only if:

$$\mathsf{eq_branching}(\mathsf{b_0}, \mathsf{b_1}) \stackrel{\text{def}}{=} \forall f \cdot (\exists b'_0 \cdot b_0 \stackrel{\text{f}}{\rightharpoonup} b'_0) \Leftrightarrow (\exists b'_1 \cdot b_1 \stackrel{\text{f}}{\rightharpoonup} b'_1)$$

In words, equal branching returns true if both blocks have the same number of children with the same flags.

Definition 5.4. Let b_0 and b_1, be two basic blocks (vanilla and diversified respectively). The set of children of b_0 and b_1 is defined as follows:

$$\mathsf{get_children}(\mathsf{b_0}, \mathsf{b_1}) \stackrel{\text{def}}{=} \{(b'_0, b'_1) \cdot \exists f \cdot b_0 \stackrel{\text{f}}{\rightharpoonup} b'_0 \wedge b_1 \stackrel{\text{f}}{\rightharpoonup} b'_1\}$$

In words, *get_children* returns the set of children that is to be explored from current basic blocks b_0 and b_1.

Definition 5.5. Basic block b is a *skip* if and only if:

$$\mathsf{is_skip}(b) \stackrel{\text{def}}{=} \sigma.\mathtt{regs} \subseteq \{\mathtt{rip}\} \wedge \sigma.\mathtt{mems} = \emptyset$$

In other words, a basic block is a skip if it does not modify memory, and the only register that is modified (if any) is the instruction pointer `rip`.

Algorithm 1 essentially is a simultaneous depth-first exploration. If both basic blocks are flagged as visited, a stuttering bisimulation for the current basic blocks b_0 and b_1 has been established. If not, both basic blocks are flagged as visited, and the comparison continues. Each block is first symbolically executed, producing symbolic states σ_0 and σ_1. The currently established stuttering bisimulation relation is updated (Line 6). This update stores that from now on, $\mathcal{R}(s_0)$ returns the set of registers modified by basic block b_0, i.e., $\mathcal{R}(s_0) = \sigma_0.\mathtt{regs}$ for all states s_0 such that $s_0.\mathtt{rip} = \sigma_0.\mathtt{rip}$. Moreover, memory mapping $\mathcal{N}$ is updated.

If we find that two basic blocks are semantically equivalent with equal branching (Line 7) the check proceeds by exploring all children. If not, then the current diversified basic block maybe a skip. In that case, the check proceeds by comparing the current vanilla basic block b_0 to the child of the skip b_1'. If diversified basic block b_1 was not a skip, then a discrepancy has been found, and the algorithm returns false (Line 16).

Theorem 5.6. *Let $g_0 = \langle B_0, \rightarrow_0, e_0\rangle$ and $g_1 = \langle B_1, \rightarrow_1, e_1\rangle$ be the control flow graphs of the vanilla and diversified binaries respectively. Let $\mathcal{R}$ and $\mathcal{M} = \gamma(\mathcal{N})$ be the mappings produced by the algorithm. The algorithm decides a divergence-blind stuttering bisimulation:*

$$\textsc{Check}(e_0, e_1) \longleftrightarrow g_0 \approx g_1 \text{ wrt. } \stackrel{c}{=}_{\langle \mathcal{R}, \mathcal{M}\rangle}$$

Proof. Soundness of the algorithm is based on the work of Fernandez et al. [69]. In that paper, it is shown that a bisimulation can be decided by a depth-first search that explores two transition systems simultaneously. Proposition 3.2 of that paper states that two deterministic transition systems are bisimilar, if and only if, a simultaneous depth-first search is not able to find a path to a pair of states with a different number of children. That is exactly what is verified by our algorithm. A key difference is that the work of Fernandez et al. is formulated for an algorithm checking strong bisimulation. Line 11 of Algorithm 1 adds an additional case for dealing with stuttering steps: the diversified world can do an arbitrary number of skips before a bisimilar node is encountered. Divergence-sensitivity is guaranteed by checking whether the number of children is always equivalent for all encountered bisimilar nodes (Line 7).

□

Chapter 6

VDB Evaluation

In this chapter, we evaluate our VDB methodology. The primary goal is to prove the soundness of the evaluated diversification techniques or detect potential counterexamples introduced by the diversification tool and provide examples of these. Additionally, it is important to understand the limitations of our methodology, particularly in terms of any unsupported assembly codes.

Section 6.1 describes the setup environment of our evaluation. Section 6.2 presents the evaluation of our methodology. Next, Section 6.3 discusses the limitation of the `VDB` algorithm. Lastly, Section 6.4 presents the overall summary of our first contribution.

6.1 Setup Environment

We apply the methodology to all 93 binaries from GNU `Coreutils 8.31`. The source code is compiled on a Linux Ubuntu 16.04 x86-64 system using `clang` with various optimization levels (`-O0` to `-O3`). All experiments were performed on a machine featuring an AMD FX-8350 CPU and 8 GB of RAM.

GNU `Coreutils 8.31` binaries are diversified by the techniques mentioned above, with various optimization levels (-O0 to -O3) depending on a diversification tool. `Coreutils` is a suitable benchmark as they are standard programs from UNIX-based systems. Moreover, these binaries provide realistic high-coverage test cases ranging from small programs (10,692 assembly LOCs) to large programs (133,065 assembly LOCs).

6.2 Soundness Evaluation

Table 6.1 presents the evaluation of our methodology. There is a total of 93 binaries from the `Coreutils`. As the level of optimization increases, we could not analyze many binaries. We organize limitation scenarios in the following way:

- If a binary cannot be diversified, this is under the *not diversified* (Not Div.) column.

Table 6.1: The evaluation of our methodology on `GNU Coreutils 8.31`

Diversification Tools		Analyzed	Not Div.	Not Dis.	Sound	Counter-example	Unsupported
		Binaries			Text Sections		
`nop` insertion [95]	`-O0`	93	0	0	12966	0	546
	`-O1`	91	0	2	7427	0	831
	`-O2`	88	0	5	7519	0	930
	`-O3`	88	0	5	7694	0	1079
CCR [117]	`-O0`	93	0	0	12805	0	697
	`-O1`	87	1	5	10808	0	961
	`-O2`	83	2	8	6681	0	868
	`-O3`	79	8	6	6619	0	836
Stack shuffling [73]	`-O0`	93	0	0	12796	0	706

Binaries = result with respect to the number of binaries

Text Sections = result with respect to the number of text sections from analyzed binaries

- If a `Ramblr` cannot disassemble a binary, this is under the *not disassembled* (Not Dis.) column.
- For all other cases, they are under the *unsupported* (Unsupported) column.

For example, consider the data for the `CCR` tool on binaries compiled with `-O1`. Out of 93 binaries, we successfully analyzed 87 using this tool. The remaining binaries include one that could not be diversified by `CCR` (column Not Div.) and five that could not be disassembled by `Ramblr` (column Not Dis.). Among the 11,769 text sections in the 87 analyzed binaries, 10,808 were examined and found to be soundly diversified. There were zero text sections that were unsoundly diversified, and 961 text sections contained behavior unsupported by our tool (see Section 6.3).

The table shows we did not find a diversification issue in any of the binaries. However, for each of the listed techniques in Table 1.1, we manually introduced bugs indicative of that technique. For example, we manually inserted one `nop` too many in the diversified version of the `wc` program and reran our tool. For each inserted bug, our methodology reports a counterexample. This report provides information on which text section that the bug is in. Moreover, it also gives the line number of parsed disassembled code for vanilla and diversified binaries for in-depth debugging purposes.

6.3 Discussion

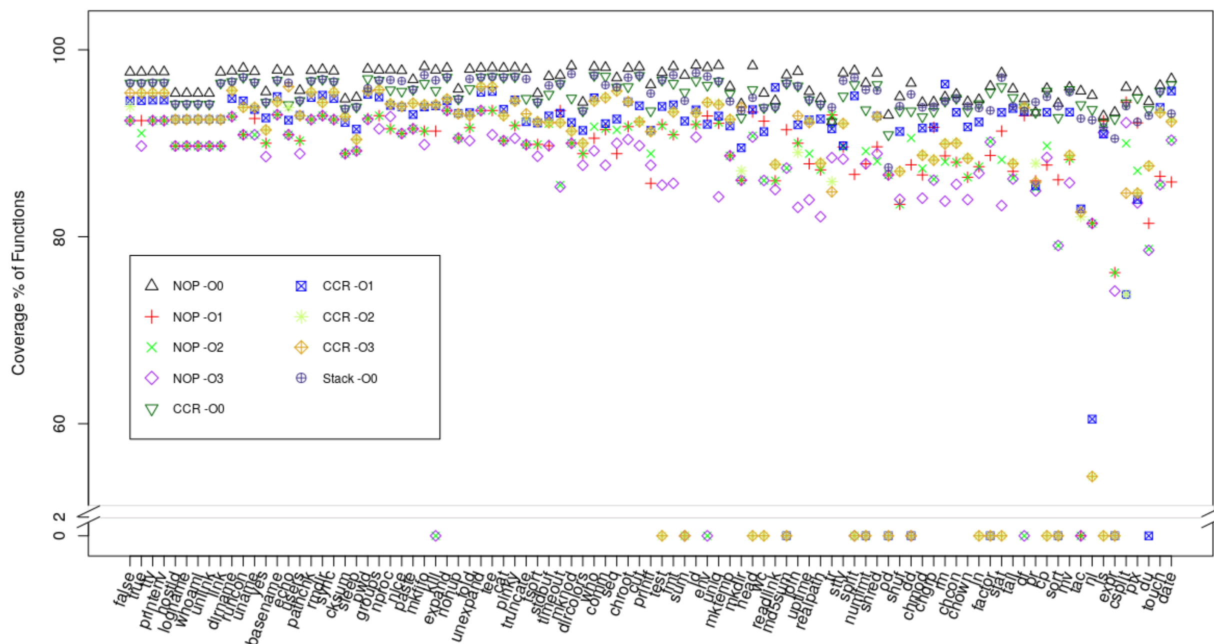


Figure 6.1: The coverage rate per diversification techniques over all GNU `Coreutils`

Our first contribution's main limitation is that we cannot deal with indirect branching (e.g., indirect jumps). Indirect branching occurs when jump- or call addresses are computed dynamically, and solving indirect branching requires assembly-level invariants on the values involved in that computation. Second, we cannot reason about shuffled local arrays, e.g., due to an `alloca` statement. The critical problem here is that the array size is not known at the assembly level.

Lastly, as the optimization level increases, a compiler performs expensive analyses and applies more aggressive transformations (e.g., perform a scalar replacement or loop transformations) to improve the binary. Therefore, as Figure 6.1 shows, as the optimization level increases, proving functional equivalence becomes more complex due to optimized instructions (e.g., packed instructions such as `puncpkhbw`). Lastly, if a disassembler cannot disassemble the binary, our tool cannot verify. Currently, our methodology only supports *diversification* techniques. Proving the functional equivalence between *code obfuscated* binaries to the vanilla binaries is more difficult due to the convoluted modifications that obfuscation techniques perform to camouflage the code.

6.4 Summary

Diversification is a security technique that produces multiple binaries that are different but semantically equal. This chapter of the dissertation presents the scalable and automated technique to establish the soundness of code diversification tools. Soundness is expressed by establishing an equivalence relation between vanilla and a diversified binary. The technique is based on disassembly, symbolic execution, establishing stack-pointer related invariants, and establishing mappings between memory regions in both the vanilla and the diversified world.

The methodology is applied to binaries compiled for the x86-64 architecture. We test the feasibility of our work with all of the binaries from `GNU Coreutils` for a variety of optimization levels (-O0 to -O3), depending on which diversification tool was applied. We evaluate three advanced diversification tools that carry out padding code insertion, instruction substitution, and the reordering of instructions, basic blocks, and text sections. Our findings indicate that our method demonstrated semantic equivalence for approximately 87% to 96% of the text sections within the binaries.

Chapter 7

Control-flow Restriction (CFR)

Security techniques verified in previous contributions are essential for understanding the implications of security measures; however, they do not actually prevent exploits. In fact, these techniques ensure that no single exploit can be duplicated across all instantiations of a potentially vulnerable binary. Therefore, even though an exploit may succeed in one instance and compromise part of the system, the system as a whole survives. Consequently, we explore a direction aimed at improving a type of security technique that aims to prevent exploits outright.

Control-Flow Integrity (CFI) is a well-known protection mechanism against code-reuse attacks [3]. CFI extracts the Control-Flow Graph (CFG) and uses this information on Indirect Control Transfer (ICT) instructions to enforce a control-flow policy. Many promising CFI variants exist in the literature, but the need for a CFG remains the main bottleneck for CFI. To address this, we revisit the problem of binary-only CFI and explore different directions in improving this technique by leveraging a deny list to remove the CFG bottleneck. We propose a CFI policy from an attacker's perspective called Control-Flow Restriction (**CFR**).

CFR aims to restrict any indirect control flow transfer to a code gadget's address. To achieve this, **CFR** relies on existing binary-only methods, such as gadget searching tools [187], disassemblers, and binary rewriters [62], to instrument a deny list for each ICT. **CFR** takes an ELF binary as input and scans the static binary for a list of unintended jump targets, generating a deny list. Next, **CFR** packs the addresses in the deny list into a bit array so that any prohibited address search can be performed in $O(1)$ time.

Section 7.1 discusses **CFR** and its relation to CFI. We present formal definitions of the **CFR** in Section 7.2. Section 7.3 discusses the motivating example of **CFR**. Finally, Section 7.4 presents an overview of the **CFR** methodology.

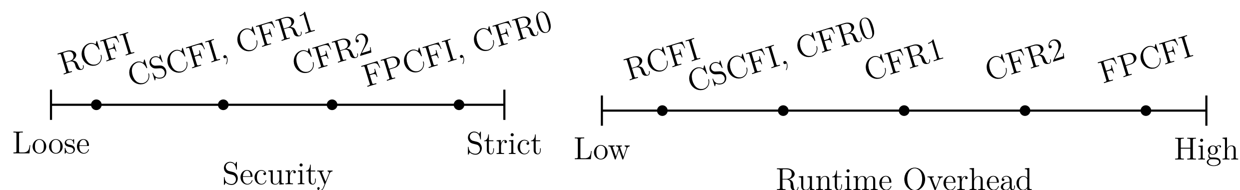


Figure 7.1: Security and runtime overhead comparison

7.1 Control-flow Restriction and CFI Relation

In this section, we first define a *control-flow policy* and different CFI variants. Then we introduce a novel security policy comparable to CFI: Control-flow Restriction (CFR).

7.1.1 Control-flow Policy

Let I be the set of instructions and let A be the set of addresses.

Definition 7.1. A *control-flow policy* is a function that given an instruction and an address returns a Boolean indicating whether the instruction is allowed to jump to the address at run-time, i.e., a function of type:

$$I \times A \mapsto \mathbb{B}$$

We use $\mathbb{P}$ to denote this type.

7.1.2 Control-flow Integrity

Control-flow Integrity (CFI) is about allowing or disallowing *control flow transfers* of a binary at runtime. CFI enforces this property by monitoring indirect control-flow transfer (ICT) instructions of the binary and comparing their transfer targets to a set of valid addresses derived from the CFG. ICT instructions can be either forward or backward. Forward ICT instructions transfer control to the target address (e.g., `call foo`). In contrast, backward ICT instructions return to the subsequent instruction of forward ICT instructions (e.g., returning from a function call through a `ret`).

The idealized version of CFI *precisely* knows which control flow transfers are allowed or disallowed. This version is called fully-precise CFI (FPCFI) [28]. FPCFI enforces its policy on both forward and backward ICT instructions. The FPCFI policy is defined by a function

$$\mathsf{FPCFI} :: \mathit{Oracle} \rightarrow \mathbb{P}$$

Here, the *Oracle* is some idealized mechanism that can perfectly decide which control flow transfers are allowed and which ones are not. The only way to obtain *Oracle* is by using innumerable binary executions to analyze all possible paths. This makes FPCFI is the most potent form of CFI, but it is infeasible to design.

Since FPCFI is impossible to enforce, various other kinds of CFI have been proposed. They differ in their *overhead*, their *practicality*, but mostly in how much *security* they enforce. Figure 7.1 depicts a high-level comparison among different variants of CFI and **CFR** policies. The loosest variant of CFI is called relaxed CFI (RCFI) [238, 240]. The RCFI policy is defined by a function

$$\mathsf{RCFI} :: \mathit{CFG} \rightarrow \mathbb{P}$$

Here *CFG* denotes a CFG. RCFI is a policy that decides which control-flow transfers are allowed based solely on the CFG. Hence, RCFI does not typically handle backward ICT instructions. This is because obtained target addresses from the CFG lack *contextual* information, distinguishing different paths that lead to a given backward ICT instruction. In other words, without any context, all backward ICT instructions are treated the same, which is unreliable.

Therefore, a stricter but less practical (because it requires additional features such as shadow stack [51]) variant of CFI, Context-Sensitive CFI (CSCFI), has been proposed [212]. In contrast to RCFI, CSCFI enforces forward and backward ICT instructions. The CSCFI policy is defined by a function

$$\mathsf{CSCFI} :: \mathbf{ShadowStack} \rightarrow \mathit{CFG} \rightarrow \mathbb{P}$$

Here **ShadowStack** denotes a stack of target addresses for backward ICT instructions. We want to emphasize that we bolded the type of **ShadowStack** to indicate that this is maintained at runtime, whereas the non-bold types mean something statically computed. In contrast to RCFI, CSCFI deploys a shadow stack to enforce a higher precision policy. The purpose of the shadow stack is to dynamically keep track of return addresses' *contexts* by storing return addresses before function calls happen. Here, the context is the function that the return address was called from. This means that CSCFI is a *stateful* policy because it requires a current state to enforce security policies.

Figure 7.1 depicts the comparison between various CFI policies. From this figure, it is clear that RCFI is relatively loose, whereas CSCFI has a relatively high overhead. Although CSCFI incurs a runtime overhead due to the deployment of a shadow stack, it is still considered the standard among CFI policies. This paper proposes a novel security policy called Control-flow Restriction (**CFR**), which enforces higher security than CSCFI but incurs a lower runtime overhead than FPCFI. At the same time, **CFR** does not require any source code or additional hardware such as Intel CET.

7.1.3 Control-flow Restriction

CFR is a policy that *restricts* transfer by blocking them if the target is from a certain *deny-list*. Restricting is done by either halting the binary's execution or prompting a warning to notify the user of potential danger. The $\textbf{CFR}_0$ policy is defined by a function

$$\textbf{CFR}_0 :: [A] \to \mathbb{P}$$

$\textbf{CFR}_0$ takes a set of *denied* addresses and then enforces a control-flow policy. For our purpose, denied addresses are all addresses of *gadgets*, short sequences of instructions that adversaries can use to construct control-hijacking attacks. However, a *DL* can include any potentially malicious address leading to dangerous functions (e.g., `system`) or a function that the user should not illegally call (e.g., `_fini` as the program should only reach this with `ret` from `main`). To find gadgets, we statically analyze a binary using the tool Ropper[1]. We will discuss the gadget finding process in Section 7.4.1.

$\textbf{CFR}_0$ is the most strict policy **CFR** can enforce. $\textbf{CFR}_0$ guarantees that control flow will only transfer to benign addresses by preventing any ICT instructions from transferring to any deny instructions. However, this policy is too strong. For instance, gadgets are not malicious individually, only when *chained* to perform arbitrary computations. Hence, if all gadget addresses are deny, $\textbf{CFR}_0$ may restrict benign execution, i.e., a *false negative*. A false negative occurs, e.g., if a benign address is deny. We will refer to such a falsely untrusted address as a *false gadget*.

Therefore, the $\textbf{CFR}_1$ policy is proposed to reduce false gadgets and increase our policy's precision. The $\textbf{CFR}_1$ policy is defined by a function

$$\textbf{CFR}_1 :: \textbf{RetAddrQueue} \to [A] \to \mathbb{P}$$

Here **RetAddrQueue** denotes a queue of target addresses for backward ICT instructions. The elaboration of this term is return address queue (RAQ), and it is comparable to a shadow stack used in the CSCFI policy. They are similar because $\textbf{CFR}_1$ enqueues a return address' context whenever a function is called. Then, when the backward ICT instruction executes, it dequeues the address and checks with the transfer target to determine the next step.

The most significant difference is that, unlike a shadow stack, RAQ does not need to create a separate stack. This is possible because we rewrite a binary to instrument the RAQ into the binary's region. Thus, RAQ bypasses the shadow stack deployment challenge [25] (mapping a stack in a binary to a shadow stack). This mapping process can be done either indirectly (to an on-demand growth stack) or directly (to a fixed-size stack). Each approach has its tradeoffs. For example, an on-demand growth stack requires additional data structure maintenance (more performance), while the fixed-size stack concerns choosing the size (more

[1] https://github.com/sashs/Ropper

memory). **CFR**$_1$ is the more precise policy than **CFR**$_0$ because it stores the context of return addresses but incurs more runtime overhead due to the deployment of the RAQ.

Lastly, the **CFR**$_2$ policy is proposed to incorporate metadata information on top of all previous policies to maximize precision. The **CFR**$_2$ policy is defined by a function

$$\textbf{CFR}_2 :: \textit{MetaInfo} \rightarrow \textbf{RetAddrQueue} \rightarrow [A] \rightarrow \mathbb{P}$$

Here *MetaInfo* denotes metadata information. The metadata contains the following information:

1. All function entry points in the binary
2. Set of allowed transfers for each indirect jump
3. Set of allowed transfers for each indirect call

Finding these addresses is considered an *undecidable* problem [123]. We can obtain this information without CFG by lowering the binary into the LLVM bitcode file [118] and applying multi-layer type analysis [142]. However, when writing this, we could not apply it to our work as one of them is a closed-source project. Therefore, we use a reverse engineering tool, Binary Ninja [2], to overapproximate this information. Binary Ninja incorporates recursive descent algorithm [173], function detection algorithm from `Nucleus` [8], and value-set analysis (VSA), a static analysis algorithm using abstract interpretation to interpret low-level code [87].

CFR$_2$ is the most precise policy **CFR** can enforce. For all forward and backward ICT instructions, we use the metadata information to determine their safety based on the type of ICT instruction. For instance, it would be incorrect to use a set of indirect jump site addresses to enforce an ICT instruction such as `call rax`. Therefore, **CFR**$_2$ applies the metadata information appropriately for each type of ICT instruction. **CFR**$_2$ does not incur too much additional runtime overhead than **CFR**$_1$, but instead, it increases the trusted computing base (TCB) of our policy. We *do not* need to trust all of Binary Ninja, only the necessary analyses for the metadata information. However, we do expect it to provide adequate metadata information for our whitelisting purposes.

7.2 Formal Definitions

7.2.1 CFR_0 Policy

For the following definitions, let *dl* be a deny list, i be an instruction, and $trgt$ be a target address of an instruction.

Definition 7.2. The control-flow policy $\textbf{CFR}_0$ is defined as:

$$\mathsf{CFR}_0(dl, i, trgt) \stackrel{\text{def}}{=} trgt \notin dl$$

That is, if and only if any target of instructions transfers to an address in a deny list, then such transfer is *restricted*.

7.2.2 CFR_1 Policy

We now introduce few more functions to help us define $\textbf{CFR}_1$ and $\textbf{CFR}_2$ policies.

$$\begin{aligned}
\mathsf{mnemonic} &:: \; I \rightarrow String \\
\mathsf{front} &:: \; \textbf{RetAddrQueue} \rightarrow A \\
\mathsf{entries} &:: \; MetaInfo \rightarrow [A] \\
\mathsf{ind_jmp_trgts} &:: \; MetaInfo \rightarrow A \rightarrow [A]
\end{aligned}$$

$\mathsf{mnemonic}(I)$ is a function that returns, given an instruction, the mnemonic of an instruction. $\mathsf{front}(\mathbf{q})$ is a function that returns, given a queue of addresses, the front element of the queue. $\mathsf{entries}(\mu)$ is a function that returns, given metadata information, the set of start addresses of all functions. $\mathsf{ind_jmp_trgts}(\mu, A)$ is a function that returns, given metadata information and an address, the set of indirect jump site addresses for that particular address.

Definition 7.3. Let $\mathbf{q}$ be a return address queue. The control-flow policy $\textbf{CFR}_1$ is defined as:

$$\text{CFR}_1(\mathbf{q}, bl, i, trgt) \stackrel{\text{def}}{=} \begin{cases} trgt = \mathsf{front}(\mathbf{q}) & \text{if } \mathsf{mnemonic}(i) = \text{``}\texttt{ret}\text{''} \\ \\ \mathsf{CFR}_0(bl, i, trgt) & \text{otherwise} \end{cases}$$

That is, if the instruction's mnemonic is a "`ret`", then we check its target with the front element of return address queue. If they match, then such transfer is *allowed*.

7.2.3 CFR$_2$ Policy

Definition 7.4. Let μ be metadata information and i.`addr` be the address of instruction i. The control-flow policy **CFR$_2$** is defined as:

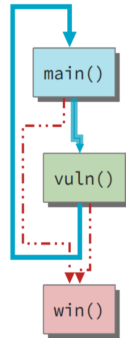
$$\text{CFR}_2(\mu, \mathbf{q}, bl, i, trgt) \stackrel{\text{def}}{=} \begin{cases} trgt \in \mathsf{entries}(\mu) & \text{if } \mathsf{mnemonic}(i) = \text{``}\mathtt{call}\text{''} \\ trgt \in \mathsf{ind_jmp_trgts}(\mu, i.\mathtt{addr}) & \text{if } \mathsf{mnemonic}(i) = \text{``}\mathtt{jmp}\text{''} \\ \text{CFR}_1(\mathbf{q}, bl, i, trgt) & \text{otherwise} \end{cases}$$

That is, if the instruction's mnemonic is a "`call`," then we check its target with the set of start addresses of all functions, and if any of them matches, then such transfer is *allowed*. Else, if the instruction's mnemonic is a "`jmp`," we check the set of indirect jump site addresses for current instruction, and if any of them matches, then such transfer is *allowed*.

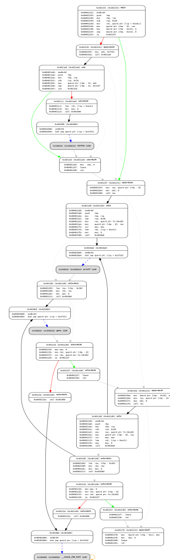
7.3 Motivating Example

```
1   typedef int (*fun_ptr)(int);
2   int win(int input) {
3     if (input == 0x1337)
4     {system("/bin/sh");}
5     return 0; }
6   int vuln(){
7     char buf[32];
8     printf("Input : ");
9     gets(buf);
10    return 0; }
11  fun_ptr funs[] = {vuln, win};
12  int main(){
13    fun_ptr fun = funs[0];
14    int unused = 0;
15    if (unused)
16    {win(0x7331);}
17    int call_0 = fun(0);
18    int call_1 = fun(0);
19    return 0;
20  }
```

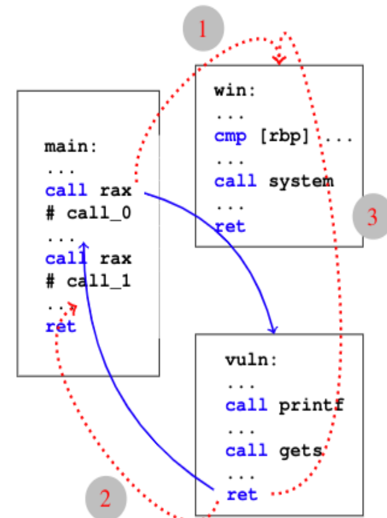
(a)



(b)



(c)



(d)

Figure 7.2: Motivating example: (a) code listing that is vulnerable to control-flow hijacking attack; (b) call graph; (c) extracted CFG of code listing from `angr` [201]; (d) intended/non-intended execution paths

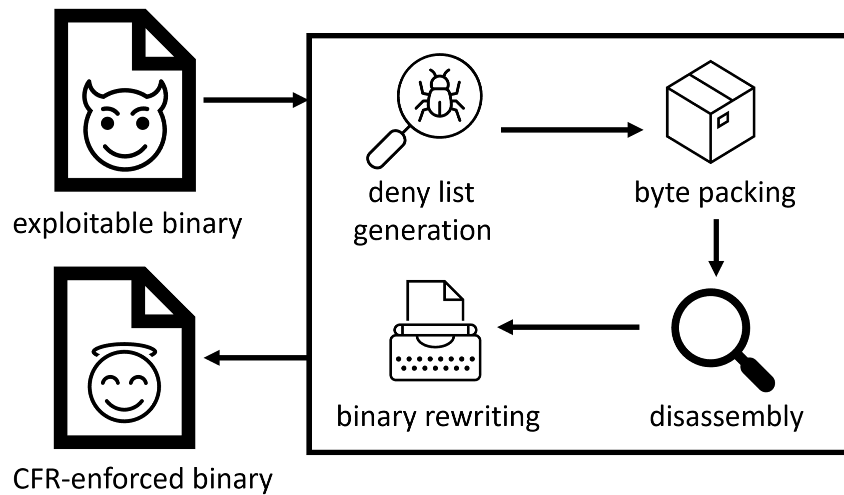


Figure 7.3: Binary hardened through **CFR**

7.3.1 Attack Vectors

Figure 7.2a shows a code listing that is vulnerable to a control-flow hijacking attack. We can see from this figure that the function `vuln` contains a function `gets` that is potentially vulnerable to buffer overflow at line 9 as it does not bound check the input. Therefore, the adversary could craft inputs to overflow the buffer and hijack the control flow to non-intended execution paths. Figures 7.2b and 7.2c depict this example's call graph and CFG, respectively.

7.3.2 CFR and CFI Comparisons

Figure 7.2d illustrates the possible execution paths of this example. Using this, we demonstrate the potential weakness of existing CFI policies. Starting with the conceptually most potent form of CFI, FPCFI can prevent non-intended transfers of ①, ②, and ③. Next, RCFI, a "loose" form of CFI that typically only enforces forward ICT instructions, can only prevent a non-intended transfer of ①.

Figure 7.3 illustrates the steps to harden a legacy binary into the **CFR**-enforced binary. **CFR** takes an ELF binary as input and scans the static binary for a list of unintended jump targets (generate a deny list). Next, **CFR** packs the addresses in the deny list into a bit array so that any prohibited address search can be $O(1)$. Afterward, we statically disassemble the binary to gather the metadata information. Lastly, we perform binary rewriting to apply **CFR** policy on all ICT instructions. When **CFR** enforced binary executes, it will check whether the ICT will transfer to denied addresses and handle cases as necessary.

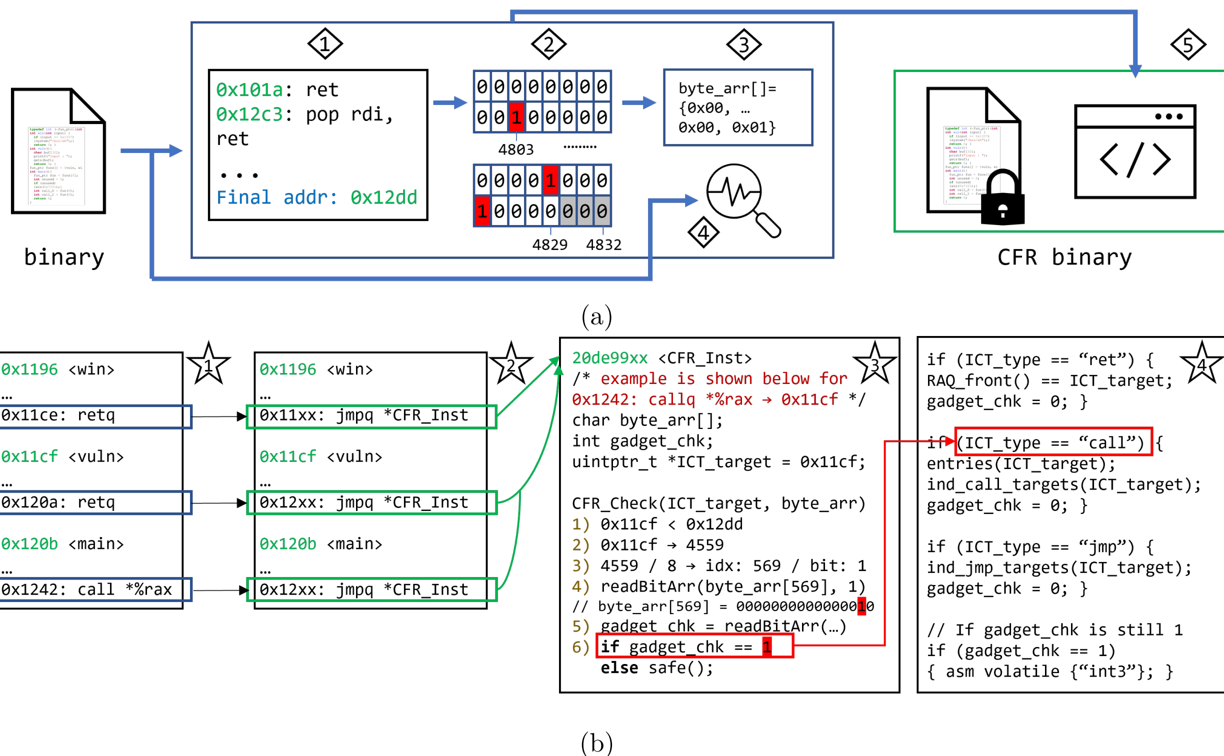


Figure 7.4: Design overview: (a) deny list generation; (b) binary rewriting and **CFR** enforcement during runtime

7.4 CFR Design Overview

7.4.1 Deny List Generation

To enable **CFR**, we first need to prepare a deny address list. Figure 7.4a presents the deny list generation in detail. First, we find all gadgets, sequence(s) of instructions that end in ICT instruction (e.g., `0x12c3: pop rdi, ret`), available in a binary ⟨1⟩. We consider gadgets to be the candidates for the deny list as they are vital to launching ROP attacks. We find gadgets using open-source tools such as Ropper [187]. The detailed step of finding gadgets is as follows: 1) Search for byte sequences ending with instructions such as `ret`, `call`, or `jmp`, depending on gadget types, and 2) Propagate backward from the ending instruction to check for validity of the previous instruction, then repeat until reaching the largest number of bytes that makes a valid instruction.

7.4.2 Byte Packing

Next, **CFR** needs to instrument a target check on each ICT. Searching through a long list of deny addresses would incur excessive overhead even when an efficient searching algorithm such as the Binary Search is used (average time complexity of $O(\log n)$). To accelerate the address searching process, we propose a *byte packing* technique. The primary goal of byte packing is to pack all deny addresses into a byte array to enable the average search time of $O(1)$ when the **CFR** needs to perform the check.

To achieve this, we first iterate each address of the deny list (the gadget's location). Next, we create a bit array and this address value gets converted into an integer (`0x12dd` $\rightarrow$ 4829), and a bit array is created based on this value. Afterward, we pad the bit array to be divisible by eight (e.g., 4829 $\rightarrow$ 4832) and set the bits for all dangerous locations, as illustrated in Figure 7.4a (⟨2⟩). For example, we first convert the deny address of `0x12c3` to an integer 4803, then set `bit_arr[4803]` $= 1$ (seen at location 4803 ⟨2⟩). The bit array is transformed into a bit string and converted into a byte array ⟨3⟩.

7.4.3 Disassembly

There may be a chance that a gadget happens to be a legal control flow target (e.g, a function address). To prevent a false positive, we exclude the "false gadgets" using Binary Ninja (⟨4⟩ in Figure 7.4a). Many open-source and commercial binary disassemblers are available. We chose Binary Ninja for its ease of use and solid performance for the required analyses among surveyed disassemblers [167]. As the detailed information regarding disassembly is outside of scope, we encourage readers to refer to the SoK of disassemblers [167] for more details.

7.4.4 Binary Rewriting

After completing necessary offline preparations, we instrument everything using `e9patch` [62]. `e9patch` rewrites a binary to redirect target instructions to a trampoline. A trampoline is a helper function that a program could jump to and return from. We chose `e9patch` because it offers high programmability and does not modify other instructions' semantics during the rewriting process (i.e., high compatibility). `e9patch` appends the **CFR** instrumentation to the end of the original binary ⟨5⟩. Its instrumentation code resides in separate memory pages; hence if the adversary attempts to access the **CFR** code illegally, it will result in a *segmentation fault*.

Figure 7.4b depicts the above-mentioned binary rewriting process and **CFR**-enforcement at runtime. `e9patch` patches all highlighted ICT instructions with a jump to trampoline functions ☆1. These will then jump to the **CFR** instrumentation (denoted green arrow

2). Afterward, when the **CFR**-enforced binary executes, it will check for the deny address using the byte packing technique 3. Here are each of the steps in detail: 1) **CFR** checks whether the ICT's target address is less than the highest deny address; If the former is greater, this implies the target address is safe (outside of deny list range), 2-3) ICT's target address is then converted into a row index and a bit location, 4) the row index and bit location are used for the `readBitArr` function to perform a one-time read of the byte array (time complexity of $O(1)$) and this will return a boolean value, 5) we store this value into `gadget_chk` variable, and 6) if `gadget_chk` returns `true`, **CFR** found a potentially dangerous transfer. This procedure is shown in 4,where it checks for the type of ICT instruction first and then checks with metadata to determine if the execution is safe.

Besides instrumenting the indirect `call`/`jmp` instructions, we also implemented a simple shadow stack. Specifically, we allocated a shadow stack on **CFR** reserved memory pages. Then we push the return address to the shadow stack on each `call` instruction and replace the `call` instruction with a `jmp` instruction to the **CFR** instrumented code. We similarly replace the `retq` instruction with a `jmp` instruction to our **CFR** instrumentation (shown in Figure 7.4b)

Chapter 8

CFR Evaluation

In this chapter, we evaluate **CFR**'s capabilities with respect to runtime overhead and its effectiveness against real-life ROP attacks. Specifically, the aim of this chapter is to understand:

1. Due to the nature of inserting trampolines via a binary rewriting method, **CFR**'s overhead could be substantial. Therefore, how expensive are these **CFR** trampolines, and is the overhead feasible?

2. The goal of **CFR** is to enforce CFI from an attacker's perspective. Hence, it is essential to perform real-life ROP attacks to test the security benefits of **CFR**.

Section 8.1 describes the setup environment of our evaluation. Section 8.2 presents an evaluation of runtime overhead using **CFR**. Next, in Section 8.3, we provide a security evaluation of **CFR** using the challenges from ROP Emporium. Lastly, Section 8.4 presents a summary of our **CFR** work.

8.1 Setup Environment

We evaluate all experiments on an Ubuntu 20.04 LTS machine with an Intel i7-4790K CPU (4.0 GHz) and 16 GB RAM. We do not manually compile COTS applications but directly use the installed version from the default download package manager from Ubuntu. This is to demonstrate the applicability of **CFR**. For the SPEC2006 benchmark, we compile them using the `GCC`-9.3 compiler with the optimization level `-O0` for compatibility purposes. For the SPEC2017 benchmark, we compile them using the GCC-9.3 compiler with the default optimization level in the configuration file. We turned off debug outputs for all tested benchmark applications to prioritize obtaining the performance data.

We chose popular COTS applications such as `bzip2, xterm`, and `gimp` as our benchmark. To ensure that functionality remains intact after rewriting, we designated test behaviors appropriate for each application to test each COTS application. We denote many applications' runtime overhead `N/A` because it is not applicable. Lastly, we applied different precision levels of **CFR** policy on both SPEC2006/2017 benchmarks to assess policies' runtime overhead.

8.2 Runtime Overhead Evaluation

Table 8.1: SPEC2006/2017 benchmarks evaluations

Binary	Original	e9patch NOP		CFR_0		CFR_1		CFR_2	
	Time(s)	Time(s)	% Diff	Time(s)	% Diff	Time(s)	% Diff	Time(s)	% Diff
400.perlbench	318	619	94.7%	619	94.7%	756	138%	844	165%
401.bzip2	286	323	13%	324	13.3%	339	18.5%	359	25.5%
403.gcc	186	267	43.5%	266	43%	315	69.4%	357	91.9%
429.mcf	189	192	1.59%	190	0.53%	197	4.23%	207	9.52%
445.gobmk	317	418	31.9%	421	32.8%	492	55.2%	524	65.3%
456.hmmer	291	297	2.06%	297	2.06%	300	3.09%	317	8.93%
458.sjeng	342	539	57.6%	571	67%	624	82.5%	700	105%
462.libquantum	287	300	4.53%	299	4.18%	301	4.88%	323	12.5%
464.h264ref	340	824	142%	835	146%	1012	198%	1134	234%
471.omnetpp	267	388	45.3%	394	47.6%	458	71.5%	511	91.4%
473.astar	259	291	12.4%	314	21.2%	303	17%	322	24.3%
433.milc	528	552	4.55%	555	5.11%	572	8.33%	593	12.3%
444.namd	545	595	9.17%	586	7.52%	606	11.2%	616	13%
453.povray	264	714	170%	727	175%	1011	283%	1102	317%
470.lbm	323	323	0%	323	0%	323	0%	335	3.71%
482.sphinx	884	1021	15.5%	1010	14.3%	1020	15.4%	1034	17%
Average %		40.8%		42.9%		60.2%		74.8%	
500.perlbench_r	333	757	127%	741	123%	939	182%	1043	213%
502.gcc_r	324	368	13.6%	369	13.9%	454	40.1%	492	51.9%
505.mcf_r	306	641	109%	613	100%	659	115%	701	129%
520.omnetpp_r	573	782	36.5%	769	34.2%	929	62.1%	983	71.6%
523.xalancbmk_r	285	526	84.6%	529	85.6%	628	120%	634	122%
525.x264_r	452	435	-3.76%	435	-3.76%	458	1.33%	476	5.31%
531.deepsjeng_r	258	354	37.2%	352	36.4%	412	59.7%	437	69.4%
541.leela_r	393	575	46.3%	574	46.1%	705	79.4%	774	96.9%
508.namd_r	197	203	3.05%	204	0.51%	204	3.55%	201	2.03%
511.povray_r	330	1237	275%	1247	278%	1597	383%	1742	428%
519.lbm_r	225	230	2.22%	232	2.22%	232	3.11%	233	3.56%
538.imagick_r	360	408	13.3%	408	13.3%	445	23.6%	470	30.6%
544.nab_r	314	313	-0.32%	313	-0.32%	314	0%	316	0.64%
554.roms_r	321	318	-0.93%	316	-1.56%	319	-0.62%	327	1.87%
Average %		53.1%		52%		76.7%		97%	

Table 8.1 presents the runtime overhead evaluation of **CFR** on SPEC2006/2017 benchmarks. The runtime performance is in seconds, and % diff represents the percent difference between the original binary and **CFR** execution time. `e9patch NOP` column represents `NOP` code instrumentation, which serves as the basis overhead of using `e9patch`. Afterward, each column represents its respective **CFR** policy ranging from **CFR**$_0$ to **CFR**$_2$.

Table 8.2: Commercial Off-The-Shelf (COTS) applications evaluations

Binary	**Original**	CFR_0		CFR_1		CFR_2		**BinCFI**		
	Time(s)	Time(s)	% Diff	Time(s)	% Diff	Time(s)	% Diff	Original Time(s)	Patched Time(s)	% Diff
sort	5.93	7.01	17.6%	7.22	21.2%	7.32	22.7%	33.3	51.8	55.4%
wc	0.34	0.38	13.1%	0.4	18.8%	0.42	25%	0.96	2.55	165%
cp	5.39	5.4	0.18%	5.41	0.41%	5.51	2.13%	4.35	4.72	8.56%
bzip2	6.73	6.76	0.55%	6.77	0.7%	6.81	1.32%	8.51	8.92	4.82%
grep	1.01	1.1	13.2%	1.12	21.8%	1.18	28.3%	1.24	3.01	142%
ssh	0.172	0.17	1.04%	0.18	3.71%	0.19	10.4%	0.02	0.03	69%
git	0.76	0.77	0.92%	0.77	2.11%	0.81	6.73%	N/A	N/A	N/A
ls	0.08	0.08	1.12%	0.1	19.4%	0.10	26%	0.14	0.19	34%
shuf	3.67	3.73	1.64%	3.8	3.54%	3.9	6.10%	2.21	2.49	12.9%
Average %		5.49%		10.2%		14.3%		61.5%		

As a side note, as BinCFI applications are based on a 32-bit processor machine, we obtained the runtime information of BinCFI applications *within* its provided virtual machine[4] because we could not execute patched applications on a 64-bit processor machine.

Table 8.3: Commercial Off-The-Shelf (COTS) applications' test behaviors and false gadget counts

Binary	Test Behavior	False Gadget(s)		
		CFR_0	CFR_1	CFR_2
sort	`sort` 10 million lines	1	0	0
wc	`wc` 10 million lines	0	0	0
cp	`cp -rf` directory	3	0	0
bzip2	`bzip2` 1G file	1	0	0
grep	`grep -Hnri "hello"` 10 million lines	0	0	0
ssh	`ssh localhost 'sleep 2>\dev\null &'`	14	6	0
git	`trickle -sd 700 git clone` repo (30 objects)	48	0	0
ls	`ls -color=tty -1a` directory with 30,000 files	0	0	0
shuf	`shuf` 10 million lines	0	0	0
evince	open 100 `MB` PDF file	3	0	0
xterm	basic operations	13	2	0
gimp	basic operations	286	8	5

Table 8.2 presents the runtime overhead evaluation of **CFR** and BinCFI on the various COTS applications. Consider, for example, the data for the `git` application in Table 8.3. The test behavior for this application is `trickle -sd 700 git_CFR2 clone` repository. We use the `trickle` tool only to limit the download speed for proper evaluation. Lastly, we present the number of false gadgets detected by different policies. As previously defined, false gadgets are the gadget addresses that **CFR** falsely flags as spurious that could unintentionally restrict the execution.

[4] http://www.seclab.cs.sunysb.edu/seclab/psi/

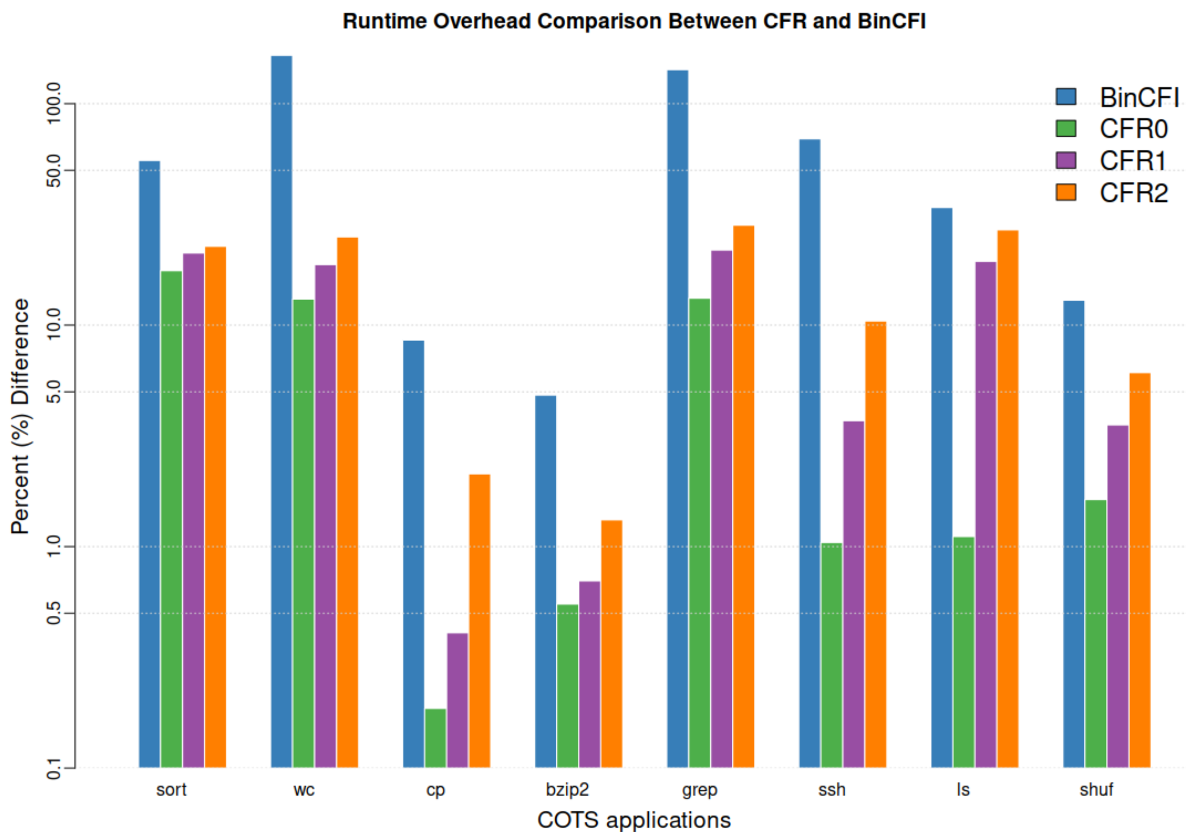


Figure 8.1: COTS applications evaluation. The percent difference formula used in this Figure is ((`modified - original`) / `original`) $\times$ 100

Figure 8.1 presents the runtime overhead comparison between our work and BinCFI [240] for COTS applications on a logarithmic scale. BinCFI is a reasonable comparison because BinCFI is a hardware-agnostic binary-level CFI policy. Moreover, BinCFI's goal is also to be scalable from small to large COTS applications. For the comparison, we applied the same behavior in the applications listed in Table 8.2. To summarize, COTS applications evaluations show that **CFR** presents an average of 5.49%/10.2%/14.3% runtime performance overhead for **CFR**$_0$/**CFR**$_1$/**CFR**$_2$ policies, respectively. In addition, **CFR**$_1$ and **CFR**$_2$ are context-sensitive control-flow security policies, whereas BinCFI is a coarse-grained CFI solution.

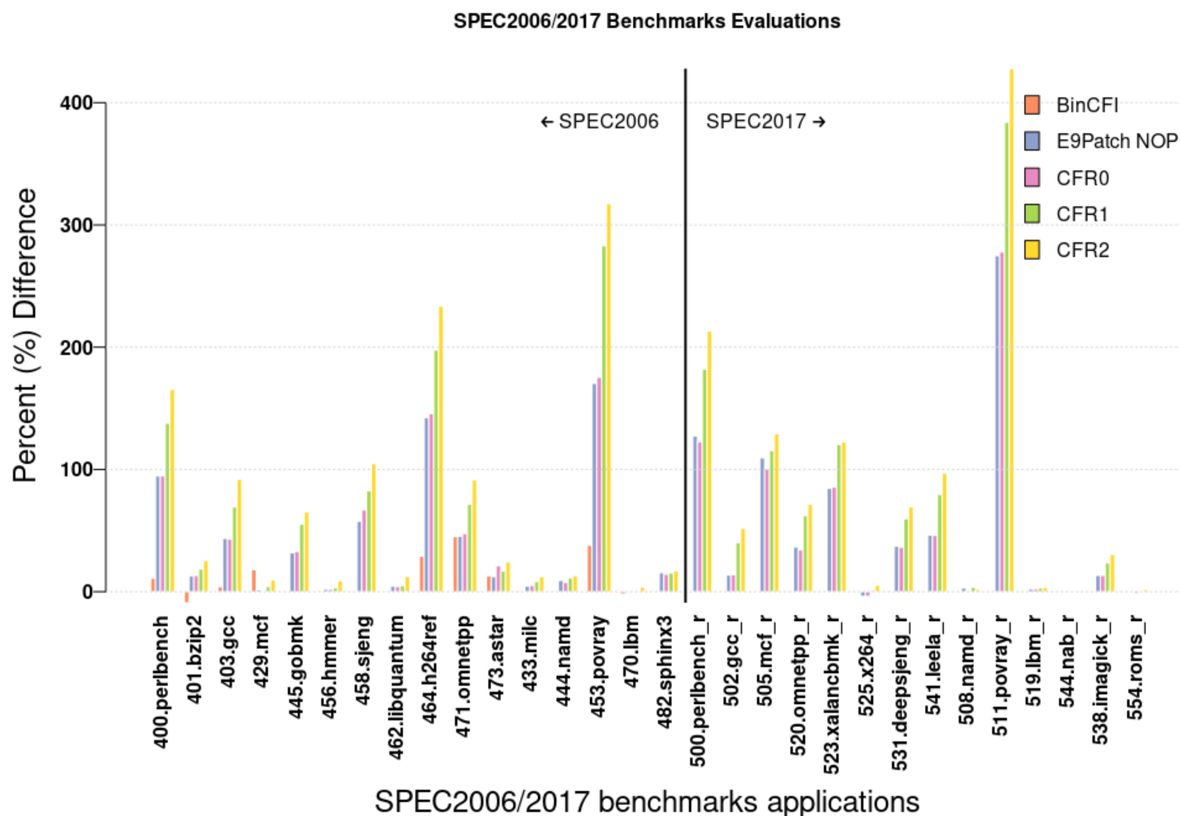


Figure 8.2: SPEC2006/2017 benchmarks evaluation. BinCFI results are from the Figure provided in [240], and **CFR** results are from our machine. BinCFI does not apply to the SPEC2017 benchmark. The percent difference formula used in this Figure is ((`modified` - `original`) / `original`) $\times$ 100

Figure 8.2 depicts runtime overhead for SPEC2006/2017 benchmarks for both BinCFI and **CFR**. Please note that BinCFI did not apply for the SPEC2017 benchmark, which means their data is unavailable. As the level of precision increases, runtime overhead correspondingly increases. As previously mentioned, runtime overhead primarily stems from the nature of trampoline insertions. Because `e9patch` does not require CFG, all patched ICT instructions first jump to the instrumentation and then transfer back to the original code.

This is different from other binary rewriting solutions [13, 16] that *inline* its instrumentation code, removing the need for additional `jmp` instructions. Consequently, for a benchmark that repeatedly uses the same multiple functions to render thousands of pixels (e.g., `povray`), due to its repeated jumping to and return from trampoline functions, overhead is high. However, it is imperative to note that average runtime overheads for all **CFR** policies across all SPEC 2006/2017 benchmarks are reasonable considering that **CFR** is a minimalistic solution.

8.3 Security Evaluation

8.3.1 Policy Evaluation

Evaluating the protection technique's strength is a complex problem. This is because it is not possible to guarantee security despite making many assumptions in a threat model. We can evaluate CFI policies both qualitatively and quantitatively. First, CFI can be qualitatively measured by checking whether it supports forward, backward, or all types of ICT instructions.

Second, there are two metrics used to measure the CFI's effectiveness quantitatively: 1) Average Indirect target Reduction (AIR) and 2) Equivalence Class (EC). AIR [240] is a known metric among CFI works to measure the CFI's strength quantitatively. To obtain AIR, one first calculates the number of eliminated targets per each ICT instruction. Next, it finds the average of this number across all ICT instructions. Unfortunately, as the number of eliminated targets does not necessarily correlate to adversaries' attack arsenals [24], the AIR metric leaves room for improvement. CFI-LB work [112] proposed a metric called EC, the number of targets that CFI cannot distinguish. In addition, they also presented a formula, $QS_{CFI} = AVG_{EC} \times LC$, or in other words, multiply the average size of all the ECs with the largest allowed targets among all ECs. The larger the value of QS_{CFI}, it would mean that the protection technique would be less secure.

Qualitatively analyzing, **CFR** can handle all ICT instructions. However, the quantitative analysis used for CFI does not suit **CFR** very well because they are similar to the false gadgets counts in our case. Hence, we perform an adversarial analysis to evaluate **CFR**'s effectiveness. In [24], the authors state the ideal standard to test the security is performing adversarial analysis. Adversarial analysis can assume an intelligent attacker that could exploit the target. But, adversarial analysis is hard to deploy in practice because it requires a human to perform analysis on a per-program basis. Furthermore, it is difficult setting the standard to reproduce the same adversarial analysis for different protection techniques.

Table 8.4: Security evaluation on ROP Emporium challenges

	`ret2win`	`callme`	`badchars`	`pivot`	`split`	`write4`	`fluff`	`ret2csu`
Attacked	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Protected by **CFR**	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Patched

Thus, we use a known set of ROP challenges called ROP Emporium as an input for our adversarial analysis. ROP Emporium is a perfect set of challenges for **CFR** because its primary goal is to launch ROP attacks. By applying **CFR** to those challenges, **CFR** needs to achieve two feats:

1. Do not allow the illegal transfer of indirect control transfer instructions.

2. Ensure that the patched application properly executes.

Table 8.4 presents our **CFR** policy evaluation on ROP Emporium challenges. To summarize, we were able to defend against all attacks except for one challenge, `ret2csu`, which we successfully defended by manually patching. We further discuss this challenge in Section 8.3.3.

Threat model We assume $W \oplus X$ protections to prevent the insertion of a malicious code or modification to a binary. In addition, the adversary can arbitrarily modify memory through a memory safety error. Therefore, the adversary's goal is to exploit memory errors to construct a control-flow hijacking attack. Lastly, we do not assume any recent hardware technologies such as Intel CET [102] or Intel PT [177].

The authors in [80] examine how the length of gadget chains can affect the severity of Return-Oriented Programming (ROP) attacks. Longer chains can perform more complex operations, making the attacks more powerful. However, the increased size of the chains also raises the execution time and the difficulty of successfully launching the attack, which makes detection easier. Therefore, if the attack is modified to make detection more difficult (e.g., by using shorter chains), the performance overhead of detection techniques would increase because it would take longer to detect the attack. However, **CFR** is an enforcement mitigation technique, and thus the severity of the attack does not impact the performance overhead.

8.3.2 False Gadgets

Table 8.3 depicts the false gadgets counts for different precision levels of **CFR** policy on COTS applications.

False negatives For COTS applications listed in Table 8.3, the **CFR**$_1$ policy reduces the majority of false gadget counts, and the **CFR**$_2$ policy eliminates the majority of remaining false gadgets. For `gimp` application, however, there remain five false gadgets that are not eliminated even with the **CFR**$_2$ policy. We will now discuss these false gadgets.

All five false gadgets result from indirect `ret` instructions returning to the address in a deny list (i.e., gadgets). **CFR**$_1$ policy correctly considers these addresses benign at first. However, they are flagged as false gadgets due to the limitation of instruction punning technique [32, 62] not providing 100% patching coverage. To expand on this point, this is because, for the **CFR**$_1$ policy, a `call` instruction saves the context for `ret` instruction. But, if it is impossible to patch all `call` instructions, then many `ret` instructions' context may not be stored appropriately. To remedy this limitation, if needed, **CFR** can temporarily whitelist these addresses benign to adapt based on a situation flexibly.

`endbr` Gadgets During our evaluation, we encountered many false gadgets with the `endbr` instruction. These are `ENDBRANCH` instructions from Intel CET [102, 138], and they mark

valid jump target addresses of indirect calls/jumps in a binary. When `endbr` instruction is unused, they have the same effect as the `NOP` instruction. Therefore, it does not change the register state and can only be used for memory alignment at worst. Consequently, we consider targets leading to `endbr` instruction as false gadgets. These gadgets are thus whitelisted in **CFR**$_2$.

8.3.3 Adversarial Analysis

We perform the adversarial analysis to evaluate the security of **CFR** policy. Unlike many CFI works that provide proof of concept, we perform actual ROP attacks on **CFR**. **CFR** is one of few code-reuse attack protection schemes that conduct adversarial analysis for security evaluation, to the best of our knowledge.

As we have previously shown, **CFR** can defend against ROP attacks on all ROP Emporium challenges except for the `ret2csu` challenge. ROP Emporium's goal is to read the `flag.txt`, which contains the text of `ROPE{a_placeholder_32byte_flag!}`, by using ROP attacks. Figure 8.3 shows how we enforce the **CFR** policy on ROP Emporium challenges during runtime.

Each line of the terminal output provides the following information: $\texttt{addr}_{\texttt{source}}$: `inst` $\rightarrow$ $\texttt{addr}_{\texttt{target}}$ = 0 (safe) / 1 (danger). In other words, **CFR** displays the source address, ICT instruction, and the target address of the ICT instruction. Then, the **CFR** result on whether the transfer target is malicious or not.

Figure 8.3a depicts when **CFR** defends the ROP attack. When **CFR** detects the attack (line 26), it outputs the transfer target of the ICT instruction (line 27). Then, it raises the user interrupt (lines 29-30). In contrast, Figure 8.3b depicts an example of where **CFR** fails to defend. First, transfer to the gadget address of `0x89a` is not caught (line 11). As a result, the content of `flag.txt` is successfully read (line 14).

`ret2csu` So why does **CFR** fail to protect `ret2csu` challenge? `ret2csu` [146] is a sub-technique of ROP first presented in Black Hat Asia 2018. The premise of this technique is to make up for missing gadgets using a function called `__libc_csu_init()`. For example, adversaries may not have gadgets that they can use to control the `rdx` register using Ropper.

This is where `ret2csu` can be used to find this needed gadget in the `__libc_csu_init()` function. `ret2csu` technique is possible because `__libc_csu_init()` is an essential function for every binary. After all, it prepares variables and binary functions to be used for the execution. Thus, initially, **CFR** fails to protect the `ret2csu` challenge because it does not consider gadget addresses outside Ropper results. Similar to indirect `jmp` instructions, this is also the imprecision caused by using external tools. Like before, this is patchable by adding `ret2csu` gadget addresses. Figure 8.3c shows the result of the successfully patched `ret2csu` challenge, where **CFR** successfully catches the transfer to the gadget address of `0x89a`.

```
1  $ python solution.py |
2  ./badchars_CFR2
3  ...
4  0x839: retq →  0xb9d = 0
5  0xbb4: retq →  0x7faca469d040 = 0
6  0x8a3: callq 0x400760 →  0x760 = 0
7  0x8c1: callq 0x400760 →  0x760 = 0
8  0x8cb: callq 0x4006e0 →  0x6e0 = 0
9  badchars by ROP Emporium
10 0x8d5: callq 0x4006e0 →  0x6e0 = 0
11 64bits
12
13 0x8df: callq 0x4008f5 →  0x8f5 = 0
14 0x90a: callq 0x400750 →  0x750 = 0
15 0x92d: callq 0x400710 →  0x710 = 0
16 0x953: callq 0x400710 →  0x710 = 0
17 0x95d: callq 0x4006e0 →  0x6e0 = 0
18 badchars are: b i c / <space> f n s
19 0x96c: callq 0x400700 →  0x700 = 0
20 > 0x984: callq 0x400730 →  0x730 = 0
21 0x999: callq 0x4009f0 →  0x9f0 = 0
22 0x9b0: callq 0x400a40 →  0xa40 = 0
23 0xadd: retq →  0x9b5 = 0
24 0x9cb: callq 0x400740 →  0x740 = 0
25 0x9d7: callq 0x4006d0 →  0x6d0 = 0
26 0xb3f: retq →  0xb34 = 1
27 Mal. action detected at 0x400b3f: retq
28 Transfer target: 0xb34
29 [1]    17454 done     python solution.py
30 | 17455 trace trap (core dumped)
31 ./badchars_CFR2
```

(a) badchars successfully defended

```
1  $ python solution.py
2  | ./ret2csu_CFR2
3  ...
4  ret2csu by ROP Emporium
5  ...
6  Call ret2win()
7  0x741: callq 0x400590 →  0x590 = 0
8  The third argument (rdx) must be
9  0xdeadcafebabebeef
10 ...
11 0x7b0: retq → 0x89a = 0
12 0x8a4: retq → 0x7b1 = 0
13 ...
14 ROPE{a_placeholder_32byte_flag!}
15 0x830: retq →  0xfffffffffc0000a = 0
16 [2]    17806 done     python solution.py
17 | 17807 segmentation fault (core dumped)
18 ./ret2csu_CFR2
```

(b) ret2csu unsuccessfully defended

```
1  $ python solution.py
2  | ./ret2csu_CFR2
3  ...
4  ret2csu by ROP Emporium
5  ...
6  Call ret2win()
7  0x741: callq 0x400590 →  0x590 = 0
8  The third argument (rdx) must be
9  0xdeadcafebabebeef
10 ...
11 0x7b0: retq →  0x89a = 1
12 Mal. action detected at 0x4007b0: retq
13 Transfer target: 0x89a
14 [2]    17825 done     python solution.py
15 | 17826 trace trap (core dumped)
16 ./ret2csu_CFR2
```

(c) ret2csu successfully patched

Figure 8.3: ROP Emporium adversarial analysis

8.4 Summary

As a second contribution of this dissertation, we presented a binary-level control-flow deny list policy called Control-flow Restriction (CFR). Our focus is to *restrict* indirect control transfer (ICT) instructions to transfer only to legitimate targets. Unlike prior works, we first assume all ICT instructions will transfer to a malicious address. Next, we search a deny list to check whether the control flow transfer is allowed. Then if necessary, we leverage the metadata information from binary-level static analysis to increase the precision of CFR to reduce false negatives (i.e., CFR falsely flags an ICT instruction to be spurious). Ultimately, we rewrite a binary with CFR policy included. To date, CFR is the first hardware-agnostic binary-level control-flow deny list policy to precisely prevent control-flow hijacking attacks.

We demonstrate the applicability of CFR by applying it to a wide variety of commercial off-the-shelf (COTS) applications. Furthermore, we perform an adversarial analysis to protect against actual ROP attacks on ROP Emporium challenges. CFR's main limitations are the potential imprecision from external tools and its runtime performance. However, we argue that the CFR's imprecision is not as severe as CFI's imprecision. CFR provides the flexibility to deny list any address as needed, something that CFI cannot easily do. Furthermore, as hardware solutions are not infallible, this underscores the need to explore binary-level solutions continuously.

Chapter 9

Shroud: Automatic Tagged Memory Compartmentalization

Binary-only solutions provide greater flexibility, practicality, and applicability compared to source-code level techniques. However, the main disadvantage of any binary-level approach is typically the excessive performance overhead resulting from the need to leverage trampoline functions, which require the technique to branch into and return from them. In recent years, commercial chip vendors have begun incorporating new security extensions into ISAs. Although many of these extensions were not originally intended to be used as security primitives, they offer researchers new opportunities to explore their capability to delegate security tasks to hardware, potentially alleviating the excessive performance overhead issue of binary-only solutions.

ARM has recently introduced a tagged memory architecture extension called Memory Tagging Extension (MTE). MTE assigns a tag to a memory region and an associated pointer to this region to detect memory overflows and bugs. Although it is a promising technology for memory sanitization, it cannot be reliably deployed as a security safeguard due to various issues. Despite these limitations, MTE represents a significant step forward in hardware-assisted security, offering a new avenue for researchers to develop more efficient security solutions that mitigate the drawbacks of traditional binary-only approaches.

In this contribution, we leverage a hardware security primitive to explore a different type of security technique related to compartmentalization. To that end, we introduce **Shroud**, a mechanism to leverage ARM MTE to compartmentalize sensitive data and safeguard the MTE for sensitive data protection. After compartmentalizing sensitive data using the MTE mechanism, **Shroud** encrypts the plain-text MTE tags in the associated pointers and only decrypts the pointer for sensitive data dereferencing.

Section 9.1 presents the motivating example of Shroud. We discuss the design overview of Shroud in Section 9.2.

9.1 Motivating Example

Listing 1 The vulnerable operations in `min-dop`[1]

```
typedef struct { int v_1, v_2; } glob_srv_struct;
char sbuf[1024] = {0};
int vuln_gv; glob_srv_struct p_srv;
g_var_t SECRET = 0x1337;
...
// read / copy client data to vuln buffer
int readInData(int clientfd, char *buf)
char buffer[8] = {0}; {
    ...
    recv_len = recv(clientfd,buffer,8,0);
    memcpy(buf, buffer, recv_len);
    return 0;
}
// check for invalid types and send error code
int checkForInvalidTypes(int type, int clientfd){
    char buffer[10];
    ...
    if (type <= 2) { ... // integer underflow
        send(clientfd, buffer, 9, 0); }
}
void do_serve(int sockfd) {
    int *p_size = 0; int *p_type = 0;
    // DOP gadgets loop dispatcher
    while (vuln_gv = ... ) {
        ...
        // mem. write violation; invokes DOP gadgets
        readInData(vuln_gv, sbuf); {
            // mem. read violation; uses int underflow
            checkForInvalidTypes(*p_type, g_clfd);
            // DOP gadgets
            if (*p_type == TYPE_ADD) { // DOP: condition
                p_srv->v_1 += *p_size; } // DOP: addition
            ...
            else {
                p_srv->v_2 = *p_size; } // DOP: assignment
        }
    }
}
```

As there have been some concerns about the MTE's viability as a security mechanism [61, 160, 228], we tested the security effectiveness of the MTE as an inspiring example of SHROUD using a minimalistic example of CVEs exploited by different non-control-data attacks: Data-

Oriented Programing (CVE 2006-5815)[1] and Heartbleed (CVE 2014-0160)[2].

Data-Oriented Programming (DOP) is an advanced non-control-data attack technique that expresses turing-complete computation by chaining *DOP gadgets* [98]. DOP gadgets are short instruction sequences that follow the format of *load* → *semantics* → *store* to simulate logical micro-operations (e.g., arithmetic, assignment, conditionals, etc.). With a successful DOP attack, the adversary can read/write memory arbitrarily.

Listing 1 shows the abridged snippet of a minimal DOP example (`min-dop`). In this listing, a *gadget dispatcher* exists in the form of a loop with a vulnerable global loop conditional variable `vuln_gv` (line 24). This variable will allow the adversary to execute the loop an arbitrary number of times. Next, inside the loop dispatcher, there exist functions that are vulnerable to out-of-bounds write/read (`readInData` and `checkForInvalidTypes`, respectively). Lastly, multiple DOP gadgets mimic a variety of different computations, such as condition checking (line 31), addition operation (line 32), and value assignment (line 35). Given that the adversaries can execute the loop dispatcher an arbitrary number of times, they can repeatedly exploit the memory vulnerability in the vulnerable function to execute DOP gadgets (i.e., stitch) until they can simulate the malicious computation.

Listing 2 The vulnerable operations in `min-heartbleed`[2]

```
1  #define BUFFERSIZE 500
2  #define COMMANDSIZE 20
3
4  int main(void) {
5      char buffer[BUFFERSIZE] = {0};
6      char command[COMMANDSIZE];
7      int payloadSize, i, j, pos;
8      // Copying a secret key in a buffer
9      strcpy(&buffer[50], "Secret key");
10
11     // No bounds checking
12     do {
13         printf("\nEnter your command below:\n");
14         scanf("%s", command);
15         if (strcmp(command, "heartbeat") == 0) {
16             // Adversaries can send arb. payloadSize
17             scanf("%s %d", buffer, &payloadSize);
18             ...
19             for ( ... ) {
20                 // Send heartbeat msg back to the user given payloadSize
21             }
22         }
23     } while (strcmp(command, "exit") != 0);
24 }
```

The Heartbleed bug is another data-only exploit discovered in OpenSSL [64]. It allows adversaries to read sensitive data such as cryptographic keys. The premise of the bug is due to a rudimentary mistake: missing boundary checks. Adversaries request the vulnerable

[1] https://github.com/mayanez/min-dop

[2] https://github.com/ParasK26/Lame-Heartbleed-Demo-buffer-over-read-in-C-

server to send back a response but with larger memory content than the buffer's maximum length. Because the vulnerable server lacks boundary checks, it will unintentionally overread the memory to fulfill adversaries' requests and copy a sensitive key and send it back. Listing 2 shows the abridged snippet of `min-heartbleed`. A secret key string gets copied onto a buffer at an arbitrary position in this listing. Afterward, `scanf` will ask the user for a heartbeat input containing a payload message and its size to exploit this example's lack of boundary check that leads to overreading the memory to print out the secret key.

Listing 3 Exploitation of MTE tagged `min-dop` example

```
> min-dop: python3 ./exploit_runner.py --gdb 1
...
INFO:exploit:EXPLOIT: Arbitrary memory read
INFO:exploit:[-13] b'0804c070'
...
INFO:exploit:[-09] b'00001337' // Secret Key
```

Listing 4 Exploitation of `min-heartbleed` example

```
>  ./heartbleeddemo_mte.out
Initializing heartbleed demo...
Enter your command below:
heartbeat malicious_read_to_leak 400
6d 61 6c 69 63 69 6f 75 73 5f   m a l i c i o u s _
72 65 61 64 5f 74 6f 5f 6c 65   r e a d _ t o _ l e
61 6b 0  0  0  0  0  0  0  0    a k . . . . . . . .
0  0  0  0  0  0  0  0  0  0    . . . . . . . . . . `
53 65 63 72 65 74 20 4b 65 79   S e c r e t   K e y
```

To test the security effectiveness of vanilla MTE instrumentation, we launch the attacks above with the MTE stack tagging [141]. To summarize, MTE stack tagging successfully prevents arbitrary memory write operation of the `min-dop` example as it tries to overwrite a *local* stack variable during the attack procedure. However, we discovered that vanilla LLVM instrumentation could not stop arbitrary memory read operations. As shown in Listing 3, we leaked the secret key by launching the DOP attack. This is possible for the DOP attack because vanilla LLVM instrumentation do not support tagging a *global* variable. In other words, as long as the attacker is powerful enough to chain necessary primitives (for this case, DOP gadgets) to underflow the vulnerable local stack data, it can arbitrarily read global variable(s), as we demonstrated. Therefore, if this stack data is relocated, then the attack is averted. Similarly in Listing 4, we overread the buffer and sniff out the stored secret key for the heardbleed attack by requesting the program to output the contents of a buffer longer than the length of the message sent[2]. This is possible as the buffer containing a secret message does not dynamically change its tag granularity upon receiving input from the user. Our security analysis on the MTE verified claims of MTE limitation as a security mechanism from previous works [61, 160, 228].

9.2 Shroud Design Overview

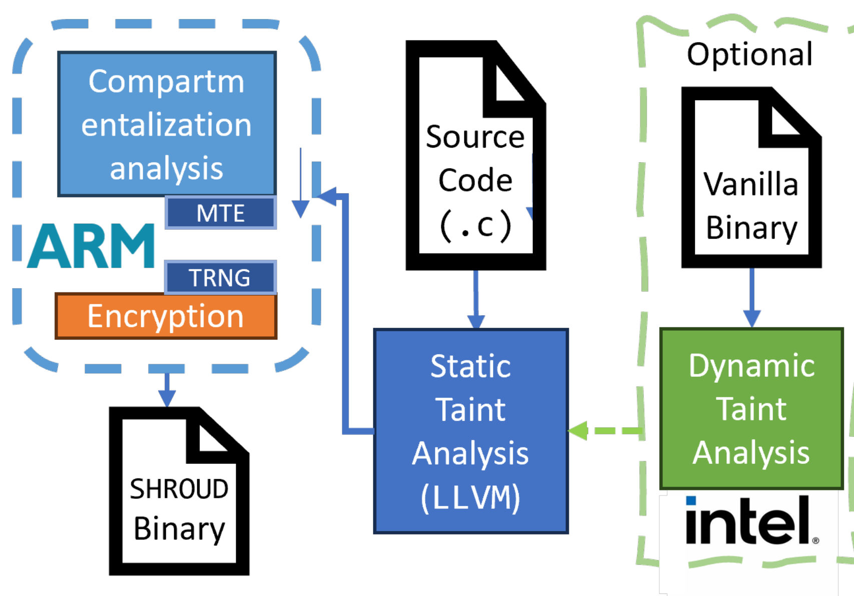


Figure 9.1: An overview of how Shroud safeguards unsafe data to protect against non-control-data attacks

Figure 9.1 illustrates the high-level overview of the Shroud procedure. The Shroud framework consists of the Intel PIN instrumentation tool [101] to perform dynamic taint analysis and various LLVM passes to perform static taint and compartmentalization analysis. If an input binary exists, we leverage dynamic taint analysis based on `libdft` [110] to track data flow between main memory and registers at byte-granularity.

We implemented Shroud on top of LLVM 13.0 [125] and a libShroud to implement the Shroud APIs. We ported the part of the LLVM StackSafetyAnalysis pass [121] and leveraged the SVF framework [207] to develop the static taint analysis. A compartmentalization analysis pass is implemented using a Module pass on LLVM. To preserve the semantics of the source code, we created an input bitcode file with `-O0` flag. Shroud requires an LLVM intermediate representation (IR) bitcode file from the source code.

9.2.1 Software Abstraction of Shroud

Before discussing the Shroud design in detail, we first present libShroud. libShroud provides a practical way to safeguard sensitive data with MTE. It contains APIs that protect the data, such as tagging and untagging a pointer with an arbitrary memory length and encrypting the target pointer. Listing 5 shows the API functions used for Shroud; we only included APIs discussed throughout the paper.

Listing 5 libShroud APIs

```
1 %/* Generate random key using the ARM TRNG */
2 uint64_t gen_rand_key()
3 /* SHROUD initialization */
4 char *shroud_init(char *address, int len);
5 /* Untag the tagged pointer with its length */
6 char *untag_granule (char *address, int len);
7 /* Retag the untagged pointer */
8 char *retag_granule (char *address);
```

Given a source code and a pointer address to the target data, `shroud_init` API can be used to initialize Shroud. This API tags and encrypts the target pointer with a one-time-pad encryption key obtained from the TRNG hardware extension. If there exists a data to the pointer's location, then `shroud_init` will initialize the memory granule relative to the existing data rather than using a default MTE granule value, which is one (16-byte). Although APIs are easy to use, if a developer wishes to use them manually, their usability depends on how much data needs to be protected and their usage throughout the program. Therefore, part of the Shroud compiler pass responsibility is automatically leveraging these APIs after relocating the target data.

Chapter 10

SHROUD Implementation

This chapter details the implementation of SHROUD, including static and dynamic taint analysis, relocation of unsafe stack variables to the heap, and an encryption scheme for protection. Static taint analysis identifies unsafe stack data, while dynamic taint analysis tracks runtime data flow. Unsafe data is relocated to heap memory with random tagging. XOR encryption is used to protect data by encrypting relocated pointer addresses with a one-time-pad key generated by ARM's TRNG, ensuring robust protection against attacks.

Section 10.1 covers the implementation of static taint analysis in our design, while Section 10.2 details the dynamic taint analysis. Section 10.3 presents the relocation analysis that converts stack variables to the heap. Finally, Section 10.4 discusses the encryption scheme used to safeguard against MTE vulnerabilities.

10.1 Static Taint Analysis

To automatically determine compartmentalization targets of stack data for SHROUD, we propose a novel approach that uses unsafe data targets as the taint source for static taint analysis. Taint analysis is a data-flow analysis based on the *source-sink* property. The idea behind this analysis is to track how targeted information (*source*) propagates through various paths (based on a graph if static, based on a runtime path if dynamic) to determine whether the information leaves through a potentially malicious destination (*sinks*).

Given an input module, we generate a worklist consisting of all functions. For each function in the worklist, we leverage an intraprocedural LLVM StackSafetyAnalysis to determine all potentially unsafe stack objects. This analysis examines the `Use` (refers to the LLVM's `Use` class) of stack memory allocator (i.e., `alloca`) instructions to check whether the target allocation is safe (i.e., whether all accesses to the target object are statically known) or not. Unsafe data for this analysis is defined as follows:

1. Unsafe pointer casts
2. Casts between pointers and integers
3. A pointer is passed as an argument to another function, where it gets cast to a potentially unsafe pointer

4. An element of an array is accessed with a statically undefined index (i.e., variable-based)

The downside of this analysis is that it may overapproximate unsafe stack data, which could lead to overprotection if directly used. Although this does not affect security guarantees, it may impact performance. By having concrete taint sources from StackSafetyAnalysis and explicit taint sinks that could cause memory safety violations, SHROUD's static taint analysis increases the precision of the StackSafetyAnalysis.

To identify taint sinks, we use the SVF [207], a scalable LLVM-based framework that provides precise value-flow and pointer analysis. We construct a flow-sensitive sparse value-flow graph (SVFG) that captures the program's def-use chains and value flows for both top-level and address-taken variables. In particular, we apply the wave propagation-based Andersen analysis [171] to generate the points-to information for the variables mentioned above. This is a flow- and context-insensitive points-to analysis that maintains the precision of Andersen analysis [7] while providing faster performance. To ensure graph generation scalability, we allow users to input the depth level for the graph construction.

Identifying sinks is an integral part of devising static taint flow analysis, but automatically determining what constitutes taint sinks is an open-ended problem and varies depending on its application [84, 200]. Typically, sinks are defined as potentially untrusted external functions not defined within a program (e.g., library API calls such as `strcpy`) [9]. However, considering *all* external functions as sinks would be an exaggeration. Therefore, to narrow down the potential sink candidates, we iterate through all callees of every callsite in the given SVFG and perform interprocedural backward dataflow analysis through each callee function's arguments to determine whether any argument *accesses* the memory object (e.g., read, write, etc.). In other words, a callee function that accesses only constant data would not be considered a sink. On the other hand, if we find access to a memory object, we consider the analyzed callee function as a sink.

Upon identifying sinks, we perform forward taint propagation analysis for each taint source in a function to track whether it reaches any of the taint sinks. After iterating through all functions, we find the intersection between the original and new candidates of unsafe stack objects (i.e., taint sources) to narrow down the compartmentalization targets.

10.2 Dynamic Taint Analysis

If a binary exists for the input module, SHROUD allows runtime information obtained from the dynamic taint analysis as an extra reference to supplement static taint analysis. Dynamic taint analysis is a type of taint analysis that tracks the data flow of predefined taint source and sinks through *runtime information* (e.g., external input through network sockets). In contrast to the static variant, the dynamic variant allows developers to determine a program's

potential "hot spots," which serves as a great starting point for what protected functions need to be. This information has been previously applied to determine the zero-day attack path probabilistically [208].

Furthermore, because a user will unlikely access all functions during the program's execution, these spots can be implied where adversaries may pass through during attack procedures. We use `libdft` [110], a dynamic data flow tracking library, to aid our dynamic taint analysis. Performing dynamic taint analysis will only provide information on tainted instructions, which does not give any information on what functions those instructions reside in. Therefore, we use Binary Ninja [2] to recover the functions containing the tainted instructions and dump out the symbol names to be included in our hybrid taint analysis.

10.3 Compartmentalization Analysis

After performing the taint analysis (default is static, optionally hybrid), Shroud uses the results to *relocate* the unsafe data to separate memory regions. To prepare for the relocation, Shroud first records the size of the unsafe stack object and all of its uses. Shroud implements this relocation using heap memory (e.g., `malloc`) and replaces all usage of the original unsafe stack objects with newly created heap memory. If the target unsafe stack object contains data, Shroud first copies this original data to the relocated destination to retain its original value. Lastly, Shroud ensures the `free` of all relocated data and performs cleanup of deprecated instructions. In contrast to the default stack tagging strategy, relocated data randomly generates a tag during execution; this effectively defeats the "increment-by-one" attack [170].

After the relocation analysis, Shroud begins instrumenting Shroud's compartmentalization API instructions to safeguard relocated data. Each data pointer's address is assigned to a random color compartment, as described in Figure 2.1, and Shroud ensures the granule size fits with its respective data size. Heap tagging is effective against many non-control-data attacks, but blindly using it is dangerous. Some situations warrant untagging and retagging to utilize heap tagging effectively, while others do not.

In the `min-heartbleed` example, an unsafe stack object (e.g., `buffer`) is used in many memory-string-related functions (e.g., `strcpy`). As previously mentioned, Shroud first relocates the `buffer` to a separate memory region and safeguards the data via compartmentalization. The key problem is that if this safeguarded data in the `strcpy` function (line 9 in Listing 6) is used straightforwardly, it leads to a segmentation fault and an unintended data leak.

A segmentation fault occurs because when the `scanf` function attempts to read the compartmentalized data, there is a tag mismatch between the function and the data. This mismatch arises from the LLVM `malloc` function being incompatible with the ARM-modified custom malloc implementation, as cited in the MTE whitepaper [195]. Therefore, we need to untag

Listing 6 Pseudo LLVM IR of SHROUD's compiler instrumentation on `min-heartbleed` example

```
1  define dso_local i32 @main() #0 {
2      entry:
3      %retval = alloca i32, align 4
4      %0 = alloca i8*, align 8
5      %1 = tail call i8* @malloc(i64 500)
6      %2 = load i8*, i8** %0, align 8
7      %3 = call i8* @shroud_init(i8* %2, i32 500))
8      %4 = getelementptr i8, i8* %0, i64 50
9      %call1 = call i8* @strcpy(i8* %4, ...))
10     ...
11     do.body:   ; preds = %do.cond, %entry
12     %call2 = call i32 (i8*, ...) @printf(...))
13     %15 = call i8* @untag_granule(i8* %0, i32 20)
14     %16 = getelementptr i8, i8* %15, i64 0
15     %call3 = call ... @__isoc99_scanf(... i8* %16)
16     %17 = call i8* @retag_granule(i8* %0)
17 }
```

the data before using the relocated data for any memory-string-related functions (as shown in line 13 in Listing 6).

However, there are other reasons why untagging is necessary. Another reason is that unintended data leakage could occur if the relocated data is not retagged according to the updated size. For instance, when relocated data is used in a function such as `scanf`, which takes in user input during runtime, its input size is undetermined at compile time. Therefore, data leakage will occur if the relocated data is not retagged with the correct compartment size after taking in the user input. When memory-string-related functions execute, SHROUD checks the updated size of the relocated data based on the input and then retags memory granule blocks accordingly (as shown in line 16 in Listing 6) automatically.

10.4 Encryption

To camouflage the compartmentalized data, SHROUD `XOR` encrypts relocated data pointer addresses with the one-time-pad (OTP) encryption key, similar to the work of PointGuard [45]. As previously mentioned in Section 9.1, the tag value is a plaintext that adversaries can sniff, which makes encryption an effective strategy to defeat the tag-sniffing attack [248]. Furthermore, we choose `XOR` encryption method because it is efficient and straightforward. A complicated scheme could make protection deployment difficult [29]. If the adversary wishes to sniff the tag, it must first obtain the one-time-pad encryption key.

SHROUD generates the OTP encryption key using the TRNG feature in the ARM v8.5 architecture [182]. TRNG is a hardware module connected to a random source (undocumented) that generates a random bit stream that can be retrieved using the `rndr` instruction. SHROUD does not need to have the number prepared ahead of time, minimizing any potential perfor-

mance overhead. Shroud generates the key per relocated data's pointer address at execution. In other words, an encryption key is not correlated to the tag value but is related to the address itself. Therefore, even if the same tag is reused for different relocated data addresses, each address will have a unique encryption key. Hence adversaries won't be able to infer any helpful information. Shroud revokes the key when the tagged pointer is untagged.

The TRNG key is impossible to predict as it is private to Shroud's process. Only the program already under the adversary's control via exploiting the vulnerability outside of Shroud's protection scope will leak this key. Even if a user obtains enough permission to run the `ptrace` tool, commands such as `PTRACE_PEEKMTETAGS`[1] will fail due to invalid input. Using the relocated data addresses as an index, we use a uthash [90] hash table to manage assigned encryption keys. Adversaries cannot leak keys from the uthash table because it is embedded into the running process (i.e., adversaries need complete control of the process during runtime to leak).

[1] https://www.kernel.org/doc/html/latest/arm64/memory-tagging-extension.html

Chapter 11

Shroud Evaluation

In this chapter, we evaluate Shroud's capabilities concerning runtime overhead and its effectiveness against security vulnerabilities. Specifically, the aim of this chapter is to understand:

1. The runtime overhead incurred by Shroud, particularly when relocating data to separate memory regions, and whether this overhead is feasible for practical use.

2. The security benefits of Shroud, demonstrated through adversarial analysis, including real-world examples and minimalistic educational exploits to assess its protection against common vulnerabilities and exposures (CVEs).

Section 11.1 describes the setup environment for our evaluation. Section 11.2 discusses the runtime overhead of using Shroud. Next, in Section 11.3, we provide a security evaluation of Shroud using adversarial analysis. Section 11.4 discusses Shroud's limitations and future work. Finally, Section 11.5 summarizes our work on Shroud.

11.1 Setup Environment

Shroud utilizes ARM MTE and TRNG features of the ARMv8.5-A architecture. For our evaluation, we installed Ubuntu on the Pixel 8 mobile phone to measure performance, as to the best of our knowledge, *no commercially available non-mobile device with the MTE feature enabled exists*. However, since the Pixel 8 does not include the TRNG feature of the ARMv8.5-A, we opted to use a pseudo-random number generator to encrypt the key. As MTE is the key feature we require and the TRNG feature will improve the performance, the evaluation we have done represents the ceiling of performance overhead and would only improve if we were to apply the TRNG feature as well. We purchased the Mac Mini M2, which contains the Apple M2 processor based on the ARMv8.5-A architecture. However, after checking the machine, we found that the MTE feature was unavailable because it is optional per the ARM specification.

11.2 Runtime Overhead Evaluation

For our runtime overhead evaluation, we first measured the overhead of data relocation after relocation analysis. Next, we follow recent works of [38, 39, 128, 139, 163] and use an open-source Byte-UnixBench benchmark[1] that provides a performance measurement of a UNIX-like system.

Table 11.1: `httpd` relocation analysis evaluations

Response Size	Vanilla	SHROUD	Overhead
`100M`	50.8s	50.86s	0.12%
`250M`	114.36s	115.61s	1.09%
`500M`	212.63s	214.26s	0.77%
`1G`	414.15s	415.25s	0.27%
Average			0.56%

We first evaluated the overhead of relocating data to measure how much performance overhead could occur from moving data to separate memory regions. Table 11.1 provides a quick summary of the results. After creating a bitcode of `httpd` version 2.4.39 using the WLLVM [176], we performed the relocation analysis (Section 10.3). We relocated all potentially unsafe stack objects from the LLVM StackSafetyAnalysis (except for the cases we omit, such as data struct) and measured the overhead of before-and-after relocation. This decision was to demonstrate the possible worst-case scenario of relocating the data. We measured the performance using ApacheBench [74] by sending 1,000 requests with file sizes ranging from 100MB to 1 GB. Although it is well known that allocating data on the stack is much faster than on the dynamic memory, the reason why the performance overhead is negligible is that for a big application such as `httpd`, dynamic memory allocation is insignificant. Therefore, the relocation analysis becomes a reasonable solution; data access performance is similar for static and dynamic memory.

[1] https://github.com/kdlucas/byte-unixbench

Table 11.2: UnixBench benchmark evaluations

Applications	Vanilla	SHROUD	Overhead
`dhry2reg`	6.12E+07*lps*	6.08E+07*lps*	0.62%
`arithoh`	4.81E+08 *lps*	4.78E+08 *lps*	0.21%
`pipe`	1.78E+06 *lps*	1.74E+06 *lps*	1.99%
`context1`	1.42E+05 *lps*	1.39E+05 *lps*	2.14%
`syscall`	4.26E+03 *lps*	4.16E+03 *lps*	2.26%
`execl`	3.42E+02 *lps*	3.31E+02 *lps*	3.42%
`whetstone -double`	8217.3 MWIPS	8084.6 MWIPS	1.61%
Average			2%

Next, we evaluated SHROUD overhead using UnixBench. UnixBench provides many tests targeted for specific operations (e.g., System Call Overhead and Pipe Throughput) and generates a different metric indicating performance per benchmark. Table 11.2 presents the result of all benchmarks except ones related to the UNIX system's performance (e.g., file operations and shell scripts). Results are shown in either loops per second (*lps*) or millions of whetstone instructions per second (MWIPS); the higher number, the better. As SHROUD could not find any sensitive data for these tests, for performance measuring purposes, we insert arbitrary data inside the main loop (if it exists) of each benchmark. Afterward, we use LIBSHROUD with an encryption/decryption mechanism to measure how SHROUD will affect the performance (we perform key generation/searching outside the main loop as that is a redundant operation).

Table 11.3: SNU_NPB (Class A) benchmark evaluations

Applications	Vanilla	SHROUD	Overhead
Integer Sort (`IS`)	0.22s	0.23s	4.55%
Embarrassingly Parallel (`EP`)	14.61s	14.61s	0%
Multigrid (`MG`)	0.71s	0.72s	1.41%
3-D FFT PDE(`FT`)	2.08s	2.11s	1.44%
Conjugate Gradient(`CG`)	0.81s	0.87s	7.41%
Pentadigaonal Solver(`SP`)	15.29s	15.6s	2.03%
Block Tridiagonal Solver(`BT`)	33.79s	33.83s	0.12%
LU Solver(`LU`)	21.26s	22.02s	3.57%
Average			2.57%

Table 11.4: lighttpd ApacheBench evaluations

Response Size	Vanilla	Shroud	Overhead
4KB	2.419s	2.45s	1.3%
16K	2.288s	2.44s	6.6%
64K	2.4s	2.5s	4.2%
256K	3.021s	3.2s	5.9%
512K	1.846s	0.87s	6.7%
1M	1.822s	1.828s	0.3%
Average			4.2%

We evaluated Shroud using the memory and computation-intensive SNU NPB-SER-C benchmark [194]. Similar to UnixBench, we protected an arbitrary buffer with libShroud. The results, presented in Table 11.3 in seconds (s), illustrate that the average performance overhead across applications is 2.11%. Furthermore, we assessed the performance overhead of Shroud on real-world applications such as lighttpd, an open-source and secure web server application. For lighttpd, we protected the pointer `password_buf` in the function `mod_authn_file_htpasswd_get`, and measured the performance using ApacheBench [74] by sending 1,000 requests with file sizes ranging from 4KB to 1MB, as shown in Table 11.4. The average overhead was 4.2%.

There exist many related works to Shroud as discussed in Section 3.7. However, a direct head-to-head comparison with Shroud is not feasible due to several reasons. Firstly, these works often cover different scopes; for example, HAKC focuses on enforcing compartmentalization within the kernel space. Secondly, some techniques, such as HDFI and CHERI, require specialized hardware that is not readily obtainable. Additionally, many of these approaches are closed-sourced, or they do not implement a compartmentalization scheme comparable to ours, thus precluding a fair comparison. Therefore, providing a direct comparison of Shroud with non-protected binaries is more appropriate to demonstrate the value of this contribution.

11.3 Security Evaluation

We conduct a security evaluation of Shroud, which includes a case study of CVEs and adversarial analysis. Replicating exact exploits for CVEs demonstrated on the x86 architecture on a completely different architecture (e.g., `aarch64`) proved challenging. Therefore, we first provide a case study of CVE bugs to illustrate Shroud's protection capabilities. Next, we use open-sourced minimalistic examples that demonstrate the aforementioned CVE attack scenarios to perform adversarial analysis and show how Shroud protects against these attacks.

11.3.1 Threat Model and Assumptions

SHROUD has a threat model like prior works related to non-control-data attacks [29, 36, 37, 98]. In detail, we assume the adversary has access to the target binaries; this includes application and shared libraries (e.g., `libShroud.so`). Furthermore, there exists a presence of memory corruption vulnerability, and the adversary knows the exact location of the target data. We assume the adversary can send arbitrary data to the target process via inputting a command at runtime, but it *does not* have the `sudo` permission. Hence, we assume a trustworthy TCB, including the OS kernel.

In the context of non-control-data attacks, the severity of certain classes of attacks can be controlled. For example, in Data-Oriented Programming (DOP) attacks, the severity is based on longer chains, which involve increased iterations of the loop dispatcher to chain DOP gadgets. However, for the Heartbleed attack, there is no flexibility in terms of controlling its severity due to the attack's simplicity. The nature of launching the attack is static, and the adversary cannot selectively target specific data within the memory. They can only control their request counts until the necessary data is successfully leaked. Therefore, it is important to note that SHROUD is an enforcement mitigation technique, and the severity of the attack has no impact with respect to performance overhead.

11.3.2 Adversarial Analysis

The adversarial analysis is performed by first launching the respective attack on the vulnerable binary to exploit it. Afterward, we protect the vulnerable binary and analyze whether the proposed security primitive can withstand the attack. As detailed in our threat model (Section 11.3.1), we assume that the adversary has full access to the binary but lacks the `sudo` ability to attach debugging tools (e.g., `ptrace`) to the binary.

Under a common attack scenario, SHROUD protects against the probabilistic memory exploitation that MTE suffers from. Normally, the adversary attempts to counterfeit a pointer by guessing the tag value to launch the attack. However, by leveraging the encryption scheme to camouflage the entirety of the tagged pointer (both tag value and address location), we prevent adversaries from easily retrieving sensitive data.

CVE 2006-5815 is a stack-based buffer overflow vulnerability within the `sreplace` function of a file server program called `proftpd` (versions 1.2 through 1.3.0) [48]. The premise of this exploit is based on an off-by-one heap overflow bug that leads to exploitation of the `sstrncpy` function, allowing the adversary to overwrite string pointers used by a public output function (e.g., `send`) to arbitrarily read memory and leak the sensitive key. `min-dop` is an open-source educational example demonstrating how similar CVE vulnerabilities can be exploited by a Data-Oriented Programming attack [98]. Although this example is not a direct replica of the CVE, it contains vulnerabilities similar to the actual CVE, making it suitable for evaluating the security of SHROUD.

CVE 2014-0160 is a vulnerability in the OpenSSL library that allows the adversary to leak sensitive data by exploiting the lack of bound-checking in the `heartbeat` feature [49]. `min-heartbleed` is a minimalistic open-source example demonstrating this by using string / memory-related functions. As previously described, the MTE-enforced binary allows an overread of the buffer by exploiting the nature of memory tagging and the `scanf` function.

The main reason buffer overreading occurs in `min-heartbleed` is the lack of update of memory granules throughout the code's execution. After the initial tagging of sensitive data's memory granule, this tag is not dynamically updated when the data is copied via the `strcpy` function or when the `scanf` function takes in user input. For example, if a user types malicious input into the `scanf` function, and the input is 22 bytes, the memory granule must be tagged with two tags, as each granule is 16 bytes long. If the adversary then attempts to read more than 32 bytes, a segmentation fault will occur.

We recognize that our compartmentalization cannot provide complete protection against buffer overreads. This limitation is not due to the design of Shroud but to the intrinsic granule size limitation, where memory granules are tagged in 16-byte increments. For instance, if data tagged with two memory granules contains a secret key within these granules, the adversary can see portions of the secret key. However, this weakness is challenging to exploit because the adversary must know the location of the secret key by other means (e.g., information leakage) to craft the exploit. If necessary, data padding can resolve this issue.

11.3.3 Sensitive Data Analysis

Table 11.5: Shroud analyses evaluations

Inputs	SSA	STA	HTA-3	HTA-2	HTA-1
`min-dop`	5	1	1	1	1
`min-heartbleed`	2	2	2	2	2
`httpd`	30	0	0	0	0
`proftpd`	112	81	55	27	5

In Shroud, various analyses are applied to automatically determine unsafe data that needs to be relocated and compartmentalized. For static analysis techniques, the main question is whether the proposed approach is practical and can be used as a general solution. In contrast, for dynamic analysis techniques, the concern is regarding the coverage; what about paths not covered during execution? Therefore, we provide three layers of analyses starting from LLVM's StackSafetyAnalysis (SSA) to Shroud, various analyses are applied to automatically determineys's Static Taint Analysis (STA) to Shroud, various analyses are applied to automatically determine's Hybrid Taint Analysis (HTA), varying from level 3 (HTA-3) to

level 1 (HTA-1) with `main` as the starting point.

Table 11.5 summarizes the unsafe stack object candidates of the before-and-after analyses. For example, `min-dop` candidates went down from five to one; the largest impact can be seen for a large application such as `proftpd`, where candidates went down from 112 to 5, depending on the analysis and the level. As a reminder, level in this context represents the depth of callee exploration of a given function. For instance, if level 2 and starting exploration function is `main` in a program with the caller-callee relationship of `main`→`foo`→`bar`→ `baz`, SHROUD, various analyses are applied to automatically determine would analyze all functions from `main` to `bar`.

To explain the false positives, let's look at the `min-dop` example. Using the LLVM SSA, there are five unsafe candidates, but four of five unsafe stack objects result from a local stack object being passed to external library functions not defined in the program (e.g., `send`). Considering all five stack objects to be unsafe based only on that knowledge would result in overprotection. Among potential false positives, LLVM SSA also considers the variable `buffer` (line 8) unsafe as it is used in the `memcpy` function.

Let's revisit Listing 1; the true positive candidate among the five candidates is in the `readInData` function (line 27), where there exists a `memcpy` that copies client data (`buffer`) to the global variable (`sbuf`) to cause the memory write safety violation. The reason why SHROUD analyses consider this particular operation as the true positive candidate is that after using the interprocedural backward dataflow analysis on each argument of the `memcpy` function, it detects that the destination variable `buf` is a pointer to a global variable `sbuf`.

SHROUD intrinsically considers global variables vulnerable for our case as any function can change its value. Although SHROUD does not safeguard the global variable as that is outside of our protection scope, we protect the client data to prevent memory write violations. Regarding the protection scope, APIs presented in Listing 5 can be applied for heap addresses, but they cannot be applied to global scope variables that reside in the *data segment* (`.data`), not the heap.

For an extensive application such as `proftpd`, SHROUD analysis can successfully narrow down the unsafe stack object candidates to include the vulnerable local `buf` variable that is used as a basis for a stack-based buffer overflow vulnerability in the `sreplace` function (CVE-2006-5815) [37] up to SHROUD HTA-2 (Hybrid Taint Analysis - Level 2). In other words, SHROUD HTA-1 cannot catch the vulnerable local variable `buf` as the exploration depth is too short. For this vulnerability, adversaries exploit the `sstrncpy` function that uses a pointer `cp`, which points to the `buf`, to overflow the buffer. SHROUD's HTA analysis can successfully reduce the number from STA of 81 to 27 that the destination argument to the `sstrncpy` argument is referring to the vulnerable stack object due to its forward taint propagation analysis leveraging the SVF [207] and `libdft` [110].

Although SHROUD HTA-2 analysis can successfully reduce the candidates from 112 to 27 and detect the existing vulnerability for `proftpd`, what about the other 26 candidates? Although

it is fair to consider these false positives, and we cannot guarantee that adversaries may not use them in other unknown attacks, this is an inherent limitation of static analysis outside this paper's scope. By proposing the hybrid taint analysis, SHROUD paves the way to combine different analyses to detect compartmentalization targets automatically to the best of its ability. In addition, overprotection does not jeopardize the application's protection strength but mainly affects the performance. Later on, we propose potentially alternative analyses to remedy this problem (Section 11.4).

11.4 Discussion

MTE Intrinsic Limitations. Although SHROUD can amend a few of these limitations, it cannot be a complete solution. However, a potential data leak is still possible because MTE's granule size must be 16 bytes. Furthermore, as the author mentioned in [195], MTE cannot detect a bug such as *intra-object-buffer-overflow* because buggy access occurs *within the same* heap- (or stack-) allocated object. Therefore, our work omits cases related to intra-objects (e.g., structs).

Performance Optimization Opportunities. When we designed a way to safeguard sensitive data on the stack, we realized that vanilla memory tagging could only provide probabilistic protection over stack objects. This is because of the limited number of tags in MTE, which allows the adversary to potentially use the brute force method to guess the tag value. Therefore, we relocate the unsafe stack object to a global and randomly allocated location. We camouflaged the relocated data pointers with random keys. Still, a better way to address this problem could be to change the tags to unreadable, and only through privileged instructions can they be modified. Although there are other possible optimization for SHROUD, such as combining pointer authentication (PAC) with the MTE to ensure the pointer integrity [148], we opted to leave this optimization as a future work to increase the applicability of SHROUD.

Precision of Static Analysis. LLVM StackSafetyAnalysis is complete in finding all potentially unsafe stack objects, but it suffers from false positives due to its overapproximate nature. Therefore, we proposed the hybrid taint analysis tailored to protecting unsafe stack objects to generalize the SHROUD's applicability. Inherent limitations of static taint analysis techniques remain because what constitutes taint sinks differs per objective and needs to be defined at compile-time. Therefore, we worked on two alternative analyses to potentially improve this: 1) Strawman Analysis and 2) Annotation Analysis.

Alternative Analyses. To alleviate the static analysis limitations, we attempted two methods to reduce false positives: 1) Strawman Analysis, 2) Annotation Analysis. Strawman analysis is based on the concept of "known attacks"; in other words, we attempt to leverage the information necessary for adversaries (e.g., the algorithm to construct the DOP attack [98]) to narrow down the potentially unsafe data and safeguard it. The downside of

this approach is the *zero-day attack*. Annotation analysis is a semi-automatic approach that allows developers to annotate unsafe data and protect them intuitively. This approach has been used in other security works [107, 148, 166], but it requires manual annotations.

11.5 Summary

As a third contribution of this dissertation, we present Shroud, a mechanism leveraging ARM's Memory Tagging Extension (MTE) to compartmentalize sensitive data. Shroud initially applies a taint analysis (default is static, optionally hybrid) to automatically identify potentially unsafe data, utilizing frameworks like `libdft` for dynamic analysis and LLVM StackSafetyAnalysis for static analysis. By combining StackSafetyAnalysis with static taint analysis using the SVF framework, Shroud narrows down unsafe data candidates and reduces false positives. Following taint analysis, Shroud relocates unsafe stack data to separate memory regions, mitigating stack-based buffer vulnerabilities and flawed stack-tagging strategies. This relocation prepares data for MTE heap tagging, allowing greater flexibility in access control. Additionally, Shroud employs a one-time-pad (OTP) encryption scheme leveraging ARM's TRNG feature to camouflage the MTE-tagged pointer, preventing tag sniffing attacks and ensuring robust protection.

We designed and implemented a prototype of Shroud on top of LLVM, evaluating its effectiveness and performance overhead. The hybrid taint analysis of Shroud was first assessed for accuracy. We then measured the runtime overhead of relocating data in the `httpd` application, assessing the performance impact in the worst-case scenario of relocating all potentially unsafe stack objects. Shroud also demonstrated its performance by running runtime overhead evaluations on lighttpd, UnixBench, and the NPB benchmark. To evaluate the security effectiveness of Shroud, we performed adversarial analysis using `min-dop` and `min-heartbleed`, demonstrating Shroud's ability to counteract similar CVE attack behaviors. These findings highlight Shroud's capability to provide fine-grained compartmentalization and robust protection against common vulnerabilities and exposures (CVEs).

Chapter 12

IBCS: Input-Based Compartmentalization System

Our previous contribution explored leveraging advancements in CPU technology to realize an efficient compartmentalization technique. However, despite our efforts, we realized that providing accurate compartmentalization necessitated developer annotations, similar to many related works [41, 86, 100, 211, 214]. Furthermore, our approach suffered from intrinsic hardware limitations that are impossible to bypass without architectural changes. Relying on annotations increases the Trusted Computing Base (TCB) and places pressure on developers to balance security against performance. While some systems, like those documented in [111, 179], automate this process, they are generally limited to embedded systems and heavily rely on specific architectural features.

Therefore, it is important to explore an architecture-agnostic solution that can provide fine-grained compartmentalization while alleviating the need for developer annotations. Vulnerabilities highlighted by exploits like Heartbleed (CVE-2014-0160) capitalize on external inputs, underscoring the need for enhanced input-driven security measures. To this end, we propose IBCS, a comprehensive compartmentalization toolchain that utilizes user input to automatically identify data for memory protection.

IBCS employs various techniques to determine these targets directly from concrete user inputs. This method mitigates the overapproximation often seen in automatically identifying compartmentalization spaces, which typically involves trade-offs between different granularities [86]. Based on user inputs, IBCS uses a hybrid taint analysis—a combination of static and dynamic approaches—to automatically generate sensitive code paths (i.e., protection regions). IBCS further analyzes each variable within these paths using novel assembly analyses to identify fine-grained compartmentalization targets. Subsequently, IBCS compartmentalizes all identified data, leveraging a novel assembly rewriting technique, which includes static verifiers to ensure correctness.

Section 12.1 presents the motivating example for IBCS. Next, we discuss the challenges of providing automatic fine-grained compartmentalization in Section 12.2. The design overview of IBCS is presented in Section 12.3.

12.1 Motivating Example

Listing 7 A Motivating Example

```
1  ...
2  #define MAXLINE 1024   /* max length of a line */
3
4  typedef struct {
5      char filename[512];
6      off_t offset;              /* for support Range */
7      size_t end;
8  } http_request;
9
10 void url_decode(char* src, char* dest, int max) {
11     char *p = src; char code[3] = { 0 };
12     while(*p && --max) {
13         if(*p == '%') {
14             memcpy(code, ++p, 2);
15             *dest++ = (char)strtoul(code, NULL, 16);
16             p += 2; } else { *dest++ = *p++; }
17     }
18     *dest = '\0';
19 }
20
21 void parse_request(int fd, http_request *req){
22     rio_t rio;
23     char buf[MAXLINE], method[MAXLINE], uri[MAXLINE];
24     char* filename = uri;
25     req->offset = 0;
26     req->end = 0;              /* default */
27     ...
28     // Possible buffer overflow
29     url_decode(filename, req->filename, MAXLINE);
30 }
31
32 void process(int fd, struct sockaddr_in *clientaddr){
33     printf("accept request, fd is %d, pid is %d\n", ...);
34     http_request req;
35     parse_request(fd, &req);
36     ...^^I
37 }
38
39 int main(int argc, char** argv){
40     ...
41     while(1){
42         connfd = accept(listenfd, ...);
43         process(connfd, &clientaddr);
44         close(connfd);
45     }
46 }
```

Most external exploits rely on server applications' external inputs (e.g., CVE-2014-0160 [64, 150]). Listing 7 presents a snippet from the web server program available in [23] that is exploitable by leveraging external inputs[1]. This example illustrates the challenges of automatically identifying and compartmentalizing an application based solely on user input.

[1] https://blog.coffinsec.com/2017/11/10/tiny-web-server-buffer-overflow-discovery-and-poc.html

After initializing necessary variables in the `main` function, the program enters an infinite loop that uses the `accept` function to accept connections and the `process` function to handle requests. A variable `req` of type `http_request` is then created and passed to the `parse_request` function. This function contains potentially exploitable code (line 29), where `req->filename` (buffer size of 512) is passed to the `url_decode` function as the destination with a maximum read size set to `MAXLINE` (1024). Consequently, an input larger than 512 bytes could overflow the buffer in `url_decode`.

The `filename` member variable must be isolated by the protection scheme to prevent buffer overruns through fine-grained compartmentalization. This isolation ensures that any excessively large input sent by adversaries is confined within its compartment, preventing it from affecting adjacent buffers or overwriting the return address to launch a control-data attack.

While several works achieve this at a coarse-grained level, automatically identifying this specific target in a fine-grained manner and then enforcing compartmentalization presents four unique challenges:

12.2 Challenges

- **Over/Underapproximation Challenge (OC)**: If IBCS records all visited functions from user input and enforces fine-grained compartmentalization on those functions, this approach might overapproximate targets, negatively impacting performance. Conversely, static analyses may result in underapproximation due to their inherent limitations [239, 244]. Striking a proper balance between over- and underapproximation is essential.

- **Identification Challenge (IC)**: Tracking user input during runtime provides accurate context-specific data, but relying solely on this information may lead to overapproximation from static analysis or underapproximation due to insufficient annotations. Dynamic analysis, unless complemented by software testing techniques such as fuzzing [127, 129], might miss potential compartmentalization targets. For example, dynamic analysis based on user input in Listing 7 failed to identify‘ `req->filename` as a target (Figure 12.1), highlighting the need for methods that minimize potential false negatives.

- **Enforcement Challenge (EC)**: Previous works have proposed various enforcement mechanisms for memory protection, many of which are manually applied [86, 100, 148, 211, 214] or rely on automated security monitors [41] that require separate privileged modes and introduce execution overhead. Relying on hardware-specific mechanisms [111, 135, 222] may limit applicability across different architectures. Developing architecture-agnostic enforcement mechanisms adaptable to various architectures is therefore vital for broader applicability.

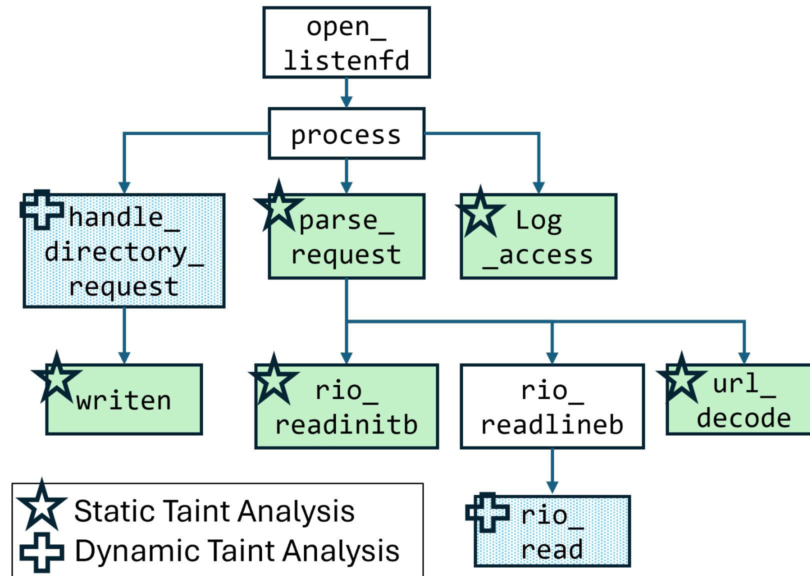


Figure 12.1: Hybrid Taint Analysis Call Graph

- **Assembly Challenge (AC)**: Instead of modifying LLVM [125] to enforce compartmentalization, we rewrite assembly code directly. This approach circumvents potential API incompatibilities of different LLVM versions and leverages resolved information such as variable stack offsets in function frames. However, accurately rewriting assembly code requires a thorough understanding of register offsets. The challenge lies in creating an infrastructure capable of correctly decoding each assembly line to identify and rewrite patch targets without altering the original program's intent. This problem is exacerbated when dealing with pointers, as they are represented as local function offsets when passed to subsequent callee functions.

12.3 IBCS Design Overview

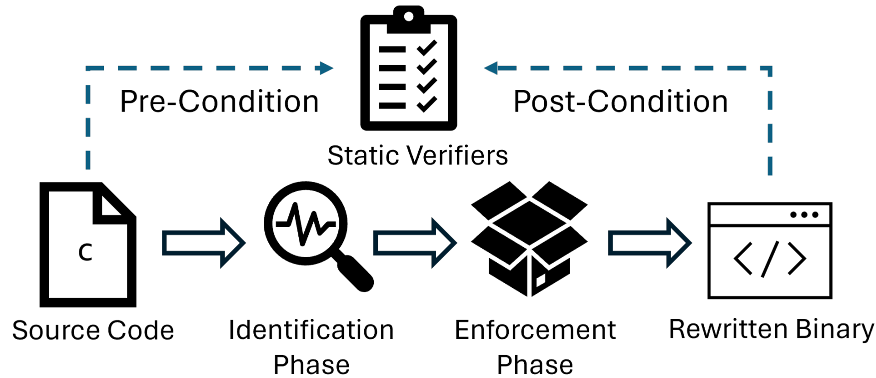


Figure 12.2: **IBCS** Design Overview

IBCS is a fully automatic compartmentalization framework for userspace applications. Figure 12.2 shows the summary of **IBCS**. Given a source code, **IBCS** first generates an assembly code and verifies it using a static verifier (discussed in Section 13.2.4) to ensure the assembly code is well-behaved with the given pre-condition. Subsequently, it identifies potentially vulnerable candidates that will be isolated by leveraging various static analyses. Afterward, **IBCS** automatically compartmentalizes these candidates by rewriting the assembly code so that the data is relocated to an isolated memory page that will protect it from potential memory exploits. Upon finishing the rewriting process, **IBCS** will verify the newly created binary file to ensure it complies with the post-condition, which checks to ensure that the assembly code is correctly rewritten.

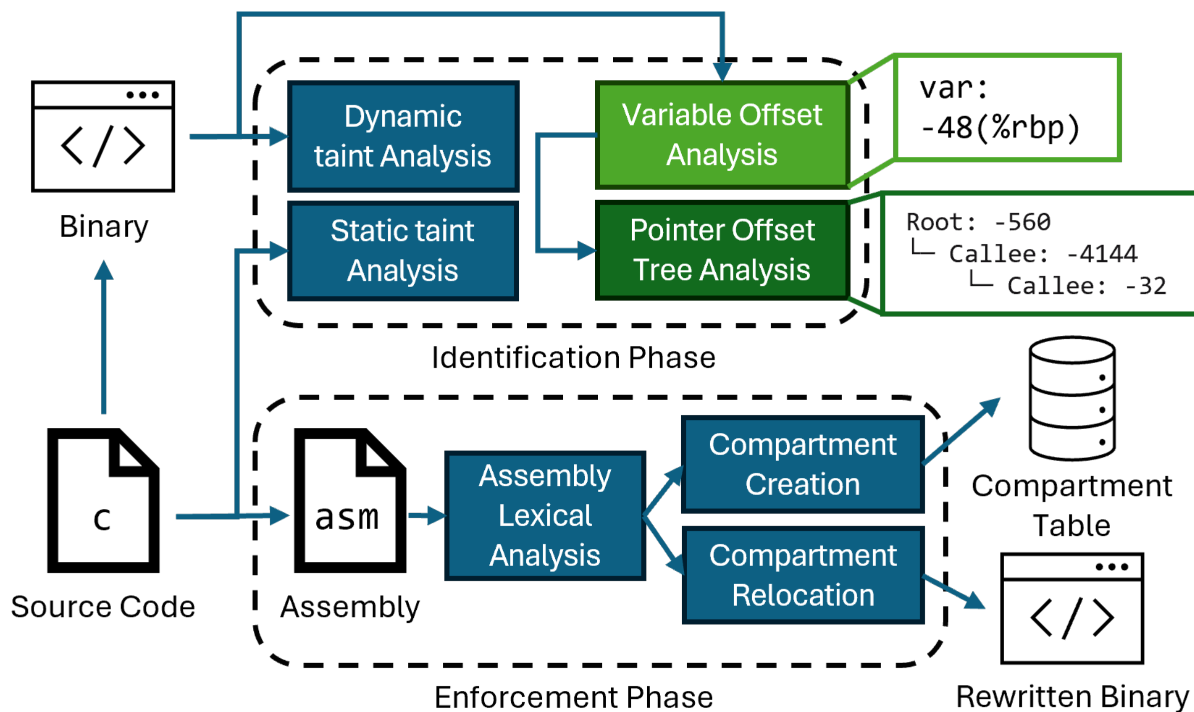


Figure 12.3: IBCS Toolchain

Figure 12.3 illustrates the detailed toolchain used by IBCS. IBCS processes source code to generate necessary intermediate files (e.g., binary, assembly code), which are then utilized in two phases: identification and enforcement, to rewrite the assembly file. For dynamic analysis, we employ the Intel PIN binary instrumentation framework [101], while static taint analysis is conducted using LLVM-9. Additional analyses related to assembly code and rewriting are implemented through Python scripts, leveraging Binary Ninja APIs [2].

Chapter 13

IBCS Implementation

In this chapter, we explain how IBCS is implemented. Determining which data should be compartmentalized is challenging due to various factors that may affect this decision. Therefore, IBCS requires different types of analysis, which we discuss in detail. The enforcement phase is also complex; it involves more than just rewriting assembly code—it requires understanding the code's purpose. To address this, we propose several analyses targeting assembly code, including assembly lexical analysis to build the Assembly Syntax Tree (AsmST). This phase involves creating detailed compartments and rewriting assembly code to ensure secure compartmentalization. It also includes static verification to check the accuracy of the rewritten binary and runtime enforcement to manage compartments during program execution.

Section 13.1 covers the implementation of identification phase of IBCS design. Next, Section 13.2 presents how the enforcement phase is implemented.

13.1 Identification Phase

IBCS identifies potential targets to compartmentalize using static information from non-stripped object files, LLVM bitcode, and runtime information from an original binary.

13.1.1 Dynamic Taint Analysis

If application programmers have detailed knowledge about which code regions might be potentially exploitable, IBCS can utilize this information. However, without such specific knowledge to guide the IBCS framework's static analysis, it first requires a user to execute a binary. This execution helps to automatically identify sensitive code paths based on user input.

The motivation behind employing dynamic analysis is to monitor functions triggered by the user's input, assuming that these functions are likely to be accessed repeatedly. Dynamic taint analysis is a vital component of this process. It involves tracking any memory byte influenced by external inputs, such as data received through network sockets, and observing its propagation through the code to identify vulnerable data points. We apply a dynamic taint analysis framework, `libdft` [110], to trace the flow of tainted data from the user's

input and concurrently generate a list of sensitive functions activated during execution. This approach enhances our ability to secure potentially exploitable sections of the code during runtime.

13.1.2 Static Taint Analysis

We utilize dynamic taint analysis to identify functions that interact with external inputs, although this method is prone to false negatives. False negatives occur when potentially vulnerable tainted data exists, yet dynamic taint analysis fails to detect it and reports no vulnerabilities. Such discrepancies may arise if the conditions for taint propagation are not properly configured to trace flows when callee functions are invoked. However, it is important to note that static taint analysis also suffers from false negatives, as discussed in ConDySTA [244]. Figure 12.1 demonstrates that static taint analysis does not identify functions like `handle_directory_request` and `rio_read`.

To address the Over/Underapproximation Challenge (**OC**), we propose a hybrid approach that combines dynamic and static taint analysis to minimize false negatives, similar to the strategy outlined in ConDySTA [244]. Although any static taint analysis method can be integrated with **IBCS**, we specifically employ SUTURE [239].

SUTURE requires the creation of a user-generated configuration file that defines an entry function—a function invoked with at least one argument containing tainted data [145]. To support this, we have developed a static variable-based taint analysis that identifies local variables of a function as potential taint sources and tracks their propagation to subsequent callee functions. This facilitates the automatic generation of the necessary configuration file needed for SUTURE. The need for this process may vary if a different static taint analysis method is employed.

13.1.3 Static Variable Offset Analysis

IBCS involves rewriting assembly code and does not rely on modifying the compiler toolchain like many previous works. Understanding each line of assembly code is crucial to rewriting the code accurately, identifying target variables, and determining appropriate modifications. Accessing generated debug information within a binary is essential for obtaining necessary details (e.g., using `objdump --dwarf=info ./input.out`). However, DWARF information, packed tightly for optimization purposes, needs proper parsing before it can be utilized effectively. To address this, **IBCS** leverages the open-source framework `pyelftools` [15] to construct a DWARF analysis that parses ELF data structures and generates disassembly information for all variables in a program.

13.1.4 Static Pointer Offset Tree Analysis

We utilize the results from static variable offset analysis to perform pointer offset analysis, aiming to accurately identify the correct register offset for pointers needing compartmentalization. For instance, in the example shown in Listing 7, the `url_decode` function is recognized as vulnerable due to a buffer overflow. The actual vulnerable data object, `req->filename`, originates from the `process` function and is passed to its nested callee functions. Within the local context of `url_decode`, `req->filename` is stored at a register offset of `-32(%rbp)`. Local assembly code rewriting in `url_decode` provides protection only within that function's scope. Despite local protection, the exploit demonstrated in Figure 7 targets the `%rip` register within the `process` function by exploiting a vulnerability in `url_decode`. Therefore, to effectively guard against this type of exploit and address the Identification Challenge (**IC**), it is crucial to correctly identify and address the register offset, as failure to do so would leave the exploit unprotected.

Algorithm 2 Pointer Offset Tree Analysis Initialization

```
1: Object 𝓑:                                                        ▷ 𝓑: binary
2:   𝓑 maps function's entry addresses to function objects 𝓕
3: Object 𝓕:                                                        ▷ 𝓕: function
4:   Methods:
5:     addr(): Returns the address of the function object 𝓕.
6:     callee_funs(): Returns a list of callee functions of the function object 𝓕.
7:     operands(): Returns a list of operands given this object is a callee function.
8: Object 𝓥:                                                        ▷ 𝓥: variable
9:   Methods:
10:    pointer(): Returns True if the variable is a pointer.
11:    offset(): Returns the register offset value for the variable.
12: Object 𝓣:                                                       ▷ 𝓣: function tree
13:   Methods:
14:     local_vars(): Returns the list of local variable of this function tree.
15:     fun(): Returns the function (𝓕) of this function tree.
16:
17: function ANALYZE_CALLEES(𝓑, 𝓕)                                  ▷ 𝓑: binary; 𝓕: function
18:     𝓕_node ← GET_NODE(𝓑[𝓕.addr()])                              ▷ 𝓕_node: function node
19:     𝓒_addrs ← DISASM(𝓑[𝓕.addr()]).callee_addrs                  ▷ 𝓒_addrs: callee addresses
20:     for each 𝓒_addr ∈ 𝓒_addrs do
21:         𝓒 ← DISASM(𝓑[𝓒_addr])                                   ▷ 𝓒: callee fun, type of 𝓕
22:         ADD_CALLEE(𝓕_node, 𝓒)
23:         𝓕_to_visit ← 𝓕_to_visit ∪ {𝓒}                           ▷ functions to visit: 𝓕_to_visit
24:     end for
25:     while 𝓕_to_visit ≠ empty do
26:         Let 𝓕 ← 𝓕_to_visit[end]
27:         𝓕_to_visit ← 𝓕_to_visit[1..end − 1]                     ▷ Remove the last element
28:         ANALYZE_CALLEES(𝓑, 𝓕)
29:     end while
30: end function
```

Algorithm 3 Generating Function and Pointer Trees

1: **function** GEN_FUN_TREES($\mathcal{B}, \mathcal{L}$) ▷ $\mathcal{B}$: binary; $\mathcal{L}$: list of functions
2: $\quad \mathcal{T}_{\text{fun}} \leftarrow$ new list ▷ $\mathcal{T}_{\text{fun}}$: function trees
3: $\quad$ **for each** $\mathcal{F} \in \mathcal{L}$ **do**
4: $\quad\quad \mathcal{R} \leftarrow \mathcal{F}$ ▷ $\mathcal{R}$: root function, type of $\mathcal{F}$
5: $\quad\quad \mathcal{T}_{\text{fun}} \leftarrow$ ANALYZE_CALLEES($\mathcal{B}, \mathcal{R}$.addr())
6: $\quad$ **end for**
7: $\quad$ **return** $\mathcal{T}_{\text{fun}}$
8: **end function**
9:
10: **function** ANALYZE_OPERANDS($\mathcal{F}, \mathcal{V}$) ▷ $\mathcal{F}$: function,$\mathcal{V}$: variable
11: $\quad$ **for each** $\mathcal{C} \in \mathcal{F}$.callee_funs() **do** ▷ $\mathcal{C}$: callee function
12: $\quad\quad$ **while** $\mathcal{C}$.operands() $\neq$ empty **do**
13: $\quad\quad\quad$ Let $\mathcal{O} \leftarrow \mathcal{C}$.operands[end]
14: $\quad\quad\quad$ $\mathcal{C}$.operands() $\leftarrow \mathcal{C}$.operands[1..end $-$ 1]
15: $\quad\quad\quad$ **if** $\mathcal{V}$.pointer() $\wedge$ $\mathcal{V}$.offset() $=$ $\mathcal{O}$.offset() **then**
16: $\quad\quad\quad\quad$ **return** *True*
17: $\quad\quad\quad$ **else if** $\mathcal{O}$.pointer() $\wedge$ $\mathcal{V}$.offset() $=$ $\mathcal{O}$.offset() **then**
18: $\quad\quad\quad\quad$ **return** *True*
19: $\quad\quad\quad$ **end if**
20: $\quad\quad$ **end while**
21: $\quad$ **end for**
22: $\quad$ **return** *False*
23: **end function**
24:
25: **function** GEN_PTR_OFFSET_TREES($\mathcal{T}_{\text{fun}}$) ▷ $\mathcal{T}_{\text{fun}}$: function trees
26: $\quad \mathcal{P}_{\text{fun}} \leftarrow$ new list ▷ $\mathcal{P}_{\text{fun}}$: pointer trees
27: $\quad$ **for each** $\mathcal{T} \in \mathcal{T}_{\text{fun}}$ **do**
28: $\quad\quad \mathcal{V}_{\text{local}} \leftarrow \mathcal{T}$.local_vars()
29: $\quad\quad$ **while** $\mathcal{V}_{\text{local}} \neq$ empty **do**
30: $\quad\quad\quad$ Let $\mathcal{V} \leftarrow \mathcal{V}_{\text{local}}$[end]
31: $\quad\quad\quad$ $\mathcal{V}_{\text{local}} \leftarrow \mathcal{V}_{\text{local}}$[1..end $-$ 1]
32: $\quad\quad\quad$ **if** ANALYZE_OPERANDS($\mathcal{T}$.fun(), $\mathcal{V}$) **then**
33: $\quad\quad\quad\quad \mathcal{R}_{\text{tree}} \leftarrow$ POP_PTR_TREE($\mathcal{T}, \mathcal{R}_{\text{tree}}$) ▷ $\mathcal{R}_{\text{tree}}$: root tree
34: $\quad\quad\quad\quad \mathcal{P}_{\text{fun}} \leftarrow \mathcal{P}_{\text{fun}} \cup \{\mathcal{R}_{\text{tree}}\}$
35: $\quad\quad\quad$ **end if**
36: $\quad\quad$ **end while**
37: $\quad$ **end for**
38: $\quad$ **return** $\mathcal{P}_{\text{fun}}$
39: **end function**
40:
41: **function** MAIN($\mathcal{B}, \mathcal{L}$) ▷ $\mathcal{B}$: binary; $\mathcal{L}$: list of functions
42: $\quad \mathcal{T}_{\text{fun}} \leftarrow$ GEN_FUN_TREES($\mathcal{B}, \mathcal{L}$) ▷ $\mathcal{T}_{\text{fun}}$: function trees
43: $\quad \mathcal{P}_{\text{fun}} \leftarrow$ GEN_PTR_OFFSET_TREES($\mathcal{T}_{\text{fun}}$) ▷ $\mathcal{P}_{\text{fun}}$: pointer trees
44: **end function**

Algorithms 2 and 3 present the methodologies we employ for pointer offset tree analysis. This analysis is structured into two phases for enhanced clarity:

1. **Constructing Function Call Trees**:
   - **Objective**: Establish the calling relationships among functions.
   - **Algorithm**: `Gen_Fun_Trees`.
   - **Input**: A binary and a list of functions from hybrid analysis.
   - **Process**:
     - Given a binary and a list of functions from hybrid analysis, this algorithm creates trees representing these relationships.
     - `Analyze_Callees` gathers addresses of callee functions for a given input function.
     - It iterates over these addresses, treating each associated function as a callee, and recursively analyzes subsequent callee functions reachable from the initial function to complete the tree generation.

2. **Generating Pointer Offset Trees**:
   - **Objective**: Analyze how pointer operations are utilized across different functions.
   - **Algorithm**: `Gen_Ptr_Offset_Trees`.
   - **Input**: The list of function trees generated in the first phase.
   - **Process**:
     - A function's local variable qualifies as the root node of a pointer offset tree based on two criteria:
       (a) If a local variable of a given function is a pointer and its offset matches the offset of a callee's operand, this indicates that the pointer is passed as an argument.
       (b) If the operand of a callee is a pointer and matches the offset of a local variable, it implies the local variable is passed as a reference to the callee.
     - Upon establishing the root node of the pointer offset tree ($\mathcal{R}_{\text{tree}}$), the tree is populated using `Pop_Ptr_Tree`, which constructs a tree of function calls based on the offset matching between function parameters and callee operands.
     - For brevity, the detailed implementation of this algorithm has been omitted.
     - Listing 8 presents an example of how the pointer offset tree analysis can narrow the accurate vulnerable target to be compartmentalized.

Listing 8 presents an example of how the pointer offset tree analysis can narrow the accurate vulnerable target to be compartmentalized.

Listing 8 Comparison of Function Offsets Before and After Pointer Offset Tree Analysis

Before:

```
1  Fun: url_decode      | Offset: -32
2  Fun: rio_readinitb   | Offset: -8
3  Fun: writen          | Offset: -56
4  Fun: parse_request   | Offset: -4144
5
6  Root Function: process, Register Offset: -560
7      |-- Function: parse_request, Register Offset: -4144
8          |-- Function: url_decode, Register Offset: -32
```

After:

```
1  Fun: process         | Offset: -560
2  Fun: writen          | Offset: -56
3  Fun: parse_request   | Offset: -1056
```

13.2 Enforcement Phase

Listing 9 Examples Demonstrating How Assembly Code Can Indirectly Obfuscate Its Purpose

```
1  # Assume %edi holds the base address of the 'req' structure
2  mov %edi, %rax  # Copy base address from %edi to %rax
3
4  # Set the 'offset' at 512 bytes from the start of 'req' to 0
5  movl $0, 512(%rax) # Write 0 to 'offset'
6
7  # Adjust %rax to directly reference the 'offset' field
8  addl $512, %rax    # Point %rax to 'offset'
9  movl $0, (%rax)    # Write 0 to 'offset'
```

13.2.1 Assembly Lexical Analysis

The first step of addressing the Enforcement Challenge (**EC**) and Assembly Challenge (**AC**) involves generating the Assembly Syntax Tree (**AsmST**) from the disassembled code. The **AsmST** provides an intuitive interface for rewriting code; however, its primary necessity stems from how certain assembly operations obfuscate the intent of instructions. For example, consider the code in Listing 7, where accessing the `offset` member in the `req` structure of `http_request` can be obfuscated, as shown in Listing 9. Both assembly code snippets perform the same function, but using an `add` instruction to access a member (e.g., `movl $0, (%rax)`) obscures the address being dereferenced in `(%rax)`. To address this issue, we convert each assembly code for a given function into Single-Static Assignment (SSA) form [50]. This process constructs an **AsmST** that provides a unique representation of each

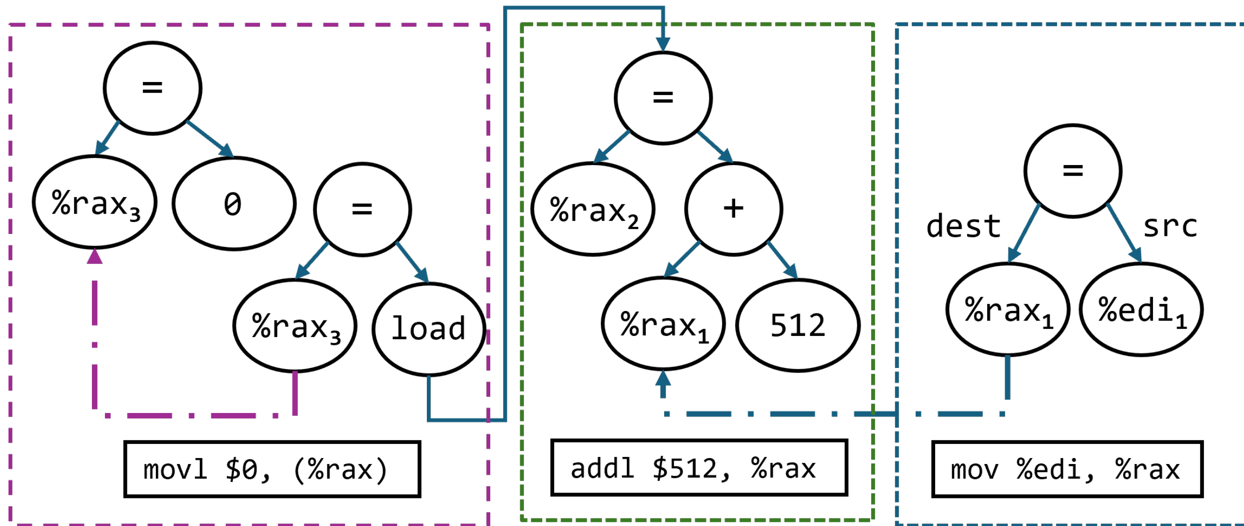


Figure 13.1: AsmST of the `movl $0, (%rax)` instruction

register at every assignment, aiding in de-obfuscating the assembly code and streamlining the rewriting process.

The design of AsmST is based on the Abstract Syntax Tree (AST) [14], which is used to represent the structural relationships within program nodes. Like the AST, edges in the AsmST denote relationships between nodes; however, the AsmST organizes these as `dest, op, src` and incorporates several node types, each representing different elements of assembly code:

- **SSA Register Nodes:** These nodes represent registers in SSA form, which helps maintain clarity about the versions of registers being used after each operation throughout the function's scope (e.g., a <$\%rax_1$> node indicates the first version of the register `%rax`).
- **Operator Nodes:** These nodes encapsulate the operators applied to operands in assembly instructions. The operator node without `dest` operand indicates a variety of operations that are used (e.g., dereferencing (`load`), sign extension (`sign`), etc.).
- **Literal Value Nodes:** Nodes for literal values represent immediate values used in operations, such as constants or offsets. These can be positive or negative.
- **Assignment Nodes:** The assignment nodes, denoted by `=`, assign the result of an operation (like an addition) to a variable or a register in SSA form.

For example, Figure 13.1 illustrates the AsmST for the code listed in Listing 9, demonstrating how `%rax` is adjusted before dereferencing the `offset` field. Initially, the assembly instruction `mov %edi, %rax` loads the base address of the `req` struct object into the SSA register $\%rax_1$. Subsequently, the base address in $\%rax_1$ is adjusted using the instruction `addl $512,`

%rax, and the updated address is then stored in the SSA register $\%\texttt{rax}_2$. Finally, $\%\texttt{rax}_3$ is dereferenced by the operation (load) node to update the offset field. The advantage of AsmST is evident as it allows tracing of the dependencies among the %rax registers back to the base address register. This traceability enables the inference that the instruction movl $0, (%rax) effectively functions as movl $0, 512(%rax), facilitating necessary assembly rewriting and optimization.

13.2.2 Fine-grained Compartment Creation

A fine-grained compartment is an isolated memory region designated to store data identified during the identification phase. After these compartments are created, a directory table is established, acting as a map for quick access to any compartment. Each variable requiring protection throughout the program code is assigned a unique offset, ensuring $O(1)$ access time.

Listing 10 Assembly Code Example Demonstrating Pointer Passing in Function Calls

```
1  process:
2  ...
3  leaq -560(%rbp), %rdx; -560(%rbp) := http_request req;
4  movq %rdx, %rsi; addr of req is passed as the arg
5  call parse_request; calls parse_request function
6  ...
7  leaq .LC61(%rip), %rax
8  movq %rax, -16(%rbp); char *msg = "Unknow Error";
9  ...
10 parse_request:
11 ...
12 movq %rsi, -4144(%rbp); req's addr is in -4144(%rbp)
13 movq -4144(%rbp), %rcx; addr of req is stored in %rcx
14 movq %rcx, %rsi; subsequently moved to the arg
15 call url_decode; calls url_decode
16 ...
17 leaq -2080(%rbp), %rcx; char buf[MAXLINE]
18 cmpl -16(%rbp), %eax; if (length == 0)
19 ...
20 url_decode:
21 movq %rsi, -32(%rbp); req's addr is in -32(%rbp)
22 movq -32(%rbp), %rax; stores req's addr in %rax
23 leaq 1(%rax), %rcx;   dereferences the req object
24 ... exploitation happens
```

For example, in Listing 10, the offset -16(%rbp) is utilized for the variables msg and length in the functions process and parse_request, respectively. In such duplicated stack offset cases across different functions, the variables msg and length are assigned respectively, with unique table offsets of 0 and 8. Table access is based on a granularity of 8 bytes because each table element holds a memory address, and on 64-bit systems, pointers are typically 8 bytes long. The table's base address is stored in the segment register (%gs). In the assembly code context, after loading the base address into an unused register (e.g., %r11) using the

rdgsbase instruction, users can easily access each compartment by adding the designated unique offset value.

13.2.3 Fine-grained Compartment Relocation

Listing 11 Example Displaying the Result of Assembly Rewriting

```
1  ; Macro Definitions
2  .macro mov_store_gs src, offset, value
3  mov \offset(%r11), %r11
4  movq \src, (%r11)  # 64-bit
5  xor %r11, %r11
6  .endm
7
8  .macro mov_load_gs dest, offset, value
9  mov \offset(%r11), %r11
10 movq (%r11), \dest  # 64-bit
11 xor %r11, %r11
12 .endm
13
14 ; Example Assembly Usage
15 movq    $0, %r11
16 test    %r11, %r11
17 jz      verifier_metadata_0
18 ; Replaced instruction metadata
19 movq    %rdx, -56(%rbp)
20 verifier_metadata_0:
21 rdgsbase %r11
22 mov_store_gs  %rdx, 0, 64
23 ; Further usage demonstrating the macros accessing moved data
24 rdgsbase %r11
25 mov_load_gs  %rax, 0, 64
26 # mov 0(%r11), %r11
27 # movq (%r11), %rax  # 64-bit
28 # xor   %r11, %r11
```

After creating compartments, **IBCS** modifies the assembly code to redirect variable access from default register offsets to designated compartments, addressing the Assembly Challenge (**AC**). Listing 11 depicts an example of such rewritten assembly code. Initially, the reserved register `%r11` is used to store the base address of the compartment table. This approach can be generalized to use a single reserved register, such as `%r15`, as suggested in [233].

A sequence of assembly instructions, `movq $0, %r11; test %r11, %r11; jz verifier_metadata_0`, is introduced to ensure that the original execution command (`movq %rdx, -56(%rbp)`) is bypassed. This metadata instruction is used for a post-condition verifier, which will be further discussed in Section 13.2.4. Metadata instruction is selectively created for instructions that store data to minimize code bloat. This verification is essential to ensure that vulnerabilities are addressed, although it may lead to functionality issues if the intended value is not correctly loaded.

Depending on the original purpose of the assembly instruction, it is rewritten with specific macros that replicate the same functionality but redirect to the compartment (e.g., `mov`

`tab_offset(%r11), %r11`) instead of using the original register offset. This careful rewriting ensures that the original intent of the assembly code is preserved while enhancing security by redirecting access to the compartmentalized memory regions.

13.2.4 Static Verification

In the spirit of LFI [233], IBCS provides static verifiers that ensure the input binary only contains supported instructions and that the modifications done to the input binary through assembly rewriting are correct. The concrete checks for the input and output verifiers are as follows. The input verifier scans every instruction in the binary and checks that the operation code is known. If an unknown operation code exists in the binary the input verification fails. In addition, the input verification takes note of operations that use reserved registers, namely `%r9`, `%r10`, and `%r11`. The output verifier looks for the pattern `test %r11, %r11` in the assembly code of the output binary which signals the beginning of a segment of the assembly code that was rewritten by IBCS. Upon detecting that pattern, the output verifier then collects information about the offset and registers used in the subsequent operations until the end of this rewritten region is detected. The register and offset information is then matched against the information collected during the assembly rewriting phase to ensure that it was rewritten correctly.

13.2.5 Runtime Enforcement

For the IBCS application to function correctly, the end-user must load the compartments table using the `LD_PRELOAD` linker environment variable (e.g., `LD_PRELOAD=./lib/table.so ./<input>.out`). Initially, the dynamic loader runs its constructor function, `create_table()`, establishing a compartment table and individual compartments. Each compartment is then linked to the table and loaded into the segment register using the `_writegsbase_u64()` function. The application can proceed as intended, with the compartments ready for access via the base address stored in the segment register. Upon completion, the destructor function `cleanup_table()` is executed to clean up the compartments and clear the table.

Chapter 14

IBCS Evaluation

In this chapter, we evaluate **IBCS**'s capabilities in terms of runtime overhead, security, and various practical considerations. Specifically, the aim of this chapter is to understand:

1. The runtime overhead introduced by **IBCS**'s compartmentalization techniques. **IBCS** relocation is based on redirecting access from the function's stack frame to an isolated memory region. As this redirection is based on changing the access target, it does not require complex trampoline insertion via binary rewriting. However, how much overhead does this cost?

2. The security benefits provided by **IBCS**, evaluated through adversarial analysis and case studies of CVEs. Although we are providing fine-grained isolation of data, what is the effectiveness of such isolation against actual security exploits?

3. The limitations and practical aspects of **IBCS**, including discussion on static analyses, assembly rewriting, and alternative compartmentalization techniques.

Section 14.1 discusses the runtime overhead of using **IBCS**. Next, in Section 14.2, we provide a security evaluation of **IBCS** using adversarial analysis. Section 14.3 discusses various aspects of **IBCS**. Finally, Section 14.4 summarizes our work on **IBCS**. Benchmarks were conducted on a 32-core Intel Xeon Silver 4110 2.10GHz CPU with 188GB of RAM.

14.1 Runtime Overhead Evaluation

To assess the effectiveness of **IBCS** on a larger scale, we evaluated the performance overhead on the Nginx web server. Due to the challenges associated with applying the full identification technique (further discussed in Section 14.3), we relied solely on dynamic taint analysis. This analysis identifies tainted paths, enabling us to patch all variables along these paths. Tainted paths refer to the routes Nginx follows while using ApacheBench [74] to measure performance.

Our approach involves compartmentalizing all local variables within the tainted paths, thus intra-procedurally isolating the data. This isolation excludes nested struct objects and global variables. Although we relocate the variables to a compartment and refer to intra-procedural use of such data within the compartment, we also maintain *shadow data*, where we store the

data at the intended stack offset. It is important to note that this stack offset is not utilized intra-procedurally. However, the shadow data is used as the argument when a variable is used in an inter-procedural context (e.g., passed as an argument to a callee function). Subsequently, after the callee function returns, we copy the updated shadow data back to the compartment.

Table 14.1: Nginx Performance Evaluations

Reqs.	File Sizes	128KB	256KB	512KB	1MB	2MB	4MB
2.5K	Orig.	0.170	0.214	0.282	0.440	0.816	1.356
	IBCS	0.178	0.223	0.286	0.450	0.837	1.358
	% diff.	4.48%	4.28%	1.64%	2.37%	2.51%	0.11%
5K	Orig.	0.162	0.212	0.294	0.427	0.7337	1.331
	IBCS	0.173	0.215	0.296	0.432	0.7343	1.354
	% diff.	7.19%	1.45%	0.74%	1.18%	0.08%	1.73%
10K	Orig.	0.155	0.188	0.259	0.422	0.723	1.425
	IBCS	0.178	0.200	0.263	0.426	0.732	1.436
	% diff.	15.03%	6.36%	1.52%	0.85%	1.21%	0.74%
20K	Orig.	0.144	0.187	0.288	0.421	0.738	1.365
	IBCS	0.160	0.194	0.290	0.26	0.746	1.483
	% diff.	10.55%	3.63%	0.77%	1.09%	1.03%	8.65%
30K	Orig.	0.147	0.193	0.264	0.424	0.749	1.355
	IBCS	0.164	0.198	0.278	0.424	0.747	1.367
	% diff.	12.00%	2.55%	5.11%	0.07%	-0.28%	0.86%

Table 14.1 compares performance between the original Nginx and **IBCS** binaries across file sizes ranging from 128KB to 4MB and request counts from 2,500 to 30,000, presenting average times per request, aggregated over 50 runs, with a standard deviation threshold of 1.2 for consistency. The percentage overhead illustrates the additional performance cost due to **IBCS**'s compartmentalization, such as initializing compartment tables and accessing compartments. Despite conventional wisdom suggesting faster data allocation on the stack than in dynamic memory, the minimal overhead observed in a large application like Nginx indicates that dynamic memory's impact is negligible.

The reason smaller file sizes, such as 128KB and 256KB, exhibit larger overheads is primarily due to the high volatility in measurement caused by the rapid completion of evaluations. For smaller files, the quick execution times result in a greater proportion of the total time consumed by the setup and teardown phases of the benchmarking process, amplifying any overheads associated with the **IBCS** modifications. As file sizes increase, the duration of the data transfer becomes more significant relative to these overheads, leading to a decrease in the relative overhead percentage, thereby validating that relocating data from the stack to a compartment is feasible as the performance difference between accessing static and dynamic memory becomes negligible.

Table 14.2: GNU `Coreutils` Rewriting Evaluations

Applications	LoCs	Variable Counts	Function Counts
`basename`	191	5	1
`pwd`	395	12	8
`wc`	875	39	7
`cp`	1227	28	7
`factor`	2662	216	42
`ls`	5309	154	91

The closest related work to **IBCS** is SCALPEL [179], as discussed in Section 3.8. SCALPEL is the only proposed technique among related works that can automatically identify, enforce, and provide finer-granularity memory protection similar to **IBCS**. However, SCALPEL utilizes a custom tagged memory architecture in RISC-V, which makes it a non-architecture-agnostic solution. Additionally, the unavailability of their implementation prevents a fair comparison. Therefore, we evaluated **IBCS** with respect to the original binary, which only leverages stack variables. This approach allows us to properly understand the cost of utilizing heap compartments and effectively measure the overhead of incorporating the protection scheme.

14.2 Security Evaluation

We evaluate the compartmentalization security benefits of **IBCS**. To bring the evaluation closer to real-world scenarios, we first present a case study of CVE bugs, illustrating the practical protective capabilities of **IBCS**. Subsequently, we employ open-sourced, minimalistic test cases that demonstrate the CVE attack scenarios for adversarial analysis [24]. This analysis also includes a larger example that simulates a potential real-life scenario, showcasing how **IBCS** safeguards against attacks. The adversarial analysis involves adopting an adversary's perspective to assess the protection scheme by identifying vulnerable data and exploiting it through respective attack scenarios. As noted in [24], the authors assert that the ideal standard for testing security is through adversarial analysis, which effectively simulates an intelligent attacker exploiting the target.

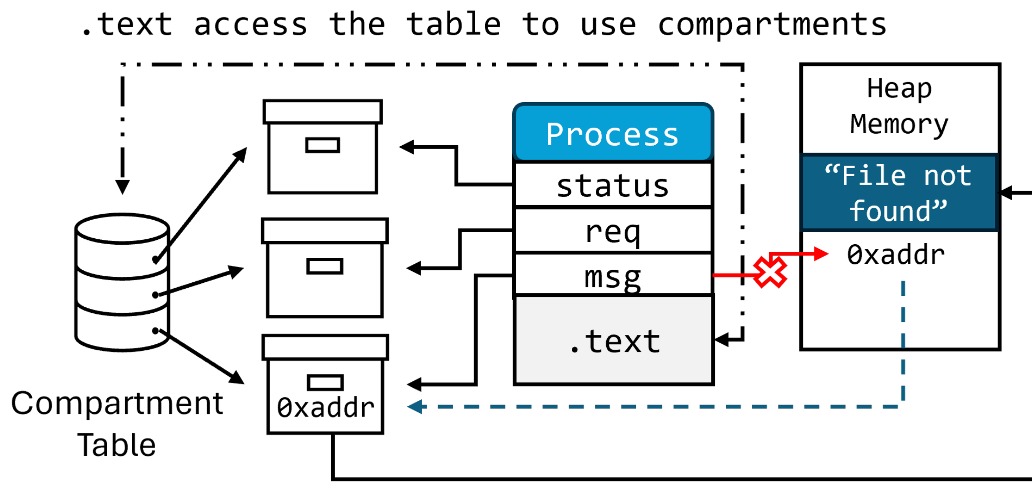


Figure 14.1: Visual Representation of **IBCS** Compartmentalization

Buffer Overflow: Buffer overflow occurs when memory buffers are accessed beyond their limits without adequate checks, allowing a malicious user to manipulate the program's memory. **IBCS** protects both stack- and heap-based overflows by isolating each stack data in the local function frame into its separate compartment, as shown in Figure 14.1. For example, in the function `process`, each local variable (such as `status` and `req`) is stored in its respective compartment during initialization. For heap-based variables such as `msg` (`char *msg = malloc(sizeof(char) * 100)`), static variable offset analysis identifies the register offset holding the dynamic memory allocation address, allowing **IBCS** to compartmentalize heap variables similarly. Examples of buffer overflow vulnerabilities include CVE-2013-2028 [149], CVE-2016-4450 [151], and CVE-2024-0684 [153].

Privilege Escalation: Privilege escalation involves gaining unauthorized access to system resources by elevating access levels, often exploiting vulnerabilities such as buffer overflows. An example is shown in Listing 7, where an attacker uses crafted input containing shellcode to overwrite the stack return address, redirecting execution to arbitrary code with elevated privileges. The attacker targets the callee function of `url_decode` to control the `process`. Mitigation typically involves assigning minimal privileges for system operations or isolating processes. **IBCS**, though focused on fine-grained data isolation, prevents privilege escalation by confining data within distinct compartments, maintaining strict isolation between an application's data segments.

Illegal Pointer Vulnerabilities: These vulnerabilities arise from improper handling or management of pointers within a program. Examples include the use of an uninitialized pointer, where a pointer is used without being initialized to a definite known value; null pointer dereference, occurring when a program attempts to read or write to a location pointed to by a pointer set to `NULL`; and dangling pointer, where a pointer continues to reference a memory location after the object it was pointing to has been freed. CVE-2016-4450 illustrates how

adversaries can exploit an illegal pointer to cause a denial of service (DoS). With IBCS, any attempt by an adversary to dereference such an illegal pointer results in a memory access exception, leading to the termination of the program's execution.

Code Injection Attack: Adversaries typically exploit known vulnerabilities to overwrite control data and redirect the program's control flow to their desired code locations [227]. With the widespread deployment of Data Execution Prevention (DEP, or W⊕X), adversaries bypass these protections by *reusing* existing code segments within the victim's program to perform arbitrary computations without injecting new malicious code [178, 213]. Recent research highlights the deployment of such code-reuse attacks in real-world scenarios [26, 54, 80], emphasizing their significant security impact. To demonstrate the effectiveness of IBCS against these attacks, we conduct adversarial analysis using a set of Linux binary exploitation tasks[1].

Similar to the previous discussions on contributions (CFR and SHROUD), as IBCS is an enforcement mitigation technique, the severity of ROP and DOP attacks has no impact on the performance overhead of IBCS.

14.3 Discussion

IBCS utilizes a variety of analyses to identify vulnerable data at the variable level automatically and compartmentalizes each piece of data to create isolation that is both quick and efficient to access. Despite IBCS's ability to patch an arbitrarily large number of variables (e.g., Nginx), several aspects still require further discussion. This section explores the potential limitations of the static analyses used in the IBCS identification phase, our rationale for rewriting assembly code, and an alternative coarse-grained compartmentalization technique that achieves function-level compartmentalization by leveraging Intel MPK.

Limitations on Static Analyses. The primary goal of IBCS is to fully automate both the identification and enforcement phases of compartmentalization. This automation involves complex tasks such as decoding the ELF binary, conducting static taint analysis, and generating pointer offset trees for all functions. These tasks become critical when pointers across the entire program need patching, not just within individual functions. A major limitation of this approach is evident in large-scale benchmarks (e.g., Nginx), where static inter-procedural analysis—which disassembles the binary to construct caller-to-callee graphs for all functions—encounters an explosion of parameter paths.

For future work, enhancing the scalability of the IBCS identification process may involve refining the pointer offset tree analysis to focus solely on assembly code, thereby bypassing the need to analyze the entire binary. This approach is promising due to several consistent aspects of assembly code: the linear execution of assembly instructions, the consistent use

[1] https://github.com/xairy/easy-linux-pwn

of argument registers to store parameters before a callee function is called, and the setup of parameters after the function's prologue. By leveraging these elements, we could redesign the algorithm for intra-procedural analysis, linking results from each function to build comprehensive function call trees without performing inter-procedural binary analysis. These trees would facilitate generating more precise and scalable pointer offset trees. Exploring the applicability of existing static taint analysis techniques represents another avenue for future research, especially given the challenges we encountered with using SUTURE for large applications.

Assembly Rewriting Discussion. Rewriting assembly code offers significant advantages for software compartmentalization and security compared to modifying the LLVM compiler [125]. Assembly rewriting enables direct interaction with low-level CPU-specific registers, which is difficult with LLVM IR due to its lack of direct register representation until the LLVM back-end process.

Challenges in modifying the LLVM back-end include managing complex structures due to alignment requirements and maintaining compatibility with existing optimizations. Such modifications can lead to incorrect code generation. Using non-stripped binaries, including DWARF information detailing variables and their offsets, facilitates precise assembly rewriting. This approach, especially with hybrid taint analysis, effectively supports assembly modifications for compartmentalization.

However, direct modifications using LLVM are constrained by their dependence on a specific compiler toolchain often tied to a particular version of LLVM. In contrast, direct asssembly rewriting provides greater specificity, reduces maintenance overhead, and is more adaptable to architecture-specific modifications. There is a potential for assembly rewriting to provide reasonable security while allowing lower performance overhead, as also demonstrated by LFI [233], enabling low-performance overhead software fault isolation (SFI). Directly rewriting assembly also facilitates the incorporation of hardware-specific extensions, potentially reducing performance and memory overheads.

Coarse-grained Compartment Binary Rewriting. We explored an alternative method for function-level compartmentalization using Intel MPK. This method involves assigning target functions to a custom section with attributes, exemplified by `void foo() __attribute __((section(".isolate_target")))`. A custom linker script ensures this section aligns at a 4096-byte memory address, as indicated by `.text BLOCK(4096) : ALIGN(4096)`. The configuration places the compartment section after `.fini` but before `.eh_frame_hdr`, maintaining functionality, as demonstrated by `.isolated_target : AT(__fini_end) {...} INSERT BEFORE .eh_frame_hdr`. This setup enables function protection using Intel MPK.

Additionally, the binary rewriting tool `e9patch` [62] is used to insert trampoline functions into the binary. To optimize patching coverage, we employ the `-finstrument-functions` compiler flag to insert profiling functions at all functions' entry and exit points. This strategy allows `e9patch` to circumvent patching complex instructions such as `ret` and instead patch the profiling functions, thereby facilitating the insertion of trampolines in larger ap-

plications like Nginx. These trampoline functions leverage the non-privileged instruction `WRPKRU` to modify the protection key rights register (`PKRU`), enabling or disabling function compartmentalization.

14.4 Summary

As a fourth contribution of this dissertation, we present the Input-Based Compartmentalization System (IBCS), a novel approach that enforces the principle of least privilege by decomposing software into isolated units with restricted capabilities. IBCS addresses the challenge of identifying compartmentalization targets by utilizing a dynamic graph generated from user inputs, integrated with static analysis to explore uncharted paths. This integration allows for accurate identification of vulnerable data targets and automated compartmentalization, significantly reducing underapproximation in enforcement. By dynamically adapting to user inputs, IBCS ensures comprehensive enforcement of compartmentalization targets while minimizing the performance penalties often associated with traditional methods.

Recent advancements in CPU technology, such as ARM's Pointer Authentication Code (PAC) and Memory Tagging Extensions (MTE), Intel's Secure Guard Extensions (SGX) and Memory Protection Key (MPK), and the Capability Hardware Enhanced RISC Instructions (CHERI), have supported more efficient compartmentalization. However, many existing techniques still require developer annotations, increasing the Trusted Computing Base (TCB) and putting pressure on developers to balance security against performance.

IBCS overcomes these challenges by dynamically adjusting to user inputs and leveraging a hybrid taint analysis graph to adapt and expand its analysis. This continuous adaptation ensures that even non-repeated user inputs contribute to a comprehensive understanding and enforcement of compartmentalization targets. We demonstrate the effectiveness of IBCS by analyzing a range of applications, from a `tiny` web server to GNU `Coreutils` to Nginx, and conducting adversarial analysis to showcase its security effectiveness.

Chapter 15

XFI: x86-based Fault Isolation

IBCS provided an automatic way to enforce fine-grained compartmentalization. However, process-level isolation was beyond its scope. Process-level isolation is a way to implement multiple *protection domains* within a single process. In the context of a system, a protection domain refers to different components of an operating system, each with its capabilities depending on its role in the system [122]. Having multiple protection domains is essential for computer systems security because separating each process ensures that errors in one process do not affect other processes, confining the errors to its own space and limiting potential damage from an exploit. These protection domains can be designated by leveraging hardware features. However, relying on hardware can limit the applicability of the protection scheme.

In contrast to relying on hardware features to set domains, a software-based approach can alleviate many of the aforementioned difficulties. Various methods exist to realize protection domains in a target system. Techniques include using Virtual Machines [203, 245] and Containerization [220, 241]. A virtual machine (VM) is a logical process that emulates hardware to mirror the functionality of a physical computer. Therefore, VM-based process isolation executes the target potentially vulnerable component(s) inside the virtual machine, creating an isolated environment from other components. Containerization is a lightweight alternative to virtualization. Instead of emulating hardware, it leverages kernel capabilities, namespaces, and control groups to run multiple virtual environments on top of a shared operating system kernel. Although these two techniques are easily deployable without requiring invasive changes to the application, they suffer from very high performance overhead due to the expensive context switching required in addition to .

Language-based Isolation [83, 85, 154] is another approach that relies on the safety of the programming language's type system. In other words, only the type of the data and the operation to be performed on that data are relevant, removing the need to worry about how the data resides in memory. For instance, WebAssembly (Wasm) is a good example of language-based isolation. However, this method requires an intensive software engineering effort, as programmers may need to rewrite legacy code to support such features. Compatibility challenges along with the complexity that comes with applying this technique can limit its applicability.

Software-based Fault Isolation (SFI) is an alternate approach that instruments checks at potentially dangerous instructions (e.g., memory access and control-flow transfer) to ensure their access is restricted within its own protection domain (i.e., within the same process).

Although implementing an efficient SFI mechanism in a RISC architecture is feasible [215, 233], achieving this in a CISC architecture can be challenging due to the nature of variable-sized instructions that require techniques such as `nop` padding [147, 234], which greatly increases the overhead for both performance and code size.

As the final contribution of this dissertation, we propose **XFI**, an x86-based SFI system that provides process-level isolation by enforcing both data-access and control-flow policies efficiently in a single address space without requiring invasive modifications to the compiler toolchain or complex instrumentation. It is also portable across any x86 machine, and it avoids requiring expensive context switches that exist in other techniques mentioned above. Unlike previous works, **XFI** rewrites assembly code, eliminating the need to modify a complex compiler's backend, and instruments necessary guards that can be verified to ensure the SFI is properly enforced.

First, the challenges associated with implementing **XFI** are discussed in Section 15.1. Subsequently, the design overview of **XFI** is presented in Section 15.2. Finally, various design considerations for **XFI** are elaborated upon in Section 15.3.

15.1 Challenges

- **Identification Challenge (IC)**: Similar to the compartmentalization scheme proposed in the **IBCS**, an important challenge in implementing the SFI scheme involves deciding which parts of the program to isolate to ensure faults do not occur. For instance, in the case of **IBCS**, we decided to leverage user input to determine what data is being used for compartmentalization. However, unlike compartmentalization, which aims to confine malicious effects within the compartment, SFI schemes need to instrument different components to perform necessary checks to ensure proper access. Therefore, properly identifying the SFI protection scope depending on the enforcement policy is an important challenge.

- **Guard Challenge (GC)**: As previously mentioned, SFI is a software instrumentation technique to ensure proper access, whether it pertains to data access or transferring control flow from one location in the code to another. In other words, SFI schemes insert "software guards" to guarantee that the addresses of these actions are within the designated, allowed region. This also means that how these guards are implemented and instrumented in a way that ensures minimal performance overhead is critical to achieving an efficient SFI scheme.

- **Compatibility Challenge**: Previously proposed SFI schemes leverage various techniques, such as rewriting a binary to instrument checks or using hardware features like segmented memory, to alleviate the complexity of implementing their respective SFI policies. However, leveraging such features raises compatibility concerns as it may re-

quire modifying the compiler toolchain, which increases the maintainability challenge and could restrict the applicability of the SFI scheme. Therefore, proposing an SFI scheme with these considerations in mind is an important challenge to tackle.

15.2 XFI Design Overview

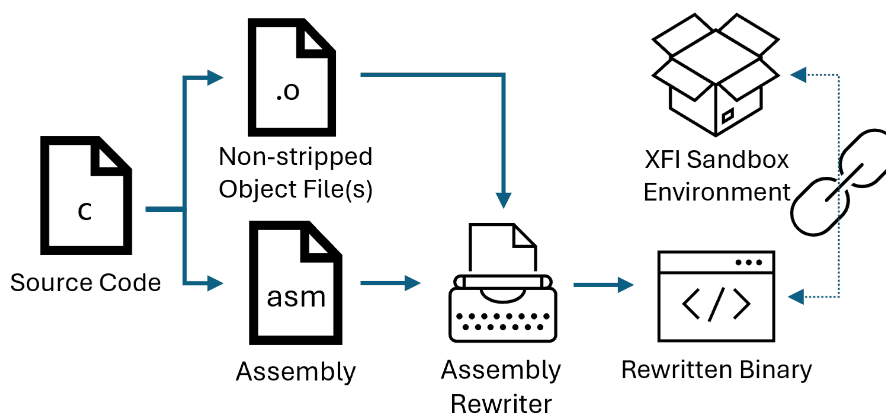


Figure 15.1: **XFI** Design Overview

XFI is a process-level SFI system for the x86 architecture that is able to provide both data-access and control-flow enforcement policies in an efficient manner. Figure 15.1 shows the summary of **XFI**. Given a source code, **XFI** generates an object file and assembly code. The object file is used to analyze and obtain the necessary information regarding different sections of the program to sandbox (e.g., `.data`), which will be used as vital information to properly rewrite the assembly code. Afterward, the results of the object file analysis and the assembly code are passed as inputs to the assembly rewriter. This rewriter is tasked with rewriting the assembly code to instrument necessary SFI policies by inserting appropriate assembly instructions. At last, a sandbox environment will be hooked with the rewritten binary to isolate it and enforce the required SFI policies.

Figure 15.2 presents the detailed toolchain used by **XFI**. First, **XFI** takes in the source code to generate an object file and assembly code using a musl compiler. We add a small snippet of code inside musl's C runtime startup file (`crt1.c`) that can calculate the base address of the executable during runtime and initializes a return address scratchpad that will be used to enforce the control-flow policy of **XFI**. We statically analyze the object file to obtain the necessary information that will be used to create a sandbox, leveraging Binary Ninja APIs [2]. Lastly, we rewrite the assembly code to enforce the necessary SFI policies depending on the type of instruction using the input assembly code.

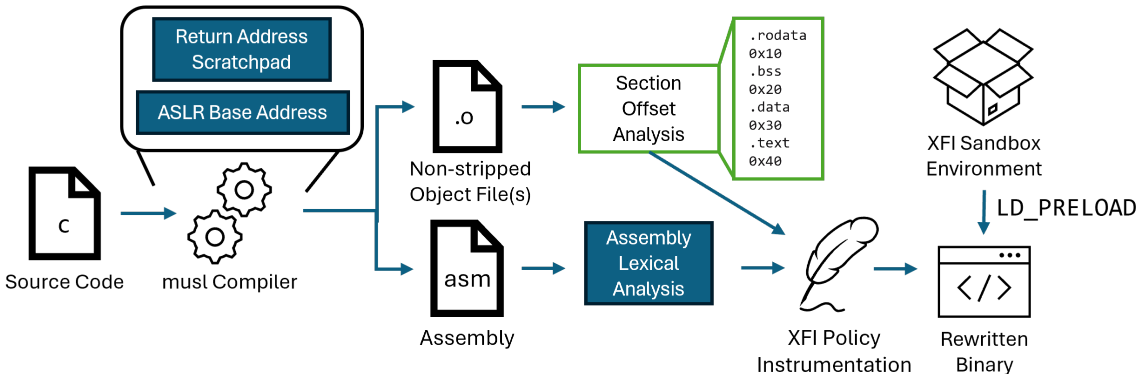


Figure 15.2: **XFI** Toolchain

15.3 Design Considerations for XFI

In contrast to prior SFI works, **XFI** rewrites an assembly code to instrument necessary SFI policies and ensure the isolation of the process. In this section, we discuss potential design choices and why we decided to settle with using the assembly code.

1. *Source Code*: Although source code allows the programmer to have a high-level view of the target program, implementing the SFI scheme at this level is not necessarily an ideal choice. The basis of any SFI scheme is to identify potentially unsafe instructions, which could be either memory accesses or control-flow transfers. Identifying such unsafe instructions at the source code level is extremely challenging. For instance, with unsafe memory access, a programmer must determine which data are dynamically allocated and manually insert checks for these data to enforce necessary guards.

   Furthermore, the programmer must parse through the source code to identify the usage of function pointers and virtual functions to infer potential indirect transfers and ensure these transfers are safe. Implementing such a protection scheme at the source-code level may also be inefficient with respect to performance overhead, and maintainability would be extremely difficult. The instrumentation may need to be frequently updated to accommodate changes to the source code.

2. *Intermediate Representation (IR)*: An intermediate representation (IR) is a code used internally by a compiler to represent input source code. For instance, in the LLVM compiler framework [125], IR is a high-level assembly language that is portable to multiple architecture targets and is an ideal choice for various compiler optimizations. This flexibility allows developers to instrument necessary techniques at a lower level than the source code before the program is fully compiled.

   Therefore, modifying LLVM IR is a popular technique used by other SFI systems (e.g., NaCl [234]) to instrument necessary policies to isolate the program. The LLVM

IR offers several advantages, such as a human-readable assembly-like syntax and the utilization of Static Single Assignment (SSA) form [50], which simplifies the implementation of compiler optimization techniques. However, the biggest downside is the complexity of implementing such techniques, as it may require modifications to the compiler's back-end that translates the IR to the target architecture. Additionally, since LLVM is frequently updated, maintaining compatibility with different versions becomes extremely difficult, and developers must keep up with changes to various APIs. This challenge is compounded by the fact that implementing compiler techniques may result in thousands of lines of code.

3. *Assembly Code*: Therefore, building on our prior contribution, we decided to rewrite the assembly code for XFI, similar to the approach taken with IBCS. Although rewriting assembly code presents various challenges, as previously discussed in the context of IBCS, directly modifying the assembly code offers significant advantages. One of the primary benefits is the transparency it provides in terms of technique implementation. Additionally, techniques instrumented at the assembly level are straightforward to debug, as there are no abstractions obscuring the operation of the assembly code.

   Moreover, since assembly code is essentially the final version of the code before it is assembled into the binary, it grants developers fine-grained control to precisely instrument necessary techniques. This approach ensures consistency across different platforms, as the syntax of assembly code, although varying by architecture, remains relatively stable and is not frequently updated. This stability further simplifies the maintenance and application of security techniques such as SFI.

Chapter 16

XFI Implementation

In this chapter, we delve into the details of how **XFI** is implemented. We discuss the protection scope of **XFI** and how SFI policies are enforced by rewriting the assembly code. First, **XFI** requires a modification of the musl compiler to help obtain the necessary information within the assembly code. Next, the object file is statically analyzed to aid in developing a sandbox creation process that will be integrated with the target process. Lastly, the assembly code is analyzed to determine how particular assembly instructions need to be rewritten to enforce the necessary SFI policies, depending on what each instruction does.

Section 16.1 discusses the preparation phase of **XFI**, which includes information regarding the basic design behind **XFI** and the modifications needed to accommodate the **XFI** policy enforcement. Section 16.2 presents how the assembly code is rewritten to enforce **XFI** policies.

16.1 XFI Preparation

16.1.1 XFI Sandbox Scheme

Before discussing the implementation details of **XFI**, we first want to discuss how ”guards” work in **XFI**, which is an important part of enforcing any SFI policies.

Similar to the work of LFI [233], the basis of the **XFI** system revolves around the formula:

$$\texttt{xfi_sandbox_address} = \texttt{xfi_base_address} + \texttt{address_offset}$$

The address resulting from this calculation should always be a valid **XFI** sandbox address. If it is not, this discrepancy will be used to detect malicious activity.

For **XFI**, we reserve the following registers:

- `%r15`: holds the base address of the **XFI** sandbox.
- `%r14`: holds the ASLR base address.
- `%r13`: holds the target address to be sandboxed and the valid **XFI** sandbox address.

To simplify the sandbox address calculation, we set the top bit of the 64-bit address as the **XFI** base address based on the order of the process being sandboxed. For example, process 1's **XFI** base address would be `0x100000000000`, process 2's **XFI** base address would be `0x200000000000`, and so on. This allows the calculation of **XFI** sandbox addresses to be distinctive and easy to manage for different processes.

Furthermore, we additionally pack this base address with the `.text` section's end address, as **XFI** also needs to enforce a control-flow policy. This information is determined based on static analysis, which will be discussed in a later section. Therefore, if an input program's `.text` section ends at the offset `0xdfa2`, the **XFI** base address would be `0x10000000dfa2`.

Lastly, **XFI** is designed around dynamically compiled position-independent code. Therefore, we assume the assembly code uses the RIP-relative addressing mode. This mode allows instructions to access memory addresses relative to the current instruction pointer (`%rip`). Additionally, this approach is critical for creating an efficient application that helps in establishing an effective sandbox mechanism.

16.1.2 Compiler Modification

XFI leverages the address offset of the data used by each instruction to efficiently obtain the sandbox address. However, this is difficult to achieve in a traditional setting due to the ubiquitous use of the Address Space Layout Randomization (ASLR) technique [198]. ASLR randomly arranges the addresses of various parts of a process without significant modifications to the actual binary; instead, the operating system ensures diversification at execution time. In other words, ASLR executes the *same* binary in different ways by loading the binary into different locations in memory. Therefore, it is not possible to directly use the offset address from the process to calculate the sandbox base address. Instead, it is necessary to first obtain the actual offset that can be used for the calculation.

Listing 12 shows the inline assembly code snippet that is used to calculate the ASLR base address. The `call 1f` instruction is used to call the label `1` while searching in the forward (`f`) direction. The following instruction marks the position in the code with label `1`, where the instruction pops the return address from the stack into the `rdi` register, effectively retrieving the current address of the instruction pointer. The subsequent instruction stores this value into the local variable `addr`. Afterward, a `while` loop is used to traverse backward one page at a time until the ELF magic number (which appears at the very beginning of an ELF binary file) is found. When this value is found, it indicates that the ASLR base address has been located and is stored in the global variable `base_address`, which can be retrieved within the assembly code.

Next, to enable the control-flow policy of **XFI**, there is a need to prepare what we call a *return address scratchpad*, which will store the return addresses. The scratchpad is allocated using `malloc` with the specified size (1024 in our case), and the pointer to this scratchpad is stored

Listing 12 ASLR Base Address Calculation

```
// Declaration of the base address variable
unsigned long long base_address = 0;

void get_base_address() {
    unsigned long long addr;
    __asm__ __volatile__(
    "call 1f\n"                // Call next instruction to push the address on the stack
    "1: pop %%rdi\n"           // Pop the return address into RDI (current address)
    "movq %%rdi, %0\n"         // Move the current address into addr
    : "=r" (addr)
    :
    : "rdi"
    );

    // Align the address to the page boundary
    addr &= 0xfffffffffffff000;

    // Traverse backward to find the start of the ELF headers
    while (1) {
        if (*(unsigned int*)addr == 0x464c457f) { // Check for ELF magic number
            base_address = addr;
            break;
        }
        addr -= 0x1000; // Move back by one page
    }
}
```

in the global variable `ret_addr_scratchpad`. This allows access to the scratchpad within the assembly code. Initializing such a scratchpad is necessary due to the limited number of registers available in the x86 architecture. While it is possible to use a caller-saved designated register to achieve this, such an approach is not feasible with a long function call chain, as each function call would require saving and restoring the register's value, adding significant overhead and complexity. Using a callee-saved register would also pose challenges, as the callee must preserve its value across function calls, further complicating the function call sequence.

All in all, we opted to modify the startup code of the C runtime (`crt1.c`), which is responsible for setting up the necessary environment before the main function of a program is called. By doing so, we avoid the need to modify the compiler toolchain and only need to add a few instrumentation codes in the startup file to provide the necessary information in the context of assembly code. This approach simplifies the implementation and integration process, as it minimizes changes to the existing compilation workflow and leverages the initialization phase to set up the required environment.

Listing 13 Return Address Scratchpad Implementation and Modified Startup Code

```
1  // Declaration of the shadow stack pointer
2  uint64_t* ret_addr_scratchpad;
3
4  void init_scratchpad(size_t size) {
5      ret_addr_scratchpad = (uint64_t*)malloc(size * sizeof(uint64_t));
6      if (!ret_addr_scratchpad) {
7          // Handle allocation failure
8          exit(1);
9      }
10 }
11 ...
12 void _start_c(long *p)
13 {
14     // Call the function to get and store the base address
15     get_base_address();
16     printf("Base address: %p\n", base_address);
17
18     // Initialize the scratchpad with enough space for 1024 return addresses
19     init_scratchpad(1024);
20     printf("Return address scratchpad: %p\n", ret_addr_scratchpad);
21
22     int argc = p[0];
23     char **argv = (void *)(p+1);
24     __libc_start_main(main, argc, argv, _init, _fini, 0);
25 }
```

16.1.3 Section Offset Analysis

Next, we leverage the Binary Ninja APIs [2] to disassemble the object file generated from the default assembly code. This allows us to identify the offset values of sections of interest, such as `.bss`, `.data`, and `.rodata`, which will be used for sandboxing purposes. Additionally, we extract symbols within each section to assist in determining which assembly instructions need to be rewritten. The focus on these specific sections is due to **XFI** being a process-level protection scheme, necessitating detailed analysis and instrumentation of data-related sections to ensure proper enforcement of **XFI** policies.

16.1.4 Sandbox Creation

Finally, a sandbox is created for a process by allocating a new memory region. We initialize a sandbox environment for a process by leveraging the `LD_PRELOAD` technique, which can *preload* a shared library and execute necessary constructor and destructor functions (if they exist) before the process starts its normal execution. Rather than migrating the process into the newly created region, we map specific sections that need to be enforced into the

sandbox. We opted for this approach to implement our enforcement policies efficiently and reduce the complexity that comes with migrating the entire code. Lastly, we store the process's respective **XFI** base address in the segment register (`%gs`) to be accessed in the assembly code.

16.2 XFI Policy Enforcement

16.2.1 Data-access Enforcement Policy

Data access in the context of assembly instructions refers to instructions like the following. Please note that this is not an exhaustive list of data access instructions, just a few prominent ones commonly used:

- `movq %rax, data(%rip)`: moves the value in the `%rax` register to the memory location specified by `data(%rip)`.
- `movq data(%rip), %rdx`: moves the value from the memory location specified by `data(%rip)` into the `%rdx` register.
- `leaq function(%rip), %rax`: computes the address of `function(%rip)` and stores it in the `%rax` register. This instruction does not access memory but rather calculates the address and places it in `%rax`.

Listing 14 illustrates various macros employed to enforce data access policies for certain instructions. Depending on the type of instruction (e.g., `mov` or `lea`), a different macro must be utilized, as the intermediate process to compute the **XFI** sandbox address may vary. At the beginning, the `.extern` directive is used to declare an external symbol `base_address`, which is initialized by musl (please refer to Section 16.1.2). Additionally, `mask` is defined per process and used as a mechanism to derive the **XFI** base address.

In the case of the `mov` instruction, the `mov_load_xfi` and `mov_store_xfi` macros are used depending on the context. The only difference between these two macros is whether the instruction is loading data (op) from memory (addr) or storing data (op) to memory (addr). The `value` parameter determines the size of the data to be moved (8, 16, 32, or 64 bits), as we need to take into consideration the different prefixes of Intel assembly code.

Here is the detailed information on what each instruction does to compute the **XFI** sandbox address.

1. `rdgsbase %r15`: reads the segment register and stores the value into `%r15`.

Listing 14 XFI Data-access Enforcement Policy

```
.section .data
    .extern base_address
    mask: .quad 0x100000000000  # Mask to keep only the topmost bit

...

.macro mov_load_xfi addr, op, value
    rdgsbase %r15
    andq    mask(%rip), %r15  # Use the fixed mask to keep only the topmost bit
    leaq    \\addr, %r14
    movq    base_address(%rip), %r13
    subq    %r13, %r14
    addq    %r15, %r14
    .if \\value == 8
    movb (%r14), \\op  # 8-bit
    .elseif \\value == 16
    movw (%r14), \\op  # 16-bit
    .elseif \\value == 32
    movl (%r14), \\op  # 32-bit
    .elseif \\value == 64
    movq (%r14), \\op  # 64-bit
    .endif
.endm

.macro mov_store_xfi addr, op, value
    rdgsbase %r15
    andq    mask(%rip), %r15  # Use the fixed mask to keep only the topmost bit
    leaq    \\addr, %r14
    movq    base_address(%rip), %r13
    subq    %r13, %r14
    addq    %r15, %r14
    .if \\value == 8
    movb \\op, (%r14)  # 8-bit
    .elseif \\value == 16
    movw \\op, (%r14)  # 16-bit
    .elseif \\value == 32
    movl \\op, (%r14)  # 32-bit
    .elseif \\value == 64
    movq \\op, (%r14)  # 64-bit
    .endif
.endm
```

2. `andq mask(%rip), %r15`: performs a bitwise `AND` operation between the **XFI** base address and a pre-defined `mask` value located at `mask(%rip)` to keep only the topmost bit in order to compute the **XFI** sandbox address.

3. `leaq addr, %r14`: loads the target address into the register `%r14` without accessing the memory.

4. `movq base_address(%rip), %r13`: loads the ASLR base address into the `%r13` register.
5. `subq %r13, %r14`: computes the address offset by subtracting the ASLR base address from the target address to obtain the static address offset.
6. `addq %r15, %r14`: generates the **XFI** sandbox address by adding the static address offset to the masked **XFI** base address.
7. Afterwards, depending on specific conditional blocks (based on the `value` provided in the macro), perform the necessary instruction to either store the data into the sandbox or load the data from the sandbox.

16.2.2 Control-flow Enforcement Policy

The control-flow enforcement policy must ensure that control transfers by the sandboxed code remain within the designated sandbox region or are restricted only to vetted external addresses. Although conceptually similar to data-access enforcement policies in ensuring that any transfers stay within the sandbox, it is more complex. For example, with the instruction `jmp %rax`, if `%rax` contains an address that bypasses the enforcement mechanism, this could lead to a violation of the data-access enforcement policy, as the **XFI** sandbox address is not properly computed.

In the context of RISC (Reduced Instruction Set Computer) architecture (e.g., ARM64), it is not possible for a branch instruction to jump into the middle of an instruction. This is due to the fixed instruction length (commonly 4 bytes). Subsequently, having a fixed instruction length allows instructions to be aligned on multiples of the 4 (`0x4`, `0x8`, etc.), ensuring that the CPU will only jump to addresses that are properly aligned and preventing jumps into the middle of an instruction.

However, in CISC (Complex Instruction Set Computer) architectures (e.g., x86), instructions can vary in length, ranging from as short as 1 byte to as long as 15 bytes. As a result, instruction alignment is not as strict as in RISC architectures (i.e., instructions can be at `0x100` and `0x103`, but `jmp 0x101` is possible). This loose alignment allows control-flow transfer instructions to land in the middle of another instruction, potentially leading to problems such as the bypassing of enforcement mechanisms. As such, many previous works enforcing control-flow policies on x86 architecture require forced alignment of instructions using the `nop` padding technique (aligned-chunk enforcement) [234], rule out this behavior and attempt to protect by preventing jumps to the middle of instructions [210], or augment different techniques such as Control-Flow Integrity [3].

Here are the control-flow transfer assembly instructions that need to be enforced:

- `jmpq %rax`: performs an unconditional jump to the address specified in the `%rax` register. This changes the flow of execution to the target address stored in `%rax`.
- `callq %rax`: calls a procedure at the address specified in the `%rax` register.
- `ret`: pops the return address from the stack and jumps to it, returning control to the calling function.

Listing 15 presents macros that we use to enforce control-flow transfer instructions. `ctrl_flow_xfi` is used to enforce control-flow policy for indirect `jmp` and `call` instructions, whereas `ret_xfi` is specialized for the `ret` instruction. Along with other symbols at the top, we define an additional external symbol `ret_addr_scratchpad`, which will hold the pointer address to the scratchpad initialized during the program's startup (please refer to Section 16.1.2). This symbol will be referred to as a way to enforce the policy for the `ret` instruction.

In contrast to the data-access enforcement macros which replace the original instruction with the macro (e.g., `mov_load_xfi func_ptr(%rip), %rdx, 64 # movq func_ptr(%rip), %rdx`), the control-flow enforcement macro is used as a guard immediately before the control-flow transfer instruction occurs. For instance, it would look like this:

```
ctrl_flow_xfi %rdx, 0
call *%rdx
```

```
ctrl_flow_xfi data(%rip), 1
jmp *data(%rip)
```

```
ret_xfi
ret
```

The `ctrl_flow_xfi` macro is designed to enforce control-flow policies by ensuring that the target address of an indirect control-flow instruction (e.g., `jmp`, `call`) is within the designated sandbox region. It takes the arguments `addr`, which represents the address, and `mem`, which differentiates whether we are handling an address stored in a register or direct memory access. Here is a detailed step-by-step description of what each instruction does:

1. `rdgsbase %r15`: reads the segment register and stores the value into `%r15`.
2. `xorq mask(%rip), %r15`: performs a bitwise `XOR` operation between the value in `%r15` and the fixed mask located at `mask(%rip)`. This operation retains only the bottommost bit of `%r15`, which also represents the end address of the `.text` section.

Listing 15 XFI Control-flow Enforcement Policy

```
1  .section .data
2      .extern base_address
3      .extern ret_addr_scratchpad
4      mask: .quad 0x100000000000  # Mask to keep only the topmost bit
5  ...
6  .macro ctrl_flow_xfi addr, mem
7      rdgsbase %r15
8      xorq    mask(%rip), %r15  # Use the fixed mask to keep only the bottommost bit
9      .if \\mem    # mem indicates whether addr is memory or reg
10     leaq    \\addr, %r14
11     .else
12     movq    \\addr, %r14
13     .endif
14     movq    base_address(%rip), %r13
15     subq    %r13, %r14
16     cmpq    %r15, %r14    # Compare %r14 with %r15
17     # Conditional jump if %r14 is not less than %r15 to the interrupt
18     jge     trigger_interrupt
19 .endm
20
21 .macro ret_xfi
22     rdgsbase %r15
23     xorq    mask(%rip), %r15  # Use the fixed mask to keep only the bottommost bit
24     movq    (%rsp), %r14
25     movq    base_address(%rip), %r13
26     subq    %r13, %r14
27     cmpq    %r15, %r14    # Compare %r14 with %r15
28     jge     trigger_interrupt
29 .endm
30 ...
31 # Control-flow enforcement interrupt
32 trigger_interrupt:
33     cmpq (%rsp), %r11
34     je safe_exec
35     int3 # If the scratchpad fails, then trace trap to indicate the policy violation
36
37 safe_exec:
38     ret
39 .Letext0:
40 .section .debug_info,"",@progbits
```

3. `.if mem`: checks whether the address is a memory address. Depending on whether the register holds the address or not, it uses either `leaq` or `movq` to load the address into the `%r14` register.

4. `movq base_address(%rip), %r13`: loads the ASLR base address into the `%r13` register.

5. `subq %r13, %r14`: computes the address offset by subtracting the ASLR base address from the target address to obtain the static address offset.

6. `cmpq %r15, %r14`: compares the static offset stored in `%r14` with the text section boundary defined in `%r15` to ensure that the address does not transfer outside of the sandbox.

7. `jge trigger_interrupt`: if the static offset (%r14) is greater than the sandbox region, then it jumps to the `trigger_interrupt` label to indicate a violation of the control-flow policy.

The `ret_xfi` macro is designed to enforce control-flow integrity for the `ret` instruction by ensuring that the return address, which is obtained using the `%rsp` register, is within the allowed sandbox region. This macro is similar to `ctrl_flow_xfi` but is specifically tailored for the `ret` instruction.

When a function returns, the `ret` instruction pops the return address from the stack, pointed to by the `%rsp` register, and transfers control to that address. By verifying the return address before allowing the `ret` instruction to proceed, the `ret_xfi` macro ensures that the return address falls within the `.text` section. However, a limitation of this macro is that it fails to handle cases where a program may need to return to internal functions (e.g., `__funs_on_exit`). Many internal functions are located in the heap, where the starting address differs from that of the stack (`0x7fff...` compared to `0x5555...`). Consequently, using the macro in its current form may result in false positives, indicating violations where there are none.

Listing 16 XFI Return Address Scratchpad

```
1  .macro store_scratchpad
2      movq    ret_addr_scratchpad(%rip), %r10  # Load scratchpad pointer
3      movq    (%rsp), %r11                     # Save caller return address
4      movq    %r11, (%r10)                     # Store return address in scratchpad
5      addq    $8, %r10                         # Increment scratchpad pointer
6      movq    %r10, ret_addr_scratchpad(%rip)  # Update global scratchpad pointer
7  .endm
8
9  .macro load_scratchpad
10     subq    $8, ret_addr_scratchpad(%rip)    # Decrement scratchpad pointer
11     movq    ret_addr_scratchpad(%rip), %r10  # Load updated pointer
12     movq    (%r10), %r11                     # Retrieve return address from scratchpad
13 .endm
```

To handle such cases, we must make use of the *return address scratchpad*, which will hold the return address of the most recent caller function. Listing 16 presents macros for the return address scratchpad. In summary, the `store_scratchpad` macro saves the return address of the caller function to a separate memory location by copying the return address

from the top of the stack (`%rsp`) into the appropriate scratchpad location. Conversely, the `load_scratchpad` macro is used to retrieve the address stored in the scratchpad.

Listing 17 Complete Enforcement of **XFI** Policies

```
main:
...
.cfi_startproc
store_scratchpad
pushq %rbp
...
mov_store_xfi func_ptr(%rip), %rax, 64 # movq %rax, func_ptr(%rip)
...
mov_load_xfi func_ptr(%rip), %rdx, 64 # movq func_ptr(%rip), %rdx
...
ctrl_flow_xfi %rdx, 0
call *%rdx
...
load_scratchpad
ret_xfi
ret
.cfi_endproc
```

Listing 17 demonstrates the comprehensive usage of all macros to enforce **XFI** policies within a function. The `store_scratchpad` macro saves the caller's return address to a scratchpad before the function's prologue. The `mov_store_xfi` macro is then used to securely store a value from `%rax` to the sandbox location using the address of `func_ptr(%rip)`, while the `mov_load_xfi` macro securely loads the value from the sandbox address of `func_ptr(%rip)` into `%rdx`, maintaining data integrity.

Before an indirect call via `%rdx`, the `ctrl_flow_xfi` macro is employed to verify that the target address is within the allowed sandbox region, thereby enforcing control-flow integrity. After the function's execution, the `load_scratchpad` macro retrieves the previously stored return address from the scratchpad. Finally, the `ret_xfi` macro first verifies whether the return address is within the sandbox; if it is not, it then proceeds to check whether the address is correct from the scratchpad before allowing the function to return, providing additional control-flow enforcement.

Chapter 17

XFI Evaluation

In this chapter, we measure **XFI**'s runtime overhead, code size overhead, and analyze the security benefits that come from using our technique. The goal of this chapter is to address the following questions:

1. What is the runtime overhead after enforcing **XFI** policies? Due to the older design of x86 assembly code, we cannot use a single instruction to perform multiple complex operations, whereas such is possible using instruction modifiers in the ARM64 architecture. Therefore, does adding more instructions cause unreasonable overhead despite the efficient design to enforce security policies?
2. In continuation of the first question, how much does the code size actually change after inserting all the macros? Does this correlate to an increase in the size of the application?
3. What is the security advantage of applying **XFI**? In contrast to **IBCS**, we provide protection at a process-level rather than fine-grained isolation of data. How does this affect **XFI**'s security goals?
4. What are the limitations and potential areas for improvement of **XFI**?

Section 17.1 discusses the runtime overhead associated with using **XFI**. Section 17.2 examines the increase in code size due to the enforcement of **XFI** policies. In Section 17.3, we provide a security evaluation of **XFI** by detailing the effectiveness of both data-access and control-flow enforcement policies. Section 17.4 addresses the limitations of **XFI** and explores potential improvements. Lastly, Section 17.5 presents a summary of **XFI**'s contributions.

17.1 Runtime Overhead Evaluation

To conduct a comprehensive evaluation of the runtime overhead introduced by **XFI**, we started by testing it on four different binaries from the `GNU Coreutils 8.31`. Subsequently, we extended our evaluation to include two distinct web-server applications: 1) `micro_httpd`[1],

[1] https://www.acme.com/software/micro_httpd/

Binary	**Source LoC**	**Original** (Seconds)	**XFI** (Seconds)	**% Overhead (Performance)**
`wc`	875	0.14	0.22	54.1%
`cp`	1227	0.08915	0.0944	5.89%
`factor`	2662	0.007	0.008	14.29%
`ls`	5309	0.034	0.040	17.65%
Average				22.48%

Table 17.1: Runtime Performance Overhead of Original and XFI Instrumented Binaries

2) a `tiny` web server application as described in [23], and 3) `SQlite`[2] [120] library. The source code for all these applications was compiled on a Linux Debian 11 x86-64 system using `gcc` in conjunction with the musl library [1], and with optimization level set to 0 (`-O0`). The experiments were executed on a machine equipped with an Intel(R) Xeon(R) Silver 4110 CPU and 188 GB of RAM.

Table 17.1 presents the summarized information on the runtime overhead across different binaries. The **Source LoC** column presents the number of source lines of code (LoC) to denote the size of the application. The **Original** column shows the runtime of the behavior defined below with the original version of the binary, and the **XFI** column shows the runtime of the **XFI** policies enforced binary. The **% Overhead** column presents the percent overhead between the original and the **XFI**-enforced binaries. As these binaries do not have pre-defined benchmarking, we evaluated each binary with the following behaviors and measured the time taken to complete each operation:

1. `wc`: counting the words in a file with 1 million lines of text.
2. `cp`: copying a large text file with 1 million lines of text to another file.
3. `factor`: factoring a large number (1234567890123456789012345678901234567890).
4. `ls`: listing the contents of a directory with 10,000 files.

Overall, there is an outlier in the evaluation for the `wc` application, which exhibits a significantly higher overhead of 54.1%. This is attributed to the repetitive execution of an instruction sequence that necessitates constant enforcement using the sandbox environment, leading to increased overhead. Furthermore, as these results are shown in seconds, the difference from 140 milliseconds to 220 milliseconds is relatively small and difficult to notice when the `wc` program executes. Although the percentage overhead is high (54.1%), the actual increase in execution time is minimal. In the context of other applications, such as web servers (Table 17.2), such excessive overhead is not observed.

[2] `https://sqlite.org/index.html`

In contrast, the other applications (`cp`, `factor`, and `ls`) show much lower overheads, with 5.89%, 14.29%, and 17.65% respectively. These results indicate that the runtime performance overhead is generally reasonable, averaging at 22.48%. This overhead is considered efficient given that the **XFI** framework enforces both data-access and control-flow policies, providing robust security protections. The varied impact on different applications highlights the importance of considering the specific operational characteristics and instruction sequences when evaluating security mechanisms.

Next, we measured performance overhead on two web server applications previously mentioned. First, we attempted to test the runtime overhead on the `micro_httpd` application after successfully patching the application. However, we encountered an issue where even directly using the application compiled from a default source code, `micro_httpd` is unable to properly handle the request into the default port 80 nor does it currently support designating a particular port with the command `-p 8080`.

Therefore, we ported the server application described in [23] (`tiny` web server) and to test the capability of **XFI** with respect to the `tiny` web server application using ApacheBench [74].

Table 17.2: `tiny` web server Performance Evaluations

Reqs.	File Sizes	64KB	128KB	256KB	512KB	1MB
`2.5K`	Orig.	0.167	0.245	0.336	0.466	0.58
	XFI	0.184	0.258	0.369	0.487	0.623
	% diff.	10.18%	5.31%	9.82%	4.51%	7.41%
`5K`	Orig.	0.205	0.28	0.327	0.446	0.657
	XFI	0.223	0.306	0.338	0.488	0.709
	% diff.	8.78%	9.28%	3.364%	4.94%	7.92%
`10K`	Orig.	0.216	0.279	0.337	0.445	0.712
	XFI	0.237	0.284	0.355	0.467	0.735
	% diff.	9.72%	1.79%	5.34%	4.94%	3.23%
`20K`	Orig.	0.238	0.297	0.365	0.464	0.635
	XFI	0.256	0.317	0.381	0.482	0.643
	% diff.	7.56%	6.73%	4.38%	3.88%	1.26%
`30K`	Orig.	0.225	0.278	0.397	0.464	0.722
	XFI	0.248	0.291	0.421	0.467	0.723
	% diff.	10.22%	4.67%	6.05%	0.65%	0.14%

Table 17.2 presents a comparative analysis of the performance between the original web server and the **XFI** binaries across various file sizes, ranging from 64KB to 1MB, and request counts from 2,500 to 30,000. The table displays the average time per request, aggregated over 30 runs, with a standard deviation threshold of 1.2 to ensure consistency. The performance overhead reflects the additional performance cost incurred due to the enforcement of **XFI** data-access and control-flow policies. Essentially, for every instruction transfer and data-access instruction, the execution of additional instructions, as previously discussed, is required.

In summary, server requests for smaller file sizes, particularly 64KB, exhibit significantly higher overhead due to the high volatility in measurements caused by the rapid completion of evaluations. For such smaller files, the impact of instrumented **XFI** policies becomes more pronounced relative to the overall time required to complete the request and response, which significantly amplifies the additional execution times introduced by the necessary operations. However, as the requested file size increases along with higher count of requests per larger file size, the performance overhead becomes considerably more reasonable.

Lastly, we attempted to apply the **XFI** policies in the SQLite shell (`sqlite3`) [120]. SQLite is primarily known as a library that applications link to for embedded SQL database functionality, but it also includes a command-line utility that ships with the library, allowing users to interact with SQLite databases directly. **XFI** was able to successfully rewrite the assembly code of `sqlite3` (`sqlite3-sqlite3.s`), but unfortunately, we encountered an error during the execution phase. During **XFI**'s sandboxing phase, the enormous size of `sqlite3` caused each domain's allocation to exceed the default memory space. Therefore, as discussed in Section 18.1.5, to support the isolation of larger applications such as the `sqlite3` binary file, a mechanism is needed to dynamically detect the domain size and allocate enough memory to support the application. However, we have documented other evaluation related to the `sqlite3` with respect to code size and binary size.

17.2 Code Size Overhead Evaluation

Binary	Source LoC	ASM LoC	Patch LoC	% Overhead (Code)
`basename`	191	245	252	202.9%
`pwd`	395	558	494	188.5%
`wc`	875	981	1690	272.3%
`cp`	1227	1082	1109	202.5%
`factor`	2662	2908	3086	206.1%
`ls`	5309	5056	12186	341.0%
Average				235.55%
`micro_httpd`	316	167	261	56.29%
`tiny` web server	449	223	692	210.31%
`sqlite3`	257673	54718	274347	401.4%

Table 17.3: Code Size Overhead of Original and XFI Instrumented Binaries

Table 17.3 presents the summarized information on the code size overhead across different binaries. Similar to Table 17.1, the **Source LoC** column presents the number of source lines of code (LoC) to denote the size of the application. The **ASM LoC** column represents only the *assembly* lines of code. In other words, we omit any directives or labels in the assembly

code and only consider assembly instructions (e.g., `mov`, `jmp`, etc.). The **Patch LoC** column shows the number of assembly instructions that **XFI** requires to be added to the original assembly code. Lastly, the **% Overhead** column computes the increase in code size after adding the **XFI** policies assembly instructions (**Patch LoC**) on top of the original assembly code (**ASM LoC**).

Binary	Source LoC	Original Size (KB)	XFI Size (KB)	% Overhead (Size)
`basename`	191	171	171	0%
`pwd`	395	184	188	2.17%
`wc`	875	244	248	1.64%
`cp`	1227	436	441	1.15%
`factor`	2662	364	373	2.47%
`ls`	5309	526	560	6.46%
Average				2.65%
`micro_httpd`	316	20	24	20%
`tiny` web server	449	36	36	0%
`sqlite3`	257673	2100	2400	14.29%

Table 17.4: Binary Size Comparison of Original and XFI Instrumented Binaries

As previously mentioned, due to the older design of the x86 architecture, **XFI** requires the insertion of extra instructions to enforce necessary policies. This results in a significant code size overhead. However, it is important to note that, with respect to binary size, the increase is not very noticeable, as presented in Table 17.4. While the code size overhead may be high, it does not affect the binary size as much as expected. The average size overhead across different binaries for `Coreutils` is only 2.65%, which indicates that the additional instructions required by **XFI** do not significantly bloat the binary. Even for a larger application such as `sqlite3`, the size overhead is only 14.29% despite the addition of many instructions, which is encouraging to see.

In Section 17.4, we will discuss potential optimization opportunities to reduce the code size overhead. These optimizations aim to make the enforcement of **XFI** policies more efficient by minimizing the number of extra instructions. This could include techniques such as reducing the number of instructions needed to load the **XFI** base address or exploring different ways to improve the macros.

Overall, while **XFI** introduces a noticeable increase in code size, its impact on the actual binary size remains relatively minor. This suggests that there is room for optimization to reduce the code size overhead without compromising the security benefits provided by **XFI**.

Although **XFI** is similar to the work of LFI [233], a direct comparison between **XFI** and LFI is not feasible because they are based on different architectures (RISC vs. CISC). Other

related works of SFI based on the x86 architecture [147, 234] are also not possible to compare against because they are based on a 32-bit architecture (x86-32), which requires segmented memory to enforce necessary policies. The later x86-64 version of NaCl proposed in [193] aimed to port the original work of NaCl [234] to the 64-bit architecture, with the goal of safely running native code from a web browser. However, at the time of writing, one of the limitations of **XFI** is the lack of support for macrobenchmarks and larger applications, making a direct head-to-head comparison unfeasible at this time.

17.3 Security Evaluation

To properly understand the security benefits that **XFI** brings, we evaluate the types of attacks that **XFI** can protect against and discuss the potential scenarios where **XFI** can defend against adversaries. In general, similar to the work of LFI [233], **XFI** is able to handle certain types of side-channel attacks, but not all, which we will further discuss in the context of different policies.

Data-access Enforcement Policies: Every SFI-related scheme needs to guarantee that all memory reads and writes by the code will be within the range of the sandbox. This is because the main goal of SFI is to minimize any potential faults within its own sandbox and prevent the code from accessing unauthorized memory regions (i.e., other processes that are loaded in the vicinity of the victim process). In that sense, enforcing data-access using **XFI** provides such a guarantee. When memory read or write instructions attempt to access an invalid **XFI** sandbox address, it will result in a fault access, and the program will subsequently exit. This subsequently means enforcing data-access policy will also prevent the adversary from performing data manipulations inside the domain that could be leveraged to perform non-control-data attacks such as Data-Oriented Programming (DOP) [98] in order to access critical data structures, such as a global variable.

With respect to side-channel attacks, **XFI**'s data-access enforcement policies provide an intra-sandbox safety policy. If a process somehow speculatively executes code within the victim's sandbox, it will only interact with data inside the sandbox, thus preventing direct access to any data outside the sandbox. In other words, it is able to intrinsically prevent "Sandbox Breakout" attacks [159]. However, in scenarios where cache lines are shared between processes, even with **XFI** policies enforced, this will not prevent adversaries from sniffing the cache state changes, which may allow them to infer sensitive information given enough data.

Control-flow Enforcement Policies: SFI-related schemes also need to ensure that all control transfers (e.g., `call`, `jmp`, and `ret`) are restricted either within their own sandbox or to vetted external addresses that are deemed to be transfer-safe (i.e., not exploitable by malicious entities). In this regard, **XFI** provides forward control-flow integrity guarantees that both indirect `call` and `jmp` instructions will transfer to code within their own designated region (i.e., inside the `.text` section). **XFI** inserts necessary guards that check the address stored in

the register before an indirect transfer occurs. For the `ret` instruction, we ensure backward-edge control-flow integrity by leveraging the *return address scratchpad*, which ensures that returns from functions only transfer to legitimate call sites. These control-flow enforcement policies help protect against Return-Oriented Programming (ROP) attacks by ensuring that control transfers can only go to authorized and intended locations. By verifying the target addresses of indirect calls, jumps, and returns, **XFI** prevents attackers from chaining together gadgets that are crucial for ROP attacks.

Side-channel attacks are also partially handled, as control-flow enforcement policies naturally prevent the adversary from tricking the CPU into mispredicting a branch to jump to a different sandbox, similar to the data-access enforcement policy. However, any side-channel attacks involving the leakage of information, such as timing attacks or power analysis, cannot be fully protected against by **XFI** and are out of scope for our work. Additionally, while **XFI** can prevent direct control-flow hijacking, it does not mitigate side-channel techniques that infer control flow through indirect means, such as monitoring cache usage patterns or branch prediction states. Thus, while **XFI** provides robust control-flow integrity, comprehensive protection against all forms of side-channel attacks would require additional measures beyond the current scope of our implementation.

With respect to the side-channel attacks we discuss in the **XFI** work (e.g., Sandbox Breakout), there is a potential way where the adversary could *evade* detection by crafting more sophisticated payloads, but because **XFI** is not a detection mitigation technique, it has no direct effect on the performance overhead. In that sense, the severity of ROP and DOP attacks also has no impact on the performance overhead of **XFI** because it is an enforcement mitigation technique.

17.4 Discussion

XFI enforces necessary SFI policies efficiently at a process level. However, it is not an iron-clad solution, and several limitations need to be discussed to explore areas for improvement. This section will cover topics such as optimizing assembly code insertions to reduce code size overhead, identifying potential security weaknesses that can be addressed, and suggesting ways to enhance the overall quality of **XFI**.

Code Size Overhead Reduction. Although the code size overhead of **XFI** does not significantly affect the overall binary size, it is still important to explore potential improvements. One obvious enhancement is the removal of the need to load the **XFI** base address every time a macro is invoked using the `rdgsbase %r15` instruction.

Since `%r15` is a callee-saved register (also known as a non-volatile register), its value should remain intact across function calls, provided that the called functions adhere to the calling convention and do not explicitly modify the `%r15` register. Therefore, it is unnecessary to always load the **XFI** base address for each macro invocation. If we can determine the entry

point of the program and load the **XFI** base address at that point, it would reduce the instruction overhead per macro call.

Improving Security Effectiveness. As previously mentioned, leakage-related side-channel attacks are not currently protected using **XFI**. Previous works such as LFI [233] leveraged architecture-related hardware features, such as Arm's `CSV2_2` extension, to extend the protection capabilities of their solution. **XFI** can also leverage various x86 hardware features, such as Speculative Store Bypass Disable (SSBD) [103] to protect against leakage-related side-channel attacks.

Furthermore, at the moment, **XFI** does not have an oracle list of addresses to vet and check whether functions using the Procedure Linkage Table (PLT) (e.g., `printf@plt`) are allowed. One simple way to mitigate this problem would be to compile the application statically (using the `-static` flag during compilation), ensuring that all code required by the program is included in the `.text` section. This way, **XFI** policies can provide extended coverage by encompassing all necessary code within the sandbox.

17.5 Summary

As the final contribution of this dissertation, we present the x86-based Fault Isolation (**XFI**), an x86-based SFI system that provides process-level isolation by enforcing data-access and control-flow policies efficiently, without requiring invasive modifications to the compiler toolchain. **XFI** is portable across x86 machines and achieves this by rewriting assembly code, thus eliminating the need to modify complex compiler backends.

XFI accomplishes this by inserting **XFI** guards and replacing assembly instructions with correlated macros to ensure that necessary policies are properly enforced. The challenges involved in implementing **XFI** include accurately determining the scope of protection for an input process, efficiently implementing necessary guards to realize isolation, and maintaining the compatibility of **XFI** without requiring extensive modifications to the existing toolchain.

XFI first statically analyzes an object file generated from the input assembly code to create a sandbox environment. Next, it rewrites the assembly code to insert the aforementioned **XFI** security policies and rewrites the binary, ensuring it will only execute if the sandbox environment is initialized before execution. We evaluate the overhead of **XFI** using GNU `Coreutils` with respect to performance, code size, and binary size, and examine the potential security benefits that **XFI** can offer.

Chapter 18

Conclusions and Future Work

Security vulnerabilities in software systems are multifaceted and can arise from various sources, including memory safety errors, code-reuse attacks, and improper input handling. Memory safety vulnerabilities, such as buffer overflows and use-after-free errors, remain prevalent due to the use of memory-unsafe languages like C and C++. Code-reuse attacks, like Return-Oriented Programming (ROP), exploit existing code to hijack control flow and perform malicious operations. Additionally, input-based vulnerabilities, where external inputs are used to trigger exploit paths, pose significant risks.

This dissertation presents a comprehensive approach to enhancing software security through multiple innovative techniques. First, it introduces a Verification of Diversified Binary (**VDB**) algorithm and present a methodology to establish the soundness of code diversification tools by ensuring an equivalence relation between vanilla and diversified binaries. This is achieved through a combination of disassembly, symbolic execution, stack-pointer related invariants, and mapping memory regions between the two versions.

Next, the dissertation presents Control Flow Restriction (**CFR**), a novel, hardware-agnostic security technique at the binary level. **CFR** protects binaries from code-reuse attacks by restricting indirect control transfer (ICT) instructions to legitimate targets. This is accomplished through a statically generated deny list and a dynamically updated return address queue, ensuring that malicious control-flow redirections are effectively thwarted.

Additionally, the dissertation introduces Shroud, a mechanism leveraging ARM's Memory Tagging Extension (MTE) to enhance data security through compartmentalization. Shroud employs taint analysis to identify unsafe data, relocates this data to separate memory regions, and uses one-time-pad encryption to secure MTE-tagged pointers. This approach mitigates stack-based vulnerabilities and prepares data for MTE heap tagging, offering robust security without significant performance overhead.

The Input-Based Compartmentalization System (**IBCS**) addresses the limitations of traditional compartmentalization methods by dynamically adapting to user inputs. By using a hybrid taint analysis graph, **IBCS** identifies and compartmentalizes sensitive data, reducing under-approximation in enforcement. This approach demonstrates the potential security benefits of compartmentalizing data in a fine-grained manner.

Finally, the x86-based Fault Isolation (**XFI**) system proposes an efficient SFI scheme based on the x86 architecture. By rewriting the assembly code, **XFI** does not require extensive

modifications to the existing toolchain. **XFI** enforces both data-access and control-flow policies, protecting against various types of side-channel attacks and ensuring fault isolation for each process. This contribution provides comprehensive protection against a wide range of attack vectors, highlighting the dissertation's role in creating a robust, adaptable security framework for modern software systems.

Table 18.1: Summary of Contributions in this Dissertation

Contribution	Threat Scope	Input	Architecture	Performance Overhead	Code Size Overhead	Binary Size Overhead
VDB (Code Div.)	BA[1][243]	Source-Code	Intel (N/A)	Low	Low-Medium	Low-High
CFR (CFI)	CA[2]	Source-Code, Binary	Intel (N/A)	High	High	High
SHROUD (Compart.)	PE,[3] MSV,[4] NA[5]	Source-Code	ARM (MTE)	Low	Medium	Low
IBCS (Compart.)	PE,[3] MSV,[4] NA[5]	Source-Code	Intel (N/A)	Low	High	Low
XFI (SFI)	CA,[2] MSV,[4] NSCA[6]	Source-Code	Intel (N/A)	Medium	High	Low

[1] Binary-only Attacks;
[2] Code-reuse Attacks;
[3] Privilege Escalation;
[4] Memory Safety Violation;
[5] Non-control-data Attacks;
[6] Non-leakage-based Side-channel Attacks

Table 18.1 presents the summary of contributions in this dissertation, highlighting how each technique addresses specific threat scopes with respect to the architecture it can be applied to, along with information regarding performance overhead and code size overhead. For instance, the code diversification technique covered in the **VDB** contribution and the CFI policies covered in the **CFR** both protect against code-reuse attacks (which are included in the class of binary-only attacks). Diversification is a pre-emptive technique that disrupts the binary layout, making it intrinsically harder for adversaries to find gadgets and perform exploitation. In contrast, CFI is a *reactive* technique that assumes the adversary will attempt

to chain gadgets to achieve control-flow hijacking and *actively* checks the intended control flow during runtime. This example reinforces the belief that the existence of different security techniques that address the same type of threat does not make one technique obsolete; rather, each provides unique strengths that can be interchangeably used in different contexts.

Furthermore, various types of overheads need to be considered when deciding which technique to apply. For example, in the context of various diversification techniques used for the **VDB** contribution (discussed in Table 1.1), the performance overhead may be low compared to the **CFR** contribution, but it requires source code, while code size and binary size overheads vary from low to high. This variation is because some techniques only require the insertion of certain metadata to perform the necessary diversification, whereas others rely on inserting additional assembly instructions. On the other hand, for **Shroud**, performance and binary size overheads are low, but the necessary instructions to relocate and apply primitives (e.g., encryption/decryption) may lead to medium code size overhead. Lastly, both **IBCS** and **XFI** contributions have high code size overhead as a large number of assembly instructions need to be augmented to perform necessary policy checks and manage compartments, but binary size overhead is low for both, along with reasonable performance overhead.

Therefore, as previously mentioned, a variety of complex and diverse vulnerabilities exist, making it essential to conduct comprehensive research that addresses multiple types of security threats. Focusing on a single type of vulnerability can lead to gaps in defense mechanisms, leaving systems exposed to other forms of attacks. A holistic approach that encompasses various vulnerabilities ensures robust security frameworks, capable of defending against a wide array of potential exploits and enhancing the overall resilience of software systems. Although various techniques may overlap to protect against the same type of threat, it is still important to explore different mitigation approaches to understand and be prepared to flexibly apply the necessary primitive when needed.

In conclusion, this dissertation advances the field of software security by proving the correctness of code diversification techniques, proposing novel security policies, developing automated tools for fine-grained compartmentalization, and efficiently enforcing necessary SFI policies for an x86 process. These contributions collectively enhance the robustness of security mechanisms against sophisticated exploits, paving the way for hybrid solutions that integrate multiple layers of protection.

18.1 Future Work

There are many future research directions for the components discussed in this dissertation. Here, we describe several potential areas for further investigation.

18.1.1 Binary-level Data-flow Generation

Data-oriented attacks have been proposed as an alternative to exploiting memory vulnerabilities without diverting control flow. In contrast to control-flow hijacking attacks that redirect indirect control transfer (ICT) instructions, a data-oriented attack attempts to alter critical data of an application to cause unintended behavior. Although data-oriented attacks have strict requirements (e.g., memory vulnerabilities and information disclosure to know the variable location), they are powerful enough to bypass even the most precise control-flow hijacking prevention techniques. Furthermore, these requirements are not as challenging to meet as they seem, making them difficult to enforce. This underscores the need to propose a binary-level data-oriented attack protection mechanism.

A Data-flow Graph (DFG) is a model where computation operations (e.g., addition, subtraction, etc.) are represented as nodes, and data flow is represented using edges. An inward arrow to the operation is considered input, while an outward arrow is regarded as output. DFG is a data-oriented model where the program is translated as a single sequence process execution and does not involve any conditional operations.

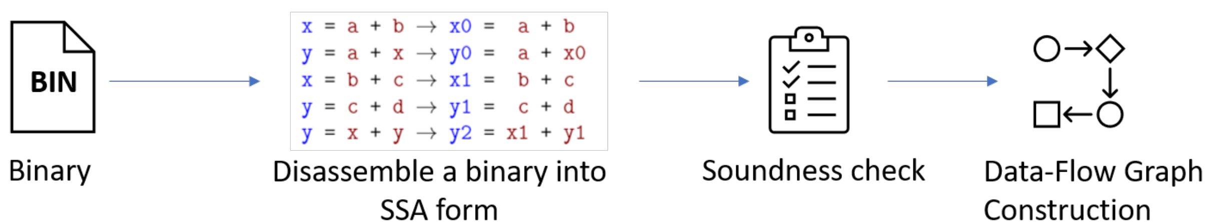


Figure 18.1: Overview of DFG construction

The first step of implementing a data-oriented attacks prevention mechanism without involving hardware is to construct the DFG. This step is relatively simple with source code. Since the compiler has access to precise information regarding the variable, it can perform data-flow analysis, namely reaching definition to statically determine which definitions may reach a certain point in the code. However, using a compiler is not possible at a binary level, which makes it more difficult. Hence, the first post-preliminary exam contribution is to construct the DFG from a binary level.

The main challenge to overcome for this contribution narrows down to two points:

1. The generation of the binary's SSA form to construct the DFG.

2. Formally defining the soundness of DFG.

Figure 18.1 presents the overview of our approach to generating a DFG. To construct the DFG, we will first recover the necessary information from the binary using a reverse-engineering binary disassembling tool. This is necessary because the first step of DFG construction is to obtain a static single assignment (SSA) form of a binary. SSA is a form of intermediate representation (IR) where every variable assigned with a value is shown at most *once* in the program text.

The main reason why SSA is obtained is that it makes data-flow analysis easier due to its simplicity to store and update. Furthermore, as distinct usage of a variable is differentiated, many optimizations are performed in DFG construction. This is important because part of the main design issues when constructing a DFG is to ensure that performance is acceptable.

Afterward, we will attempt to formally verify the *soundness* of constructed DFG. Soundness in this context refers to checking whether the constructed DFG does not have any *false positive* nodes. In other words, it does not contain any node that did not exist in the original code nor does not have any node which will yield an error. To achieve this, we will deploy symbolic execution and model checking techniques to ensure there is no fault in the generated DFG.

18.1.2 Data-flow Restriction

After constructing the DFG, we propose a data-oriented attack security policy called Data-flow Restriction (DFR). This contribution aims to implement a binary-level protection mechanism against data-oriented attacks using the binary rewriting technique. As previously explained, data-oriented data attacks leverage non-control data (e.g., a data variable) to bypass control-flow hijacking prevention techniques. Such bypass is possible because a technique such as CFI ensures that indirect control transfer instructions do not transfer to unwanted locations and does not consider the integrity of value stored in a variable.

Data-flow Integrity [29] has been proposed to provide a method to counter data-oriented attacks. However, DFI at the moment is only available at a source code level. In addition, existing data-oriented attacks defense mechanisms surveyed in [37] show that none of the existing works operate solely on binary-level. All of them are either an extension of C language [189], hardware-assisted [164, 205] or require a kernel-level mechanism [55] to protect against data-oriented attacks. The main challenge to overcome for this contribution is determining what instructions to patch and rewrite the binary using the constructed DFG to enforce the DFR.

Figure 18.2 presents a preliminary overview of DFR. First, we will leverage our own constructed DFG and organize necessary data to determine the possible value sets for each variable read in a binary. Next, we will insert this metadata during the binary rewriting

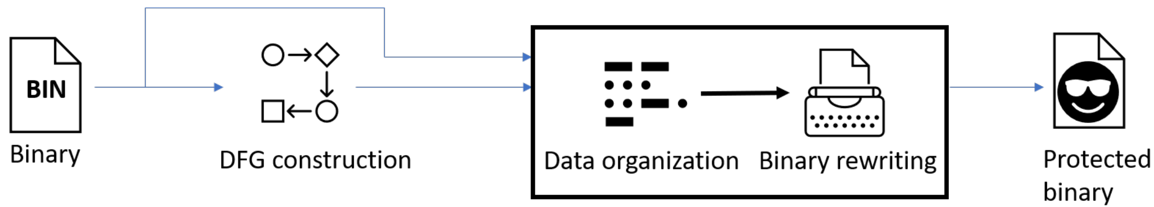


Figure 18.2: Overview of DFR

procedure, where we will patch all instructions that read a variable. When a patched binary executes, before a read instruction runs, it will first check whether a read target contains a value within a possible value set analyzed during the DFG construction process. If the data is not part of the set, then this will be considered a potential danger. Afterward, the execution will either provide a warning or halt altogether.

18.1.3 Scalable Pointer Offset Tree Analysis

The primary goal of **IBCS** is to fully automate both the identification and enforcement phases of compartmentalization. This automation involves complex tasks such as decoding the ELF binary, conducting static taint analysis, and generating pointer offset trees for all functions. These tasks become critical when pointers across the entire program need patching, not just within individual functions. A major limitation of this approach is evident in large-scale benchmarks (e.g., Nginx), where static inter-procedural analysis—which disassembles the binary to construct caller-to-callee graphs for all functions—encounters an explosion of parameter paths.

For future work, enhancing the scalability of the **IBCS** identification process may involve refining the pointer offset tree analysis to focus solely on assembly code, thereby bypassing the need to analyze the entire binary. This approach is promising due to several consistent aspects of assembly code: the linear execution of assembly instructions, the consistent use of argument registers to store parameters before a callee function is called, and the setup of parameters after the function's prologue. By leveraging these elements, we could redesign the algorithm for intra-procedural analysis, linking results from each function to build comprehensive function call trees without performing inter-procedural binary analysis. These trees would facilitate generating more precise and scalable pointer offset trees. Exploring the applicability of existing static taint analysis techniques represents another avenue for future research, especially given the challenges we encountered with using SUTURE for large applications.

18.1.4 Heap Compartmentalization of IBCS

Listing 18 Heap Value Initialization in Both C and Assembly Code

C Source Code

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int *heap_var = (int *)malloc(sizeof(int));
    *heap_var = 42;
    return 0;
}
```

Assembly Code

```asm
main:
...
call    malloc@plt              # Call malloc to allocate memory
movq    %rax, -8(%rbp)          # Store the returned pointer in heap_var
...
movq    -8(%rbp), %rax
movl    $42, (%rax)             # Assign value 42 to *heap_var
```

In Section 14.2, we mentioned that *conceptually* **IBCS** can compartmentalize heap variables similarly to stack-based buffers. However, we did not explicitly clarify *how* this can be done. Therefore, as future work, we discuss potential directions for achieving this. The important challenge in supporting heap compartmentalization is successfully identifying heap variables. This can be difficult using only DWARF information because heap variables are typically categorized as pointers (`DW_TAG_pointer_type`). However, not all pointers are heap variables. For example, a string in C (e.g., `char *text = "Hello World!"`) is a pointer variable but not a heap variable. Thus, the first task in supporting heap compartmentalization is to accurately identify heap variables among various types of pointers.

Listing 18 presents how the heap variable is referenced in the assembly code for a given C source code. There are multiple ways to achieve this. The first method involves directly analyzing the assembly code. Upon finding the first reference to a given sensitive data (e.g., `-8(%rbp)`), one can analyze the immediate preceding instruction to check whether it is a function that creates a heap variable (in this example, `malloc`). However, this can be a daunting task without considering the details of how to relocate these heap compartments into the compartment table.

To remedy this, a second method takes advantage of the fact that heap initialization functions ultimately rely on `malloc` in their internal procedures. By modifying `malloc` either in the source code or by hooking a new wrapper function, one could insert a unique sequence of assembly instructions (via inline assembly) to refer to the memory page (compartment) that will be stored in the compartment table after the function calls. Figure 18.3 presents a rough visual example of how this will look (subject to change).

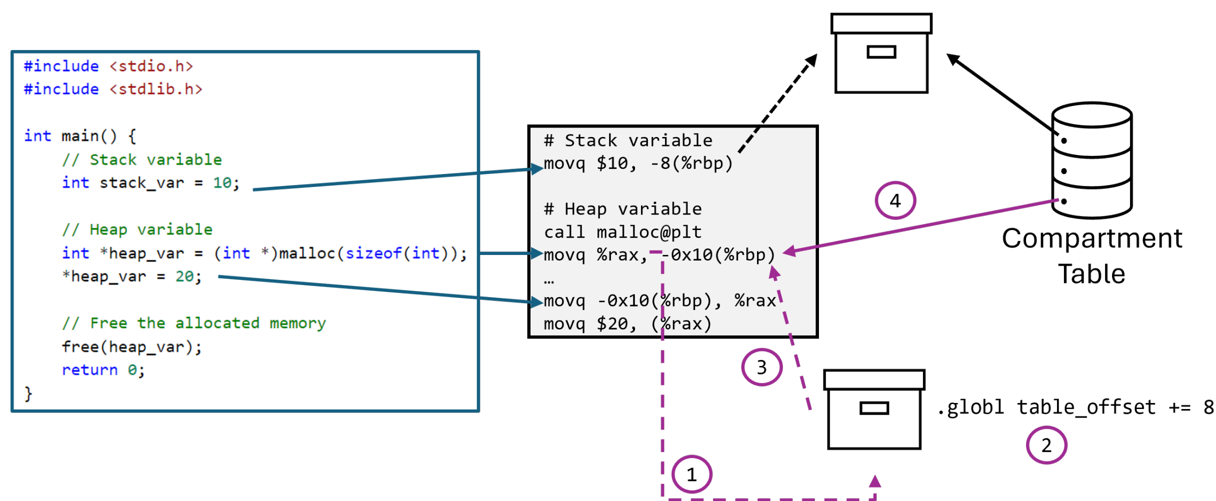


Figure 18.3: **IBCS** Heap Compartmentalization Overview

As a reminder, a stack variable (e.g., `0x8(%rbp)`) simply needs to be relocated to the pre-generated compartment when the compartment table is generated. In contrast, for a heap variable, upon the `malloc` function call, it first needs to generate a compartment ((1)). Next, a global table offset variable (`table_offset`) needs to be updated ((2)), and then stored in the respective stack offset where the heap variable will reside ((3)). Finally, the compartment needs to be registered into the compartment table with the updated table offset ((4)).

The challenge and limitation of this method are that the compartment table size needs to be pre-defined to support the potential number of heap variable compartments (as they will be initialized during the actual function calls of `malloc`). Additionally, ensuring that new heap variables are properly placed in the compartment table with the up-to-date `table_offset` value, which now needs to be globalized throughout the entire program, adds complexity. Heap compartmentalization work would greatly benefit from future work on Scalable Pointer Offset Tree Analysis being extended to be applicable solely at the assembly code level so that, as long as the heap is compartmentalized, all other heap usages will intrinsically refer to the heap compartmentalization.

Multiple Domains Support: At the time of writing, **XFI** is designed to efficiently isolate multiple domains but currently only supports single-domain isolation. To extend **XFI** to support multiple domains, a buddy allocator system needs to be implemented. This memory management technique allocates memory for multiple domains by dividing the memory pool into smaller chunks called "buddies." Depending on the number of processes to be isolated, the memory pool is divided evenly, with each process receiving a unique sandbox memory allocation, ensuring separation and isolation.

Full Support on x86: The control-flow enforcement policy provided by **XFI** does not fully address the core difficulties of implementing SFI on x86, such as handling variable-

sized instructions. Currently, **XFI** aims to provide basic control-flow policy enforcement by ensuring that control-flow transfers reside within a known region and leveraging the concept proposed in CFI [3] to ensure that the return address is valid. Improving this aspect could enhance **XFI**'s security guarantees. Possible approaches include adopting the `nop` padding proposed in PittSFIeld [147] or deploying a hardware-enforced CFI like Intel Control-flow Enforcement Technology (CET) [102], which uses mechanisms such as Indirect Branch Tracking (IBT) and the `endbr64` instruction at valid indirection targets.

Macro Optimizations: Finally, while we have optimized the **XFI** macros to a considerable extent, there may still be opportunities for further optimization. Additional improvements in macro design could further reduce the code size overhead, making **XFI** even more efficient. For instance, the author of LFI [233] suggests leveraging the syntax of `%gs:r15d` to safely rewrite the necessary assembly instructions. This approach, however, would not work in the current design of **XFI** as it needs to pack the `.text` section end address.

18.1.5 Supporting Different Types of Binaries for IBCS and XFI

Listing 19 Comparison of Assembly Code Optimization Levels

-O0 Optimization Level

```
1 movl    $5, -4(%rbp)         # int a = 5;
2 movl    $10, -8(%rbp)        # int b = 10;
3 movl    -4(%rbp), %eax       # Load a into %eax
4 addl    -8(%rbp), %eax       # Add b to %eax
5 movl    %eax, -12(%rbp)      # Store result in c
6 movl    -12(%rbp), %esi      # Load c for printf
7 leaq    .L.str(%rip), %rdi   # Load format string
8 movl    $0, %eax
9 call    printf@PLT
```

-O2 Optimization Level

```
1 movl    $15, %edi             # Combine a and b directly
2 leaq    .L.str(%rip), %rsi    # Load format string
3 movl    $0, %eax
4 call    printf@PLT
```

Higher Optimization Support: Although the assembly code rewriting techniques offer promising benefits, they also have their own disadvantages. By using a higher level of compiler optimization (e.g., `-O2`), the assembly code becomes so optimized that we are unable to determine the location of variables. Listing 19 depicts an example where a code that executes `int c = a + b;` (assuming `a = 5` and `b = 10`) has all those variables optimized away at a higher level (`-O2`). This could be an interesting area for future work to investigate, further paving the way for using assembly code as a realm for rewriting the code.

Larger Application Support: Although we were able to support the Nginx application for the **IBCS** work, attempting to rewrite assembly code for other larger applications proved

challenging for various reasons. For instance, every large application uses different `Makefile` and compilation strategies, making the process of rewriting assembly code less intuitive than we had hoped. The challenge lies in how to compile these applications after the rewriting process is completed. Furthermore, in the case of **IBCS**, if Pointer Offset Tree Analysis could rely solely on assembly code without the need for static analysis, rewriting the assembly code would be significantly easier. This is because all references to the original pointer location would not need to be patched, unlike our current approach.

Furthermore, in the context of **XFI**, we have attempted to support larger applications such as `sqlite3` [120]. However, we faced a challenge where, in order to isolate domains of such applications, each domain requires an excessively large memory allocation which the current version of **XFI** cannot support. Therefore, being able to dynamically determine the size of such domains and flexibly isolate each domain to its own space will be an interesting area for future work.

Bibliography

[1] musl libc. URL https://www.musl-libc.org/.

[2] Vector 35. Binary ninja. URL https://binary.ninja/.

[3] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595932267. doi: 10.1145/1102120.1102165. URL https://doi.org/10.1145/1102120.1102165.

[4] Alex Aiken. Moss: A system for detecting software plagiarism. URL http://www.cs.berkeley.edu/~aiken/moss.html.

[5] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 263–277. IEEE, 2008.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019. ISSN 2475-1421. doi: 10.1145/3290353. URL http://doi.acm.org/10.1145/3290353.

[7] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

[8] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, 2017. doi: 10.1109/EuroSP.2017.11.

[9] Steven Arzt. Static data flow analysis for android applications. 2017.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

[12] Jeffrey M Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[13] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[14] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[15] Eli Bendersky. pyelftools: Parsing elf and dwarf in python, 2020.

[16] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, page 9–16, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308496. doi: 10.1145/2024569.2024572. URL https://doi.org/10.1145/2024569.2024572.

[17] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, page 8, USA, 2003. USENIX Association.

[18] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, page 17, USA, 2005. USENIX Association.

[19] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[20] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, page 30–40, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305648. doi: 10.1145/1966913.1966919. URL https://doi.org/10.1145/1966913.1966919.

[21] Sergey Bratus, ME Locasto, and ML Patterson. Exploit programming: From buffer overflows to "weird machines" and theory of computation. 2011.

[22] Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical computer science*, 59 (1-2):115–131, 1988.

[23] Randal E Bryant and David Richard O'Hallaron. *Computer systems: a programmer's perspective*. Prentice Hall, 2011.

[24] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), April 2017. ISSN 0360-0300. doi: 10.1145/3054924. URL https://doi.org/10.1145/3054924.

[25] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999, 2019. doi: 10.1109/SP.2019.00076.

[26] Nicholas Carlini and David Wagner. {ROP} is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.

[27] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 385–399, USA, 2014. USENIX Association. ISBN 9781931971157.

[28] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 161–176, USA, 2015. USENIX Association. ISBN 9781931971232.

[29] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[30] Dong-Kyu Chae, Jiwoon Ha, Sang-Wook Kim, BooJoong Kang, and Eul Gyu Im. Software plagiarism detection: A graph-based approach. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 1577–1580, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2263-8. URL http://doi.acm.org/10.1145/2505515.2507848.

[31] Sagar Chaki and James Ivers. Software model checking without source code. *Innov. Syst. Softw. Eng.*, 6(3):233–242, September 2010. ISSN 1614-5046. doi: 10.1007/s11334-010-0125-0. URL http://dx.doi.org/10.1007/s11334-010-0125-0.

[32] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 320–332, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062344. URL https://doi.org/10.1145/3062341.3062344.

[33] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns.

In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 559–572, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302456. doi: 10.1145/1866307.1866370. URL https://doi.org/10.1145/1866307.1866370.

[34] Ligeng Chen, Zhongling He, and Bing Mao. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 88–98, 2020. doi: 10.1109/DSN48063.2020.00028.

[35] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security*, ICISS '09, page 163–177, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642107719. doi: 10.1007/978-3-642-10772-6_13. URL https://doi.org/10.1007/978-3-642-10772-6_13.

[36] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX security symposium*, volume 5, page 146, 2005.

[37] Long Cheng, Hans Liljestrand, Thomas Nyman, Yu Tsung Lee, Danfeng Yao, Trent Jaeger, and N. Asokan. Exploitation techniques and defenses for data-oriented attacks, 2019.

[38] Haehyun Cho, Jinbum Park, Donguk Kim, Ziming Zhao, Yan Shoshitaishvili, Adam Doupe, and Gail-Joon Ahn. Smokebomb: effective mitigation against cache side-channel attacks on the arm architecture. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 107–120, 2020.

[39] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupe, and Gail-Joon Ahn. Vik: Practical mitigation of temporal memory safety violations through object id inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 271–284, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507780. URL https://doi.org/10.1145/3503222.3507780.

[40] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. ISBN 1-55860-470-7. URL http://dl.acm.org/citation.cfm?id=645923.671005.

[41] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. {ACES}: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 65–82, 2018.

[42] DL Clutterbuck and Bernard A Carré. The verification of low-level code. *Software Engineering Journal*, 3(3):97–111, 1988.

[43] Frederick B. Cohen. Operating system protection through program evolution. volume 12, page 565–584, GBR, October 1993. Elsevier Advanced Technology Publications. URL https://doi.org/10.1016/0167-4048(93)90054-9.

[44] Jonathan Corbet. x86 nx support, 2004. URL https://lwn.net/Articles/87814/.

[45] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard™: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[46] Jedidiah R Crandall and Frederic T Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 221–232. IEEE, 2004.

[47] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780. IEEE, 2015.

[48] CVE. Cve-2006-5815: Stack-based buffer overflow in proftpd, 2006. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2006-5815.

[49] CVE. Cve-2014-0160: Heartbleed bug in openssl, 2014. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

[50] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13 (4):451–490, 1991.

[51] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 555–566, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332453. doi: 10.1145/2714576.2714635. URL https://doi.org/10.1145/2714576.2714635.

[52] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*, STC '09, page 49–54, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587882. doi: 10.1145/1655108.1655117. URL https://doi.org/10.1145/1655108.1655117.

[53] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, page 40–51, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305648. doi: 10.1145/1966913.1966920. URL https://doi.org/10.1145/1966913.1966920.

[54] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of {Coarse-Grained}{Control-Flow} integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, 2014.

[55] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *NDSS*, 2017.

[56] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 266–280, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. URL https://doi.acm.org/10.1145/2908080.2908126.

[57] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.

[58] Kasif Dekel. Ghosthook – bypassing patchguard with processor trace based hooking, 2017. URL https://www.cyberark.com/resources/threat-research-blog/ghosthook-bypassing-patchguard-with-processor-trace-based-hooking.

[59] Christian DeLozier, Kavya Lakshminarayanan, Gilles Pokam, and Joseph Devietti. Hurdle: Securing jump instructions against code reuse attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 653–666, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378506. URL https://doi.org/10.1145/3373376.3378506.

[60] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[61] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2019.

[62] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, pages 151–163, 2020.

[63] Zoran Durić and Dragan Gašević. A source code similarity system for plagiarism detection. *Comput. J.*, 56(1):70–86, January 2013. ISSN 0010-4620. doi: 10.1093/comjnl/bxs018. URL http://dx.doi.org/10.1093/comjnl/bxs018.

[64] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[65] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, August 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele.

[66] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[67] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, 2006.

[68] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 901–913, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813646. URL https://doi.org/10.1145/2810103.2813646.

[69] Jean-Claude Fernandez and Laurent Mounier. Verifying bisimulations "on the fly". In *FORTE*, volume 90, pages 95–110, 1990.

[70] Edward A Feustel. On the advantages of tagged architecture. *IEEE Transactions on Computers*, 100(7):644–656, 1973.

[71] David Flater and David Flater. *Software Science Revisited: Rationalizing Halstead's System Using Dimensionless Units*. US Department of Commerce, National Institute of Standards and Technology, 2018.

[72] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[73] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, pages 67–72. IEEE, 1997.

[74] The Apache Software Foundation. ab - apache http server benchmarking tool, 2021. URL http://httpd.apache.org/docs/2.2/en/programs/ab.html.

[75] Vincenzo Frascino. Arm v8.5 memory tagging extension. In *Proceedings of the Linux Plumbers Conference, Lisbon, Portugal*, volume 10, 2019.

[76] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. pages 238–255, 2008.

[77] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 585–598, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344654. doi: 10.1145/3037697.3037716. URL https://doi.org/10.1145/3037697.3037716.

[78] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 40, USA, 2012. USENIX Association.

[79] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, page 575–589, USA, 2014. IEEE Computer Society. ISBN 9781479946860. doi: 10.1109/SP.2014.43. URL https://doi.org/10.1109/SP.2014.43.

[80] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using {Gadget-Chain} length to prevent {Code-Reuse} attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, 2014.

[81] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 227–242. IEEE, 2018.

[82] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*, volume 19. USENIX Association Berkeley, 1996.

[83] Li Gong and Roland Schemers. Implementing protection domains in the javatm development kit 1.2. In *NDSS*. Citeseer, 1998.

[84] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[85] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[86] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1016–1031, 2015.

[87] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1787–1804, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/guo.

[88] Aditi Gupta, Javid Habibi, Michael S Kirkpatrick, and Elisa Bertino. Marlin: Mitigating code reuse attacks using code randomization. *IEEE Transactions on Dependable and Secure Computing*, 12(3):326–337, 2014.

[89] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[90] Troy D Hanson and Arthur O'Dwyer. uthash: A hash table for c structures, 2013.

[91] T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. Software complexity analysis using halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pages 1109–1113. IEEE, 2017.

[92] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor:{Intra-Process} isolation for {High-Throughput} data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, 2019.

[93] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

[94] Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, and Jack Davidson. Zipr++: Exceptional binary rewriting. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, FEAST '17, page 9–15, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450353953. URL https://doi.org/10.1145/3141235.3141240.

[95] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-5524-7. URL https://doi.org/10.1109/CGO.2013.6494997.

[96] Shohreh Hosseinzadeh, Samuel Laurén, Sampsa Rauti, Sami Hyrynsalmi, Mauro Conti, and Ville Leppänen. Obfuscation and diversification for securing cloud computing. In *Enterprise Security: Second International Workshop, ES 2015, Vancouver, BC, Canada, November 30–December 3, 2015, Revised Selected Papers*, pages 179–202. Springer, 2017.

[97] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, 104:72 – 93, 2018. ISSN 0950-5849.

[98] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[99] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1470–1486, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243797. URL https://doi.org/10.1145/3243734.3243797.

[100] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. {KSplit}: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, 2022.

[101] Intel. Pin - a dynamic binary instrumentation tool. URL https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[102] Intel. *Control-flow Enforcement Technology Preview*, 2017. https://binpwn.com/papers/control-flow-enforcement-technology-preview.pdf.

[103] Intel. Speculative execution side channel mitigations. Technical report, Intel Corporation, 2018. URL https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[104] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. *Compiler-Generated Software Diversity*, pages 77–98. Springer New York, New York, NY, 2011.

[105] Todd Jackson, Andrei Homescu, Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Diversifying the software stack using randomized nop insertion. In Sushil Jajodia, Anup K. Ghosh, V. S. Subrahmanian, Vipin Swarup, Cliff Wang, and Xiaoyang Sean Wang, editors, *Moving Target Defense*, volume 100 of *Advances in Information Security*, pages 151–173. Springer, 2013. ISBN 978-1-4614-5415-1.

[106] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.30. URL https://doi.org/10.1109/ICSE.2007.30.

[107] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. Annotating, tracking, and protecting cryptographic secrets with cryptompk. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 650–665. IEEE, 2022.

[108] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015. doi: 10.1109/SPRO.2015.10.

[109] Paul A Karger. Limiting the damage potential of discretionary trojan horses. In *1987 IEEE Symposium on Security and Privacy*, pages 32–32. IEEE, 1987.

[110] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 121–132, 2012.

[111] Arslan Khan, Dongyan Xu, and Dave Jing Tian. Ec: Embedded systems compartmentalization via intra-kernel isolation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2990–3007. IEEE, 2023.

[112] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng. Adaptive call-site sensitive control flow integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 95–110, 2019.

[113] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339–348. IEEE, 2006.

[114] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[115] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42314-1. URL http://dl.acm.org/citation.cfm?id=647170.718283.

[116] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 437–452, 2017.

[117] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *2018 IEEE symposium on security and privacy (SP)*, pages 461–477. IEEE, 2018.

[118] Lukáš KORENČIK. *Decompiling binaries into LLVM IR using McSema and Dyninst*. PhD thesis, Masarykova univerzita, Fakulta informatiky, 2019.

[119] Tim Kornau et al. *Return oriented programming for the ARM architecture*. PhD thesis, Master's thesis, Ruhr-Universität Bochum, 2010.

[120] Jay Kreibich. *Using SQLite*. " O'Reilly Media, Inc.", 2010.

[121] Volodymyr Kuznetzov, Laszlo Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018.

[122] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[123] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1:323–337, 1992.

[124] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014.

[125] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[126] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7): 107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

[127] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[128] Zhipeng Li, Yan Ding, Xiaofan Chen, Pan Dong, Chenlin Huang, Liantao Song, and Peng Wang. Agile approach on the performance prediction of arm trustzone-based mandatory access control security enhancement. In *2021 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 1083–1090. IEEE, 2021.

[129] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[130] Yu Liang, Xinjie Ma, Daoyuan Wu, Xiaoxiao Tang, Debin Gao, Guojun Peng, Chunfu Jia, and Huanguo Zhang. Stack layout randomization with minimal rewriting of android binaries. In Soonhak Kwon and Aaram Yun, editors, *Information Security and Cryptology - ICISC 2015*, pages 229–245, Cham, 2016. Springer International Publishing.

[131] Jarred Adam Ligatti. *Policy enforcement via program monitoring*. Princeton University, 2006.

[132] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[133] Hans Liljestrand, Carlos Chinea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N Asokan. Color my world: Deterministic tagging for memory safety. *arXiv preprint arXiv:2204.03781*, 2022.

[134] Jay P. Lim and Santosh Nagarakatte. Automatic equivalence checking for assembly implementations of cryptography libraries. pages 37–49, 2019. URL http://dl.acm.org/citation.cfm?id=3314872.3314880.

[135] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger

Kapitza, et al. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, 2017.

[136] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: Cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 667–678, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5937-5. URL http://doi.acm.org/10.1145/3238147.3238199.

[137] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM. ISBN 1-59593-339-5. URL http://doi.acm.org/10.1145/1150402.1150522.

[138] Hongjiu Liu. *Control-flow Enforcement Technology*, 2018. URL https://www.linuxplumbersconf.org/event/2/contributions/147/attachments/72/83/CET-LPC-2018.pdf.

[139] Yu Liu, Kejiang Ye, and Cheng-Zhong Xu. Performance evaluation of various risc processor systems: A case study on arm, mips and risc-v. In *International Conference on Cloud Computing*, pages 61–74. Springer, 2021.

[140] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1607–1619. ACM, 2015. doi: 10.1145/2810103.2813690. URL https://doi.org/10.1145/2810103.2813690.

[141] LLVM. Memory-tagged sanitizer design documentation, 2018. URL https://llvm.org/docs/MemTagSanitizer.html.

[142] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1867–1881, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3354244. URL https://doi.org/10.1145/3319535.3354244.

[143] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065034. URL https://doi.org/10.1145/1064978.1065034.

[144] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 389–400, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. URL https://doi.acm.org/10.1145/2635868.2635900.

[145] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, 2017.

[146] Hector Marco-Gisbert and Ismael Ripoll. return-to-csu: a new method to bypass 64-bit linux aslr. In *Black Hat*, March 2018. URL https://www.blackhat.com/asia-18/. Black Hat Asia 2018, Black Hat ; Conference date: 20-03-2018 Through 23-03-2018.

[147] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security Symposium*, volume 10, pages 209–224, 2006.

[148] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakc. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[149] MITRE. Cve-2013-2028, 2013. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028.

[150] MITRE. Cve-2014-0160, 2013. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[151] MITRE. Cve-2016-4450, 2016. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4450.

[152] MITRE. Trends in real-world cwes: 2019 to 2023, 2023. URL https://cwe.mitre.org/top25/archive/2023/2023_trends.html.

[153] MITRE. Cve-2024-0684, 2024. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-0684.

[154] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999.

[155] Magnus O. Myreen, Michael J. C. Gordon, Konrad Slind, and Rockwell Collins Usa. Decompilation into logic — improved.

[156] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[157] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 31–40, 2010.

[158] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 175–184, 2014.

[159] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. Swivel: Hardening {WebAssembly} against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1433–1450, 2021.

[160] Pascal Nasahl, Robert Schilling, Mario Werner, Jan Hoogerbrugge, Marcel Medwed, and Stefan Mangard. *CrypTag: Thwarting Physical and Logical Memory Vulnerabilities Using Cryptographically Colored Memory*, page 200–212. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450382878. URL https://doi.org/10.1145/3433210.3453684.

[161] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250746. URL https://doi.org/10.1145/1250734.1250746.

[162] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[163] Zhenyu Ning, Chenxu Wang, Yinhua Chen, Fengwei Zhang, and Jiannong Cao. Revisiting arm debugging features: Nailgun and its defense. *IEEE Transactions on Dependable and Secure Computing*, (01):1–1, 2021.

[164] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, and Ahmad-Reza Sadeghi. Hardscope: Thwarting dop with hardware-assisted run-time scope enforcement, 2018.

[165] Hamed Okhravi. A cybersecurity moonshot. *IEEE Security and Privacy*, 19(3):8–16, 2021.

[166] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1937. IEEE, 2021.

[167] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask, 2020.

[168] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Practical software diversification using in-place code randomization. In *Moving Target Defense*, 2013.

[169] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 241–254. USENIX Association, 2019. URL https://www.usenix.org/conference/atc19/presentation/park-soyeon.

[170] Aditi Partap and Dan Boneh. Memory tagging: A memory efficient design, 2022. URL https://arxiv.org/abs/2209.00307.

[171] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *2009 International Symposium on Code Generation and Optimization*, pages 126–135. IEEE, 2009.

[172] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

[173] Potchik, Brian. Architecture Agnostic Function Detection In Binaries. https://binary.ninja/2017/11/06/architecture-agnostic-function-detection-in-binaries.html, 2017. Online.

[174] Manish Prasad and Tzicker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, 2003.

[175] David A Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. URL http://dl.acm.org/citation.cfm?id=2032305.2032360.

[176] Tristan Ravitch. Wllvm: Whole-program llvm, 2014.

[177] James R. Reinders. *Processor Tracing*, 2013. https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html.

[178] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[179] Nick Roessler and André DeHon. Scalpel: Exploring the limits of tag-enforced compartmentalization. volume 18, pages 1–28. ACM New York, NY, 2021.

[180] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009. ISSN 0167-6423. URL http://dx.doi.org/10.1016/j.scico.2009.02.007.

[181] Mark Rutland. Armv8. 3 pointer authentication. *Linux Security Summit 2017*, 2017.

[182] Markku-Juhani O. Saarinen, G. Richard Newell, and Ben Marshall. Building a modern trng: An entropy source interface for risc-v. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, ASHES'20, page 93–102, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380904. doi: 10.1145/3411504.3421212. URL https://doi.org/10.1145/3411504.3421212.

[183] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[184] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167 (1-2):131–170, 1996.

[185] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[186] Pierangela Samarati and Sabrina Capitani De Vimercati. Access control: Policies, models, and mechanisms. In *International school on foundations of security analysis and design*, pages 137–196. Springer, 2000.

[187] Sascha Schirra. Ropper github webpage, 2020. https://github.com/sashs/Ropper, Online, accessed 01/23/2023.

[188] Len Sassaman, Meredith L Patterson, Sergey Bratus, and Anna Shubina. The halting problems of network stack insecurity. *USENIX; login*, 36(6):22–32, 2011.

[189] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Ben Zorn. Yarra: An extension to c for data integrity and partial safety. csf '11.

[190] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for {PKU-based} memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, 2022.

[191] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, 2015.

[192] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *In Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE*, pages 45–54. IEEE Computer Society, 2002.

[193] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary {CPU} architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[194] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE international symposium on workload characterization (IISWC)*, pages 137–148. IEEE, 2011.

[195] Kostya Serebryany. Arm memory tagging extension and how it improves c/c++ memory safety. *Login USENIX Mag*, 44(5), 2019.

[196] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.

[197] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[198] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, page 298–307, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581139616. doi: 10.1145/1030083.1030124. URL https://doi.org/10.1145/1030083.1030124.

[199] Yarden Shafir and Alex Ionescu. R.I.P ROP: CET internals in windows 20h1, 2020. URL https://windows-internals.com/cet-on-windows/.

[200] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1527–1543. IEEE, 2020.

[201] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[202] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 157–168, New York, NY, USA, 2006. ACM. ISBN 1-59593-263-1. URL http://doi.acm.org/10.1145/1146238.1146256.

[203] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[204] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE symposium on security and privacy*, pages 574–588. IEEE, 2013.

[205] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[206] Mikhail Styugin, Vyacheslav Zolotarev, Anton Prokhorov, and Roman Gorbil. New approach to software code diversification in interpreted languages based on the moving target technology. In *2016 IEEE 10th International Conference on Application of Information and Communication Technologies (AICT)*, pages 1–5. IEEE, 2016.

[207] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[208] Xiaoyan Sun, Jun Dai, Peng Liu, Anoop Singhal, and John Yen. Using bayesian networks for probabilistic identification of zero-day attack paths. *IEEE Transactions on Information Forensics and Security*, 13(10):2506–2521, 2018. doi: 10.1109/TIFS.2018.2821095.

[209] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 143–156, 2020.

[210] Gang Tan et al. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends® in Privacy and Security*, 1(3):137–198, 2017.

[211] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[212] Victor van der Veen, Dennis Andriesse, Enes Goktacs, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 927–940, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813673. URL https://doi.org/10.1145/2810103.2813673.

[213] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689, 2017.

[214] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by) pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 266–282, 2022.

[215] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[216] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *In The Network and Distributed System Security Symposium*, NDSS '17, 2017. doi: 10.14722/ndss.2017.23225.

[217] Shuai Wang, Pei Wang, and Dinghao Wu. Composite software diversification. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 284–294. IEEE, 2017.

[218] Wei Wang, Zhong Shao, Xinyu Jiang, and Yu Guo. A simple model for certifying assembly programs with first-class function pointers. In *Proceedings of the 2011 Fifth International Conference on Theoretical Aspects of Software Engineering*, TASE '11, pages 125–132, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4506-6. doi: 10.1109/TASE.2011.16. URL http://dx.doi.org/10.1109/TASE.2011.16.

[219] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 157–168, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316514. URL https://doi.org/10.1145/2382196.2382216.

[220] Junzo Watada, Arunava Roy, Ruturaj Kadikar, Hoang Pham, and Bing Xu. Emerging trends, techniques and open issues of containerization: A review. *IEEE Access*, 7: 152443–152472, 2019.

[221] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for {UNIX}. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[222] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[223] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[224] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 367–382, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king.

[225] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[226] Gregory Wroblewski. General method of program code obfuscation. 2002.

[227] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. {KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1187–1204, 2019.

[228] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G Neumann, Simon W Moore, Robert NM Watson, et al. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 545–557, 2019.

[229] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.

[230] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 363–376, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. URL http://doi.acm.org/10.1145/3133956.3134018.

[231] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 70–82, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. URL http://doi.acm.org/10.1145/349299.349313.

[232] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*, pages 101–104. IEEE, 2012.

[233] Zachary Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. 2024.

[234] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[235] William D Young and John Mchugh. Coding for a believable specification to implementation mapping. In *1987 IEEE Symposium on Security and Privacy*, pages 140–140. IEEE, 1987.

[236] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-assisted fine-grained code-reuse attack detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, page 66–85, Berlin, Heidelberg, 2015. Springer-Verlag. ISBN 9783319263618. doi: 10.1007/978-3-319-26362-5_4. URL https://doi.org/10.1007/978-3-319-26362-5_4.

[237] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI*, volume 8, pages 225–240, 2008.

[238] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE symposium on security and privacy*, pages 559–573. IEEE, 2013.

[239] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. Statically discovering high-order taint style vulnerabilities in os kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 811–824, 2021.

[240] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 337–352, USA, 2013. USENIX Association. ISBN 9781931971034.

[241] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei Zhou. A comparative study of containers and virtual machines in big data environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 178–185. IEEE, 2018.

[242] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2019.

[243] Xianglong Zhang, Wei Wang, Peng Xu, Laurence T Yang, and Kaitai Liang. High recovery with fewer injections: Practical binary volumetric injection attacks against dynamic searchable encryption. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5953–5970, 2023.

[244] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, and Jianwei Niu. Condysta: Context-aware dynamic supplement to static taint analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 796–812. IEEE, 2021.

[245] Xin Zhao, Kevin Borders, and Atul Prakash. Virtual machine security systems. *Advances in Computer Science and Engineering*, 1:339–365, 2009.

[246] Mengya Zheng, Xingyu Pan, and David Lillis. Codex: Source code plagiarism detection based on abstract syntax tree. In *AICS*, 2018.

[247] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 558–569, 2014.

[248] Mohamed Tarek Ibn Ziad, Miguel A Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-fat: Architectural support for low overhead memory safety checks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 916–929. IEEE, 2021.