

The Continuing Arms Race

*Code-Reuse Attacks
and Defenses*

Edited by
Per Larsen
Ahmad-Reza Sadeghi



The Continuing Arms Race

ACM Books

Editor in Chief

M. Tamer Özsu, *University of Waterloo*

ACM Books is a new series of high-quality books for the computer science community, published by ACM in collaboration with Morgan & Claypool Publishers. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

The Continuing Arms Race: Code-Reuse Attacks and Defenses

Editors: Per Larsen, *Immunant, Inc.*

Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*

2018

Frontiers of Multimedia Research

Editor: Shih-Fu Chang, *Columbia University*

2018

Shared-Memory Parallelism Can Be Simple, Fast, and Scalable

Julian Shun, *University of California, Berkeley*

2017

Computational Prediction of Protein Complexes from Protein Interaction Networks

Sriganesh Srihari, *The University of Queensland Institute for Molecular Bioscience*

Chern Han Yong, *Duke-National University of Singapore Medical School*

Limsoon Wong, *National University of Singapore*

2017

The Handbook of Multimodal-Multisensor Interfaces, Volume 1: Foundations, User Modeling, and Common Modality Combinations

Editors: Sharon Oviatt, *Incaa Designs*

Björn Schuller, *University of Passau and Imperial College London*

Philip R. Cohen, *Voicebox Technologies*

Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*

Gerasimos Potamianos, *University of Thessaly*

Antonio Krüger, *German Research Center for Artificial Intelligence (DFKI)*

2017

Communities of Computing: Computer Science and Society in the ACM

Thomas J. Misa, Editor, *University of Minnesota*

2017

Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining

ChengXiang Zhai, *University of Illinois at Urbana-Champaign*
Sean Massung, *University of Illinois at Urbana-Champaign*
2016

An Architecture for Fast and General Data Processing on Large Clusters

Matei Zaharia, *Stanford University*
2016

Reactive Internet Programming: State Chart XML in Action

Franck Barbier, *University of Pau, France*
2016

Verified Functional Programming in Agda

Aaron Stump, *The University of Iowa*
2016

The VR Book: Human-Centered Design for Virtual Reality

Jason Jerald, *NextGen Interactions*
2016

Ada's Legacy: Cultures of Computing from the Victorian to the Digital Age

Robin Hammerman, *Stevens Institute of Technology*
Andrew L. Russell, *Stevens Institute of Technology*
2016

Edmund Berkeley and the Social Responsibility of Computer Professionals

Bernadette Longo, *New Jersey Institute of Technology*
2015

Candidate Multilinear Maps

Sanjam Garg, *University of California, Berkeley*
2015

Smarter Than Their Machines: Oral Histories of Pioneers in Interactive Computing

John Cullinane, *Northeastern University; Mossavar-Rahmani Center for Business and Government, John F. Kennedy School of Government, Harvard University*
2015

A Framework for Scientific Discovery through Video Games

Seth Cooper, *University of Washington*
2014

Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers

Bryan Jeffrey Parno, *Microsoft Research*
2014

Embracing Interference in Wireless Systems

Shyamnath Gollakota, *University of Washington*
2014

The Continuing Arms Race

Code-Reuse Attacks and Defenses

Per Larsen

Immunant, Inc.

Ahmad-Reza Sadeghi

Technische Universität Darmstadt

ACM Books #18



Copyright © 2018 by the Association for Computing Machinery
and Morgan & Claypool Publishers

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews—without the prior permission of the publisher.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan & Claypool is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

The Continuing Arms Race

Per Larsen, Ahmad-Reza Sadeghi, editors

books.acm.org

www.morganclaypoolpublishers.com

ISBN: 978-1-97000-183-9 hardcover

ISBN: 978-1-97000-180-8 paperback

ISBN: 978-1-97000-181-5 eBook

ISBN: 978-1-97000-182-2 ePub

Series ISSN: 2374-6769 print 2374-6777 electronic

DOIs:

10.1145/3129743 Book	10.1145/3129743.3129749 Chapter 5
10.1145/3129743.3129744 Preface	10.1145/3129743.3129750 Chapter 6
10.1145/3129743.3129745 Chapter 1	10.1145/3129743.3129751 Chapter 7
10.1145/3129743.3129746 Chapter 2	10.1145/3129743.3129752 Chapter 8
10.1145/3129743.3129747 Chapter 3	10.1145/3129743.3129753 References
10.1145/3129743.3129748 Chapter 4	

A publication in the ACM Books series, #18

Editor in Chief: M. Tamer Özsu, *University of Waterloo*

First Edition

10 9 8 7 6 5 4 3 2 1

Contents

Preface xi

Chapter 1 How Memory Safety Violations Enable Exploitation of Programs 1

Mathias Payer

- 1.1 Memory Safety 4
- 1.2 Data Integrity 8
- 1.3 Confidentiality 10
- 1.4 Data-Flow and Control-Flow Integrity 11
- 1.5 Policy Enforcement 15
- 1.6 An Adversary's Toolkit 16
- 1.7 Conclusion 22

Chapter 2 Protecting Dynamic Code 25

Gang Tan, Ben Niu

- 2.1 Overview of Challenges and Solutions 26
- 2.2 Type-Based CFG Generation 28
- 2.3 Handling Dynamically Linked Libraries 39
- 2.4 Handling Just-In-Time Compiled Code 48
- 2.5 Related Work 58
- 2.6 Conclusion 60

Chapter 3 Diversity and Information Leaks 61

Stephen Crane, Andrei Homescu, Per Larsen, Hamed Okhravi, Michael Franz

- 3.1 Software Diversity 62
- 3.2 Information Leakage 63

- 3.3 Mitigating Information Leakage 64
- 3.4 Address Oblivious Code Reuse 69
- 3.5 Countering Address-Oblivious Code Reuse 70
- 3.6 Evaluation of Code-Pointer Authentication 74
- 3.7 Conclusion 78

Chapter 4 Code-Pointer Integrity 81

*Volodymyr Kuznetsov, László Szekeres, Mathias Payer,
George Candea, R. Sekar, Dawn Song*

- 4.1 Introduction 81
- 4.2 Related Work 84
- 4.3 Threat Model 87
- 4.4 Design 87
- 4.5 The Formal Model of CPI 97
- 4.6 Implementation 102
- 4.7 Evaluation 109
- 4.8 Conclusion 116

Chapter 5 Evaluating Control-Flow Restricting Defenses 117

*Enes Göktaş, Elias Athanasopoulos, Herbert Bos,
Georgios Portokalidis*

- 5.1 Introduction 117
- 5.2 Control-Flow Restricting Defenses 119
- 5.3 Security Analysis 122
- 5.4 Quantifying Gadget Availability in CFR 127
- 5.5 Proof-of-Concept Exploit against CFR 131
- 5.6 Summary 135

Chapter 6 Attacking Dynamic Code 139

Felix Schuster, Thorsten Holz

- 6.1 Goals and Attacker Model 140
- 6.2 Counterfeit Object-Oriented Programming 142
- 6.3 Loopless Counterfeit Object-Oriented Programming 157
- 6.4 A Framework for Counterfeit Object-Oriented Programming 160
- 6.5 Proof-of-Concept Exploits 162
- 6.6 Discussion 168
- 6.7 Security Assessment of Existing Defenses 173
- 6.8 Conclusion 179

Chapter 7 Hardware Control Flow Integrity 181

*Yier Jin, Dean Sullivan, Orlando Arias, Ahmad-Reza Sadeghi,
Lucas Davi*

- 7.1 Introduction 181
- 7.2 Threat Model and Assumptions 184
- 7.3 Requirements 185
- 7.4 Modeling CFI 186
- 7.5 Constructing a Precise Stateful CFI Policy 190
- 7.6 Hardware-Enhanced CFI: Design and Implementation 192
- 7.7 Security Evaluation 200
- 7.8 Performance Evaluation 205
- 7.9 Related Work 208
- 7.10 Conclusion 210

Chapter 8 Multi-Variant Execution Environments 211

Bart Coppens, Bjorn De Sutter, Stijn Volckaert

- 8.1 General Design of an MVEE 212
- 8.2 Implementation of GHUMVEE 217
- 8.3 Inconsistencies and False Positive Detections 220
- 8.4 Comprehensive Protection against Code-Reuse Attacks 233
- 8.5 Relaxed Monitoring 241
- 8.6 Evaluation 250
- 8.7 Conclusion 259

References 261

Contributor Biographies 283

Preface

Our societies are becoming increasingly dependent on emerging technologies and connected computer systems that are increasingly trusted to store, process, and transmit sensitive data. While generally beneficial, this shift also raises many security and privacy challenges. The growing complexity and connectivity offers adversaries a large attack surface. In particular, the connection to the Internet facilitates remote attacks without the need for physical access to the targeted computing platforms. Attackers exploit security vulnerabilities in modern software with the ultimate goal of taking control over the underlying computing platforms. There are various causes of these vulnerabilities, the foremost being that the majority of software (including operating systems) is written in unsafe programming languages (mainly C and C++) and by developers who are by-and-large not security experts.

Memory errors are a prominent vulnerability class in modern software: they persist for decades and still are used as the entry point for today's state-of-the-art attacks. The canonical example of a memory error is the stack-based buffer overflow vulnerability, where the adversary overflows a local buffer on the stack, and overwrites a function's return address. While modern defenses protect against this attack strategy, many other avenues for exploitation exist, including those that leverage heap, format string, or integer overflow vulnerabilities.

Given a memory vulnerability in the program, the adversary typically provides a malicious input that exploits this vulnerability to trigger malicious program actions not intended by the benign program. This class of exploits aims to hijack the control flow of the program and differs from conventional malware, which encapsulates the malicious code inside a dedicated executable that needs to be executed on the target system and typically requires no exploitation of a program bug.

As mentioned above, the continued success of these attacks is mainly attributed to the fact that large portions of software programs are implemented in type-unsafe languages (C, C++, or Objective-C) that do not guard against malicious program inputs using bounds checking, automatic memory management, etc. However, even

type-safe languages like Java rely on virtual machines and complex runtimes that are in turn implemented in type-unsafe languages out of performance concerns. Unfortunately, as modern applications grow more complex, memory errors and vulnerabilities will likely continue to exist, with no end in sight.

Regardless of the attacker's method of choice, exploiting a vulnerability and gaining control over an application's control flow is only the first step of an attack. The second step is to change the behavior of the compromised application to perform malicious actions. Traditionally, this has been realized by injecting malicious code into the application's address space, and later executing the injected code. However, with the widespread enforcement of data execution prevention (DEP), such attacks are more difficult to launch today. Unfortunately, the long-held assumption that only code injection posed a risk was shattered with the introduction of code-reuse attacks, such as return-into-libc and return-oriented programming (ROP). As the name implies, code-reuse attacks do not require any code injection and instead repurpose benign code already resident in memory.

Code-reuse techniques are applicable to a wide range of computing platforms: x86-based platforms, embedded systems running on an Atmel AVR processor, mobile devices based on ARM, PowerPC-based Cisco routers, and voting machines deploying a z80 processor. Moreover, the powerful ROP technique is Turing-complete, i.e., it allows an attacker to execute arbitrary malicious code.

In fact, the majority of state-of-the-art run-time exploits leverage code-reuse attack techniques, e.g., against Internet Explorer, Apple QuickTime, Adobe Reader, Microsoft Word, or the GnuTLS library. Even large-scale cyberattacks such as the popular Stuxnet worm, which damaged Iranian centrifuge rotors, incorporated code-reuse attack techniques.

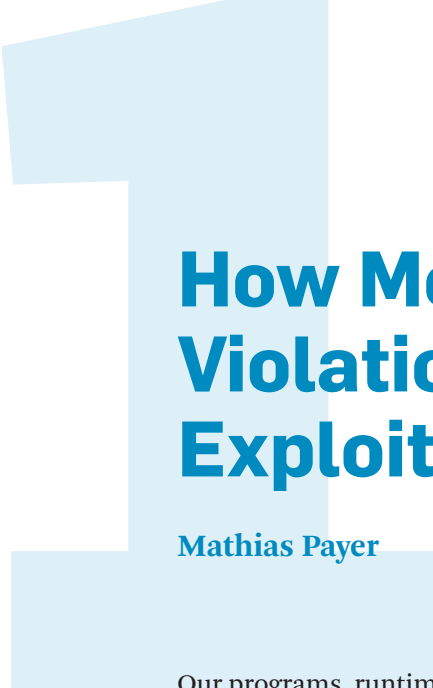
Indeed, even after more than three decades, memory corruption exploits remain a clear and present danger to the security of modern software and hardware platforms. This is why the research community both in academia and industry have invested major efforts in the recent past to mitigate the threat. Various defenses have been proposed and even deployed by Google, Microsoft, Intel, etc. The most prominent defenses are based on enforcement (e.g., Control-Flow Integrity [CFI]) or randomization (also known as software diversity) of certain program aspects. Both types of defenses have distinct advantages and disadvantages. Randomization makes it hard for attackers to chain together their attack gadgets, is efficient, and can be applied to complex software such as web browsers. It can have different levels of granularity, from a simple Address Space Layout Randomization (ASLR) to fine-grained randomization at function or even instruction level. However, randomization requires high entropy and all randomization schemes are inherently

vulnerable to information leakage. CFI, on the other hand, provides guarantees that the application does not deviate from the intended program flow, yet it requires a security and efficiency tradeoff: i.e., coarse-grained CFI have been shown to be vulnerable, and fine-grained CFI can be inefficient without hardware support. Researchers have been working on improving these schemes with novel ideas in both software and hardware design. Many proposed defenses have been bypassed by other attacks, generating a large body of literature on this topic.

It seems that the arms race between attackers and defenders continues. Although researchers have raised the bar for adversaries, there are still a number of challenges to tackle against sophisticated attacks. Despite all the recent proposals on various defenses, we cannot claim that the problem is entirely solved. However, our community has gained much insight through recent results on how and to what extent we need to employ certain design principles to significantly reduce the effect of code-reuse attacks.

The main purpose of this book is to provide readers with some of the most influential works on run-time exploits and defenses. We hope that this material will inspire readers and generate new ideas and paradigms.

Per Larsen
Ahmad-Reza Sadeghi
February 2018



How Memory Safety Violations Enable Exploitation of Programs

Mathias Payer

Our programs, runtime systems, operating systems, and hypervisors are, to a large extent, written in low-level languages like C or C++. These systems languages were initially designed more than 30 years ago when performance was the key metric and security was, at best, a side note. Our systems languages do not enforce memory safety and force the programmer to include necessary safety checks. Coding guidelines and code quality continuously improved over time due to an increased awareness of security. Along with this awareness, language standards and compilers evolved as well and became more powerful, especially compilers that now offer (optional) safety checks. Yet, despite these improvements some attack surface remains. Adversaries can abuse bugs that cause memory safety violations to change the semantics of the program, executing the adversary's desired behavior. Memory safety issues are a problem not just for legacy software that was developed decades ago but also for new software, such as Google Chrome or Microsoft Edge, both large software projects that started just a couple of years ago under strict coding and testing guidelines.

Missing checks in source code are the root cause of different forms of memory safety issues. Examples of such safety issues are (i) buffer overflows or arbitrary memory corruption if a pointer dereferences memory outside the bounds of the underlying memory object, (ii) use-after-free conditions if a reference to a previously freed memory object is dereferenced, and (iii) type confusion if a base object is cast down to a subtype. Invalid pointers are easily created by mistake: sloppy pointer arithmetic, off-by-N mistakes in loop bounds, or integer errors (e.g., an

integer value is incremented past its maximum and flips into a negative value or a truncated type is used in a comparison, especially when used with memory allocation functions). Adversaries can leverage these bugs to change the semantics of the executing program by, e.g., modifying variables that are used in comparisons outside of regular control flow, overwriting code pointers (references in the data region that point to a code segment) to redirect execution, or leaking information by copying sensitive data to an output stream.

There is a fundamental disparity in difficulty between defending software and attacking it. While a programmer must fix all possible bugs and defenses must cover all possible attack vectors to secure a system, an adversary only needs to find one exploitable bug to gain control of the system. This disparity makes defense challenging. Finding and fixing through formal methods, static bug-finding techniques, or fuzz testing is important. However, due to the large state space of programs, it is unlikely that all bugs will ever be found. Systems therefore must rely on defense mechanisms to protect themselves against attacks. Each defense mechanism enforces some security policy. Runtime monitors observe the state of the program according to the underlying security policy and terminate the process if a violation is detected. When designing defenses, researchers must consider if the defense will fundamentally stop an attack vector and provide a real solution or if it will merely “raise the bar.” Arguing how much a defense raises the bar is challenging and may be questionable. For example, adversaries tend to invent and develop automated tools that bypass defenses. A fundamental problem when evaluating defenses is the lack of quantitative metrics and benchmarks to assess security qualities and measuring *how much* and *how effectively* a defense raises the bar. Another interesting design decision is *where* or *when* the attack is detected. Memory safety stops the process before the adversary writes a single illegal byte to the address space while other mechanisms such as stack canaries allow modification of the memory but detect corruption later, when the data is used. Security is an arms race between defenders slowly improving defenses while ensuring availability and compatibility while adversaries continue to probe for new weaknesses.

We discuss the attack surface through memory safety errors for applications and runtime systems written in low level languages. In the *Eternal War in Memory* survey [Szekeress et al. 2013, Szekeress et al. 2014], the authors discuss defenses along four attack vectors: code injection, control-flow hijacking, data-only attacks, and information leaks. Attacks follow the path of least resistance, subject to the adversary’s constraints. From a high-level perspective, an adversary wants to (i) inject, change, or modify code, (ii) hijack the control flow of the application to

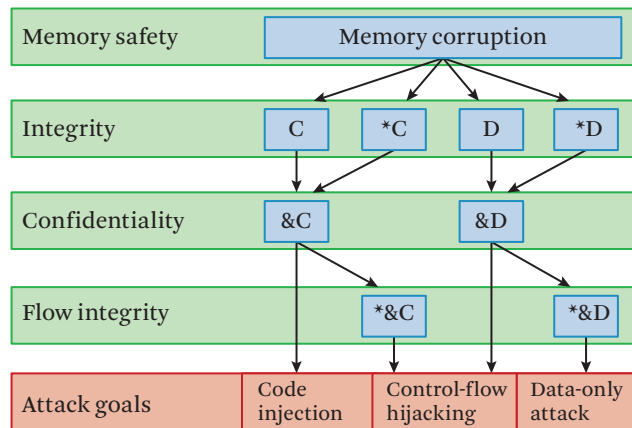


Figure 1.1 Attack surface, showing attack paths from the initial memory safety violation to code injection, control-flow hijacking, and data-only attacks. In the figure, C shows code, D shows data, asterisk (*) marks pointers, ampersand (&) marks addresses, and *& marks a dereference operation, e.g., *&C tells us that a code pointer is being dereferenced.

locations that are outside the valid control flow, or (iii) modify data in some way to either leak information or set up non-control-data attacks.

Figure 1.1 shows the attack surface reachable through memory safety errors. An adversary may leverage memory safety errors to execute any of the pictured attack paths. Defenses can stop the attack at any layer before the attack successfully completes. Note that defenses at any layer always allow the adversary to launch a denial of service attack as the underlying bug is still in the program and defenses are (usually) restricted to terminating the offending thread or process. Recovery from memory safety violations without program-specific recovery code is hard due to the semantic gap between execution, source code, and programmer intention. The root cause of the violation could be manifold, e.g., an invalid pointer computation, a memory object that was freed too early, or a mishap in an integer computation. Defenses therefore opt to terminate the offending thread or process, potentially causing denial of service but protecting the integrity of the system and the confidentiality of data.

Starting from a defender's perspective, we follow Figure 1.1 and discuss different security policies, beginning with memory safety at a high level in Section 1.1, followed by data integrity in Section 1.2, confidentiality in Section 1.3, and data-flow integrity in Section 1.4. Security policies running on a system can be enforced

through different mechanisms, which are introduced in Section 1.5. Moving to the adversary's point of view, we introduce techniques and practices that achieve the desired level of control in Section 1.6. These techniques allow understanding of fundamental attack vectors used to compromise systems through memory safety vulnerabilities. We then finish with concluding remarks in Section 1.7.

1.1 Memory Safety

Memory corruption, the absence of memory safety, is the root cause of current high-profile attacks and the foundation of a plethora of different attack vectors. Memory safety is a program property that guarantees memory objects can only be accessed with the corresponding capabilities. At an abstract level, a pointer is a capability to access a certain memory object or memory region [Hicks 2014, Nagarakatte et al. 2009]. A pointer receives capabilities whenever it is assigned and is then allowed to access the pointed-to memory object. The capabilities of a memory object describe the size or area, validity, and type of the underlying object. Capabilities are assigned to a memory object when it is created. The initial pointer returned from the memory allocator receives these capabilities and can then pass them, through assignment, to other pointers. Memory objects can be created explicitly by calling the allocator, implicitly for global data by starting the program, or implicitly for the creation of a stack frame by calling a function. The capabilities are valid as long as that memory object remains alive. Pointers that are created from this initial pointer receive the same capability and may only access the object inside the bounds of that object, and only as long as that object has not been deallocated. Deallocation, either through an explicit call to the memory allocator or through removal of the stack frame by returning to the caller, destroys the memory object and invalidates all capabilities.

Pointer capabilities cover three areas: bounds, validity, and type. The bounds of a memory object encode spatial information of the memory object. *Spatial memory safety* ensures that pointer dereferences are restricted to data *inside* the memory object. Memory objects are only valid as long as they are allocated. *Temporal safety* ensures that a pointer can only be dereferenced as long as the underlying object *stays allocated*. Memory objects can only be accessed if the pointer has the correct type. *Type safety* ensures that the object's type matches one of the *compatible types* according to type inheritance. The C/C++ family of programming languages allows invalid pointers to exist, i.e., a pointer may point to an invalid memory region that is out of bounds or no longer valid. A memory safety violation only triggers when such an invalid pointer is dereferenced.

Memory safety can be enforced at different layers. Language-based memory safety makes it impossible for the programmer to violate memory safety by, e.g., checking each memory access and type cast (Java, C#, or Python) or enforcing a strict static type system (Rust). Systems that retrofit memory safety to C/C++ are commonly implemented at the compiler level due to the availability of pointer and type information. Techniques that retrofit memory safety for C/C++ must track each pointer and its associated bounds for spatial memory safety, validity for temporal memory safety, and associated type for type safety.

1.1.1 Spatial Memory Safety

Spatial memory safety ensures that a pointer can only dereference data that is within the bounds of the assigned object. If an out-of-bounds pointer is dereferenced, a spatial memory safety violation is signaled, terminating the process. Language-based solutions like CCured [Necula et al. 2005] and Cylcone [Jim et al. 2002] enforce a stricter type system on top of the loose typing that C/C++ offer and make memory safety constraints explicit. Compiler-based solutions like SoftBound [Nagarakatte et al. 2009] allocate a disjoint metadata store to track all pointers and their associated bounds. In addition, both language-based and compiler-based solutions instrument pointer assignment and pointer arithmetic to propagate the underlying bounds information and prepend every dereference with an explicit check of whether the pointer is still valid. SoftBound favors a disjoint metadata storage where the address of the pointer is used to look up metadata information. Disjoint metadata eases portability and enables seamless integration of memory safety solutions [Nagarakatte et al. 2015].

Spatial memory safety violations happen if a pointer is incremented past the bounds of the object, e.g., in a loop or through pointer arithmetic:

```
char *c = (char*)malloc(16);
for (int i = 0; i <= 16; i++) {
    // buffer overflow for i == 16
    *c = i;
}
// violation through a direct write
c[16] = 42;
```

1.1.2 Temporal Memory Safety

Temporal memory safety ensures that the pointer can only reference live memory objects. If the underlying memory object is no longer valid, e.g., because it is freed (for heap objects) or the function returned (for stack objects), dereferencing a

stale pointer results in undefined behavior. CETS [Nagarakatte et al. 2010] retrofits temporal memory safety on top of C/C++. In addition to metadata for each pointer, CETS also allocates metadata for each object. Both pointer and object receive an associated identifier, and only if the identifiers match is the object valid and the dereference operation allowed.

Temporal memory safety violations happen if the underlying memory object was freed:

```
char *c = malloc(16);
char *d = c;
free(d);
// violation as c no longer points to a valid object
c[12] = 23;
```

1.1.3 Type Safety

Type safety is a programming language concept that assigns each allocated memory object an associated type. Typed memory objects may only be used at program locations that expect the corresponding type. Casting operations allow an object to be interpreted under a different type. Casting is allowed along the inheritance chain. Upward casts (upcasts) move the type closer to that of the root object so that the type becomes more generic, while downward casts (downcasts) specialize the object to a subtype. For example, consider an inheritance chain from the root object *Animal* over *Mammal* to the specialized type *Ape*. If we have a reference of type *Mammal*, an upcast could cast a *Mammal* into an *Animal* and a downcast could cast a *Mammal* into an *Ape*. The type hierarchy is specified by the programmer according to the semantics of the source programming language. In low-level languages like C or C++, type safety is not explicit and a memory object can be reinterpreted in arbitrary ways. C++ provides a vast set of type cast operations. Static casts are only checked at compile time to ensure that the two types are compatible. Dynamic casts execute a slow runtime check, which is only possible for polymorphic classes with virtual functions; otherwise, no vtable pointer—to identify the object’s type—is available in the memory object layout. Reinterpret casts allow reclassification of a memory object under a different type. Static casts have the advantage that they do not incur any runtime overhead but are purely checked at compile time. Static casts lack any runtime guarantees, and objects of the wrong type may be used at runtime. For example, Listing 1.1 shows a type violation where an object of the base type can be used as a subtype after an illegal downcast. Reinterpretation of casts allows the programmer to explic-

```

class B {
    int b;
};
class D: B {
    int c;
    virtual void d() {}
};
:
:
B *Bptr = new B;
// Type violation:
D *Dptr = static_cast<D*>B;
Dptr->c = 0x43; // Type confusion!
Dptr->d();      // Type confusion!

```

Listing 1.1 Type violation after illegal downcast.

itly break the underlying type assumptions and reassign a different type to the pointer or underlying memory object. Due to the low-level nature of C++, a programmer may write to the raw memory object and change the underlying object directly.

Ideally, a program can statically be proven type safe. Unfortunately, this is not possible for C/C++ and defenses have to resort to runtime checks. If we make all casts in the program explicit and check them for correctness, then we can ensure that the runtime type conforms to the statically assumed type at compile time. UBSan [Project 2013] follows this approach. Unfortunately, enforcing dynamic casts results in prohibitive performance overhead and compatibility issues. UBSan requires manual blacklisting for non-polymorphic classes. In addition, the existing type-checking infrastructure in C++ was optimized under the assumption that only few runtime checks would be executed and is therefore inherently slow. CaVer [Lee et al. 2015] vastly extends the coverage of UBSan by using a disjoint metadata table (similar to other memory safety techniques) to allow explicit type checks on non-polymorphic objects as well. Unfortunately, the instrumentation results in prohibitively high performance overhead. Also, CaVer only supports objects on the heap allocated through the `new` operator; stack objects (`alloca`) and other heap objects (`malloc`) are not supported. TypeSan [Haller et al. 2016] extends coverage over CaVer and reduces performance overhead, enabling an always-on type-checking solution.

1.2 Data Integrity

Data integrity is a policy that ensures integrity of data in the process's address space. For efficiency reasons, individual data integrity policies only protect a subset of data. The protected data may depend on the underlying type, location, or when it was allocated. Defense mechanisms often rely on integrity as a basis, i.e., protecting code regions or data regions against modification. For example, any defense that relies on some form of runtime check relies on code integrity as the adversary could otherwise simply rewrite the underlying code, removing the checks. Data integrity is often implemented through hardware extension, e.g., virtual memory allows setting data on a per-page basis as readable, writable, and/or executable. Each virtual memory page may have different permissions, and the memory management unit issues a page fault if the permissions do not match, e.g., a read-only page is written, an unmapped page is read, or control is transferred to a non-executable page. Software techniques can increase the flexibility and granularity but usually result in higher overhead. Write-Integrity Testing (WIT) [Akritidis et al. 2008] is a software integrity approach. WIT uses points-to analysis at compile time to assign a label to each object and to each memory write. At runtime, the label of each object is recorded and write instructions verify that the labels match. The effectiveness of WIT is limited by the completeness and over-approximation of the points-to analysis when individual labels are merged due to imprecision. Software-based Fault Isolation (SFI) [Wahbe et al. 1993, Hiser et al. 2006, Kiriansky et al. 2002, Ford and Cox 2008, Payer and Gross 2011, Payer 2012, Yee et al. 2009] allows a fine-grained separation and protection of code and data through instrumentation (and verification) of the code. Software masking, a form of SFI, can protect a certain memory area from adversary access by masking each pointer before it is dereferenced. Such a masking scheme can be implemented by prepending each dereference with a logical AND instruction of the memory address.

Most systems enforce data integrity for code and a subset of data. Code is generally marked executable and immutable. For data, the subset of immutable data, such as `const` objects, is generally marked read-only, as are support data structures used by the dynamic loader, such as symbol tables, linkage tables between different libraries, or the kernel export table.

1.2.1 Code Integrity

Code integrity enforces that existing code in a process is immutable. Combining non-executable data (e.g., through virtual memory) and code integrity results in $W \oplus X$ —writable xor executable data [de Raadt 2005, PaX Team 2004b]. Data Execu-

tion Protection (DEP) [Andersen and Abella 2004] can be implemented in two ways: through non-executable data or by instruction set randomization. Non-executable data ensures that only well-marked regions that contain code can be executed. Before the introduction of the no-execute flag on commodity hardware by AMD and Intel around 2006, protection against code injection relied on a combination of segmentation and software techniques [van de Ven 2004], similar to software-based fault isolation [Wahbe et al. 1993, Hiser et al. 2006, Kiriansky et al. 2002, Ford and Cox 2008, Payer and Gross 2011]. Code integrity works well for static programs but is inherently incompatible with just-in-time compilation, dynamic library loading, and binary translation. These features require changing code at runtime, making the enforcement of code integrity policies challenging. For example, an adversary may have a small window of opportunity to compromise code while it is writable. Also, protection of just-in-time compiled code remains an open problem, with some defenses trying to contain control flow to the area where code is dynamically generated [Niu and Tan 2014b].

1.2.2 Code-Pointer Integrity

Ideally, full memory safety would stop all forms of memory corruption, but it is currently prohibitively expensive. Code-Pointer Integrity (CPI) [Kuznetsov et al. 2014a] is an integrity policy that enforces full safety for a subset of data (code pointers). CPI separates control data structures from non-control data structures and enforces that only program code that is supposed to change control data structures is allowed to do so. Code that handles data is not allowed to change code pointers, i.e., a data pointer is forbidden to modify a code pointer. The CPI mechanism uses a type-based analysis to identify sensitive pointers that must be protected. The compiler analysis classifies each pointer as a sensitive pointer or as a data pointer. Sensitive pointers are code pointers and the transitive closure of any pointer pointing to sensitive pointers. The set of sensitive pointers is then protected through memory safety checks, carving out a subset of protected data that includes all code pointers and any data that may be used to modify code pointers. Protecting only a subset of all data allows CPI to drastically reduce the overhead while protecting against any control-flow hijack attack or modification of a code pointer through a memory safety violation. Note that CPI also relies on code integrity, and the mechanism works well for code pointers on the heap or in global data.

1.2.3 Stack Integrity

A thread's stack is a data structure where buffers and code pointers are frequently allocated in close proximity to each other. Protecting code pointers against targeted

overwrites and buffer overflows is challenging due to the volatility and frequent changes. Stack-based protections also face the challenge of frequent allocations and deallocations of stack frames; the individual overhead must therefore be minimal. *Stack integrity* ensures that data on the stack remains integrity protected against illegal modifications.

Strong protections for function returns enforce stack integrity by leveraging the relationship between function calls and returns. A mechanism that enforces stack integrity ensures that any backward-edge transfers can only return to the most recent prior caller. This property can be enforced by storing the prior call sites in a shadow stack or guaranteeing memory safety on the stack, i.e., if the return instruction pointers are immutable, then stack integrity trivially holds. Control-flow enforcement for function returns is “easier” than for indirect function calls because function calls and returns form a symbiotic relationship that can be leveraged in the design of the defense, i.e., a function return always returns to the location of the previous call. Such a relationship does not exist for indirect function calls. Depending on the mechanism, at least return pointers are protected by shadow stacks [Payer et al. 2015c, Dang et al. 2015] and at best the majority of data for safe stacks [Kuznetsov et al. 2014a].

1.3 Confidentiality

Defenses based on confidentiality are probabilistic: if a data value is unknown, the adversary is restricted to guessing the correct value. For defenses that hide memory addresses, randomization results in a search space of $\frac{1}{2^{32}}$ for 32-bit systems or $\frac{1}{2^{64}}$ for 64-bit systems and is highly unlikely to succeed. While practical implementations cannot leverage the full virtual memory space, it remains large enough in practice. Randomization-based defenses hide locations of sensitive data, e.g., code pointers, flags, or privileged data, from an adversary. Through the use of virtual memory, data can be spaced out in a vast address space where the majority of pages remain unmapped. Without knowing the precise address, an adversary will likely trigger a segmentation fault when trying to guess the correct location. All modern systems use Address Space Layout Randomization (ASLR) [Durden 2002, Bhatkar et al. 2003, Bhatkar et al. 2005] to randomize and hide the location of the base addresses of individual segments, e.g., data regions, code regions, and loader data structures of individual shared libraries, the heap, dynamically allocated memory areas, and the stacks of individual threads. Knowing neither the absolute addresses nor the offsets between individual regions, adversaries must rely on either just-in-time exploit construction [Snow et al. 2013] or some form of information leak or side channel to infer the addresses.

For code, randomization of the base address of the code region is often not enough, as adversaries can recover the locations of all functions with a single leaked code pointer if they know the relative offset to the desired function. For a single distribution or operating system, all installations generally share the same files, which allows adversaries to learn such offsets. Fine-grained code randomization schemes [Kil et al. 2006, Hiser et al. 2012, Wartell et al. 2012, Bigelow et al. 2015, Crane et al. 2015, Crane et al. 2015, Larsen et al. 2014] address the problem of well-known intra-file offsets by diversifying files on a per-system or per-process basis by permuting functions and basic blocks to increase the randomness inside single code regions. These systems introduce fine-grained diversity between individual instances of the program and may dynamically rediversify at specific intervals. Data Space Randomization (DSR) [Bhatkar and Sekar 2008, Cowan et al. 2003] increases diversity for data, changing the data or pointer representation dynamically by either encrypting the pointer or introducing an additional layer of indirection to hide the actual data regions from the adversary.

Stack canaries [Hiroaki and Kunikazu 2001] are an early attempt at stack integrity that relies on confidentiality of the canary. Random values are placed before return addresses on the stack. Stack canaries are integrity checked before the function returns to the caller. If an adversary uses a stack-based continuous buffer overflow [One 1996] to hijack control flow, then the canary is corrupted and the check before the return will fail. Unfortunately, it does not protect against direct targeted overwrites of the return address.

All randomization-based defenses are prone to information leaks; if an adversary can leak and infer the location or value of the protected data, then the defense can be bypassed. Randomization-based defenses are usually an additional step the adversary needs to compromise, making the attack harder and less likely to succeed. Recently, several attacks have shown that information hiding only gives limited security and adversaries can often recover the target addresses if a larger piece of data (such as a stack or metadata table) is hidden in the virtual address space [Göktaş et al. 2016, Oikonomopoulos et al. 2016, Evans et al. 2015a].

1.4 Data-Flow and Control-Flow Integrity

Data-flow and control-flow integrity both ensure that data values are valid. Compared to integrity, flow integrity ensures the security property not when data is written but when data is read. Every time a data value is used, a flow integrity check ensures that the value is benign.

Instead of preventing data corruption, Data-Flow Integrity (DFI) [Castro et al. 2006] detects corruption by checking read instructions. This prohibits corrupted

data from being used in computations but allows data in the process to be corrupted through memory safety violations. An adversary can use a corrupted pointer to modify a data value, but the corrupted value cannot be used in a later computation. DFI enforces reaching definitions for data values. Only program locations that are supposed to write a data value pass the check whenever data is read. DFI records the location for each memory write. For each read operation, DFI encodes the set of benign write locations and, at runtime, checks if the last write to that location came from such a benign location. In program terms, when reading the `isAdmin` flag, the read check ensures that the value was last written from a code location that is allowed to write to that address. By delaying the check until the read instruction, DFI can achieve lower overhead than memory safety. Compared to WIT, DFI protects read instructions instead of write instructions. DFI shares the same limitations due to a similar points-to analysis and over-approximation of merged labels.

Control-Flow Integrity (CFI) detects control-flow hijacking attacks by limiting the targets of control-flow transfers. Since the initial idea for the CFI defense mechanism [Abadi et al. 2005a] and the first (closed source) prototype were presented in 2005, a plethora of alternate CFI-style defenses have been proposed and implemented (see Burow et al. [2016], Burow et al. [2017] for a survey). Any CFI mechanism consists of two abstract components: the often static analysis component that recovers the Control-Flow Graph (CFG) of the application with different levels of precision and the dynamic enforcement mechanism that restricts control flows according to the generated CFG (see Figure 1.2).

Listing 1.2 shows a simple program with five functions. The `foo` function uses a function pointer, and the CFI mechanism injects both a forward-edge (e.g., indirect function call) and a backward-edge (function return) check. The function pointer points to either `bar` or `baz`. Depending on the forward-edge analysis, different sets of targets are allowed at runtime.



Figure 1.2 A CFI mechanism restricts control-flow hijacking to targets that are valid according to a pre-determined control-flow graph (black arrows). All other targets are rejected (red arrows).

```

void aa();
void ab();
void ac();
void ad(int, int);

void foo(int usr) {
    void (*func)();

    // func points to either bar or baz
    if (usr == MAGIC)
        func = aa;
    else
        func = ac;

    // forward-edge CFI check
    // depending on the precision of CFI:
    // a) all functions {aa, ab, ac, ad, foo} are valid
    // b) functions with prototype "void (*)()" are valid
    //    i.e., {aa, ab, ac}
    // c) only address-taken functions are valid, i.e., {aa, ac}
    CHECK_CFI_FORWARD(func);
    func();

    // backward-edge CFI check
    CHECK_CFI_BACKWARD();
}

```

Listing 1.2 Simple program that showcases different granularities of precision for CFI mechanisms. Depending on the precision of the static analysis, only a subset of targets is valid.

For forward edges, the CFG generation enumerates all possible targets, often based on information from the source language such as symbol information, function prototypes, or class inheritance information. Switch statements in C/C++ are a good example as the different targets are statically known and the compiler can generate a fixed jump table and emit an indirect jump with a bound check to guarantee that the target used at runtime is one of the valid targets in the switch statement. For indirect function calls through a function pointer, the analysis becomes more complicated as the target may not be known a priori. Common source-based analyses use a type-based approach and, looking at the function prototype of the function pointer that is used, enumerate all matching functions. Different CFI mechanisms use different forms of type equality and use, e.g., any defined function, functions with the same arity (number of arguments), or functions with the same signature

(arity and equivalence of argument types) to distinguish valid call targets. At runtime, any function with matching signature is allowed.

Just looking at function prototypes likely yields several collisions where functions are defined that may never be called in practice. The analysis therefore over-approximates the valid set of targets. In practice, the compiler can check which functions are *address taken*, i.e., there is a source line that generates the address of the function and stores it. The CFI mechanism may reduce the number of allowed targets by intersecting the sets of equal function prototypes and the set of address-taken functions. For virtual calls, i.e., indirect calls in C++ that depend on the type of the object and the class relationship, the analysis can further leverage the type of the object to restrict the valid functions, e.g., the default constructors of all classes have the same signature but only the constructors of the subset of related classes are feasible.

The constructed CFG for the forward edge is stateless, i.e., the *context of the execution* is not considered and each control-flow transfer is independent of all others. On one hand, at runtime only one target is allowed for any possible transfer, namely, the target address currently stored at the memory location of the code pointer. CFG construction, on the other hand, over-approximates the number of valid targets with different granularities, depending on the precision of the analysis. Some mechanisms take path constraints into consideration [Niu and Tan 2015, van der Veen et al. 2015] and check (for a limited depth) if the path taken through the application is feasible by using a dynamic analysis approach that validates the current execution [van der Veen et al. 2015]. So far, only a few mechanisms look at the path context as this incurs dynamic tracking costs at runtime.

CFI can be enforced at different levels. Sometimes the analysis phase (CFG construction) and enforcement phase even overlap [van der Veen et al. 2015, Payer et al. 2015c], for instance, when considering path constraints. Most techniques have two fundamental mechanisms, one for forward-edge transfers and one for backward-edge transfers. Figure 1.2 shows how CFI restricts the set of possible target locations by executing a runtime monitor that validates the target according to the constructed set of allowed targets. If the observed target is not in that set, the program terminates. For forward-edge transfers, the code is often instrumented with some form of equivalence check. The check ensures that the target observed at runtime is in the set of valid targets. This can be done through a full set check or a simple type comparison that, e.g., hashes function prototypes and checks if the hash for the current target equals the expected hash at the call site. The hash for the function can be embedded inline in the function, before the function, or in an orthogonal metadata table.

Most mechanisms for the forward edge are stateless and allow an attacker to redirect control flow to any valid location as identified by the CFG analysis. Limiting the size of the target sets constrains the attacker on the forward edge. Future extensions of CFI should become context sensitive to increase the protection guarantees for the forward edge. If implemented correctly (i.e., considering language-specific information to increase precision [Schuster et al. 2015] and building the analysis on high-level data available at compile time), CFI is an efficient mitigation that constrains the control flow of the process, protecting against control-flow hijacking. Adversaries may still corrupt memory and data-only attacks are still in scope. For the forward edge, a strong mechanism must consider language-specific semantics to restrict the set of valid targets as much as possible.

Backward-edge transfers are harder to protect than forward-edge transfers as they require context sensitivity to be effective. When using a stateless, context-insensitive approach, the attacker may redirect the control flow to any valid call site when returning from a function [Carlini and Wagner 2014, Davi et al. 2014, Göktaş et al. 2014a, Carlini et al. 2015e]. This imprecision often leaves enough targets for an attacker. The backward edge should therefore be protected through a stack integrity mechanism like a shadow stack or safe stack.

1.5 Policy Enforcement

Security defenses rely on a set of assumptions about the generated code and the runtime system. Most assumptions are only encoded indirectly (if at all) and not enforced throughout the software stack. Due to the complexity of the overall software stack, formal verification of components is restricted to high assurance domains with notable exceptions, including the CompCert verified compiler and the seL4 verified kernel, efforts that cannot be repeated for all software.

Many defenses rely on immutable code and non-executable memory regions. These two constraints can be enforced by hardware at the page table level but do not always hold due to exceptions. Another constraint that many code-reuse defenses depend on are immutability of vtables, jump tables, and other sensitive data. Even if the defense ensured that the assumption held at the compiler level where the initial transformation/instrumentation was carried out, there are no guarantees that the assumption remains enforced at runtime. Any software along the stack, compiler optimizations, assembler, linker, runtime loader, or operating system, may break the assumption. For example, the dynamic loader may copy read-only data into a writable section at runtime that then remains writable [Ge et al. 2017], favoring compatibility and performance over security. If a vtable of a class is defined

in a library but the executable implements the constructor, the loader relocates the vtable to a writable data section in the executable at runtime. An attacker may then freely modify the actual vtable code pointers while defense mechanisms like CFI will verify the correctness of the underlying vtable pointer. Another example is a compiler optimization that removes an inserted security check based on the assumption of certain language semantics or a certain memory model. While the compiler is free to do so according to the programming language, an attacker need not adhere to the semantics of the programming language [D’Silva et al. 2015].

Due to compatibility and performance trade-offs, full memory safety is rarely used in practical systems. As other defenses are not complete, allowing either remaining attack vectors or partial attacks, secondary defenses are needed. In addition, layering defenses can decrease the trust needed in a defense mechanism. Secondary defenses like intrusion detection, logging, auditing, or system call monitors allow a second line of defense to detect if a program is compromised.

System call monitors [Provos 2003, Goldberg et al. 1996, Alexandrov et al. 1999, Cowan et al. 2000, Acharya and Raje 2000, Bauer 2006, Fetzer and Suesskraut 2008, Watson et al. 2010] enforce a per-application policy that limits the damage a compromised process can do to the system, e.g., limiting the commands that can be executed and files that can be accessed. Such monitors can be implemented through various enforcement mechanisms, from in-kernel solutions [Wright et al. 2002] to ptrace-based solutions.

1.6 An Adversary’s Toolkit

Adversaries have a clear target in mind when designing attacks. The attack usually follows the simplest path that achieves the desired target under the constraints of the adversary. The goals of the adversary are either to leak information or to gain certain privileges on a system.

Ideally, memory safety (Section 1.1) stops all attacks by detecting or protecting against the initial memory safety violation. As a first step, the adversary needs to modify the process state, leveraging the memory safety violation to corrupt code, code pointers, data, or data pointers. At the level of memory safety, the adversary does not have any capabilities to compromise the process state because the attack is detected when an invalid pointer is dereferenced for reading or writing. Defenses that target the integrity of code or data (Section 1.2) stop the attack at this level. Having privileges to read or write arbitrary data is not enough; due to the large virtual address space, defenses can “hide” sensitive or confidential data by shuffling and randomizing their locations and values (Section 1.3). Defenses that protect

data-flow integrity can still stop an attack despite a powerful adversary that has the ability to change data and knows the addresses of the targets. Data-flow integrity stops the attack whenever the modified data is used. For example, a defense may stop the process whenever a modified code pointer is dereferenced and used to transfer control (Section 1.4). If the attack is not stopped by a defense, then the adversary may execute arbitrary computation in the compromised process. Security policies at all these levels may be enforced through different techniques, relying on either modified hardware, a compiler, or binary rewriting (Section 1.5). To make permanent changes and to interact with the environment, the adversary must rely on system calls or memory-mapped files. Auxiliary or high-level defenses evaluate the state of a process externally to detect anomalies. For example, a defense may check system calls and arguments used by the process to decide, for each system call, if it is in the set of allowed system calls that this program may execute.

1.6.1 Code Corruption

Code corruption allows an adversary to control what code is executed. An adversary may inject new code or modify existing code to change the computation of the program to an arbitrary adversary-controlled computation. Code corruption and code injection were the most common attack vector due to its relative simplicity until the enforcement of code integrity. If the adversary has access to an executable code region and control over the instruction pointer, code corruption is the most straightforward attack vector. Code corruption allows the adversary to gain full control over all memory in the process and to execute arbitrary system calls on behalf of the process [One 1996, Durden 2002, Butler and Anonymous 2004]. See Figure 1.3 for an example of a stack smashing attack that redirects control flow to injected code in a stack-based buffer. Note that this attack assumes either that no stack-based defenses or code injection defenses are present or that they can be circumvented.

Modern systems enforce a separation between code and data regions through an executable bit on a per-page basis. Code corruption modifies memory that is mapped as executable, thereby changing existing instructions or inserting new instructions into the executing program. Code injection adds new code to the process memory by either remapping a data region as executable, writing data to an executable region, or forcing a just-in-time code generator to emit specific code in an executable area. For a successful code corruption attack, the adversary must both circumvent code integrity and recover certain addresses, e.g., the location where code should be emitted.

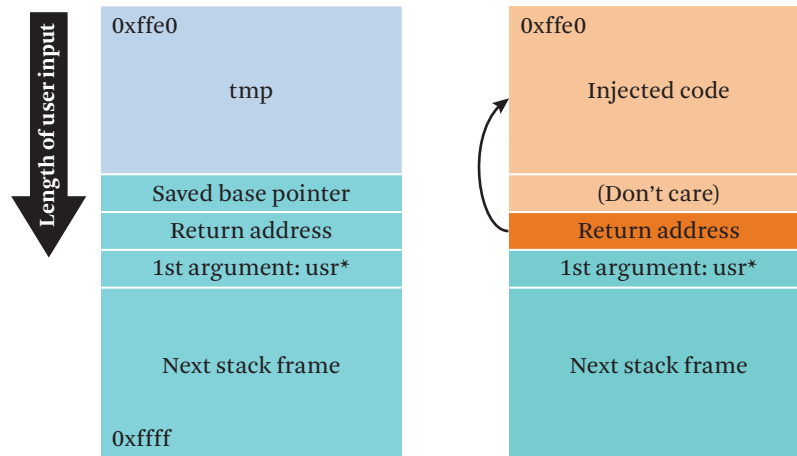


Figure 1.3 Straightforward stack-based code injection and control-flow hijacking through the stored return instruction pointer.

1.6.2 Control-Flow Hijacking

Control-flow hijacking happens whenever an adversary manages to redirect control flow of the program from expected locations (i.e., following control flow as defined through the program’s control-flow graph) to an adversary-controlled location. Common architectures like ARM, x86, x86-64, or MIPS allow two different ways to transfer control (branch) from one location in memory to another location. On the instruction level, control-flow transfers are encoded either direct or indirect. For direct branches, the target is usually a relative distance from the current instruction pointer, and the distance is encoded as an immediate offset in the instruction itself (e.g., `0x75 0x02` encodes a `jne` instruction—jump if the zero bit in the flags is 0—which transfers control two bytes forward in execution flow for x86). Under the assumption of code integrity, i.e., code remains immutable throughout the execution of the program, an adversary cannot modify the target of these instructions as the immediate offset is encoded as part of the immutable instruction itself. While the target of the branch is immutable, an adversary may control the data that is used in the comparison to set the flags and thereby influence whether or not the branch is taken for conditional branches.

Indirect branches allow transfer of control flow to an arbitrary location in memory. The target is usually encoded as an absolute address to a code location, and the indirect control-flow transfer instruction dereferences a memory location or uses the value in a register (e.g., `0xff 0xd0` encodes an indirect call through the `eax/rax`

register on x86/x86-64). An adversary with knowledge of the location of the code pointer may modify the target through an arbitrary memory write. Many platforms have several types of indirect control-flow transfers for different use cases. For example, function returns (`ret` on x86) transfer control indirectly to the address that is at the top of the stack, indirect jumps (`jmp` on x86) transfer control to the specified address in a register or memory location, and indirect calls (`call` on x86) push the current instruction pointer (after the call) to the stack and execute an indirect jump. Indirect control-flow transfers are further divided into *forward-edge* control-flow transfers and *backward-edge* control-flow transfers. Forward-edge control-flow transfers direct control flow forward to a new location and are used in indirect jump and indirect call instructions, which are mapped at the source code level to switch statements, indirect calls, or virtual calls. The backward edge is used to return to a location that was used in a forward-edge transfer earlier, i.e., when returning from a function call. For simplicity, we do not discuss interrupts, interrupt returns, and exceptions in detail.

Memory corruption allows the adversary to modify data outside of the program semantics defined by the programmer. In programs, code pointers, data pointers, sensitive data, and other data are not separated and they are all accessible through the same address space. By changing program semantics, an adversary can use any memory write to change a code pointer. The only isolation between different data types is enforced by the semantics defined through the programmer. If the program is buggy, these semantics may break down and allow arbitrary changes in semantics, leading to so-called weird machines [Bratus et al. 2011]. For example, the programmer may have intended to change a data value on the stack, but due to a miscalculation through an offset that is under the adversary's control, the memory write ends up corrupting a code pointer on the process's heap. In control-flow hijack attacks, adversaries compromise a pointer, make it point to a memory location that contains a code pointer, and corrupt the original code pointer with a target address. When the program follows the corrupted code pointer, the control flow is redirected from a benign intended program location to a location under the adversary's control and the control-flow hijack attack is complete.

1.6.3 Code Reuse

Adversaries use control-flow hijacking to redirect the control flow of the program to alternate executable locations. With the widespread use of defenses enforcing data execution prevention [Andersen and Abella 2004], adversaries had to adjust attacks to achieve code execution. Whenever code injection is infeasible,

the attacker must resort to code reuse. In a code-reuse attack, the adversary repurposes existing code in alternate ways. The existing code of the application is analyzed and broken into small components called *gadgets*. Gadgets usually end in an indirect control-flow transfer (e.g., a return, indirect call, or indirect jump instruction) to allow the adversary to redirect execution to the next gadget. Gadgets can be stitched together in arbitrary ways, allowing adversaries to inject Turing-complete programs by modifying code pointers and supplemental data.

Over time, different flavors of code reuse have been developed. Starting with return to libc [Wojtczuk 1998], privileged functions like `mmap` or `mprotect` were reused to remap injected data as executable. Return-Oriented Programming (ROP) [Roemer et al. 2012] generalized return to libc attacks to Turing-complete ROP payloads, and Jump-Oriented Programming (JOP) [Checkoway et al. 2010, Bletsch et al. 2011] finally moved from return instructions to indirect jump or call instructions to transfer control between individual gadgets. In response to advances in defense techniques, attacks have been refined over the last couple of years [Schuster et al. 2015, Evans et al. 2015, Carlini et al. 2015e, Snow et al. 2013], becoming increasingly powerful. Figure 1.4 shows a stack-based ROP attack. Note that the simplified example assumes that no stack-based defenses are in place but code integrity protects against code injection in a stack-based buffer. In addition, the attacker needs to know or learn the locations of the desired gadgets.

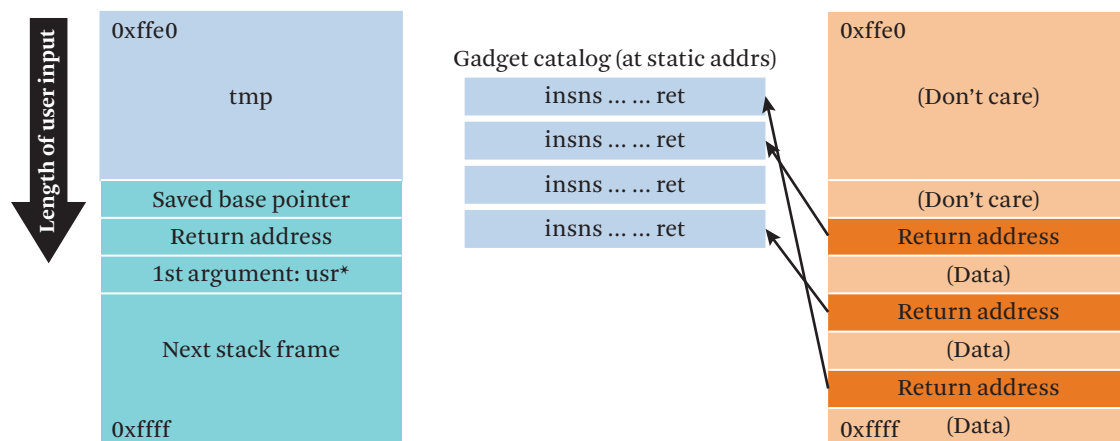


Figure 1.4 Stack-based ROP attack, redirecting control flow to a set of gadgets. This example assumes x86 calling conventions where all parameters are passed on the stack.

1.6.4 Data-Only Attacks

Data-only attacks exclusively modify data that is never loaded into the program counter of the CPU, excluding all code pointers. Data-only attacks may write to data that is used in comparisons or control-flow decisions, but the modified data never encodes a target address for the program counter. Data-only attacks can be realistic threats [Chen et al. 2005] and allow adversaries to compromise systems without hijacking the control flow, simply bending it along the valid control-flow graph of the program [Carlini et al. 2015e, Evans et al. 2015, Hu et al. 2015]. By modifying data, an adversary may still influence the control flow of the application. For example, redirecting control flow to an if-branch instead of an else-branch by modifying the tested condition through a memory corruption causes a trace through the code that is not intended by the programmer.

Data-only attacks can be grouped into two different classes of increasing complexity: non-control-data attacks and control-flow bending attacks. *Non-control-data attacks* modify a sensitive data area like a flag or string. For example, an attack could compromise the string that is passed to an `exec()` system call, enabling privileged mode for a JavaScript interpreter that allows unprivileged code from the web to execute with local privileges, or change the user id of a process in the kernel to 0 (root). These are realistic threats that exploit a program that enforces different privileges for individual components or restricts privileges inside the program itself. *Control-flow bending attacks* increase the sophistication of data-only attacks by “bending” the control flow along valid edges in the control-flow graph across multiple basic blocks. In the short C example below, assuming that no benign execution would allow `usr` to point to the third argument `a`, a control-flow bending attack could force, through the memory error in line 6, both of the if-branches to execute.

```

1 void vulnerable(int *usr, int usr2, int a)
2   if (a) {
3     :
4   }
5   // memory corruption
6   *usr = usr2;
7   if (!a) {
8     :
9   }
10 }
```

In control-flow bending attacks, the adversary uses the side effects of the executed basic blocks to modify the program state. The memory safety violation allows an initial setup of the program state that in turn allows steering the control flow along the path that executes the desired computations as side effects. For example, targets for modification could be interwoven data structures and pointers.

1.7 Conclusion

Memory safety violations continue to pose an important threat despite a long history of research. Existing solutions offer either complete protection but result in prohibitive performance overhead and compatibility issues, or partial protection with low overhead and high compatibility. Attacks can be stopped at several levels of abstraction, offering different trade-offs between defense compatibility, strength, and performance. Table 1.1 shows a summary of different defense mechanisms and their overhead, strength, and compatibility.

Current systems leverage only code integrity, ASLR, and stack canaries. The status quo protects against code injection attacks but lacks in protection against code-reuse or data-only attacks. These advanced attack vectors are still possible by circumventing probabilistic defenses like ASLR. In the near future, CFI and stack integrity will likely be deployed, building on code integrity and increasing the protection against code-reuse attacks. Going forward, we encourage the research community to develop more principled defenses against memory corruption that stop code-reuse attacks as well as data-only attacks.

Table 1.1 Summary of Defense Mechanisms and Corresponding Policies

Policy	Technique	Strength ^a	Performance Overhead ^b	Compatibility ^b	Weakness ^c
Spatial Memory Safety	SoftBound, CCured, Cyclone	Targeted	80%	High	Overhead
Temporal Memory Safety	CETS	Targeted	> 100%	High	Overhead
Type Safety	TypeSan, CaVer	Targeted	5–50%	High	Overhead
Data Integrity	Write Integrity Testing	High	10–25%	Composition	Over-approximation, unprotected reads, field protection
Code Integrity	Page flags	Partial	< 1%	High	Dynamically generated code
Return Integrity	Stack canaries	Low	< 5%	High	Information leaks, direct writes
Data Space Randomization	Data Space Randomization (DSR)	Partial	15–25%	Composition	Over-approximation, information leaks
Address Space Randomization	Address Space Layout Randomization	High	< 10%	High	Information leaks
Data-Flow Integrity	Data-Flow Integrity	Partial	100–200%	Composition	Over-approximation
Control-Flow Integrity	Control-Flow Integrity	Targeted	< 10%	Composition	Over-approximation

a. Strength lists the power of a mechanism: Targeted (completely stopping a type of memory corruption), High (mostly covering an issue), Partial (partially covering an issue), Low (somewhat hindering the adversary).

b. Performance overhead and compatibility list reported performance impact and potential compatibility issues: High (works for most software), Composition (does not work with shared libraries).

c. Weakness lists potential drawbacks of a policy.



Protecting Dynamic Code

Gang Tan, Ben Niu

One important aspect of making Control-Flow Integrity (CFI) practical is to support *modularity*. In the context of CFI, modularity refers to the ability to perform instrumentation of modules of an application separately, without considering other modules, and to link instrumented modules into an executable on demand. Modularity support is of crucial importance for accommodating dynamic linking, which loads libraries at runtime and is frequently used in modern software systems since it allows different software vendors to work on or update modules independently. Just-In-Time (JIT) compilation also requires modularity support because a piece of newly generated code by a JIT compiler can be considered a module, which is to be “linked” with the JIT compiler.

In both dynamic linking and JIT compilation, not all code is available statically. The code of a dynamically linked library is available only after the library has been loaded at runtime, which may happen during the middle of a program execution; in JIT compilation, the binary code of a function is available only after the function has been compiled on the fly. We call this kind of code *dynamic code* since it is available only after a runtime event.

Unfortunately, many CFI systems require all modules of an application, including libraries, to be available at the static instrumentation time. They perform a global analysis on all code to construct a global Control-Flow Graph (CFG). Instrumentation schemes in many CFI systems also assume global code properties. For instance, instrumentation in the classic CFI [Abadi et al. 2005a] inserts identifiers (representing a class of indirect branches and targets) before branch targets and inserts checks before indirect branches to make sure the right identifiers according to the CFG are at the targets. The identifiers are embedded as instructions in code and cannot appear in the rest of the code. However, this property cannot be

guaranteed without inspecting the whole program. Many other CFI instrumentation techniques also do not support modularity; the reasons for this are discussed in Section 2.5.

In this chapter, we present a modular CFI system that extends CFI to support dynamic code. We start with an overview of the main challenges and solutions when dealing with dynamic code in Section 2.1. In Section 2.2, we present a compiler-assisted, type-based scheme that allows efficient CFG construction in the presence of dynamic code. In Section 2.3, we present a modular CFI system that supports dynamic libraries by adopting the type-based CFG construction process and a technique for supporting multi-threading. In Section 2.4, we show that, with some adjustments, the modular CFI system can support JIT compilation. We discuss related work in Section 2.5 and conclude in Section 2.6.

The main results of this chapter were published in previous conference papers [Niu and Tan 2014a, Niu and Tan 2014b] and in a Ph.D. dissertation [Niu 2015]. In this chapter, we streamline the presentation by integrating the previous papers and by highlighting the most important ideas and messages that are of interest to future CFI research. Certain details are omitted and interested readers are referred to the previous papers. Furthermore, experimental results included in this chapter may be different from those in the conference publications because we have made improvements since their publication.

2.1 Overview of Challenges and Solutions

When designing a system that enforces CFI on programs that can load or generate new code dynamically, there are a number of challenges the system needs to address. We next highlight the main challenges and briefly discuss the solutions in our modular CFI system.

2.1.1 Challenges of Supporting Modularity

In CFI, the enforced Control-Flow Graph (CFG) is the security policy, telling what targets that an indirect branch can transfer its control to. We use an indirect branch to refer to either an indirect call (a call via a register or memory operand), an indirect jump (a jump via a register or memory operand), or a return instruction.

A traditional CFI system builds a static CFG for a program ahead of time. However, it is extremely difficult to build a static CFG when a program incorporates dynamic code, simply because it is hard to predict what dynamic libraries will be loaded or what binary code will be dynamically generated during JIT compilation. Since a modular approach (which does not rely on global analysis or instrumen-

tation) is preferred, CFG construction should be deferred to the runtime, in an incremental way. That is, whenever a new piece of dynamic code is incorporated into the main program, we should construct a new CFG for the program with the new code. It is worth mentioning that the control-flow edges for the code in the main program may also change after new code is added. For example, suppose a function named f in the program P contains a return instruction. The program's internal CFG allows the return instruction to return to any caller of f in P . After P is linked with a library M , the return instruction can also return to any caller of f in M .

Therefore, the CFG has to be updated after a piece of dynamic code is incorporated. The dynamic nature of the CFG when dealing with dynamic code poses multiple challenges:

- How to efficiently generate a new CFG with high precision when dynamic code is incorporated. There are clearly a wide range of design choices for generating the new CFG. Since an online algorithm is necessary, a good choice has to balance efficiency and CFG precision.
- How to update the CFG safely and efficiently at runtime in the presence of multithreading. When there are multiple threads, one thread may use the current CFG to decide whether an indirect branch control transfer is legal, while another thread may load a library and trigger an update of the CFG. Therefore, we have to avoid the well-known read/write race problem when dynamically updating the CFG.
- How to ensure that dynamic code is instrumented to respect the current and future CFGs. Since we cannot trust dynamic code, a mechanism must be there to check for its trustworthiness so that its indirect branch control transfers respect the current CFG. Furthermore, since the CFG evolves during program execution because of future dynamic code, the instrumentation must be designed to accommodate future CFGs.

2.1.2 MCFI Solutions

We have designed a system called Modular CFI (MCFI [[Niu and Tan 2014a](#)]), which accommodates dynamic code including dynamically linked libraries and Just-In-Time (JIT) compiled code. Before delving into details, we outline how MCFI solves the aforementioned challenges.

For runtime CFG generation, MCFI adopts a compiler-assisted, type-based scheme. When compiling a module, MCFI takes the module's source code and propagates type information from source code to binary code. For library code whose source code is available, the compilation is performed by a modified LLVM

compiler. In the case of JIT compilation, a modified JIT compiler generates type information for the JITted code. As a result, an MCFI executable module contains both binary code and type information, which will be used for CFG generation. When MCFI modules are linked either statically or dynamically, their type information is combined and used to generate a CFG. The CFG generation process is type based; for instance, an indirect call through a function pointer is allowed to target all functions whose types are compatible with the type of the function pointer. Our experience suggests that the type-based process is efficient and generates higher-precision CFGs when compared to other CFI systems. Drawbacks are that it requires source code and sometimes requires modest effort to adapt type-unsafe source code.

For thread-safe CFG operations, MCFI wraps look up and update operations into transactions, inspired by Software Transactional Memory (STM [Shavit and Touitou 1995]). A CFG lookup transaction performs speculative reads (assuming no CFG update transactions are running) and is extremely efficient. We have adapted generic STM algorithms and designed a lightweight STM implementation that performs transactions efficiently in MCFI's context.

For ensuring the trustworthiness of dynamic code, MCFI has a verifier component, which verifies that instrumentation is inserted into the dynamic code at the right places. MCFI's instrumentation scheme is designed so that when CFG changes there is no need to patch the instrumentation. For that, MCFI represents the CFG in separate tables outside the code region. When CFG changes, only those tables need to be updated and the instrumentation in the code stays the same.

Overall, these ideas make MCFI efficient (with less than 5% overhead on average), support modularity, and generate high-precision CFGs. A CFI survey [Burow et al. 2016] has independently verified these claims. Specifically, they found MCFI generates the highest-precision CFGs among all current CFI systems, and its efficiency is on par with other state-of-the-art systems (even though most other systems lack the support of JIT compilation and dynamic libraries).

2.2 Type-Based CFG Generation

Generating CFGs for C/C++ applications is a seemingly easy task. However, it is quite difficult to statically approximate control flows of many C/C++ features, including calls through function pointers and virtual method calls, because their control flows depend on runtime data. The matter is worse when dealing with dynamic code because not all code is available at once—only one module is available at a time.

For a modular CFG generation approach, an MCFI module contains auxiliary information for CFG generation, in addition to code and data. A module can be the main application, a library, or a piece of code generated on the fly. When MCFI modules are statically or dynamically linked, not only are their code and data linked but their auxiliary information is also merged into the combined module. This design allows modules to be independently instrumented and linked statically or dynamically. The combined module can enforce a CFG that is built using a combination of the individual modules' CFGs.

There is a range of choices for what kind of auxiliary information MCFI can attach to a module for CFG generation. The richer the auxiliary information is, the better it can enable the generation of a precise CFG. On the other hand, richer auxiliary information implies more analysis time is needed for generating and merging the information and for producing a CFG from the information. Since MCFI cannot afford long analysis time, there is a trade-off between CFG precision and CFG generation efficiency.

MCFI attaches type information to modules and uses type matching for efficient online CFG generation. Specifically, an MCFI module comes with the types of its functions and its function pointers. The benefit of this design is that it can efficiently generate a relatively precise CFG. Moreover, the type information for an individual module can be generated by an augmented compilation toolchain. Finally, combining type information of multiple modules during linking is a simple union operation.

We next describe the type-based CFG generation scheme. We start by discussing how types are used to compute control-flow edges out of indirect branches compiled from features in C. We then discuss CFG generation for C++ features, including virtual method calls and exceptions. We finish by presenting under what conditions our CFG generation process is sound, along with experimental results. In the discussion, we present how control-flow edges are computed only for indirect branches because control flows for direct branches and non-branching instructions can be precisely computed and checked before code execution (and therefore do not need instrumentation).

2.2.1 CFG Generation for C

The C language has many programming constructs that may be compiled to code containing indirect branches. We next enumerate those and discuss how MCFI handles them.

C Function Pointers. A function pointer in C points to some global function. A function call through a function pointer at the C source-code level is typically

implemented through an indirect call at the machine-code level, but the compiler may implement it through an indirect jump if it is a tail call. For an indirect branch (either an indirect call or an indirect jump) through a function pointer that points to a function of type τ , MCFI allows it to call any function as long as (1) the function's address is taken in the code, and (2) the function's type is some τ' that is equivalent to type τ .

Taking a function's address means that the function's address is assigned to a function pointer somewhere in the code. Note that function addresses can also be taken at runtime by the `libc` function `dlsym`. Therefore, we changed `dlsym`'s implementation so that before `dlsym` returns a valid function address, an MCFI runtime trampoline is called to mark the function's address as taken and update the CFG so that function pointers with the equivalent type can legitimately call the function. In a follow-up work [Niu and Tan 2015], we show that by carefully handling address-taken events, we can further refine the CFG.

Returns. To compute control-flow edges out of return instructions, we construct a call graph, which tells how functions get called by direct or indirect calls. Using the call graph, control-flow edges out of return instructions can be computed: if there exists an edge from a call node to a function, return instructions in the function can return to the return address following the call node.

In addition, modern compilers perform tail-call elimination to save stack space. If a return instruction immediately follows a call instruction during code emission, the return is eliminated and the call is replaced with a jump. We handle this case in the following way: if in function f , there is a call node calling g , and g calls h through a series of tail jumps, then an edge from the call node in f to h is added to the call graph. As a result, the return instructions in h can return to the return address following the call node in f .

Signal Handlers. In Linux, signal handlers are usually not called by application code¹, so their return instructions do not return to the application code. Instead, signal handlers return to a user-space code stub set up by the OS kernel and the code stub invokes the `sigreturn` system call. MCFI provides new function attributes for developers to annotate signal handlers in the source code, and the compiler inlines the code stub into every signal handler during code generation. Each signal handler is associated with a special type signature to ensure it never becomes an indirect call target. This design helps mitigate SigReturn-Oriented Programming

1. If a signal handler is invoked by application code, we can change the code to duplicate the handler so that the copy is never invoked by application code.

```

memcpy:
    .:  instructions omitted
    .:
__mcfi_return_memcpy:
    mcfi-ret # MCFI-instrumented return

    .section MCFIMetadata,"",progbits
    .ascii  "memcpy : void* (*)(void*, void*, size_t)"

```

Figure 2.1 Metadata annotation for the assembly code of `memcpy`.

(SROP) attacks [Bosman and Bos 2014]. SROP attacks modify the signal-handling stack (which stores the context information, including the program counter when a signal is raised) in user space so that the execution of `sigreturn` may jump to attacker-controlled locations. By inlining the code stub that contains `sigreturn` in signal handlers, MCFI makes `sigreturn` system calls unreachable from other application code. As a result, attackers need to trigger real signals to execute a `sigreturn` system call; without the inlining, attackers could just transfer the control to the code stub for executing a `sigreturn`.

Assembly Code. Some C programs may contain assembly code for performance and for architecture-specific code. MCFI requires developers to manually annotate the instructions in assembly code with type information and branch targets. For example, some `libc` functions, such as `memcpy`, are implemented using manually written assembly code for performance. Figure 2.1 shows part of `memcpy`'s assembly-code implementation. The return instruction is changed to MCFI's instrumented return instruction called `mcfi-ret` (the instrumentation will be explained later). To generate the CFG, type information needs to be added to the assembly instructions of `memcpy`. Moreover, its instrumented return instruction should be annotated so that the MCFI runtime knows which indirect branch performs the function return operation for `memcpy`. To achieve these results, we insert an MCFI-specific label `__mcfi_return_memcpy` for identifying `memcpy`'s instrumented return and add a string (enclosed in double quotes) in a newly created section called `MCFIMetadata` to identify the type information.

Other Control-Flow Features in C. A `longjmp` in C returns (implemented through an indirect-jump instruction) to the address set up by a `setjmp` call. MCFI connects the `longjmp`'s indirect jump to the return addresses of all `setjmp` calls. Other functions, such as `setcontext` and `getcontext`, are handled in a similar way.

Switch and indirect `goto` statements are typically compiled to direct jumps or jump-table-based indirect jumps; their targets are statically computed and embedded in read-only code or jump tables, so they do not need instrumentation.

2.2.2 CFG Generation for C++

In addition to the cases we discussed for C, the C++ language has many more programming features that may be compiled to code containing indirect branches. We next enumerate those additional cases in C++.

Virtual Method Calls. C++ supports multiple inheritance and virtual methods. A virtual method call through an object is usually compiled to an indirect call (or an indirect jump with tail call optimization). A virtual call on an object is resolved during runtime through dynamic dispatch. Which method it invokes depends on the actual class of the object. Similar to `SafeDispatch` [Jang et al. 2014], MCFI performs Class Hierarchy Analysis (CHA) [Dean et al. 1995] on C++ code. This analysis tracks the class hierarchy of a C++ program and determines, for each class *C* and each virtual method of *C*, the set of methods that can be invoked when calling the virtual method through an object of class *C*; these methods might be defined in *C*'s subclasses. MCFI allows a virtual method call to target all methods determined by CHA. Also, note that to support C++ multiple inheritance, the compiler may generate thunks [Itanium C++ ABI 2017], which are simple trampolines that first adjust the `this` pointer and then jump to the corresponding virtual method. The thunks may be used to fill virtual tables instead of their corresponding virtual methods and therefore can be called by virtual method invocation as well. We associate each thunk with the same meta-information as its corresponding virtual method and add it to the class hierarchy as well for CFG generation.

Next we use a toy C++ example in Figure 2.2 to demonstrate the basic idea. We define a class *A* and its subclass *B* as well as their virtual methods. In function `fx`, virtual method `foo` is invoked with respect to a class *A* object pointer; in function `fy`, `foo` is invoked with a class *B* object pointer. According to the class hierarchy, `a->foo()` at line 16 possibly targets `A::foo` and `B::foo`, while `b->foo()` at line 19 can target `B::foo`. Note that `a->foo()` should not reach `A::foo const` at line 3 or `B::foo const` at line 10, because their type qualifiers do not match: `a->foo()` calls a non-constant virtual method, but `A::foo const` and `B::foo const` are constant methods.

It should be pointed out that CHA is a whole-program analysis. To support modularity, MCFI emits a class hierarchy for each module and combines modules' class hierarchies at link time.

```

1 class A {
2 public:
3     virtual void foo() const {}
4     virtual void foo() {}
5     virtual void bar() const {}
6     virtual void bar() {}
7 };
8 class B : public A {
9 public:
10     virtual void foo() const {}
11     virtual void foo() {}
12     virtual void bar() {}
13 };
14 :
15 void fx(A *a) {
16     a->foo();
17 }
18 void fy(B *b) {
19     b->foo();
20 }

```

Figure 2.2 A C++ example of virtual method calls.

C++ Function Pointers. C++ supports two kinds of function pointers: (1) those that point to global functions (similar to C) or static member methods of classes and (2) those that point to non-static member methods. Function pointers of these two kinds have different static types. Their target sets are disjoint and they are handled differently by compilers. Figure 2.3 shows a code example of the two kinds of function pointers.

Function pointer `fp` is of the first kind. It is assigned to the address of a global function `getpagesize` at line 2. At line 3, the function pointer is invoked via an indirect call (or indirect jump if it is a tail call). To identify its targets, MCFI adopts a type-matching method (similar to how calls through functions pointers in C are handled): an indirect branch via a function pointer that points a function of type τ can target any global function or static member method whose static type is equivalent to τ and whose address is taken in the code.

Function pointer `memfp` at line 9 is of the second kind, which is also called a method pointer. The code reuses the class definitions in Figure 2.2. According to the C++ semantics, we allow an indirect branch through such a method pointer

```

1 typedef int (*Fp)();
2 Fp fp = &getpagesize;
3 std::cout << (*fp)();
4 :
5 typedef void (A::*memFp)() const;
6 :
7 void fz(const A *a) {
8     memFp memfp = &A::foo;
9     (a->*memfp)();
10 }

```

Figure 2.3 An example of C++ function pointers.

that points to methods of type τ to target any virtual or non-virtual member method defined in the same class whose type is equivalent to τ , whose address is taken, and whose type qualifiers such as `const` match the method pointer's. (LLVM IR does not support function type qualifiers, so we changed Clang and LLVM to propagate C++ member method type qualifiers to the LLVM IR as metadata.) Moreover, for each matched virtual member method, we search the class hierarchy to find in derived classes all virtual methods whose types and qualifiers match and add those functions to the target set, because, for example, if a B object pointer is passed at line 7, `B::foo const` will be called. Consequently, the method pointer dereference `(a->*memfp)()` at line 9 can possibly reach `A::foo const`, `A::bar const`, or `B::foo const` in Figure 2.2 at line 3, 5, or 10, respectively.

Exception Handling. C++ exceptions are handled by compilers and libraries that implement the Itanium C++ ABI. In this ABI, C++ exception handling is the joint work of the compiler, a C++-specific exception handling library such as `libc++abi`, and a C++-agnostic stack-unwinding library such as `libunwind`. We next give a summary about how MCFI handles indirect branches involved in exception handling; interested readers can refer to a previous publication [Niu and Tan 2014b] for details.

By analyzing carefully how the Itanium C++ ABI handles exceptions, all indirect branch instructions involved in exception handling can be handled using the strategies we have discussed (CHA and the type-matching method), except for one indirect branch. During stack unwinding, if the current stack frame has a catch clause that matches the type of the thrown exception, control is transferred to the

catch clause via an indirect branch; we call this indirect branch `CatchBranch`. If a type-matching catch is not found, the stack unwinding should be continued. However, if there is a cleanup routine that is used to destroy objects constructed in `try` statements, the cleanup routine needs to run before the unwinding continues. It turns out that the same indirect branch (`CatchBranch`) is used to transfer the control to the cleanup routine, but with a different target address. For simplicity, our implementation connects `CatchBranch` to all catch clauses and cleanup routines. To support separate compilation, MCFI modifies the LLVM compiler to emit a table recording addresses of all catch clauses and cleanup routines in each module, and these tables are combined during linking.

If an exception object is caught but not rethrown, `libc++abi` invokes the object's destructor, which is registered when calling `__cxa_throw`. The invocation is through an indirect call. Possible targets of this call in a module can be statically computed by tracking `__cxa_throw` invocations. As a result, MCFI's compiler emits these target addresses for each module and the runtime combines them at link time.

Global Constructors and Destructors. The constructors of global and local static objects are invoked before the `main` function of a C++ program, and their destructors are called after the `main` function returns. LLVM generates stub code for each such object. The stub code directly invokes the constructor and registers the destructor using either `__cxa_atexit` or `atexit` defined in `libc`. The addresses of the stub code are arranged in the binary and iterated by an indirect call (called `CtorCall`) in `libc` before `main`. After `main`, another `libc` indirect call (called `DtorCall`) iterates the registered destructors to destroy objects. Both `CtorCall` and `DtorCall`'s targets are statically computable by analyzing the compiler-generated stub code.

Lambda Functions. C++11 provides lambda functions, whose related control-flow edges (returns) are also supported by our CFG generation. Compilers automatically convert lambda functions to functors, which are classes with `operator()` methods that are called by direct branches. The return edges of the `operator()` methods can be handled in the same way as those of other functions.

2.2.3 CFG Soundness

Our method for CFG generation is type based. An indirect call through a function pointer to a global function is allowed to call any global function whose type matches the function pointer's type. The class hierarchy analysis, which is used to resolve virtual method calls, is also based on static types. As a result, if a C or

C++ program misuses types using features such as arbitrary type casts, then our CFG generation method may construct a CFG whose edges do not cover all dynamic control flow of the program; enforcement of such a CFG would break the program’s execution. We call such a CFG unsound.

We believe that MCFI can generate a sound CFG for a *memory-safe* C/C++ program (for memory safety, see [Nagarakatte et al. 2009]) if the program satisfies a *compatibility condition*: it has no bad type casts from or to types that contain function pointer types. Next we define this condition using the following simplified C types:

$$\tau := \text{int} \mid \text{void} \mid \tau_* \mid (\tau_1 \rightarrow \tau_2) \text{ fptr} \mid \text{struct} \{ \tau_1 f_1; \dots; \tau_n f_n \}$$

We use τ_* for regular C pointer types and “ $(\tau_1 \rightarrow \tau_2)$ fptr” for types of pointers to functions that take τ_1 values and return τ_2 values. The C standard routinely distinguishes pointers to functions from pointers to data; so we have different syntaxes for them to avoid confusion. In C, “ $(\tau_1 \rightarrow \tau_2)$ fptr” is written as “ $(\tau_2)(*)(\tau_1)$ ”. Type “struct $\{ \tau_1 f_1; \dots; \tau_n f_n \}$ ” is for a struct type with n fields and the i th field has name f_i and is of type τ_i .

We define a function $\text{has_fptr}(\tau)$, which returns true if and only if τ contains a function pointer type:

$$\text{has_fptr}(\tau) = \begin{cases} \text{true} & \text{if } \tau = (\tau_1 \rightarrow \tau_2) \text{ fptr} \\ \text{false} & \text{if } \tau = \text{int or void} \\ \text{has_fptr}(\tau_1) & \text{if } \tau = \tau_1* \\ \text{has_fptr}(\tau_1) \vee \dots \vee \text{has_fptr}(\tau_n) & \text{if } \tau = \text{struct} \{ \tau_1 f_1; \dots; \tau_n f_n \} \end{cases}$$

Then the compatibility condition is stated as follows: a type cast from a τ_1 value to a τ_2 value is compatible if $\neg \text{has_fptr}(\tau_1)$ and $\neg \text{has_fptr}(\tau_2)$. The condition essentially enforces that values of function pointer types cannot be forged.

For C++, $\text{has_fptr}(\tau)$ should be extended to cover the case when τ is a class: it returns true if the class contains a virtual method because the implementation of classes uses a virtual table pointer that points to a table with virtual method pointers.²

Not every C/C++ program satisfies the above compatibility condition, so they first need to be retrofitted to meet the condition. We built a Clang-based static checker

2. MCFI allows each virtual method call to reach any virtual method implementation according to the class hierarchy; so the condition can be relaxed to allow type casts between two classes (or class pointers) as long as there exists an inheritance path in the class hierarchy between the two classes.

that can report incompatible type casts to facilitate the retrofitting process. The checker is executed after Clang generates the Abstract Syntax Tree (AST), which explicitly represents type casts.

We investigated how much effort it takes to make SPEC CPU2006 C/C++ programs and Google V8 comply with the compatibility condition. A detailed experience report can be found in a Ph.D. dissertation [Niu 2015]. In summary, we found programs require little or no changes to make them compatible with MCFI's CFG generation mechanism. For Google V8, which has over 555,000 lines of code, we modified only 10 lines using a wrapper approach. For the C programs in SPEC CPU2006, we made 6 lines of code changes for `perlbench`, about 30 lines for `gcc`, and 1 line for `libquantum`. We also tried our approach on the seven C++ programs in SPEC CPU2006 as well as `libc++`, `libc++abi`, and `libunwind` for a total over 620,000 lines of code, only 35 lines of code (all in `povray`) needed to be changed to generate sound CFGs. In addition, all the generated CFGs were tested on data sets that come with those benchmarks.

2.2.4 CFG Precision

We measured the precision of the CFGs generated by MCFI. Table 2.1 shows the numbers. The “IBs” column lists the total number of instrumented indirect branches, with the number of those indirect branches that have targets shown in parentheses. Some indirect branches may not have targets; for example, the `libc` function `aio_cancel` is never called by any of the benchmarks, so its return instruction has nowhere to target. The “IBTs” column presents the number of all indirect branch targets; the “EQCs” column presents the number of equivalence classes of the indirect branch targets. Moreover, the “Avg IBTs / IB” column lists how many targets an indirect branch has on average, and the “Avg IBs / IBT” column shows the number of indirect branches that could reach the same target on average. As can be seen, MCFI supports fine-grained CFGs. The average targets per indirect branch and average indirect branches per target are much less than coarse-grained CFI, which could be as many as the number of indirect branch targets and indirect branches, respectively. Further, indirect branches can reach more targets in some programs (e.g., `403.gcc`) than others, and it is because those programs have function pointers that could point to many functions or have functions that are called in many places. For example, `403.gcc` defines many functions for emitting instructions, and those functions are all indirectly callable and share the same type signature.

Table 2.1 CFG Statistics for SPEC CPU2006 C/C++ Benchmarks

SPEC CPU2006	IBs (with matching targets)	IBTs	EQCs	Average	
				IBTs / IB	IBs / IBT
400.perlbench	3,327 (2,399)	18,379	1,039	722	95
401.bzip2	1,711 (943)	4,065	505	33	8
403.gcc	6,108 (5,039)	50,413	2,321	1,244	125
429.mcf	1,625 (875)	3,852	493	34	8
433.milc	1,825 (1,030)	5,880	625	36	7
444.namd	4,796 (3,042)	17,620	1,314	154	27
445.gobmk	3,908 (3,119)	14,557	944	949	204
447.dealII	13,624 (8,361)	61,464	3,225	1,035	141
450.soplex	6,305 (4,407)	22,418	1,847	175	35
453.povray	6,275 (4,355)	28,738	2,027	374	57
456.hmmmer	2,038 (1,136)	7,907	682	93	14
458.sjeng	1,777 (1,010)	4,827	560	32	7
462.libquantum	1,688 (917)	4,170	514	35	8
464.h264ref	2,455 (1,616)	7,047	793	41	10
470.lbm	1,612 (867)	3,840	485	35	8
471.omnetpp	7,791 (5,526)	35,772	2,203	456	71
473.astar	4,770 (2,994)	16,763	1,325	159	29
482.sphinx3	1,893 (1,071)	6,432	652	39	7
483.xalancbmk	31,167 (27,117)	97,265	7,970	1,103	308

Compared to coarse-grained CFI techniques with several equivalence classes supported, MCFI's CFGs can generate two to three orders of magnitude more equivalence classes. For instance, CCFIR [Zhang et al. 2013] and binCFI [Zhang and Sekar 2013] allow an indirect call to target any function whose address is taken; therefore, all such functions are included in one equivalence class. CCFIR and binCFI also allow any return instruction to target any instruction following a call, combining all return sites in one equivalence class. The classic CFI's instrumentation [Abadi et al. 2005a] can support a fine-grained CFG, but for implementation convenience its CFG generation also allows all indirect calls to target any function whose address

is taken. NaCl [Yee et al. 2009] and MIP [Niu and Tan 2013] enforce chunk-based CFI in which an indirect branch can target any chunk beginning; it enforces even less-precise CFGs.

A Ph.D. dissertation [Niu 2015] presents other statistics about the generated CFGs, including edge distribution among indirect branches and how precision can be improved by disabling tail-call elimination.

2.3 Handling Dynamically Linked Libraries

The type-matching method generates a fine-grained CFG for an application in the presence of dynamic code. The CFG is then enforced on the application. Next we detail the major components in MCFI for the CFG enforcement, using dynamically linked libraries as the motivating application; later on we show that the same components can also accommodate JIT compilation. We start, however, with a discussion about the threat model. MCFI adopts CFI's concurrent attacker model [Abadi et al. 2005a]. The model allows a strong adversary, which is treated as a separate thread running in parallel with user threads. The attacker thread can read and write any memory (subject to memory page protection). Consequently, the attacker can corrupt writable memory between any two instructions in the user program. However, it is assumed that machine registers of a thread cannot be directly modified by the attacker thread. The attacker can still affect registers indirectly by corrupting memory. As an example, if the program reads from a region of writable memory to a register, then the register's value is under the attacker's control because the attacker can write any value to that region of memory.

In addition, to prevent arbitrary code execution, a trusted MCFI runtime enforces the invariant that no memory regions are both writable and executable at the same time. The invariant is enforced when an application is initially loaded by the runtime. The runtime sets up a separate code and data region. Code is loaded into the code region, which is executable and readable but not writable. Note that the code region can include some read-only data, such as jump tables. The data region is readable and certain parts are writable but not executable. The invariant also holds when dynamically linking libraries. New libraries are loaded into unoccupied parts in the code and data regions.

2.3.1 CFG Encoding: ID Tables

Following the classic CFI [Abadi et al. 2005a], MCFI encodes a CFG by first partitioning indirect branch targets into equivalence classes and associates each with an Equivalence Class Number (ECN). To remove the global ID uniqueness requirement

in the classic CFI, ECNs are pulled out of the code section and stored in a runtime data structure consisting of two separate tables. These tables are conceptually maps from addresses to IDs, each of which contains an ECN and other components.

Encoding a CFG as separate ID tables has several benefits. First, IDs in the tables can overlap with the numbers in the code section, eliminating the global ID uniqueness assumption in the classic CFI. Second, the instrumentation code before indirect branches is parameterized over the ID tables and remains the same after loading libraries. Therefore, code pages for applications and libraries can be shared among processes, saving memory and application launch time. Third, centralized ID tables enable favorable memory cache effect and fast table updates using parallel memory-copy mechanisms of the CPU.

MCFI maintains two ID tables that encode the CFG. The branch ID table, called the *Bary table*, maps from an indirect branch location to the location's branch ID, which contains the ECN of the equivalence class of addresses the branch is allowed to jump to in the CFG. The target ID table, called the *Tary table*, maps from an address to an ID showing the equivalence class to which the address belongs. With the ID tables, instrumenting an indirect branch is straightforward. Take the example of a return instruction located at address l . The instrumentation can first use the Bary table to look up the branch ID for address l , use the Tary table to look up the target ID for the actual return address, and check whether the branch ID is the same as the target ID.

We briefly discuss how an ID is encoded and how the two tables are represented in memory. It is machine dependent and we discuss only the case for x86; more details can be found in a previous publication [Niu and Tan 2014a]. An MCFI ID is 8 bytes long and is stored at an 8-byte aligned memory address so that a single x64 memory-access instruction can atomically access it. An ID contains a 28-bit ECN in the higher four bytes, a 28-bit version number in the lower four bytes, and eight reserved bits (which are those least significant bits of the eight bytes). It allows 2^{28} different equivalence classes in programs. The version number in an ID supports table-access transactions and is used to detect whether a check transaction should be aborted and retried. The ID-encoding scheme allows 2^{28} different version numbers. MCFI inserts extra nop instructions into the program to force indirect branch targets to be eight-byte aligned so that addresses that are not eight-byte aligned cannot be possible indirect branch targets; consequently, the tables have entries only for eight-byte aligned code addresses and the size of a table is the same as the code size. Furthermore, the eight reserved bits in an ID have fixed bit values “0, 0, 0, 0, 0, 0, 0, 1” and are there to prevent programs from using indirect branch targets that are not eight-byte aligned. In particular, if an

indirect branch uses an address that is not eight-byte aligned, the eight-byte target ID loaded from the Tary table will be invalid (i.e., it will not have the special bit values in the least significant bits). Instrumentation before an indirect branch will then be able to detect this and abort the indirect branch.

The Tary table is represented as an array of IDs indexed by code addresses. If a code address is not a possible indirect branch target, the corresponding array entry contains all zeros; otherwise, it contains the ID of the code address. This design enables efficient lookups and updates. Recall that the Bary table conceptually maps indirect branch locations to branch IDs. One observation is that instruction addresses are known once they are loaded in memory. Therefore, when a module is loaded into the code region, MCFI's loader patches the code to embed constant Bary table indexes that correspond to correct branch IDs in branch-ID read instructions. In this design, the Bary table does not need entries for code addresses that do not hold indirect branches (in contrast, the Tary table has all-zero entries even for addresses that are illegal indirect branch targets). Furthermore, all branch IDs loaded from the Bary table are valid IDs as long as the loader embeds the correct table indexes in branch-ID read instructions.

2.3.2 Memory Layout and Table Protection

The Bary and Tary tables need to be protected at runtime so that application code cannot directly change them. Figure 2.4 shows the memory layout of an application protected with MCFI. The application should have been compiled and instrumented by MCFI's compilation toolchain. The application and all its instrumented libraries are loaded into a sandbox created by the MCFI runtime. The sandbox can be realized using Software-based Fault Isolation (SFI [Wahbe et al. 1993]) or hardware support (e.g., segmentation on x86-32). We use the scheme described in ISboxing [Deng et al. 2015] to create the SFI sandbox. In detail, the sandbox for running applications is within $[0, 4 \text{ GB})^3$, and the MCFI compiler instruments each indirect memory write instruction by adding a `0x67` prefix, which is the 32-bit address-override prefix. The prefix forces the CPU to clear all upper 32 bits after computing the target address. As a result, code in the sandbox cannot arbitrarily execute or write memory pages outside the sandbox, but has to invoke trampolines provided by the MCFI runtime; these trampolines allow the untrusted code to

3. The maximum sandbox size can be extended to 64 TB on x64 if the sandboxing technique in PittSField [McCamant and Morrisett 2006] is used or if the MCFI runtime is implemented as a kernel module.

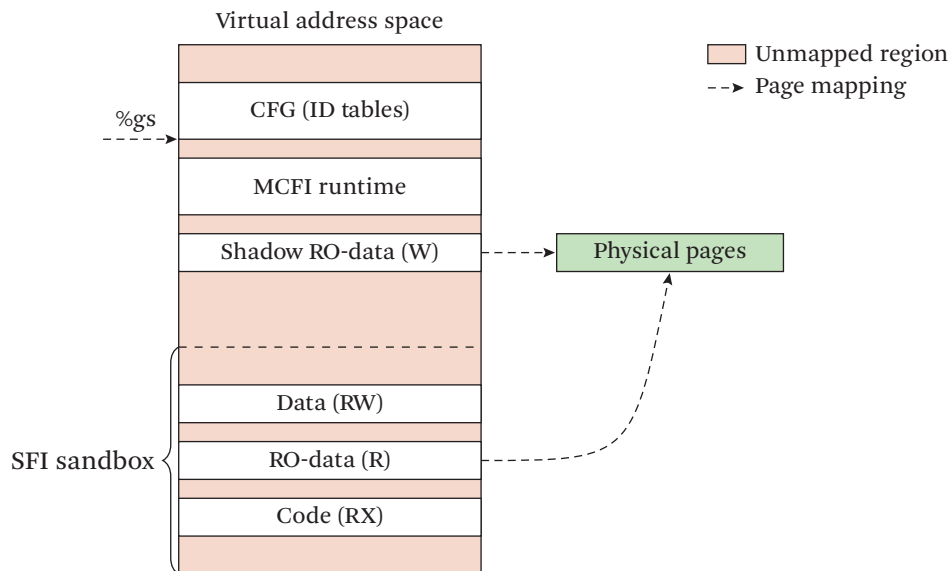


Figure 2.4 Memory layout of MCFI. “R,” “W,” and “X” appearing in parentheses denote the Readable, Writable, and eXecutable memory page permissions, respectively. The “RO-” prefix means Read-Only.

escape the sandbox safely. The runtime also maintains the invariant that no memory pages in the sandbox are writable and executable simultaneously. In addition, the runtime guarantees that read-only data, such as jump tables, are not writable. Consequently, those system calls that might subvert the invariant are replaced with runtime trampolines. For instance, the `mmap`, `munmap`, and `mprotect` system calls in `libc` are all rewritten to invoke the relevant runtime trampolines that are checked. The MCFI runtime and the encoded CFG, i.e., Bary and Tary tables, are stored outside the sandbox. The ID tables are read-only from the application’s perspective, but writable by the runtime.

MCFI uses the `%gs` segment register to index both the Bary and Tary tables. Inside the sandbox, MCFI always loads the code in `[4MB, 4GB)` to comply with the x64 Linux ABI, and the region `[0, 4MB)` is always unmapped. MCFI allocates `[%gs+68KB, %gs+4MB)` for the Bary table, and `[%gs+4MB, %gs+4GB)` for the Tary table. MCFI unmaps `[%gs, %gs+64KB)` for trapping calls to the NULL pointer and uses the page `[%gs+64KB, %gs+68KB)` for storing trampolines, which are pointers to MCFI runtime services. Applications can be modified to jump to the trampolines

to safely escape the sandbox. For example, `jmpq %gs:65536` would transfer the control to the first trampoline MCFI installs.

Figure 2.4 also shows parallel mapping of some read-only data, which include the GOT.PLT data in Linux. The PLT (Procedure Linkage Table) has a list of entries that contain glue code emitted by the compiler to support dynamic linking. Code in the PLT entries uses target addresses stored in the GOT.PLT table (GOT is short for Global Offset Table). The GOT.PLT table is adjusted during runtime by the linker to dynamically link modules. However, security weakness results from the GOT.PLT table's writability, as demonstrated by a recent attack [Davi et al. 2014].

To address this security concern, MCFI sets the GOT.PLT table to be read-only inside the sandbox and creates outside the sandbox a shadow GOT.PLT table, which is mapped to the same physical pages as the in-sandbox GOT.PLT table. All changes to the GOT.PLT table are therefore performed by the MCFI runtime, which ensures that each entry's value is the address of either the dynamic linker or a function whose name is the same as the corresponding PLT entry's name. The parallel-mapping scheme takes advantage of the virtual memory mechanism available to user-space programs and can be achieved by shared memory mechanisms provided by OSes (e.g., `shm_open`, `ftruncate`, and `mmap libc` calls in Linux). As we will later see, the same parallel-mapping idea is used to support JIT compiled code.

2.3.3 CFG Enforcement: Transactions and Code Instrumentation

Since concurrent ID table reads and writes are possible, a synchronization mechanism must be designed for maintaining the consistency of the tables. Otherwise, the tables may reach some intermediate state that allows for illegal control-flow transfers. A simple lock-based scheme for accessing tables could be adopted, but it would incur a large performance penalty due to MCFI's table-read-dominant workloads: dynamic linking is a rare event compared to the use of indirect branches; even in a JIT environment, such as Google's V8 JavaScript engine, the number of indirect branch executions is roughly 10^8 times the CFG updates triggered by dynamic code installation.

Our solution is to wrap table operations into transactions and use a custom form of Software Transactional Memory (STM) to achieve safety and efficiency. We use two kinds of transactions:

Check transaction (TxCheck). This transaction is executed before an indirect branch. Given the address where the indirect branch is located and the address that the indirect branch targets, the transaction reads the branch ID

and the target ID from the tables, compares the two IDs, and takes actions if the IDs do not match. This transaction performs only table reads.

Update transaction (TxUpdate). This transaction is executed during dynamic linking. Given the new IDs generated from the new CFG after linking a library, this transaction updates the Bary and Tary tables.

The reason why a transaction-based approach is more efficient is that the check transaction performs speculative table reads, assuming there are no other threads performing concurrent writes; if the assumption is wrong, it aborts and retries. This technique suits our context well and provides needed efficiency.

The details of how transactions are implemented is in a previous publication [Niu and Tan 2014a]. We mention only that MCFI uses a transaction algorithm that is customized for MCFI's compact representation of IDs. MCFI could adopt standard STM algorithms to implement the transactions. However, those algorithms are generic and separate metadata (e.g., the version numbers) from real data (the ECNs). As a result, they require multiple instructions for retrieving metadata and real data, and multiple instructions for comparing metadata and real data to check for transaction failure and CFI violation. We micro-benchmarked the TML [Dalessandro et al. 2010] algorithm, a state-of-the-art sequence-lock-based STM algorithm particularly optimized for read-dominant workloads, and found it is substantially slower than MCFI's custom transaction algorithm, which puts metadata and real data in a single word. The compact representation enables MCFI to use a single instruction to retrieve both real data and metadata and a single instruction to check for transaction failure.

MCFI's ID-encoding scheme supports 2^{28} versions, and it might encounter the ABA problem [Dechev 2011]. For example, an attacker may load over 2^{28} modules and exhaust the MCFI's version number space. This is unlikely in practice, even for JIT compiled code. Security is violated only if the program has at least 2^{28} code updates during a check transaction. To avoid the ABA problem, MCFI maintains a counter of executed update transactions and makes sure it does not hit 2^{28} . After the completion of an update transaction, if every thread is observed to have finished using old-version IDs (when it invokes a system call or a runtime trampoline), the counter is reset to zero.

2.3.4 Code Verification

We mentioned before that it is necessary to verify a piece of dynamic code such as a library before it is incorporated into the main application. The verification process

ensures that the code is instrumented in the right way to respect the CFG and the instrumentation cannot be bypassed.

The verifier maintains three sets of code addresses in the code:

Pseudo-instruction start addresses (PSA). This address set remembers the start addresses of all *pseudo instructions*. A pseudo instruction is an instruction preceded by necessary instrumentation code. An indirect branch must be checked to consult the ID tables to determine whether the branch is legal. Before a memory write, the memory address should be masked so that the write cannot corrupt ID tables. Specifically, we define a pseudo instruction as (1) a *checked indirect branch*, which is MCFI's check-transaction instruction sequence for checking a register r followed by an indirect branch through r ; (2) a *masked memory write*, which is a memory write through r preceded by the 32-bit address-override prefix; or (3) an instruction that is neither an indirect branch nor an indirect memory write.

Indirect branch targets (IBT). This address set remembers all possible indirect branch targets.

Direct branch targets (DBT). This address set remembers all direct branch targets.

The critical invariant of the three sets is $IBT \cup DBT \subseteq PSA$. That is, all indirect and direct branch targets must be start addresses of pseudo instructions. With this invariant, it is impossible to jump to the middle of an instruction. Furthermore, it is impossible to transfer the control to an indirect branch or a memory write without executing its preceded MCFI check, which is necessary for CFI and SFI.

The three address sets are built incrementally with the installation of new code. The initial sets are built using information in the main application module, which contains code, data, and meta-information. In particular, the initial PSA and DBT are built with information in disassembled code with the help of the CFG, which tells where to start disassembling; no full disassembly is actually necessary because PSA requires identifying only instruction boundaries and DBT requires identifying only direct branch targets. The initial IBT is built once a CFG is built because the CFG tells where indirect branches can target. When a new piece of dynamic code (e.g., a library) in the form of an MCFI module is installed, the verifier updates the three address sets; that is, it computes PSA' , IBT' , and DBT' after taking the new module into consideration. The new sets are computed using a similar process to how the initial sets are computed.

With the new address sets, the verifier checks $IBT' \cup DBT' \subseteq PSA'$ and makes sure that indirect branches and memory-write instructions are appropriately instrumented; in particular, only checked indirect branches and masked memory writes are allowed.

2.3.5 Dynamic Library Loading and Unloading

We have discussed the major components for supporting dynamic libraries, including CFG generation, encoding, and enforcement. Next we describe the steps when a library is loaded and unloaded.

MCFI allows a multi-threaded program to load new libraries dynamically and transfer the control to the libraries' code. The dynamic linking is jointly performed by MCFI's dynamic linker, CFG generator, and runtime. The dynamic linker itself is instrumented by MCFI and runs within the sandbox, just like other program modules. Before any program module is loaded, the dynamic linker is first loaded in memory. The program modules' GOT entries are set to the dynamic linker's entry point. In detail, dynamically linking a library is performed in the following steps:

1. **Module preparation.** A running program invokes MCFI's dynamic linker (by jumping to a PLT entry or invoking `dlopen`) to load a new library. The dynamic linker loads the library in the sandbox and sets the library code to be writable but not executable. Then the linker analyzes the library and generates new PLT target addresses.
2. **New CFG generation.** The linker invokes the CFG generator to generate a new CFG for the original program with the new library. PLT entries are connected to functions with matching names. New IDs are generated for the Bary and Tary tables. Further, the runtime patches the in-sandbox library so that the library's code has the Bary table indexes embedded in instructions that read branch IDs. Next, the code pages are set to read-only and statically verified (as discussed in Section 2.3.4). Then the code pages of the library are set to be executable but not writable.
3. **ID table updates.** The linker passes the new PLT target addresses to the runtime and executes an update transaction, adjusting the IDs in the tables as well as modifying entries in the GOT to use the new PLT target addresses.

In addition to dynamic library loading, MCFI supports dynamic library unloading. When a library is unloaded, all indirect branch targets inside the library's code are marked invalid, achieved by changing the validity bits of relevant IDs in the Tary table to be all zeros. This prevents all threads from entering the library's code

since checks before indirect branches that target the library code would fail (as relevant IDs have been set to zeros) and there cannot be direct branches that target the library. However, there might be threads currently running or sleeping in the library's code. Hence, it is unsafe to reclaim the library's code pages immediately; otherwise, those pages could be refilled with code of a newly loaded library and the sleeping threads would resume and execute unintended instructions. To safely handle this situation, MCFI asynchronously waits until it observes that all threads have executed at least one system call or runtime trampoline call; we instrument each `syscall` instruction in `libc` to increment a per-thread counter when a `syscall` instruction is executed. Then the runtime can safely reclaim the memory allocated for the library after every counter has been incremented.

2.3.6 Implementation and Performance Evaluation

The MCFI toolchain has two tools: an LLVM-based C/C++ compiler, which performs code instrumentation and generation of CFG-related metadata; and a runtime that loads instrumented modules and monitors their execution.

The MCFI compiler is modified from Clang/LLVM-3.5, with a diff result of about 4,500 lines of changes. In summary, the changes to LLVM propagate metadata such as class hierarchies and type information, which are used for CFG generation. The metadata is inserted into the compiled ELF (embeddable and linkable format) as new sections. In addition, each MCFI-protected application runs with instrumented libraries. Therefore, we also modified and instrumented standard C/C++ libraries, including the `musl libc`, `libc++`, `libc++abi`, and `libunwind`. Moreover, since signal handlers are sandboxed in the same way as regular application code, the signal-handling stack for each thread should be in the sandbox. Therefore, after a new thread is created, the `libc` code is changed to allocate a memory region inside the sandbox and execute `sigaltstack` to switch the stack to the in-sandbox region, which is released when the thread exits.

The MCFI runtime consists of around 11,000 lines of C/assembly code. The runtime is position independent and is injected to an application's ELF executable as its interpreter. When the application is launched, the Linux kernel loads and executes the runtime first. The runtime then loads the instrumented modules into the sandbox region, creates shadow regions, and patches the code accordingly. The CFG is generated using the metadata in the code modules.

Execution-Time Overhead. We ran SPEC CPU2006 benchmarks over the reference data sets for three times and calculated the average running time. Then we compared the running time of MCFI-protected programs (including the CFG generation

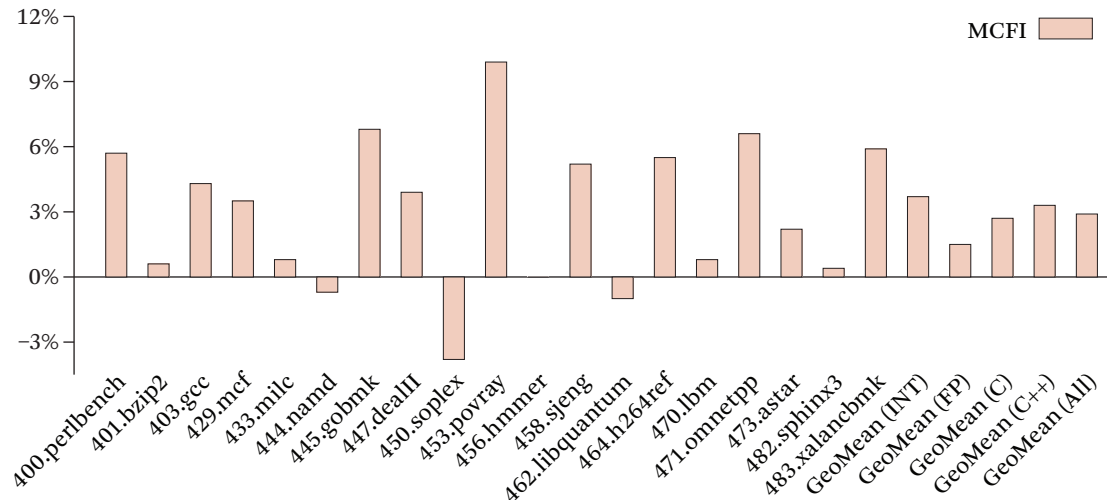


Figure 2.5 MCFI runtime overhead on SPEC CPU2006 C/C++ benchmarks.

time) with that of native programs and calculated the overhead, shown as percentages in Figure 2.5. On average, MCFI slows down program execution by 2.9%.

Two points are worth mentioning. First, notice that several benchmarks (e.g., 450.soplex) run faster with MCFI’s instrumentation. We replaced the MCFI instrumentation with nops and still observed the acceleration (e.g., 0.6% faster for 450.soplex); therefore, we believe it is because of the extra alignments that MCFI requires for indirect branch targets. Second, MCFI’s overhead is correlated with the execution frequency of indirect branches. We calculated the correlation using the Pearson correlation coefficient and got a result of 0.74, which indicates strong correlation.

2.4 Handling Just-In-Time Compiled Code

With some adjustments the same MCFI design can accommodate Just-In-Time (JIT) compilation. JIT compilers dynamically emit native code to writable memory pages and then execute the native code. In what follows, we use the term *JITted code* for the native code generated by a JIT compiler on the fly.

In the setting of ahead-of-time compilers, programs generated by compilers are targets of attacks, but compilers themselves are not since the compilers’ code is not part of the executable. In contrast, JIT compilers take untrusted programs as input and are themselves targets of attacks. Memory errors in JIT compilers can

allow attackers to inject new code or reuse JITted code in unexpected ways (e.g., [Song et al. 2015, Snow et al. 2013]). As a result, accommodating JIT compilation in MCFI brings the following new challenges:

- Our goal is to enforce CFI for a JIT compiler as well as JITted code generated by the JIT compiler, so we need to compute a single CFG for both parts. The CFG for the JIT compiler can be generated from source-level information acquired using the trusted MCFI compilation toolchain, but JITted code cannot use this process, as its compilation through the JIT compiler is not trusted. Therefore, one challenge is how to securely and efficiently generate a CFG for JITted code and merge it with the JIT compiler’s CFG (even when we cannot trust the JIT compiler).
- Since JIT compilers emit code on the fly and the code might be corrupted by attackers, another challenge is how to securely install, modify, and delete JITted code.

RockJIT, a general system that is built on top of MCFI, addresses these challenges. RockJIT hardens both the JIT compiler and JITted code, but by enforcing different levels of CFG precision on the JIT compiler and JITted code, its overhead is much smaller than previous work on protecting JIT compilation and its security is stronger. Our evaluation of Google’s V8 engine shows that RockJIT-hardened V8 can remove over 99.97% of functionality-irrelevant indirect branch edges from NaCl-JIT-hardened V8, and it is only 11.7% slower than the vanilla V8.

2.4.1 JIT Compiler Architecture and Threat Model

We first review the typical architecture of JIT compilers and use it to motivate the threat model of RockJIT.

Common JIT Architecture. We investigated a range of JIT compilers, including Google V8 (JavaScript), Mozilla TraceMonkey (JavaScript), Oracle HotSpot (Java), Facebook HHVM (PHP), and LuaJIT (Lua). We found that their architectures share many commonalities and can all be represented by the diagram in Figure 2.6. A JIT compiler emits JITted code in the code heap and executes it. The code heap is readable (R), writable (W), and executable (X). A typical JIT compiler contains the following major components:

Baseline executor. When a program starts running, its execution is the job of the baseline executor. Oftentimes, the baseline executor is an interpreter, which is easy to implement but slow. For instance, HotSpot has an interpreter

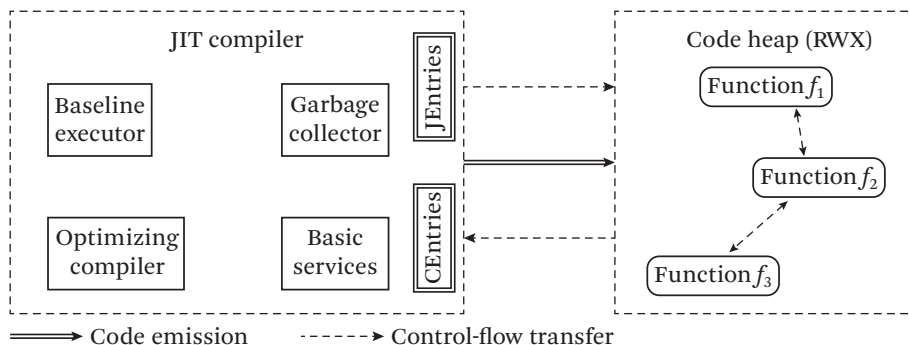


Figure 2.6 The common architecture of modern JIT compilers.

that interprets Java bytecode. The baseline executor may have a different implementation from an interpreter. For example, the baseline executor of V8 compiles JavaScript source code directly to unoptimized native code.

Optimizing compiler. During the execution of a program by the baseline executor, the JIT compiler performs runtime profiling to identify hot code and to infer types in the case of dynamically typed languages. Based on the runtime profile, the optimizing compiler generates optimized native code. JIT compilers can have quite different designs for their optimizers. For example, V8 profiles method execution and optimizes a whole method at a time. In contrast, TraceMonkey profiles execution paths (e.g., a hot loop) and performs trace-based optimization [Gal et al. 2009].

Garbage collector. Managed languages provide automatic memory management, which is supported by a garbage collector. Most garbage collectors implement common algorithms such as concurrent mark and sweep.

Basic services. A JIT compiler also provides runtime services, including support for debugging, access to internal states for performance tuning, and foreign function interfaces for enabling interoperation between managed code and native code.

For performance, all JIT compilers we inspected are developed in C/C++. Since the calling convention of C/C++ is different from that of JITted code, which is JIT-compiler specific, JIT compilers introduce interfaces to allow context switches between the code of the compiler and JITted code. In Figure 2.6, the interfaces are shown as JEntries and CEntries; both are implemented by indirect branches.

`JEntries` transfer control to JITted code and `CEntries` transfer control to the JIT compiler. As an example of `JEntries` in V8, the initial control transfer from the JIT compiler to the code heap is through an indirect call (`JEntry`) in a code stub called `JEntryStub`. As an example of `CEntries`, V8 provides services (or functions) such as JavaScript object creation and object property access. When JITted code invokes these services, the control is first transferred to a stub called `CEntryStub` with a register containing the address of the target service function. Within `CEntryStub`, an indirect call (`CEntry`) through the register is executed to transfer the control to the service function. Moreover, it should be noted that both `CEntries` and `JEntries` could be dynamically generated (e.g., the JIT compiler can emit JITted functions that directly invoke `CEntries` to efficiently call a JIT-engine service).

Threat Model. On top of MCFI's concurrent attacker threat model, RockJIT's threat model makes further assumptions about a JIT compiler. It assumes the JIT compiler's code is benign but may contain vulnerabilities. The JITted code can contain malicious logic since it is compiled from source code that might be provided by the attacker. The malicious logic aims to launch attacks such as code injection and JIT spraying.

We further make two assumptions about the JIT compiler. First, we assume that context switches between the JIT compiler and JITted code occur through a set of interface functions; that is, only through one of those `JEntries` and `CEntries` in Figure 2.6 can the control transfer between the JIT compiler and JITted code. This assumption enables different CFG precision on the JIT compiler and JITted code. Second, we assume JITted code, when executed normally (i.e., no jumps to the middle of instructions), does not contain direct system call invocations and privileged instructions. The JITted code can, however, invoke one of the `CEntries` to request services such as OS system calls from the JIT compiler (after appropriate security checking by the compiler). These two assumptions hold in all the JIT compilers we have inspected. Even if a certain JIT compiler violates these assumptions, it should be easy to modify it to make the assumptions hold.

2.4.2 RockJIT Architecture

RockJIT's architecture is visualized in Figure 2.7. It provides services to a JIT compiler and monitors its security. An existing JIT compiler, such as V8, is modified slightly to cooperate with RockJIT. It is then compiled and instrumented by MCFI's compilation toolchain to generate an MCFI module. The module is loaded by RockJIT into a sandbox. After loading, RockJIT generates a control-flow graph for the JIT

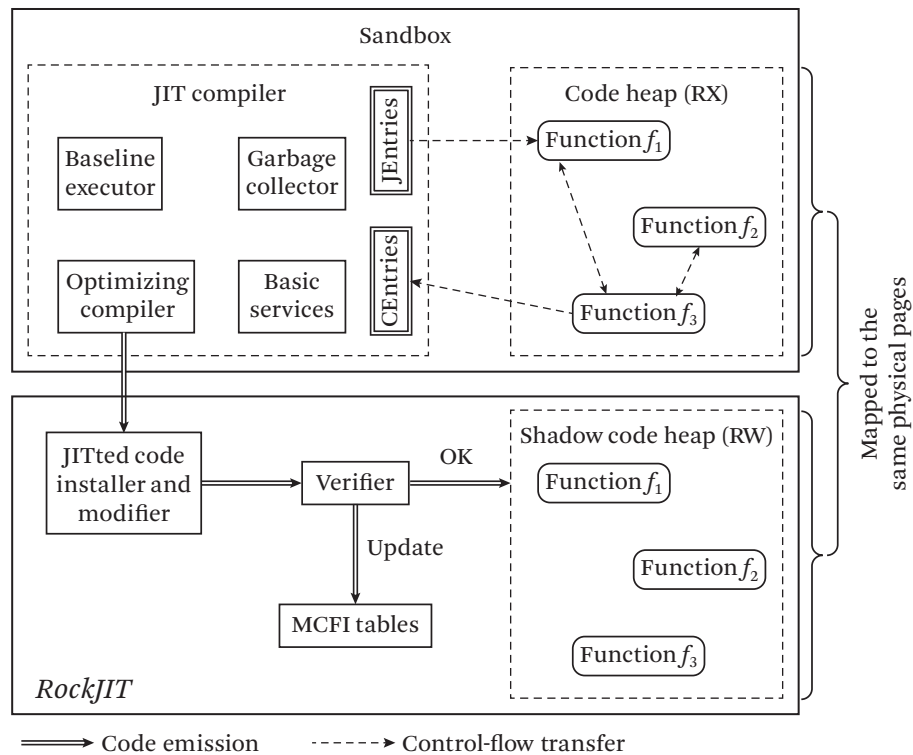


Figure 2.7 The architecture of RockJIT.

compiler based on the auxiliary type information in the module, constructs MCFI tables that encode the control-flow graph, and starts execution of the JIT compiler.

The sandbox around the JIT compiler and JITted code restricts their control flow according to the tables and also restricts their memory access to inside the sandbox. The JIT compiler can request services provided by RockJIT via a set of well-defined interface functions. For example, to prevent code in the sandbox from changing memory protection arbitrarily, all direct system calls for changing memory mapping and memory pages' protection bits are forbidden; instead, the code can invoke services provided by RockJIT to issue such system calls in a managed way.

To rule out code injection attacks, RockJIT guarantees that no memory pages are writable and executable at the same time, similar to Data Execution Protection (DEP). One thorny issue we mentioned before is that the code heap (i.e., memory pages that hold JITted code) is made both writable and executable in typical JIT compilers.

To address this issue, RockJIT uses a parallel mapping scheme in which a *shadow code heap* is added outside the sandbox. This is similar to what NaCl-JIT does [Ansel 2014]; the same parallel mapping scheme was presented earlier (in Section 2.3.2) to mitigate the security weakness of GOT.PLT tables. The shadow code heap is in RockJIT's private memory; it is mapped to the same physical pages as the code heap in the sandbox but with *different permissions*. In particular, the code heap in the sandbox is made readable and executable but not writable. The shadow code heap is made readable and writable but not executable.⁴ Because the JIT compiler is restricted to access memory inside the sandbox, the JIT compiler cannot directly modify the shadow code heap for runtime code manipulation. Instead, it invokes the services of RockJIT to install new native code or modify existing native code. RockJIT performs verification on the native code to check a set of properties for security. If the verification succeeds, RockJIT installs the new code in the shadow code heap and updates MCFI tables using a new control-flow graph that takes the new code into account. Since the shadow code heap maps to the same physical pages as the in-sandbox code heap, the code heap is filled with the same code, which can then be invoked by the JIT compiler.

RockJIT CFG Generation. RockJIT enforces control-flow integrity on both the JIT compiler and JITted code, but it applies different levels of precision on each. For the JIT compiler, RockJIT applies a C++ CFG generation strategy detailed in Section 2.2 to produce a fine-grained CFG offline; it takes into consideration C++ semantics, such as virtual method calls. In contrast, the CFG for JITted code is coarse grained in the sense that all its indirect branches share a common set of targets. The JIT compiler is modified to emit not only native code but also information about indirect branch targets. RockJIT then constructs the coarse-grained CFG for the new code and combines it with the old CFG.

The approach of hybrid CFI precision in RockJIT is the result of a careful balance between security and performance. First, the JIT compiler's code is where the majority of the code is, and it contains dangerous system call invocations. Since its code is statically available, constructing a fine-grained CFG offline for the JIT compiler increases security substantially as recent work has shown that coarse-grained CFI can suffer from ROP attacks [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014]. On the other hand, JITted code is frequently generated on the fly, and for performance it is important that verification and new CFG generation do

4. Since the shadow code heap is controlled by trusted RockJIT, whether it is executable or not does not affect security; we make it not executable, following the principle of least privilege.

not have high performance overhead. Verification and CFG generation algorithms for coarse-grained CFI run much faster. Some readers may wonder whether coarse-grained CFI for JITted code might jeopardize security. We do not believe that is the case because one of our assumptions is that JITted code cannot contain dangerous instructions such as system calls, a property that is enforced by RockJIT's verifier; such instructions are required in an attack. JITted code can still request system call services from the JIT compiler, but the JIT compiler is hardened through fine-grained CFI: security is maintained as long as sufficient checks are placed before system calls for the set of control-flow paths in a fine-grained CFG, which is a much smaller set than the one in a coarse-grained CFG.

One point worth mentioning is that, thanks to the verifier, the JIT compiler is not in the Trusted Computing Base (TCB) even though it performs runtime code generation. Native code generated by the JIT compiler is first checked to obey a set of safety properties before it is installed. The verifier is in the TCB, but it is much smaller than the JIT compiler.

2.4.3 JITted Code Manipulation

The code heap maintained by a JIT compiler is where code is dynamically managed. It consists of multiple regions of code such as functions. A JIT compiler dynamically installs, deletes, and modifies code regions. New code regions are frequently generated by the compiler and installed in the code heap. When a code region is no longer needed, the JIT compiler can delete it from the code heap and reuse its memory for future code installation. Runtime code modification is mostly used in performance-critical optimizations. As an example, *inline caching* [Deutsch and Schiffman 1984, Hölzle et al. 1991] is a technique that is used in JIT compilers to speed up access to object properties. In this technique, a JIT compiler modifies native code to embed an object property, such as a member offset after the property has been accessed for the first time, avoiding expensive object-property access operations in the future. Another example of runtime code modification happens in V8 during code optimization. V8 profiles function and loop execution to identify hot functions and loops. It performs optimization on the hot code to generate an optimized version. Afterward, runtime code patching is performed on the unoptimized code to transfer its control to the optimized version through a process called on-stack replacement [Hölzle et al. 1992].

Since RockJIT enforces CFI, it is necessary to check security for each step of runtime code installation, deletion, and modification. In RockJIT, a JIT compiler cannot directly manipulate the code heap, which does not have the writable permission. Instead, RockJIT provides services to the JIT compiler for code installation,

deletion, and modification. One worry for runtime code manipulation is thread safety: one thread is manipulating code, while another thread may see partially manipulated code. We next discuss the detailed steps involved in RockJIT's code manipulation and how thread safety is achieved.

Code Installation. For code installation, the JIT compiler invokes RockJIT's code installation service and sends a piece of native code, the target address where the native code should be installed, and meta-information about the code for constructing new address sets used in the verification process. The code installation service then performs the following steps:

1. Verification is performed on the new piece of code following the process in Section 2.3.4: the verifier performs verification on the code and updates the address sets to PSA' , IBT' , and DBT' . In addition, the RockJIT verifier checks the following properties:
 - (a) The code contains only instructions that are used for a particular JIT compiler. This set of instructions is usually a small subset of the native instruction set and can be easily derived by inspecting the code-emission logic of a JIT compiler. Importantly, this subset cannot contain system calls and privileged instructions—one of our assumptions.
 - (b) There are no direct branches from the JITted code region to the JIT compiler region (and vice versa). Recall one of the assumptions in the threat model is that control can be transferred between the two regions only through a set of well-defined interface functions.
2. If the verification succeeds, the code is copied to the shadow code heap at an address computed from the start address where the code should be installed.
3. The runtime ID tables used by MCFI are updated to take into account the new code. Since coarse-grained CFI is enforced on JITted code, only information in IBT' is needed to update the tables.

There are a few notes worth mentioning about the above steps. First, the verification of benign programs is expected to succeed if there are no bugs in the JIT compiler. A verification failure indicates a bug that should be fixed. Second, it is important that the MCFI tables are updated *after* copying the code, not before. During the copying process, the code becomes partially visible to the JIT compiler as the code heap is mapped to the same physical pages as the shadow code heap. However, since the MCFI tables have not been updated yet, no branches can jump

to the new code, avoiding the situation in which one thread is installing some new code and another thread branches to partially installed code.

Code Deletion. Deletion of JITted code is similar to library unloading discussed in Section 2.3.5, with the only difference being that before deactivating all targets in the code being deleted, RockJIT should make sure there are no direct branches targeting the code that is to be deleted.

Code Modification. If the new code region has the same internal pseudo instruction boundaries and native instruction boundaries as the old code region, and only the native instructions are modified, and the new code passes verification, RockJIT follows NaCl-JIT's approach to replace the old code with the new code. Otherwise, code modification is implemented as a code deletion followed by a code installation.

2.4.4 Modification to a JIT Compiler

Existing JIT compilers need to be modified to work with RockJIT, and our experience involved adapting Google's V8 JavaScript engine (3.29.88.19). To adapt V8's x64 source, we modified 1,934 lines of its source code: 1,914 lines were changed to make it generate MCFI-compatible code and invoke RockJIT's services for runtime code manipulation; 20 lines were added for fixing bad type casts (detailed in Section 2.2) that prevent sound CFG generation. This experience demonstrates that modifying an existing JIT compiler to work with RockJIT requires modest effort. Most of the changes to V8 were in its code-emission logic to make the generated code compatible with MCFI:

- Code-emission functions that generate indirect branches were modified to generate checked indirect branches. We could directly use MCFI's instrumentation (i.e., the transactions) to rewrite the JITted code, but for coarse-grained CFG enforcement, we use a simplified CFI-check implementation (detailed in [Niu 2015]).
- Code-emission functions for indirect memory writes were modified to generate masked memory writes. The sandbox resides in the [0, 4GB) memory. Therefore, an indirect memory write is prefixed with 0x67 (the 32-bit address-override prefix) to clear the first 32 bits of a 64-bit address.

Another part we modified was to accommodate online code patching. When V8 emits certain optimized native code, it reserves some bytes in the code in anticipation of future code patching (for a process called deoptimization). The

Table 2.2 CFG Statistics of the Google V8 JavaScript Compiler

SPEC CPU2006	IBs (with matching targets)	IBTs	EQCs	Average	
				IBTs / IB	IBs / IBT
V8	35,775 (29,609)	116,919	9,696	808	205

original V8 reserves 13 bytes for this purpose. RockJIT needs more bytes because of extra MCFI checks; we had to reserve 24 bytes instead. Next, changes were made to V8 to invoke code installation, deletion, and modification services provided by RockJIT at appropriate places. Finally, since V8 emits JEntries and CEntries on the fly, RockJIT provides services for V8 to securely install those JEntries and CEntries as well as their type signatures to enable CFG generation. Therefore, changes were made to V8 to invoke those services.

Compared to related work, RockJIT changes around 60% less code than NaCl-JIT, which changed over 5,000 lines of code for the x64 version of V8. NaCl-JIT requires more changes because (1) it disallows the mix of code and data in the JIT-compiled code and V8 has to be changed to separate code and data, whereas RockJIT's CFI allows the mixture of code and data as long as data cannot be reached from code with legal control flow; and (2) NaCl-JIT uses the ILP32 programming model on x64, while the native V8 uses the LP64 model; therefore, it has to change nearly the entire code-emission logic.

2.4.5 Evaluation

We compiled the modified Google V8 JavaScript compiler to a stand-alone executable using the MCFI toolchain and measured the generated CFG and performance overhead.

CFG Statistics. RockJIT supports fine-grained CFGs for the JIT compiler. Table 2.2 presents the details of the CFG constructed for V8. Similar to SPEC CPU2006 C++ benchmarks, thousands of equivalence classes are supported, and the average number of targets of indirect branches and average number of indirect branches targeting an address are much less than coarse-grained CFI, which could be the number of indirect branch targets and the number of indirect branches, respectively. Compared to NaCl-JIT, which enforces a form of coarse-grained CFI on V8's code, RockJIT's CFG removes about 99.7% of indirect branch edges from NaCl-JIT's CFG.

Execution-Time Overhead. We measured the slowdown of RockJIT-instrumented V8 over Octane 2 benchmarks. On average, 11.7% runtime overhead is incurred by RockJIT.

Also, we separately calculated runtime-overhead results for the subset of benchmarks that were included in Octane 1 (the predecessor of Octane 2) since related systems use Octane 1 for evaluation. RockJIT incurs only 3.1% overhead on average. Compared to other JIT-compiler hardening systems, such as NaCl-JIT [Ansel et al. 2011], librando [Homescu et al. 2013], and SDCG [Song et al. 2015], MCFI incurs less overhead and provides better security.

2.5 Related Work

Most fine-grained CFI systems do not support modularity [Erlingsson et al. 2006, Wang and Jiang 2010, Akritidis et al. 2008, Zeng et al. 2011, Zeng et al. 2013a, Davi et al. 2012, Pewny and Holz 2013, Criswell et al. 2014]. For instance, HyperSafe [Wang and Jiang 2010] statically constructs a table for each indirect branch, and the table contains all target addresses that the indirect branch can jump to. The program is changed so that table indexes, not target addresses, are used. Before an indirect branch, an index is converted into a target address using information in the table. HyperSafe's CFI precision is higher than the classic CFI's, which uses the notion of equivalence classes for efficiency at the expense of precision. However, HyperSafe does not support separate compilation because it requires a whole-program analysis to build the tables and those tables cannot be changed at runtime. As another example, WIT's CFI enforcement [Akritidis et al. 2008] uses a color table. The color table is built by static analysis on the compiler IR code. Each indirect call is statically assigned a color, and functions that the call can target are assigned the same color. The color table is represented during runtime, and dynamic checks consult the table before an indirect call. WIT's color table is similar to MCFI's tables. But WIT relies on a whole-program analysis to construct its table and does not support dynamically linked libraries.

A few CFI systems have modularity support. However, the modularity support of ForwardCFI [Tice et al. 2014] introduces time windows for attacks during dynamic module linking. MCFI [Niu and Tan 2014a] is the first fine-grained CFI technique that supports dynamic code linking, and RockJIT extends MCFI to cover JIT compilation. MCFI requires source code, and a couple of CFI systems aim to enforce CFI at the binary-code level, including Lockdown [Payer et al. 2015c] and vfGuard [Prakash et al. 2015]. They can work with programs without source code, at the expense of performance and CFG precision.

Our follow-up work π CFI [Niu and Tan 2015] enforces per-input CFGs. In MCFI, CFG generation is performed at runtime, and as a result it cannot afford advanced static analysis; this can cause CFG precision loss. However, π CFI shows that CFG precision can be significantly improved using dynamic information. Specifically, it enforces a per-input CFG and is more precise than an all-input CFG, which has to consider all possible inputs. Recent attacks such as Control-Flow Bending [Carlini et al. 2015e] and Control Jujutsu [Evans et al. 2015] show methods of attacking conventional fine-grained CFI systems, but fortunately π CFI can mitigate those attacks.

Code-Pointer Integrity (CPI [Kuznetsov et al. 2014a]) is a recent system that isolates all data related to code pointers into a protected safe memory region and thus can mitigate control-flow hijacking attacks. It is also a compiler-based framework and has low execution overhead. However, it incurs high memory overhead. Furthermore, CPI does not directly enforce a control-flow graph. The control-flow graph provided by CFI methods such as MCFI and π CFI is valuable to other software-protection mechanisms because they can use it to perform static-analysis-based optimization and verification [Zeng et al. 2011].

Work Related to the Security of JIT Compilers. RockJIT’s goal of improving the security of JIT compilation is shared by several other systems. Perhaps the closest work is NaCl-JIT [Ansel et al. 2011], which applies SFI to constraining both a JIT compiler and JITted code. To prevent SFI checks from being bypassed, NaCl-JIT enforces aligned-chunk CFI similar to PittSFIeld [McCamant and Morrisett 2006], which enforces coarse-grained CFGs. In contrast, RockJIT applies fine-grained CFI on the JIT compiler and therefore provides stronger security. NaCl-JIT also has high performance overhead. Its aligned-chunk CFI requires insertion of many nop instructions to make indirect branch targets aligned at chunk boundaries. NaCl-JIT reports nops account for half of the sandboxing cost. Largely because of this, its performance overhead is around 51%, while RockJIT’s overhead is only 11.7%.

In addition, software diversification has been used to harden JIT compilation. The librando system [Homescu et al. 2013] inserts a random amount of nops in the JITted code. In addition, it uses a technique called constant blinding, which replaces instructions that have constant operands with other equivalent instruction sequences to mitigate JIT spraying [Blazakis 2010]. Due to its black-box implementation, librando has to disassemble the JITted code, modify the code, and re-assemble the new code. It incurs a significant overhead (265.8%). Other systems, including INSERT [Wei et al. 2011], JITSafe [Chen et al. 2013], and RIM [Wu et al. 2012], also employ diversification techniques similar to librando’s.

Readactor [Crane et al. 2015] leverages execute-only pages supported by virtualization and runs randomized JIT engine and JITted code inside those pages. Most of these diversification-based systems protect only JITted code, not the JIT compiler. Even Readactor needs to temporarily allow writable code during JITted code installation. In comparison, RockJIT can eliminate JIT spraying attacks and enforces CFI on both the JIT compiler and JITted code. On the other hand, since software diversification techniques are orthogonal to CFI, it is perhaps beneficial to deploy both defenses in a JIT compiler, following the principle of defense in depth.

Another mitigation mechanism for JIT is to separate the write permission from the execution permission for the code heap. For instance, SDCG [Song et al. 2015] stores the shadow code heap in another process and emits code to the process through inter-process communication. However, the process-based separation incurs higher overhead than RockJIT's SFI-based separation. JITDefender [Chen et al. 2011] and JITSafe [Chen et al. 2013] drop the write permission of the code heap whenever it is not needed. However, before dropping the permission, those code pages may have already been modified by the attacker for arbitrary code execution. More importantly, they cannot prevent JIT spraying attacks, which do not require modifying the code heap.

2.6 Conclusion

MCFI is the first efficient CFI instrumentation that supports separate compilation. It addresses the challenge of how to support dynamically linked libraries for CFI in the presence of multi-threaded code, using a novel approach based on transactions. It is also extended to cover JIT compilation in RockJIT. In a previous publication [Niu 2015], we also show how it can be extended to support the interoperation between MCFI-instrumented modules and non-instrumented modules.

In evaluating MCFI and RockJIT, we have implemented a compilation toolchain, which instruments C and C++ programs. Our experiments on SPEC CPU2006 benchmarks and other programs show that MCFI imposes about 5% execution-time overhead on average. The MCFI toolchain has been open sourced and is available for download at the following URL: <http://github.com/mcfi>.



Diversity and Information Leaks

Stephen Crane, Andrei Homescu, Per Larsen,
Hamed Okhravi, Michael Franz

Almost three decades ago, the Morris Worm infected thousands of UNIX workstations by, among other things, exploiting a buffer-overflow error in the *fingerd* daemon [Spafford 1989]. Buffer overflows are just one example of a larger class of memory (corruption) errors [Szekeres et al. 2013, van der Veen et al. 2012]. The root of the issue is that systems programming languages—C and its derivatives—expect programmers to access memory correctly and eschew runtime safety checks to maximize performance. There are three possible ways to address the security issues associated with memory corruption. One is to migrate away from these legacy languages that were designed four decades ago, long before computers were networked and thus exposed to remote adversaries. Another is to retrofit the legacy code with runtime safety checks. This is a great option whenever the, often substantial, cost of runtime checking is acceptable. In cases where legacy code must run at approximately the same speed, however, we must fall back to targeted mitigations, which, unlike the other remedies, do not prevent memory corruption. Instead, mitigations make it harder, i.e., more labor intensive, to turn errors into exploits.

Since stack-based buffer overwrites were the basis of the first exploits, the first mitigations were focused on preventing the corresponding stack smashing exploits [Levy 1996]. The first mitigations worked by placing a canary, i.e., a random value checked before function returns, between the return address and any buffers that could overflow [Cowan et al. 1998]. Another countermeasure that is

now ubiquitous makes the stack non-executable. Since then, numerous other countermeasures have appeared and the most efficient of those have made it into practice [Meer 2010]. While the common goal of countermeasures is to stop exploitation of memory corruption, their mechanisms differ widely. Generally speaking, countermeasures rely on randomization, enforcement, isolation, or a combination thereof. Address space layout randomization is the canonical example of a purely randomization-based technique. Control-Flow Integrity (CFI [Abadi et al. 2005a, Burow et al. 2016]) is a good example of an enforcement technique. Software-fault isolation, as the name implies, is a good example of an isolation scheme. Code-Pointer Integrity (CPI [Kuznetsov et al. 2014a]) is an isolation scheme focused on code pointers. While the rest of this chapter focuses on randomization-based mitigations, we stress that the best way to mitigate memory corruption vulnerabilities is to deploy multiple different mitigation techniques, as opposed to being overly reliant on any single defense.

3.1 Software Diversity

Randomization, or software diversity [Cohen 1993, Larsen et al. 2014], essentially hides implementation details, such as the memory layout, from adversaries. This means that adversaries cannot rely on code, variables, or other program artifacts residing at a known location. This idea has similarities with biodiversity wherein some fraction of animals in a herd will have immunity against environmental hazards due to random differences in their immune systems. One can also draw parallels to kinetic warfare insofar that belligerents seek to conceal their locations to avoid becoming an easy target.

Because adversaries in the digital domain seek to exploit implementation flaws that trigger invalid memory accesses, the inputs that cause the unintended behavior are highly implementation dependent. This is why randomization of the code layout has a destabilizing effect on code-reuse attacks that depend on code snippets (*gadgets* in ROP parlance [Shacham 2007]) residing at known addresses.

Adversaries generally have two ways to bypass diversified binaries: guessing or reconnoitering their target. Repeatedly mounting an attack that crashes the victim program [Bittau et al. 2014, Shacham et al. 2004, Evans et al. 2015a] has visible side effects that often facilitate detection. Information leakage, on the other hand, is often silent and leaves few traces, if any, on the victim system. In the rest of this chapter, we focus on bypasses of diversity relying on information leakage, particularly code layout disclosure, and the countermeasures available to defenders.

3.2 Information Leakage

In their seminal paper on stack guards, Cowan et al. mention that their techniques are not impossible to bypass, but to do so would require the attacker to examine the entire memory image of the program [Cowan et al. 1998 (p. 4)]. The tacit assumption is that the attacker cannot easily leak the memory contents of a running program. Their follow-up work focusing on pointers also cites the difficulty of accessing process memory in their security argument: “To obtain the key, the attacker would either have to already have permission to manipulate the process with debugging tools (e.g., ptrace) or would have to have already successfully perpetrated a buffer overflow attack against the process” [Cowan et al. 2003]. Strackx et al. [2009] were the first to examine what they termed the “Memory Secrecy Assumption” underpinning randomizing defenses at the time. The gist of their argument is that memory secrecy relies on the absence of memory corruption vulnerabilities, an assumption that, if valid, would also obviate the need for memory corruption mitigations, such as ASLR, stack canaries, and other diversity techniques. Information leakage can arise from format string vulnerabilities that cause the defective program to print out internal data or code rather than the intended output. Strackx et al. point out that buffer over-reads are a more common source of information leakage and demonstrate a concrete attack in which ASLR and ProPolice [Etoh and Yoda 2000] can be bypassed thanks to such over-reads.

Serna [2012] highlighted that type confusion and use-after-free vulnerabilities as well as application-specific vulnerabilities also facilitate information leakage. The presentation also highlighted that the widespread deployment of ASLR and stack canaries in all modern operating systems had made information leakage a requirement to write reliable exploits. Most importantly, Serna noted that the combination of attacker-controlled scripting and memory corruption errors put adversaries in a powerful position.

Snow et al. [2013] translated Serna’s observation into practice by using an overflowed buffer object to systematically scan the memory of the process running a malicious script. Just-in-time code-reuse, JIT-ROP, attacks generalize previous attacks and are worth summarizing here. The general goal of JIT-ROP is to find as many mapped code pages as possible by starting from a small root set of known pages. The discovery of additional code pages happens by recursively scanning each page for references to other pages and adding these pages to a working set. In context of browsers, the JIT-ROP technique is used to break out of a sandboxed scripting environment, such as a JavaScript VM hosted by a browser. This lets the adversary execute arbitrary code with all permissions granted to the operating system process. To do so, the adversary tricks an unsuspecting user into visiting a web page

-serving a malicious script. The script constructs a write-what-where primitive out of a memory corruption vulnerability such that the adversary can access any mapped location within the virtual address space of the process. Since the code layout is not known to the adversary a priori, the exploit fails if it touches unmapped memory and the resulting segmentation fault is not handled by the program. Segmentation faults are avoided by scanning for pointers to code in the data memory surrounding the overflowed object (using a priori knowledge of the heap layout). Next, the exploit scans the code page identified by the code pointer. Since the virtual-to-physical memory mapping happens at the page granularity, it is always safe to scan an entire page, which is usually 4KiB in size. Snow et al. realized that they could implement a disassembler in JavaScript to recover references between code pages and use the recovered references to discover additional code pages recursively. The recursive disassembly step terminates when the script has discovered enough code snippets to mount a traditional code-reuse attack.

3.3 Mitigating Information Leakage

[Backes and Nürnberger \[2014\]](#) were first out of the gate with a response to JIT-ROP attacks. Their technique, Oxymoron, splits the code segment into 4KiB pages. Furthermore, any code reference to another page is indirected through a lookup table. The base of the lookup table is hidden using the vestiges of x86 segmentation. This prevents the recursive disassembly step in the JIT-ROP attack. An interesting aspect of Oxymoron is that the scheme was designed to allow code pages to be shared among processes. This is an important optimization for shared libraries and one that is overlooked by most of the academic literature although it is crucial in practice.

[Davi et al. \[2015\]](#) presented a different response to JIT-ROP attacks—Isomeron—motivated by their finding that the original JIT-ROP technique could be modified slightly to bypass Oxymoron. The key to the Oxymoron bypass was the finding that data memory contains enough pointers to discover enough code pages to mount an attack, even if it is not possible to discover additional pages through inter-page references thanks to Oxymoron. Virtual method tables for the C++ dispatch mechanism, for example, enable pointer harvesting and lessen the need for recursive disassembly. The Isomeron defense [[Davi et al. 2015](#)] frustrates return-oriented programming techniques by cloning each program function and randomly picking between original and function clones during execution. Code-reuse exploits need

not use returns to chain gadgets, so the Isomeron technique has shortcomings of its own.

[Backes et al. \[2014\]](#) advocated for a more principled way to counter information leakage: preventing read accesses to code pages. Their implementation—eXecute-no-Read or just XnR—presented a work-around for all x86 processors whose memory management units lack native support for executable, non-readable pages. To work around this limitation, XnR prevents reads by clearing the present bit for nearly all code pages. Normally, the CPU uses the present bit to track which pages are present in RAM and or paged out to disk. Accesses to a page with the present bit cleared, causes the CPU to generate a page fault which the operating system handles by reading the missing page from the pagefile. XnR modifies the operating system's page fault handler to mark XnR pages present (without evicting their contents) if and only if the present bit was cleared to prevent read accesses *and* if the page fault was triggered by an instruction fetch, i.e., an attempt to execute the page was made. If, on the other hand, the fault was generated by a read access to an executable page, the XnR page fault handler terminates the program before any memory contents can be leaked. The number of page faults to handle determines the overhead of the XnR approach. To avoid excessive slowdowns, XnR keeps a small window of recently executed pages readable and executable—and thus exposed to information leaks. However, XnR uses a sliding window of two to eight pages to limit the amount of code that can be leaked at any point in the execution.

[Gionta et al. \[2015\]](#) developed a system—HideM—that similarly made code pages unreadable but does so by using the Translation Look-aside Buffer (TLB) in a special way known as TLB-desynchronization. On processors that use separate TLBs for data and code, the two TLBs are usually kept in sync, which gives an executing process the same view of its address space regardless of the type of access. HideM configures the memory management unit such that accesses to the same virtual address translate to different physical addresses depending on the access type. This way, instruction fetches proceed as intended whereas read accesses—whether malicious or not—go to a different physical copy of the text section. To ensure that legitimate reads to constant data stored on code pages function correctly, HideM zeros out all instructions in the readable copy of the text section while preserving all embedded constant data. This is a point in favor of HideM since XnR does not explicitly address the problem of reading embedded constants. On the other hand, most modern processors have unified TLBs and thus do not support TLB-desynchronization as required by HideM.

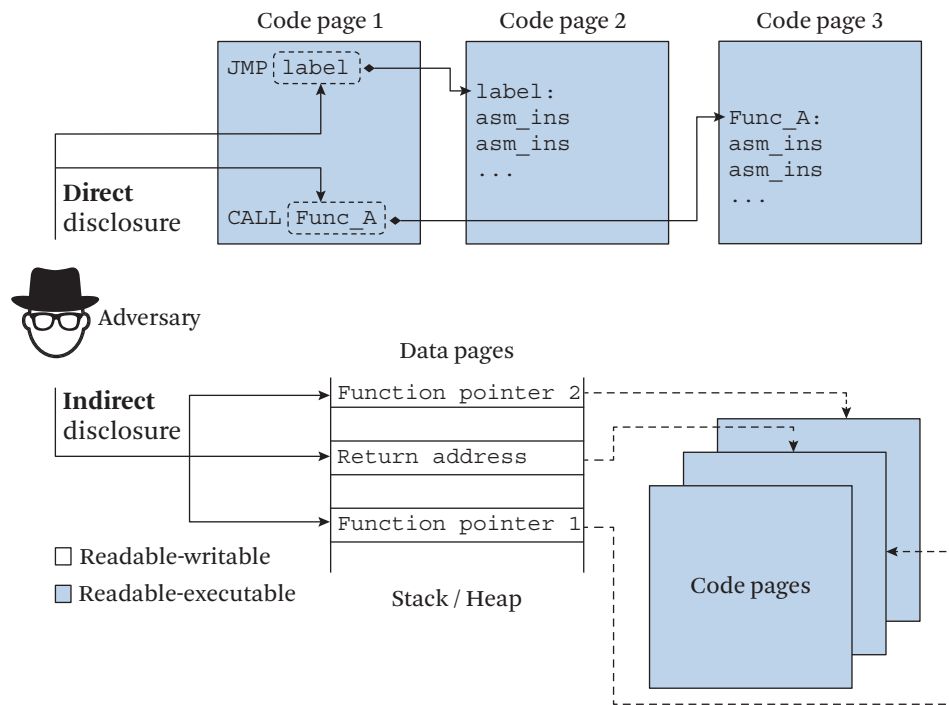


Figure 3.1 Direct and indirect memory disclosure. (Based on Crane et al. [2015])

While XnR and HideM goes a long way toward preventing *direct* leakage through adversarial reads, adversaries can also make inferences about the code layout by inspecting code pointers stored in the data segments of a running process. The difference between these two types of leakage is illustrated in Figure 3.1. The defenses we’ve discussed so far have protected the code pages and references between them (top half of figure) but not references from data pages to code pages (bottom half of figure). The utility of leaking a function pointer or return address when code pages cannot be read directly depends on the granularity of the code layout diversity. If each individual instruction is placed at a random location [Hiser et al. 2012], such leaks mainly facilitate whole-function reuse. However, the most granular diversity techniques tend to have high overheads [Larsen et al. 2014] and may prevent page sharing between processes [Backes and Nürnberg 2014, Crane et al. 2016].

Crane et al. [2015] built a system—Readactor—that explicitly seeks to prevent both direct and indirect leakage of code layout. Rather than emulating execute-no-

read permissions, Readactor leverages the extended page translation mechanisms found in modern processors (circa 2008 and onward) to accelerate hypervisors. Memory accesses inside virtual machines undergo two levels of address translation: (i) guest virtual to guest physical translation and (ii) guest physical to host physical translation. The effective permission of an access to host physical memory is the intersection of the permissions used in the two translation steps. Unlike the first translation step, which forces read permissions on executable pages, the second translation step can represent true execute-only memory permissions. The Readactor system used a lightweight hypervisor to activate the extended page tables on a per-process basis to protect individual applications running on a traditional host system, i.e., outside a traditional hypervisor. Rather than allowing accesses to constant data on code pages, Redactor used a modified compiler to eliminate all such reads. The major open source C/C++ compilers later stopped emitting constants on code pages for performance reasons, which also benefits execute-only techniques.

Readactor tackles indirect leakage by introducing a pointer indirection layer so no pointer stored in a readable memory region points directly to its target. All that adversaries can observe are pointers into a special execute-only area containing trampolines (direct jumps) to the actual functions. Because the trampolines are stored on pages with execute-only memory, they cannot be dereferenced by an exploit. Adversaries therefore cannot learn the locations of functions in the absence of hardware-level side channels [Gras et al. 2017] or implementation errors. Readactor also demonstrated that just-in-time compiled code can be made compatible with execute-only memory with modest effort; the need to also protect JITed code from indirect disclosure was highlighted but not implemented. The necessity of avoiding indirect disclosure of JITed code was reiterated by Maisuradze et al. [2017].

A few variations of and extensions to the basic ideas behind XnR, HideM, and Readactor are worth mentioning. Schuster et al. [2015] demonstrated a new type of code-reuse attack called Counterfeit Object Oriented Programming (COOP), which is capable of bypassing control-flow integrity defenses that are not C++ aware. C++ awareness, in this context, simply means using information about class hierarchies to further constrain the set of outgoing control-flow edges at a C++ virtual method call site. C++-aware CFI is straightforward to implement when program source code is available, whereas techniques to recover class hierarchies via binary analysis took a while to appear [Pawlowski et al. 2017]. Since COOP attacks execute entire C++ methods without regard for the actual code layout, such attacks can also bypass defenses such as Readactor. COOP attacks are not entirely layout agnostic, however; they require knowledge of the layout of C++ objects and the layout of C++ virtual

method tables. Since objects must be stored in RW memory, their layouts are difficult to hide. Vtables, on the other hand, contain a mix of data and pointers to code, the latter part of which can be hidden and randomized along the lines of the Readactor system. Crane et al. [2015] presented a counter to COOP attacks called Readactor++ that splits virtual method tables into two parts: one containing data and another containing code (direct jump trampolines to virtual methods). The code part, called the `xvtable`, is protected by execute-only permissions, and randomized. To prevent brute force attacks, dummy entries that are never activated during normal program execution are added to each `xvtable` [Crane et al. 2013].

Supporting execute-only memory is not always straightforward and most approaches rely on using the memory management unit in unconventional ways. For systems where MMU “tricks” are infeasible, such as systems having a simpler memory protection unit, execute-only permissions can be enforced in software [Braden et al. 2016] using techniques conceptually similar to software-fault isolation [Wahbe et al. 1993, McCamant and Morrisett 2006].

Lu et al. [2015] demonstrated that it is possible to use a pointer indirection layer to prevent indirect leakage *without* using execute-only memory to protect against direct leakage. Their proposed solution, ASLR-Guard, uses the vestiges of x86 segmentation support to hide the location of a table that translates between code locators (visible to adversaries) and actual code addresses (hidden). Lu et al. argue that without a way to disclose code addresses, there is no need to prevent against direct leakage since a 64-bit virtual address space is large enough to resist brute force attempts at finding an ASLR’ed code segment. Later research on crash resistance and allocation oracles have undermined that assumption [Gawlik et al. 2016, Oikonomopoulos et al. 2016, Göktaş et al. 2016]. On a practical level, the ASLR-Guard implementation does not bound the growth of code locators and thus its memory overhead.

Chen et al. [2017] demonstrated support for execute-only memory for sourceless binaries. Specifically, their NORAX system is able to protect 64-bit ARM (AArch64) binaries. Notably, the AArch64 platform offers native support for execute-only memory, unlike current x86 CPUs. A general challenge of binary analysis and assembly is to accurately separate code and data. Code misclassified as data (data misclassified as code) can lead to page faults when using DEP (execute-only memory) to mitigate exploits. NORAX addresses this challenge using a combination of offline binary rewriting and online load/runtime monitoring. The offline step conservatively estimates code regions and moves data bytes embedded in these regions to a new data section. The original data bytes are overridden with unique magic numbers that are recognized by the NORAX loader and runtime monitor.

This lets the NORAX loader adjust any references to the original data bytes, which are now inaccessible since all code is mapped with execute-only permissions. If an attempt to read a code page happens at runtime, the NORAX runtime monitor determines whether the associated access violation was generated by a legitimate access (missed by the offline analysis) or whether it is a malicious access, which should cause program termination.

3.4 Address Oblivious Code Reuse

Rudd et al. [2017] explored the security properties of an ideal version of leakage-resilient code diversity, i.e., one that is not weakened by implementation-level flaws. Their finding was that even an ideal implementation does not stop all types of code reuse. The reason is that code-hiding mechanisms, such as execute-only memory, only apply to code pages, not code locators (e.g., function pointers and return addresses or pointers to Readactor trampolines). Code locators must be readable and writable for the program to function properly. Even with defenses such as Readactor and ASLR-Guard in place, adversaries can manipulate code locators used in place of traditional code pointers.

Rudd et al. used a data memory disclosure vulnerability to observe the state of a protected program as it executes. The fact that programs execute in a way that inherently leaks information about the state of execution enables profiling of the code indirection layer. Adversaries can correlate the execution state of their own unprotected program instance to that of a remote, protected instance at the time of the memory disclosure. Therefore, profiling can inform adversaries that a code identifier points to a function F in the protected program (without revealing the address of F). Adversaries can use this mapping from code identifiers to the underlying functions to construct a position-independent, whole-function code-reuse attack. Rudd et al. called this *Address-Oblivious Code Reuse* (AOCR) since the attack executes all code through code identifiers without any knowledge of the actual code layout.

Although AOCR attacks are possible, they require more effort to construct than their position-dependent equivalent. First of all, the state of the system changes rapidly, which makes it challenging to correctly time memory disclosures of code identifiers. If the target application is multi-threaded, however, memory corruption allows an adversary to manipulate the variables controlling entry to a critical section. Mutexes, for instance, are usually set by a thread as it enters the mutex such that other threads wanting to enter will suspend until the first thread has exited the critical section protected by the mutex. For instance, an adversary may

use one thread T_A to manipulate the mutex in a way that causes another thread T_B to block. This gives the adversary a chance to inspect memory without the timing unpredictability resulting from the execution of T_B .

Once the adversary has discovered a mapping from code locators to functions, he must find a way to (i) hijack the control flow, (ii) pass proper arguments to functions used in the exploit, and (iii) chain function calls. The control flow can be hijacked by using memory corruption to swap a code locator with the code locator corresponding to the first function in the malicious call chain. Rudd et al. solved the second challenge by reusing functions that read all their arguments from global variables. This requires knowledge of how global variables are laid out, but that too can be profiled and, in contrast to code, global variables must be readable. The third challenge, chaining calls through code locators, was solved using Malicious Loop Redirection (MLR). This technique requires the vulnerable application to contain a loop whose body contains an indirect call site. Specifically, the loop must (1) have a loop condition that is attacker controllable and (2) call functions through code pointers/locators. An ideal loop looks like this:

```
while (task) { task->fptr(task->arg); task = task->next; }
```

where `task` points to a linked list of (`fptr`, `arg`) pairs in attacker-controlled memory. Note that register randomization is not an effective defense because the semantics of the call dictates that the first argument is taken from `task->arg` and moved to `rdi` to conform to the `x86_64` ABI. Note that MLR is conceptually similar to the loop-gadget concept in COOP and Subversive-C code-reuse attacks [Lettner et al. 2016, Schuster et al. 2015].

Using these techniques, Rudd et al. demonstrated working AOCR attacks against two popular web servers protected by Readactor: Nginx and the Apache HTTP Server. Readactor served as a stand-in for leakage-resilient diversity techniques in general since it is the most comprehensive implementation of leakage-resilient diversity available. Note that approaches based on *destructive code reads* [Tang et al. 2015, Werner et al. 2016] are also vulnerable to AOCR since these attacks never attempt to read the actual code. Snow et al. demonstrated additional attacks specifically targeting destructive-code-read techniques [Snow et al. 2016].

3.5 Countering Address-Oblivious Code Reuse

Recall that code-pointer hiding via trampolines already limits the set of addresses that are reachable from an attacker-controlled indirect branch. Even if an attacker discloses all trampoline pointers, only function entries, return sites, and individual

instructions inside trampolines are exposed. We therefore implemented an extension to the Readactor code-pointer hiding mechanism, which we call Code-Pointer Authentication (CPA). CPA adds authentication after direct calls and before indirect calls to prevent the control-flow hijacking step as explained in Section 3.4 and thus mitigate AOCR attacks. One of the benefits of randomization-based defenses is that they do not rely on static program analysis, an advantage which helps them scale to complex code bases. To avoid relying on static program analysis, we must use different techniques to authenticate direct and indirect calls since we do not know the set of callees in advance.

3.5.1 Authenticating Direct Calls and Returns

Our general approach to authenticate direct calls uses cookies. A cookie is simply a randomly chosen value that is loaded into a register by the caller and read out and checked against an expected value by the callee. For returns, the callee loads another cookie into a register before returning, and the register is checked for the expected value directly after the return. Each function has two unique, random cookies: one to authenticate direct calls to the function (forward cookie, FC) and another to authenticate returns (return cookie, RC). Because the instructions that set and check cookies are stored in execute-only memory and the register storing the cookie is cleared directly after the check, attackers cannot leak or forge the cookies.

Our prototype implementation chooses cookie values at compile time and inserts these values into the execute-only code. A full-featured implementation could randomize the cookie values at load time so they vary between executions. This could easily be accomplished by marking all cookie locations during compilation, iterating over these locations during program initialization, and writing new cookies into the code before re-protecting the memory with execute-only permission.

The left side of Figure 3.2 shows how we authenticate an example direct function call from `foo` to `bar`. Dark gray labels indicate how we extend the Readactor code-pointer hiding technique with authentication cookies. Before transferring control to the direct call trampoline `t_bar` along control-flow edge ①, we load `bar`'s forward cookie into a scratch register. Edge ② transfers control from `t_bar` to `bar`. The prologue of `bar` checks that the register contents match the expected forward cookie value and clears the register to prevent spilling its contents to memory. Before the `bar` function returns along edge ③, we load the backward cookie for `bar` into the same scratch register. At the return site in `foo`, we check that the register contains the backward cookie identifying `bar` as the callee. The return site then clears the register.

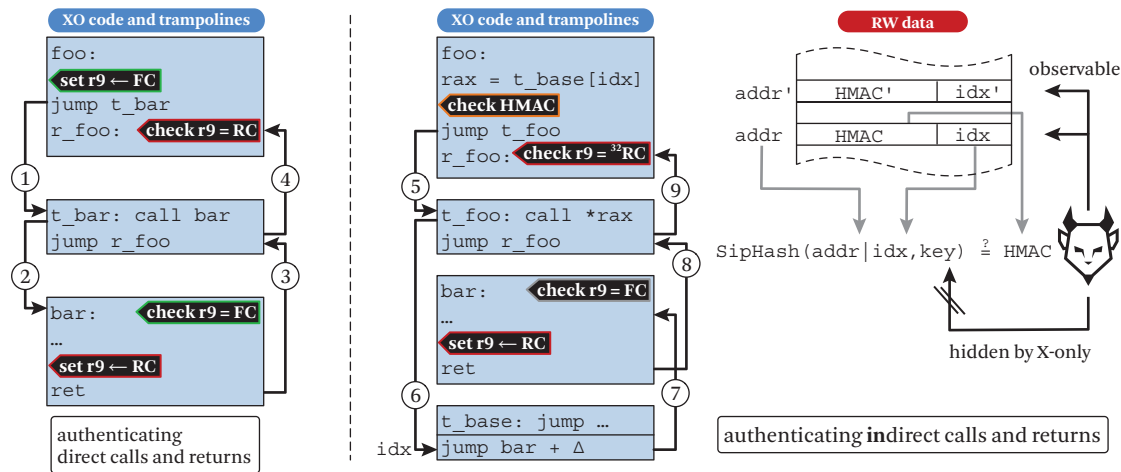


Figure 3.2 Code-pointer authentication. Direct calls and returns are illustrated in the leftmost third of the figure; indirect calls and returns are shown in the rightmost two-thirds. Light gray boxes contain execute-only code and white boxes contain data. Dark gray labels show where we insert additional instructions to prevent address harvesting attacks. The $\stackrel{?}{=}$ operator in the check after edge 9 indicates that we only check the lower 32 bits of the return cookie.

The return address pushed on the stack by the call instruction in `t_bar` leaks the location of the following jump instruction as well as the direct call itself. If the adversary manipulates an indirect branch to execute control-flow edge ②, the check at the target address will cause the forward cookie check to fail and thus the attack to fail. Analogously, redirecting control to flow along edge ④ will cause the check at `r_foo` to fail.

3.5.2 Securing Indirect Calls and Returns

Without static program analysis, we don't know the target of an indirect call at compile time and thus enforce bounds on the program control flow. Cookies, as used in the direct call case, are therefore not applicable to indirect calls. However, we can still authenticate that the function pointer used in an indirect call was correctly stored and not maliciously forged without requiring any static analysis.

All function pointers in a program protected by Readactor are actually pointers to trampolines that obscure the true target address. Inspired by the techniques of CCFI [Mashtizadeh et al. 2015], we change the representation of trampoline pointers (which are stored in attacker-observable memory) to allow for authentication.

In Readactor’s code-pointer hiding mechanism, a trampoline pointer is simply the address of the forward trampoline. With CPA, the trampoline pointer representation is composed of a 16-bit index (`idx`) into a table of trampolines (starting at `t_base`) and a 48-bit Hash-based Message Authentication Code (HMAC). We show two such pointers on the right side of Figure 3.2. Using a trampoline index prevents leakage of the forward trampoline pointer address since the base address of the array of forward trampolines `t_base` can be hidden in execute-only code. We found that programs need less than 2^{16} forward pointers in practice, so it suffices to use the lower 16 bits of a 64-bit word for the index (this can be adjusted as needed for larger applications). We compute the HMAC by hashing the index along with the least significant 48 bits of its virtual memory address. With this HMAC we can detect if the adversary tries to replace a code pointer with another pointer harvested from a different memory location. We find that SipHash [Aumasson and Bernstein 2012], which is optimized for short messages, is a good choice of HMAC for our approach.

The middle third of Figure 3.2 illustrates the case where the function `foo` calls `bar` indirectly through a function pointer. Again, dark gray labels highlight our extensions to Readactor’s code pointer hiding technique. The indirect call site in `foo` loads the (HMAC, index) pair from memory, recomputes the HMAC using the (address, index, key) tuple, and compares the two (see rightmost third of Figure 3.2). If HMACs match, the index is used to lookup the address of the forward pointer which is subsequently used to execute control-flow edge ⑥. Note that the forward trampoline that creates edge ⑦ does not target the first instruction in `bar`; instead, we add a delta to the address of `bar` to skip the forward cookie check that authenticates direct calls to `bar` (e.g., edge ②).

As explained in Section 3.4, AOCR attacks swap two pointers to hijack the program control flow. Because the address of the pointer is used to compute the HMAC, moving the pointer without re-computing the HMAC will cause the HMAC check before all indirect calls to fail unless the two (address, index) pairs collide in the hash. Attackers can still harvest and swap (HMAC, index) pairs stored to the same address at different times. See Section 3.6.1 for a more complete security analysis.

Returns from indirect calls make up the fourth and final class of control flows that we must authenticate. The callee sets a return cookie before the callee returns and checks the cookie at the return site; see edges ⑧ and ⑨ in Figure 3.2. We again clear the cookie register directly after the check to prevent leaks. The cookie check at the end of arrow ⑨ must pass for all potential callees. Therefore, we set the lower 32 bits of all backward cookies to the same global random value and only

check the lower halfword of the backward cookie at the return site. This ensures that returns only target return sites; however, any return instruction can target indirect-call-preceded gadgets under this scheme. We did not reuse any indirect call-preceded gadgets in our harvesting attack since these are also protected by register randomization and callee-saved stack slot randomization. It is possible to further restrict returns from indirect calls by taking function types into account. Rather than setting the lower 32 bits of return cookies to the same random value, we can use different random values for different types of functions.

3.6 Evaluation of Code-Pointer Authentication

3.6.1 Security

Code-pointer authentication prevents reuse of the remaining exposed trampoline pointers, even if the attacker has harvested all available trampoline locations. This authentication mitigates AOCR attacks. To show how, we systematically consider each possibly exposed indirect branch target in turn.

Direct call trampoline entry (edge ① in Figure 3.2). An attacker can harvest the location of the backward jump (`jump r_foo`) in the call trampoline from the return address on the stack. In the original Readactor defense, it is possible to compute the address of the previous instruction from this pointer and invoke `t_bar`.

With direct call authentication, each direct callee function checks that its specific, per-function cookie is set prior to calling it. If the attacker cannot forge the callee function’s cookie, this check will fail. We store the cookie as an immediate value in execute-only memory and pass it to the callee in a register. After performing the cookie check, the callee clears the register. Thus, direct call cookies cannot leak to an adversary, and the attacker has a 2^{-64} chance of successfully guessing the correct 64-bit random cookie value. Since the attacker cannot forge a correct cookie before an indirect branch to a direct call cookie, direct call trampoline entry points are unavailable as destinations for an attack.

Direct call trampoline return (edge ③ in Figure 3.2). Harvesting a return address corresponding to a direct call trampoline gives the attacker the location of the backward jump in a call trampoline. In Readactor, this destination allows the attacker to invoke a call-preceded gadget beginning at `r_foo` in the example.

We also protect these destinations with an analogous, function-specific return cookie. Directly before a callee function returns, it sets its function-

specific return cookie. The return site verifies that the expected callee’s return cookie was set before continuing execution. This prevents the attacker from reusing this destination unless the control-flow edge would be allowed during normal program execution.

Indirect call trampoline entry (edge ⑤ in Figure 3.2). Similarly, an attacker can harvest indirect call trampoline locations from the stack and dispatch to the beginning of an indirect call trampoline. However, this destination is trivial to attackers, since they must set another valid, useful destination for the indirect call before invoking the trampoline. The attack could always dispatch straight to this final destination instead of to the indirect call trampoline. Thus, we do not need to protect indirect call trampoline entry points from reuse.

Indirect call trampoline return (edge ⑧ in Figure 3.2). Analogous to the direct call case, the attacker can dispatch to the backward edge of an indirect call trampoline to invoke an indirect-call-preceded gadget. This is a more challenging edge to protect without static analysis, since the indirect call site cannot know which function-specific return cookie to check.

Since the caller does not know the precise callee, we enforce a weaker authentication check on indirect call return destinations. By splitting return cookies into a global part and function-specific part, we can still ensure that the return site must be invoked by a return, not an indirect call. We believe that the fine-grained register randomization implemented in Readactor largely mitigates the threat of indirect-call-preceded gadget reuse, since the attacker cannot be sure of the semantics of the gadget due to execute-only memory.

Function trampolines (edge ⑥ in Figure 3.2). Function trampoline harvesting and reuse is the easiest attack vector against code-pointer hiding schemes. In Readactor, after harvesting function trampolines, the attacker can overwrite any return address or function pointer with a valid function trampoline destination and perform whole-function reuse.

We prevent reuse of function trampolines by changing the function-pointer format to include an HMAC tying the function pointer to a specific memory address. This prevents reuse of function pointers from returns as well as most swaps of function pointers in memory.

Since function pointers are no longer memory addresses in our authentication scheme, the attacker cannot use a function pointer as a return address

at all. The return would interpret the address as an HMAC-Idx pair and fail to verify the HMAC, crashing the program.

Function pointers cannot be swapped arbitrarily under this defense, since the pointer is tied to its address in memory by the HMAC. If a pointer P at address A is moved to address B , the HMAC check will fail when loaded from address B . Thus the attacker must either forge a valid HMAC or have harvested P from the targeted location in memory at a previous point in execution.

HMAC Forgery. We first address the possibility of forging a valid HMAC for a function and pointer address pair without ever having seen a valid HMAC for that pair. SipHash is designed to be forgery resistant, thus the probability of correctly forging a valid HMAC for a pointer at an address not previously HMACed is expected to be 2^{-48} , based on the size of the HMAC tag. Additionally, since we can store the HMAC key in execute-only memory, an attacker cannot disclose the 128-bit key and thus is limited to brute-forcing this key.

Replay Attacks. As in other pointer encryption schemes [Mashtizadeh et al. 2015, Cowan et al. 2003], HMACs do not provide temporal safety against replay attacks on function pointers. That is, a function pointer can be harvested at one point in program execution and later rewritten to the same address.

3.6.2 Performance

To evaluate the performance of our code-pointer authentication, we applied the protections on top of the Readactor++ system. We measured the performance overhead of both direct call authentication and function-pointer authentication on the SPEC CPU2006 benchmark suite. These results are summarized in Figure 3.3. All benchmarks were measured on a system with two Intel Xeon E5-2660 processors clocked at 2 Ghz running Ubuntu 14.04.

With all protections enabled, we measured a geometric mean performance overhead of 9.7%. This overhead includes the overhead from basic Readactor call and jump trampolines and compares favorably with the 6.4% average overhead reported by Crane et al. [2015]. We also measured the impact of direct call authentication and indirect call authentication individually (labeled DCA and ICA in the figure, respectively). We found that indirect code-pointer authentication generally adds more overhead (6.7% average) than direct code-pointer authentication (5.9% average), although this is strongly influenced by the program workload, specifically the percentage of calls using function pointers.

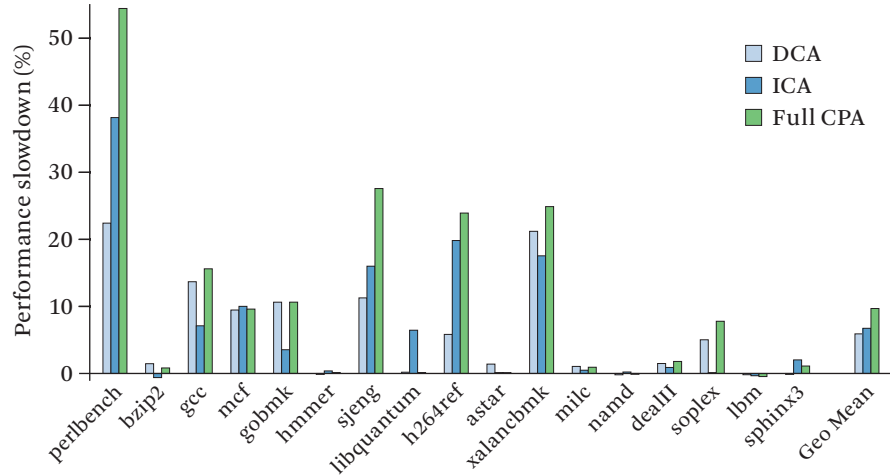


Figure 3.3 Performance overhead of code-pointer authentication on SPEC CPU2006. All measurements include the overhead of the Readactor++ transformations [Crane et al. 2015].

We observed that `h264ref` stands out as an interesting outlier for indirect call authentication. This benchmark repeatedly makes a call through a function pointer in a hot loop. To make matters worse, the target function is a one-line getter, thus our instrumentation dominates the time spent in the callee. This benchmark in particular benefits greatly from inlining the HMAC verification to avoid the extra call overhead. To speed up HMAC verification, especially in this edge case, we implemented a small (128 byte), direct-mapped, hidden cache of valid HMAC entries. This hidden cache is only accessed via offsets embedded in execute-only memory and is thus tamper resistant. Before recomputing an HMAC, the verification routine checks the cache to see if the HMAC is present.

We found three corner cases in SPEC where we could not automatically compute a new HMAC when a function pointer was moved. This is because the program first casts away the function-pointer type and then copies the pointer inside a struct. We had to insert a single manual HMAC in `gcc` and another in `povray` to handle these edge cases. `perlbench` stores function pointers in a growable list, which is moved during reallocation. Since our prototype does not yet instrument the `libc realloc` function, we had to manually instrument these operations. The CCFI [Mashtizadeh et al. 2015] HMAC scheme requires similar modifications. Finally, Readactor is not fully compatible with C++ exception handling, so we were not able to run `omnetpp` and `povray`, which require exception handling.

3.7 Conclusion

There are three ways to bypass diversity-type mitigations. The first is to target unprotected areas, the second is to employ brute force guessing, and the third relies on information leakage. The first two ways are relatively straightforward to counter through good engineering. The third option, however, remains the most challenging to fully address. Although it is possible to prevent leakage (perhaps modulo hardware side channels) of the code layout, address-oblivious attacks, though technically complex, are feasible. It is possible to mitigate address-oblivious code reuse, too, although the solution we designed and evaluated adds overhead and complexity to what was initially a fairly simple defense strategy.

If history is any guide, retrofitting security into fundamentally insecure languages without hampering performance will remain an open research challenge in the foreseeable future. The specific strand of research presented here is not the “one true answer” to all security problems; just as is the case with mitigation alternatives, such as CFI and CPI. Instead, we describe our broader expectations for the short, medium, and long term based on recent industry developments:

- In the *short* term, deploying better mitigations is the best option. This is not a particular insight of ours; one simply has to look at the direction in which major software developers are headed. At the time of writing, work is under way to improve the granularity of code randomization schemes, and hardware support for execute-only memory is forthcoming for Intel and already available for ARM. Although deployment of leakage-resilient diversity, as enabled by these techniques, is unlikely to stop all exploits, it does considerably raise the bar on attackers. At the same time, control-flow integrity techniques are supported by all major compilers, and hardware support is similarly forthcoming from both Intel and ARM. Diversity and CFI are not mutually exclusive techniques, and either will stop a sufficiently determined adversary on its own. Rather, we believe a combination of disparate exploit mitigations will offer the best return on investment.
- Unlike the short-term options, *medium*-term options will require some source code changes. Access control mechanisms, such as SELinux, when correctly implemented, help implement the principle of least privilege such that vulnerabilities in unprivileged code cannot be used to carry out privileged operations. Legacy applications are unlikely to be broken into independent submodules based on the privileges they require, however. Therefore, manual refactoring may be required to realize the full potential of access control mechanisms. Similarly, techniques that retrofit type and memory

safety into legacy C/C++ code require that bad casts and invalid memory accesses are removed from the application before a protected version can be released.

- Whereas medium-term options may require minor changes and fixes to existing source code, the best *long*-term option is likely to very gradually retire C/C++ code. This will take multiple decades, and some code bases may simply be abandoned as the software landscape changes anyway. The reason we mention language mitigation, however long it may take, is that it brings with it several important *secondary* benefits. Reduction of technical debt and the resulting productivity benefits are chief among these. C and its derivatives reflect the age in which they were designed. For instance, C programmers must declare variables and functions defined outside the current translation unit such that the compiler can emit code in a single pass over the input files. Modern programming languages reflect the current reality that computing cycles are cheap and programmer attention scarce. Moreover, [Balasubramanian et al. \[2017\]](#) show that the features of the Rust systems programming language can support security capabilities, such as zero-copy software fault isolation, that cannot be implemented efficiently in traditional languages. Only by abandoning the languages in the C family, which have been spectacularly successful at any rate, can we make systems programming more productive, safe, and accessible.

Acknowledgments

This material is based upon work partially supported by the Department of Defense under Defense Advanced Research Projects Agency (DARPA) contract FA8750-15-C-0124, Air Force contracts FA8721-05-C-0002 and FA8702-15-D-0001, and by the National Science Foundation under awards CNS-1513837 and CNS-1619211.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the Air Force, the National Science Foundation, or any other agency of the U.S. Government.

Code-Pointer Integrity

Volodymyr Kuznetsov, László Szekeres, Mathias Payer,
George Candea, R. Sekar, Dawn Song

In this chapter, we describe code-pointer integrity (CPI), a new design point that *guarantees the integrity of all code pointers* in a program (e.g., function pointers, saved return addresses) and thereby prevents all control-flow hijack attacks that exploit memory corruption errors, including attacks that bypass control-flow integrity mechanisms, such as control-flow bending [Carlini et al. 2015e]. We also describe code-pointer separation (CPS), a relaxation of CPI with better performance properties. CPI and CPS offer substantially better security-to-overhead ratios than the state of the art, and they are practical (CPI and CPS were used to protect a complete FreeBSD system and over 100 packages like `apache` and `postgresql`), effective (prevented all attacks in the RIPE benchmark), and efficient: on SPEC CPU2006, CPS averages 1.2% overhead for C and 1.9% for C/C++, while CPI's overhead is 2.9% for C and 8.4% for C/C++.

This chapter is organized as follows: we introduce the motivation and key ideas behind CPI and CPS (Section 4.1), describe related work (Section 4.2), introduce our threat model (Section 4.3), describe CPI and CPS design (Section 4.4), present the formal model of CPI (Section 4.5), describe an implementation of CPI (Section 4.6) and the experimental results (Section 4.7), and then conclude (Section 4.8).

4.1 Introduction

System code is often written in memory-unsafe languages. This makes it prone to memory errors that are the primary attack vector to subvert systems. Attackers exploit bugs, such as buffer overflows and use-after-free errors, to cause memory corruption that enables them to steal sensitive data or execute code that gives them control over a remote system [Wojtczuk 1998, Nergal 2001, Checkoway et al. 2010, Bletsch et al. 2011].

The goal of CPI is to secure system code against all *control-flow hijack* attacks, which is how attackers gain remote control of victim systems. Low-level languages like C/C++ offer many benefits to system programmers, and CPI makes these languages safe to use while preserving their benefits, not the least of which is performance. Before expecting any security guarantees from systems, we must first secure their building blocks.

There exist a few protection mechanisms that can reduce the risk of control-flow hijack attacks without imposing undue overhead. Data Execution Prevention (DEP) and $W \oplus X$ [van de Ven 2004] use memory page protection to prevent the introduction of new executable code into a running application. Unfortunately, DEP is defeated by code-reuse attacks, such as return-to-libc [Nergal 2001] and return-oriented programming (ROP) [Wojtczuk 1998, Bletsch et al. 2011], which can construct arbitrary Turing-complete computations by chaining together existing code fragments of the original application. Address Space Layout Randomization (ASLR) [PaX Team 2004a] places code and data segments at random addresses, making it harder for attackers to reuse existing code for execution. Alas, ASLR is defeated by pointer leaks, side-channel attacks [Hund et al. 2013], and just-in-time code-reuse attacks [Snow et al. 2013]. Finally, stack cookies [Cowan et al. 1998] protect return addresses on the stack but only against continuous buffer overflows.

Many defenses can improve upon these shortcomings but have not seen wide adoption because of the overheads they impose. According to a recent survey [Szekeres et al. 2013], these solutions are incomplete and bypassable via sophisticated attacks and/or require source code modifications and/or incur high performance overhead. These approaches typically employ language modifications [Jim et al. 2002, Necula et al. 2005], compiler modifications [Cowan et al. 2003, Akritidis et al. 2008, Dhurjati et al. 2006, Nagarakatte et al. 2009, Serebryany et al. 2012], or rewrite machine code binaries [Niu and Tan 2013, Zhang and Sekar 2013, Zhang et al. 2013].

Control-flow integrity (CFI) protection [Abadi et al. 2005a, Burow et al. 2016, Li et al. 2011, Zhang et al. 2013, Zhang and Sekar 2013, Niu and Tan 2014a], a widely studied technique for practical protection against control-flow hijack attacks, was recently demonstrated to be ineffective [Carlini et al. 2015e, Evans et al. 2015, Göktaş et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014] against attacks that are adjusted to the constraints of the underlying defense.

Existing techniques cannot both *guarantee protection* against control-flow hijacks and impose *low overhead* and *no changes* to how the programmer writes code. For example, memory-safe languages guarantee that a memory object can only be

accessed using pointers properly based on that specific object, which in turn makes control-flow hijacks impossible. But this approach requires runtime checks to verify the temporal and spatial correctness of pointer computations, which inevitably induces undue overhead, especially when retrofitted to memory-unsafe languages. For example, state-of-the-art memory safety implementations for C/C++ incur $\geq 2\times$ overhead [Nagarakatte et al. 2010]. We observe that, in order to render control-flow hijacks impossible, it is sufficient to guarantee the integrity of code pointers, i.e., those that are used to determine the targets of indirect control-flow transfers (indirect calls, indirect jumps, or returns).

This chapter describes code-pointer integrity (CPI), a way to enforce precise, deterministic memory safety for all code pointers in a program. The key idea is to split process memory into a *safe region* and a *regular region*. CPI uses static analysis to identify the set of memory objects that must be protected in order to guarantee memory safety for code pointers. This set includes all memory objects that contain code pointers and all data pointers used to access code pointers indirectly. All objects in the set are then stored in the safe region, and the region is isolated from the rest of the address space (e.g., via hardware protection). The safe region can only be accessed via memory operations that are proven at compile time to be safe or that are safety-checked at runtime. The regular region is just like normal process memory: it can be accessed without runtime checks and, thus, with no overhead. In typical programs, the accesses to the safe region represent only a small fraction of all memory accesses (6.5% of all pointer operations in SPEC CPU2006 need protection). Existing memory safety techniques cannot efficiently protect only a subset of memory objects in a program; rather, they require instrumenting *all* potentially dangerous pointer operations.

CPI fully protects the program against all control-flow hijack attacks that exploit program memory bugs. CPI requires no changes to how programmers write code since it automatically instruments pointer accesses at compile time. CPI achieves low overhead by selectively instrumenting only those pointer accesses that are necessary and sufficient to formally guarantee the integrity of all code pointers. The CPI approach can also be used for data, e.g., to selectively protect sensitive information like the process UIDs in a kernel.

We also describe code-pointer separation (CPS), a relaxed variant of CPI that is better suited for code with abundant virtual function pointers. In CPS, all code pointers are placed in the safe region, but pointers used to access code pointers indirectly are left in the regular region (such as pointers to C++ objects that contain virtual functions). Unlike CPI, CPS may allow certain control-flow hijack attacks, but it still offers strong guarantees and incurs negligible overhead.

Finally, we describe SafeStack, a component of both CPI and CPS that protects code pointers on the stack. SafeStack is integrated into the Clang compiler starting with version 3.7.0.

The experimental evaluation of CPI and CPS shows that these techniques impose sufficiently low overhead to be deployable in production. For example, CPS incurs an average overhead of 1.2% on the C programs in SPEC CPU2006 and 1.9% for all C/C++ programs. CPI incurs on average 2.9% overhead for the C programs and 8.4% across all C/C++ SPEC CPU2006 programs. CPI and CPS are effective: they prevented 100% of the attacks in the RIPE benchmark and the recent attacks [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014] that bypass CFI, ASLR, DEP, and all other Microsoft Windows protections. We compile and run with CPI/CPS a complete FreeBSD distribution along with over 100 widely used packages, demonstrating that the approach is practical.

4.2 Related Work

A variety of defense mechanisms have been proposed to date to answer the increasing challenge of control-flow hijack attacks, some of them described in Chapter 1. Figure 4.1 compares the design of the different protection approaches to our approach.

Enforcing memory safety ensures that no dangling or out-of-bounds pointers can be read or written by the application, thus preventing the attack in its first step. Cyclone [Jim et al. 2002] and CCured [Necula et al. 2005] extend C with a safe type system to enforce memory safety features. These approaches face the problem that there is a large (unported) legacy code base. In contrast, CPI and CPS both work for unmodified C/C++ code. SoftBound [Nagarakatte et al. 2009] with its CETS [Nagarakatte et al. 2010] extension enforces *complete* memory safety at the cost of 2–4× slowdown. Tools with less overhead, like BBC [Akritidis et al. 2009], only *approximate* memory safety. LBC [Hasabnis et al. 2012] and Address Sanitizer [Serebryany et al. 2012] detect continuous buffer overflows and (probabilistically) indexing errors, but can be bypassed by an attacker who avoids the red zones placed around objects. Write Integrity Testing (WIT) [Akritidis et al. 2008] provides spatial memory safety by restricting pointer writes according to points-to sets obtained by an over-approximate static analysis (and is therefore limited by the static analysis). Other techniques [Dhurjati et al. 2006, Akritidis 2010] enforce type-safe memory reuse to mitigate attacks that exploit temporal errors (use-after-frees).

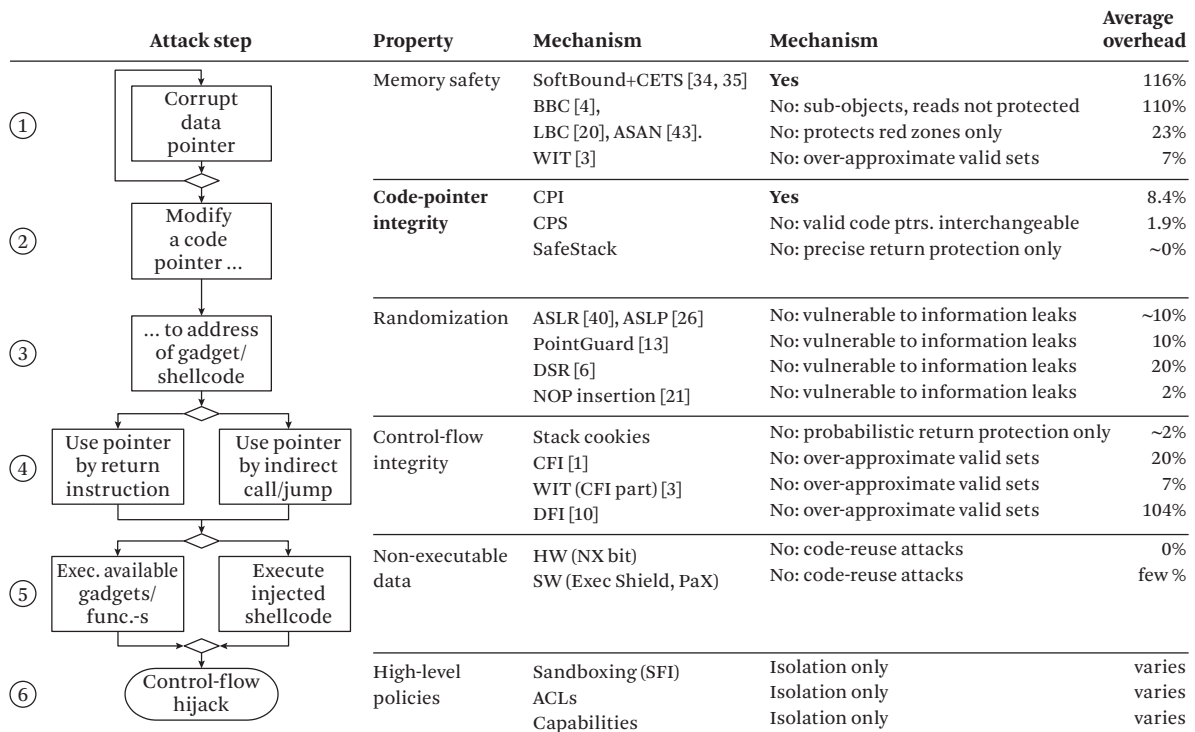


Figure 4.1 Summary of control-flow hijack defense mechanisms aligned with individual steps that are necessary for a successful attack. The diagram on the left is a simplified version of the complete memory corruption diagram in [Szekeres et al. 2013].

CPI by design enforces spatial and temporal memory safety for a subset of data (code pointers) in step 2 of Figure 4.1. Our Levee prototype currently enforces spatial memory safety and may be extended to enforce temporal memory safety as well (e.g., how CETS extends SoftBound). We believe CPI is the first to stop all control-flow hijack attacks at this step.

Randomization-based confidentiality techniques, like ASLR [PaX Team 2004a] and ASLP [Kil et al. 2006], mitigate attacks by restricting the attacker’s knowledge of the memory layout of the application in step 3. PointGuard [Cowan et al. 2003] and DSR [Bhatkar and Sekar 2008] (which is similar to probabilistic WIT) randomize the data representation by encrypting pointer values but face compatibility problems. Software diversity [Homescu et al. 2013] allows fine-grained, per-instance code randomization. Randomization techniques are defeated by information leaks through,

e.g., memory corruption bugs [Snow et al. 2013] or side-channel attacks [Hund et al. 2013].

Control-flow integrity [Abadi et al. 2005a] ensures that the targets of all indirect control-flow transfers point to valid code locations in step 4. All CFI solutions rely on statically pre-computed context-insensitive sets of valid control-flow target locations. Many practical CFI solutions simply include every function in a program in the set of valid targets [Zhang et al. 2013, Zhang and Sekar 2013, Li et al. 2011, Tice et al. 2014]. Even if precise static analysis would be feasible, CFI could not guarantee protection against all control-flow hijack attacks, but rather merely restrict the sets of potential hijack targets. Indeed, recent results [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014] show that many existing CFI solutions can be bypassed in a principled way. CFI+SFI [Zeng et al. 2011], Strato [Zeng et al. 2013a], and MIPS [Niu and Tan 2013] enforce an even more relaxed, statically defined CFI property in order to enforce software-based fault isolation. CCFI [Mashtizadeh et al. 2014] encrypts code pointers in memory and provides security guarantees close to CPS. Data-flow-based techniques, like Data-Flow Integrity (DFI) [Castro et al. 2006] or Dynamic Taint Analysis (DTA) [Schwartz et al. 2010], can enforce that the used code pointer was not set by an unrelated instruction or to untrusted data, respectively. These techniques may miss some attacks or cause false positives, and have higher performance costs than CPI and CPS. Stack cookies, CFI, DFI, and DTA protect control-transfer instructions by detecting illegal modification of the code pointer whenever it is used, while CPI protects the load and store of a code pointer, thus preventing the corruption in the first place. CPI provides precise and provable security guarantees.

In step 5, the execution of injected code is prevented by enforcing the non-executable (NX) data policy, but code-reuse attacks remain possible.

High-level policies, e.g., restricting the allowed system calls of an application, limit the power of the attacker even in the presence of a successful control-flow hijack attack in step 6. Software Fault Isolation (SFI) techniques [McCamant and Morrisett 2006, Erlingsson et al. 2006, Castro et al. 2009, Yee et al. 2009, Zeng et al. 2011] restrict indirect control-flow transfers and memory accesses to part of the address space, enforcing a sandbox that contains the attack. SFI prevents an attack from escaping the sandbox and allows the enforcement of a high-level policy, while CPI enforces the control flow inside the application.

CPI and CPS rely on an instruction-level isolation mechanism to enforce the separation of code pointers from the rest of the data in program memory (Section 4.4). This book describes multiple implementations of such mechanisms (Section 4.6.3), including implementations that provide precise isolation guarantees (based on

hardware- or software-enforced isolation) as well as probabilistic guarantees (based on randomization and information hiding). Evans et al. [2015a] demonstrated that one of the implementations with probabilistic guarantees can be bypassed in practical settings. Their attack cannot subvert the other implementation alternatives presented in this book (see Section 4.6.3 and [Kuznetsov et al. 2015]).

4.3 Threat Model

This chapter is concerned solely with control-flow hijack attacks, namely, ones that give the attacker control of the instruction pointer. The purpose of this type of attack is to divert control flow to a location that would not otherwise be reachable in that same context, had the program not been compromised. Examples of such attacks include forcing a program to jump (i) to a location where the attacker injected shellcode, (ii) to the start of a chain of return-oriented program fragments (“gadgets”), or (iii) to a function that performs an undesirable action in the given context, such as calling `system()` with attacker-supplied arguments. Data-only attacks, i.e., those that modify or leak unprotected non-control data, are outside the scope of our discussion.

We assume powerful yet realistic attacker capabilities: full control over process memory but no ability to modify the code segment. Attackers can carry out arbitrary memory reads and writes by exploiting input-controlled memory corruption errors in the program. They cannot modify the code segment because the corresponding pages are marked read-executable and not writable, and they cannot control the program-loading process. These assumptions ensure the integrity of the original program code instrumented at compile time, and enable the program loader to safely set up the isolation between the safe and regular memory regions. Our assumptions are consistent with prior work in this area.

4.4 Design

We now present the terminology used to describe our design, then define the code-pointer integrity property (Section 4.4.1), describe the corresponding enforcement mechanism (Section 4.4.2), and define a relaxed version that trades some security guarantees for performance (Section 4.4.4). We further formalize the CPI enforcement mechanism and sketch its correctness proof (Section 4.5).

We say a pointer dereference is *safe* iff the memory it accesses lies within the target object on which the dereferenced pointer is based. A *target object* can either be a memory object or a control-flow destination. By *pointer dereference* we mean accessing the memory targeted by the pointer, either to read/write it (for data

pointers) or to transfer control flow to its location (for code pointers). A *memory object* is a language-specific unit of memory allocation, such as a global or local variable, a dynamically allocated memory block, or a sub-object of a larger memory object (e.g., a field in a struct). Memory objects can also be program specific, e.g., when using custom memory allocators. A *control-flow destination* is a location in the code, such as the start of a function or a return location. A target object always has a well-defined lifetime; for example, freeing an array and allocating a new one with the same address creates a different object.

We say a pointer is *based on* a target object X iff the pointer is obtained at runtime by (i) allocating X on the heap; (ii) explicitly taking the address of X if X is allocated statically, such as a local or global variable, or is a control-flow target (including return locations, whose addresses are implicitly taken and stored on the stack when calling a function); (iii) taking the address of a sub-object y of X (e.g., a field in the X struct); or (iv) computing a pointer expression (e.g., pointer arithmetic, array indexing, or simply copying a pointer) involving operands that either are themselves based on object X or are not pointers. This is a slightly stricter version of C99’s “based-on” definition: we ensure that each pointer is based on at most one object.

The execution of a program is *memory safe* iff all pointer dereferences in the execution are safe. A program is memory safe iff all its possible executions (for all inputs) are memory safe. This definition is consistent with the state of the art for C/C++, such as SoftBounds+CETS [Nagarakatte et al. 2009, Nagarakatte et al. 2010]. Precise memory safety enforcement [Nagarakatte et al. 2009, Necula et al. 2005, Jim et al. 2002] tracks the based-on information for each pointer in a program to check the safety of each pointer dereference according to the definition above; the detection of an unsafe dereference aborts the program.

4.4.1 The Code-Pointer Integrity (CPI) Property

A program execution satisfies the code-pointer integrity property iff all its dereferences that either dereference or access sensitive pointers are safe. *Sensitive pointers* are code pointers and pointers that may later be used to access sensitive pointers. Note that the sensitive pointer definition is recursive, as illustrated in Figure 4.2. According to case (iv) of the based-on definition above, dereferencing a pointer to a pointer will correspondingly propagate the based-on information; e.g., an expression $*p = \&q$ copies the result of $\&q$, which is a pointer based on q , to a location pointed to by p , and associates the based-on metadata with that location. Hence, the integrity of the based-on metadata associated with sensitive pointers requires that pointers used to update sensitive pointers be sensitive as well (we discuss implications of relaxing this definition in Section 4.4.4). The notion of a sensitive pointer

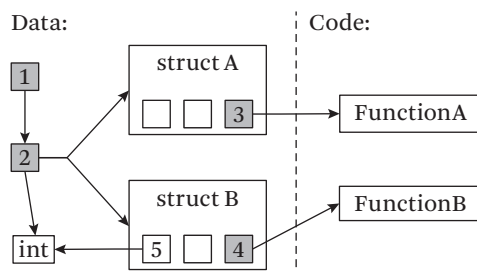


Figure 4.2 CPI protects code pointers 3 and 4 and pointers 1 and 2 (which may access pointers 3 and 4 indirectly). Pointer 2 of type `void*` may point to different objects at different times. The `int*` pointer 5 and non-pointer data locations are not protected.

is dynamic. For example, a `void*` pointer 2 in Figure 4.2 is sensitive when it points at another sensitive pointer at runtime, but it is not sensitive when it points to an integer.

A memory-safe program execution trivially satisfies the CPI property, but memory safety instrumentation typically has high runtime overhead, e.g., $\geq 2\times$ in state-of-the-art implementations [Nagarakatte et al. 2010]. Our observation is that only a small subset of all pointers is responsible for making control-flow transfers, and so, by enforcing memory safety only for control-sensitive data (and thus incurring no overhead for all other data), we obtain important security guarantees while keeping the cost of enforcement low. This is analogous to the control-plane/data-plane separation in network routers and modern servers [Altekar and Stoica 2010], with CPI ensuring the safety of data that influences, directly or indirectly, the control plane.

Determining precisely the set of pointers that are sensitive can only be done at runtime. However, the CPI property can still be enforced using any over-approximation of this set, and such over-approximations can be obtained at compile time, using static analysis.

4.4.2 The CPI Enforcement Mechanism

We now describe a way to retrofit the CPI property into a program P using a combination of static instrumentation and runtime support. Our approach consists of a *static analysis* pass that identifies all sensitive pointers in P and all instructions that operate on them (Section 4.4.2.1), an *instrumentation* pass that rewrites P to “protect” all sensitive pointers, i.e., store them in a separate, safe memory region and associate, propagate, and check their based-on metadata (Section 4.4.2.2), and

an instruction-level *isolation* mechanism that prevents non-protected memory operations from accessing the safe region (Section 4.4.2.3). For performance reasons, we handle return addresses stored on the stack separately from the rest of the code pointers using a *safe stack* mechanism (Section 4.4.3).

4.4.2.1 CPI Static Analysis

We determine the set of sensitive pointers using type-based static analysis: a pointer is sensitive if its type is sensitive. Sensitive types are pointers to functions, pointers to sensitive types, pointers to composite types (such as `struct` or `array`) that contain one or more members of sensitive types, or universal pointers (i.e., `void*`, `char*`, and opaque pointers to forward-declared `structs` or `classes`). A programmer could additionally indicate, if desired, other types to be considered sensitive, such as `struct ucred` used in the FreeBSD kernel to store process UIDs and jail information. All code pointers that a compiler or runtime creates implicitly (such as return addresses, C++ virtual table pointers, and `setjmp` buffers) are sensitive as well.

Once the set of sensitive pointers is determined, we use static analysis to find all program instructions that manipulate these pointers. These instructions include pointer dereferences, pointer arithmetic, and memory (de-)allocation operations that call either (i) corresponding standard library functions, (ii) C++ `new/delete` operators, or (iii) manually annotated custom allocators.

The derived set of sensitive pointers is over-approximate: it may include universal pointers that never end up pointing to sensitive values at runtime. For instance, the C/C++ standard allows `char*` pointers to point to objects of any type, but such pointers are also used for C strings. As a heuristic, we assume that `char*` pointers that are passed to the standard `libc` string manipulation functions or that are assigned to point to string constants are not universal. Neither the over-approximation nor the `char*` heuristic affect the security guarantees provided by CPI: over-approximation merely introduces extra overhead, while heuristic errors may result in false violation reports (though we never observed any in practice).

Memory manipulation functions from `libc`, such as `memset` or `memcpy`, could introduce a lot of overhead in CPI: they take `void*` arguments, so a `libc` compiled with CPI would instrument all accesses inside the functions, regardless of whether they are operating on sensitive data or not. CPI's static analysis instead detects such cases by analyzing the real types of the arguments prior to being cast to `void*`, and the subsequent instrumentation pass handles them separately using type-specific versions of the corresponding memory manipulation functions.

We augmented type-based static analysis with a data-flow analysis that handles most practical cases of unsafe pointer casts and casts between pointers and integers. If a value v is ever cast to a sensitive pointer type within the function being analyzed, or is passed as an argument or returned to another function where it is cast to a sensitive pointer, the analysis considers v to be sensitive as well. This analysis may fail when the data flow between v and its cast to a sensitive pointer type cannot be fully recovered statically, which might cause false violation reports (we have not observed any during our evaluation). Such casts are a common problem for all pointer-based memory safety mechanisms for C/C++ that do not require source code modifications [Nagarakatte et al. 2009].

A key benefit of CPI is its selectivity: the number of pointer operations deemed to be sensitive is a small fraction of all pointer operations in a program. As we show in Section 4.7, for SPEC CPU2006, the CPI type-based analysis identifies for instrumentation 6.5% of all pointer accesses; this translates into a reduction of performance overhead of 16–44× relative to full memory safety.

Nevertheless, we still think CPI can benefit from more sophisticated analyses. CPI can leverage any kind of *points-to* static analysis, as long as it provides an over-approximate set of sensitive pointers. For instance, when extending CPI to also protect select non-code-pointer data, we think DSA [Lattner and Adve 2005, Lattner et al. 2007] could prove more effective.

4.4.2.2 CPI Instrumentation

CPI instruments a program in order to (i) ensure that all sensitive pointers are stored in a safe region, (ii) create and propagate metadata for such pointers at runtime, and (iii) check the metadata on dereferences of such pointers.

In terms of memory layout, CPI introduces a safe region in addition to the regular memory region (Figure 4.3). Storage space for sensitive pointers is allocated in both the safe region (the *safe pointer store*) and the regular region (as usual); one of the two copies always remains unused. This is necessary for universal pointers (e.g., `void*`), which could be stored in either region depending on whether they are sensitive at runtime or not, and also helps to avoid some compatibility issues that arise from the change in memory layout. The address in regular memory is used as an offset to look up the value of a sensitive pointer in the safe pointer store.

The *safe pointer store* maps the address $\&p$ of sensitive pointer p , as allocated in the regular region, to the value of p and associated metadata. The metadata for p describes the target object on which p is based: lower and upper address bounds of the object and a temporal id (see Figure 4.3). The layout of the safe pointer store is similar to metadata storage in SoftBounds+CETS [Nagarakatte et al. 2010],

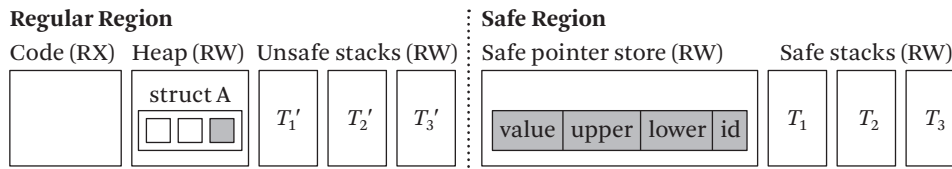


Figure 4.3 CPI memory layout: The safe region contains the safe pointer store and the safe stacks. The location of a sensitive pointer on the left (shaded) remains unused, while the value of this pointer and its metadata are stored in the safe pointer store. The safe stacks T_1, T_2, T_3 have corresponding stacks T'_1, T'_2, T'_3 in regular memory to allocate unsafe stack objects.

except that CPI *also* stores the value of p in the safe pointer store. Combined with the isolation of the safe region (Section 4.4.2.3), this allows CPI to guarantee full memory safety of all sensitive pointers without having to instrument all pointer operations.

The instrumentation step changes instructions that operate on sensitive pointers, as found by CPI’s static analysis, to create and propagate the metadata directly following the based-on definition in Section 4.4.1. Instructions that explicitly take addresses of a statically allocated memory object or a function, allocate a new object on the heap, or take an address of a sub-object are instrumented to create metadata that describes the corresponding object. Instructions that compute pointer expressions are instrumented to propagate the metadata accordingly. Instructions that load or store sensitive pointers to memory are replaced with CPI intrinsic instructions (Section 4.4.2.3) that load or store both the pointer values and their metadata from/to the safe pointer store. In principle, call and return instructions also store and load code pointer, and so would need to be instrumented, but we instead protect return addresses using a safe stack (Section 4.4.3).

Every dereference of a sensitive pointer is instrumented to check at runtime whether it is safe, using the metadata associated with the pointer being dereferenced. Together with the restricted access to the safe region, this results in precise memory safety for all sensitive pointers.

Universal pointers (`void*` and `char*`) are stored in either the safe pointer store or the regular region, depending on whether they are sensitive at runtime or not. CPI instruments instructions that cast from non-sensitive to universal pointer types to assign special “invalid” metadata (e.g., with the lower bound greater than the upper bound) for the resulting universal pointers. These pointers, as a result, would never be allowed to access the safe region. CPI intrinsics for universal pointers would only

store a pointer in the safe pointer store if it had valid metadata, and only load it from the safe pointer store if it contained valid metadata for that pointer; otherwise, they would store/load from the regular region.

CPI can be configured to simultaneously store protected pointers in both the safe pointer store and regular regions, and check whether they match when loading them. In this debug mode, CPI detects all *attempts* to hijack control flow using non-protected pointer errors; in the default mode, such attempts are silently prevented. This debug mode also provides better compatibility with non-instrumented code that may read protected pointers (e.g., callback addresses) but not write them.

Modern compilers contain powerful static analysis passes that can often prove statically that certain memory accesses are always safe. The CPI instrumentation pass precedes compiler optimizations, thus allowing them to potentially optimize away some of the inserted checks while preserving the security guarantees.

4.4.2.3 Isolating the Safe Region

The safe region can only be accessed via CPI intrinsic instructions, and they properly handle pointer metadata and the safe stack (Section 4.4.3). The mechanism for achieving this isolation is architecture dependent.

On x86-32, we rely on hardware segment protection. We make the safe region accessible through a dedicated segment register, which is otherwise unused, and configure limits for all other segment registers to make the region inaccessible through them. The CPI intrinsics are then turned into code that uses the dedicated register and ensures that no other instructions in the program use that register. The segment registers are configured by the program loader, whose integrity we assume in our threat model; we also prevent the program from reconfiguring the segment registers via system calls. None of the programs we evaluated use the segment registers.

Certain architectures provide other hardware-enforced isolation mechanisms that can be used to protect the safe region. For instance, Intel recently introduced the memory protection keys extension and the MPX extension [Intel 2013] for the x86-64 architecture. These extensions could be used to isolate the safe region with low overhead, as discussed in Section 4.6.4.

On other architectures, CPI can protect the safe region using precise Software Fault Isolation (SFI) [Castro et al. 2009]. SFI requires that all memory operations in a program are instrumented, but the instrumentation is lightweight: it could be as small as a single and operation if the safe region occupies the entire upper half of the address space of a process. In our experiments, the additional overhead introduced by SFI was less than 5%.

On 64-bit architectures, CPI could also protect the safe region using randomization and information hiding. The fact that no addresses pointing into the safe region are ever stored in the regular region is what makes perfect information hiding possible. For example, the x86-64 architecture no longer enforces the segment limits; however, it still provides two segment registers with configurable base addresses. Similar to x86-32, we use one of these registers to point to the safe region; however, we choose the base address of the safe region at random and rely on preventing access to it through information hiding. Unlike classic ASLR, though, our hiding is leak-proof: since the objects in the safe region are indexed by addresses allocated for them in the regular region, no addresses pointing into the safe region are ever stored in regular memory at any time during execution. Hiding large regions through randomization is not secure [Oikonomopoulos et al. 2016]; however, when the size of the safe region is small (e.g., when the region is implemented as a hash table, as discussed in Section 4.6), the 48-bit address space of modern x86-64 CPUs makes guessing the safe region address impractical at least in some usage scenarios.

We further analyze the security and performance implications of the safe region protection mechanisms described above in Section 4.6.

Since sensitive pointers form a small fraction of all data stored in memory, the safe pointer store is highly sparse. To save memory, it can be organized as a hash table, a multi-level lookup table, or a simple array relying on the sparse address space support of the underlying OS. We implemented and evaluated all three versions, and we discuss the fastest choice in Section 4.6.

4.4.3 The Safe Stack

CPI treats the stack specially, in order to reduce performance overhead and complexity. This is primarily because the stack hosts values that are accessed frequently, such as return addresses that are code pointers accessed on every function call, as well as spilled registers (temporary values that do not fit in registers and compilers store on the stack). Furthermore, tracking which of these values will end up at runtime in memory (and thus need to be protected) vs. in registers is difficult, as the compiler decides which registers to spill only during late stages of code generation, long after CPI's instrumentation pass.

A key observation is that the safety of most accesses to stack objects can be checked statically during compilation, hence such accesses require no runtime checks or metadata. Most stack frames contain only memory objects that are accessed exclusively within the corresponding function and only through the stack pointer register with a constant offset. We therefore place all such proven-safe ob-

jects onto a *safe stack* located in the safe region. The safe stack can be accessed without any checks. For functions that have memory objects on their stack that do require checks (e.g., arrays or objects whose address is passed to other functions), we allocate separate stack frames in the regular memory region. In our experience, less than 25% of functions need such additional stack frames (see Table 4.4). Furthermore, this fraction is much smaller among short functions, for which the overhead of setting up the extra stack frame is non-negligible.

The safe stack mechanism consists of a static analysis pass, an instrumentation pass, and runtime support library. The analysis pass identifies, for every function, which objects in its stack frame are guaranteed to be accessed safely and can thus be placed on the safe stack; return addresses and spilled registers always satisfy this criterion. For the objects that do not satisfy this criterion, the instrumentation pass inserts code that allocates a stack frame for these objects on the regular stack. The runtime support allocates regular stacks for each thread and can be implemented either as part of the threading library, as we did on FreeBSD, or by intercepting thread create/destroy, as we did on Linux. CPI stores the regular stack pointer inside the thread control block, which is pointed to by one of the segment registers and can thus be accessed with a single memory read or write.

Our safe stack layout is similar to double stack approaches in ASR [Bhatkar et al. 2005] and XFI [Erlingsson et al. 2006], which maintain a separate stack for arrays and variables whose addresses are taken. However, we use the safe stack to enforce the CPI property instead of implementing software fault isolation. The safe stack is also comparable to language-based approaches like Cyclone [Jim et al. 2002] or CCured [Necula et al. 2005] that simply allocate these objects on the heap, but our approach has significantly lower performance overhead.

Compared to a shadow stack like in CFI [Abadi et al. 2005a], which duplicates return instruction pointers outside of the attacker’s access, the CPI safe stack presents several advantages: (i) all return instruction pointers and most local variables are protected, whereas a shadow stack only protects return instruction pointers; (ii) the safe stack is compatible with uninstrumented code that uses just the regular stack, and it directly supports exceptions, tail calls, and signal handlers; and (iii) the safe stack has near-zero performance overhead (Section 4.7.2), because only a handful of functions require extra stack frames, while a shadow stack allocates a shadow frame for every function call.

The safe stack can be employed independently from CPI, and we believe it can replace stack cookies [Cowan et al. 1998] in modern compilers. By providing precise protection of all return addresses (which are the target of ROP attacks today), spilled registers, and some local variables, the safe stack provides substantially stronger

security than stack cookies, while incurring equal or lower performance overhead and deployment complexity.

4.4.4 Code-Pointer Separation (CPS)

The code-pointer separation property trades some of CPI's security guarantees for reduced runtime overhead. This is particularly relevant to C++ programs with many virtual functions, where the fraction of sensitive pointers instrumented by CPI can become high since every pointer to an object that contains virtual functions is sensitive. We found that, on average, CPS reduces overhead by $4.3\times$ (from 8.4% for CPI down to 1.9% for CPS), and in some cases by as much as an order of magnitude.

CPS further restricts the set of protected pointers to code pointers only, leaving pointers that point to code pointers uninstrumented. We additionally restrict the definition of based-on by requiring that a code pointer be based only on a control-flow destination. This restriction prevents attackers from “forging” a code pointer from a value of another type, but still allows them to trick the program into reading or updating wrong code pointers.

CPS is enforced similarly to CPI, except (i) for the criteria used to identify sensitive pointers during static analysis and (ii) that CPS does not need any metadata. Control-flow destinations (pointed to by code pointers) do not have bounds because the pointer value must always match the destination exactly, hence no need for bounds metadata. Furthermore, they are typically static and hence do not need temporal metadata either (there are a few rare exceptions, like unloading a shared library, which are handled separately). This reduces the size of the safe region and the number of memory accesses when loading or storing code pointers. If the safe region is organized as a simple array, a CPS-instrumented program performs essentially the same number of memory accesses when loading or storing code pointers as a non-instrumented one; the only difference is that the pointers are being loaded or stored from the safe pointer store instead of their original location (universal pointer load or store instructions still introduce one extra memory access per such instruction). As a result, CPS can be enforced with low performance overhead.

CPS guarantees that (1) code pointers can only be stored to or modified in memory by code-pointer store instructions, and (2) code pointers can only be loaded by code-pointer load instructions from memory locations to which a code-pointer store instruction previously stored a value. Guarantee (1) restricts the attack surface, while guarantee (2) restricts the attacker's flexibility by limiting the set of locations to which the control can be redirected—the set includes only entry points of functions whose addresses were explicitly taken by the program during its execution. Combined with the safe stack, CPS precisely protects return addresses.

In contrast, CFI [Abadi et al. 2005a] allows any vulnerable instruction in a program to modify any code pointer; it only checks that the value of a code pointer, when used in an indirect control transfer, is within the set of allowed destinations, as defined by a specific implementation of CFI. For instance, coarse-grained CFI implementations [Zhang and Sekar 2013, Zhang et al. 2013] define allowed destinations as any function defined in a program (for function pointers) or that directly follows a call instruction (for return addresses). To illustrate this difference, consider the case of the Perl interpreter, which implements its opcode dispatch by representing internally a Perl program as a sequence of function pointers to opcode handlers and then calling in its main execution loop these function pointers one by one. Fine-grained CFI statically approximates the set of legitimate control-flow targets, which in this case would include all possible Perl opcodes. CPS, however, permits only calls through function pointers that are actually assigned. This means that a memory bug in a CFI-protected Perl interpreter may permit an attacker to divert control flow and execute any Perl opcode, whereas in a CPS-protected Perl interpreter the attacker could at most execute an opcode that exists in the running Perl program.

For C++ programs, CPS protects not only code pointers but also virtual table pointers. The abundance of virtual table pointers in most C++ programs gives an attacker sufficient freedom to induce malicious program behavior by only chaining existing virtual functions through corresponding existing call sites [Schuster et al. 2015]. Including virtual table pointers in the set of sensitive pointers protected by CPS prevents such attacks.

CPS provides strong control-flow integrity guarantees and incurs low overhead (Section 4.7). We found that it prevents all recent attacks designed to bypass CFI [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014, Carlini et al. 2015e, Evans et al. 2015]. We consider CPS to be a solid alternative to CPI in those cases when CPI’s (already low) overhead seems too high.

4.5 The Formal Model of CPI

This section presents a formal model and operational semantics of the CPI property and a sketch of its correctness proof. Due to the size and complexity of C/C++ specifications, we focus on a small subset of C that illustrates the most important features of CPI. Due to space limitations we focus on spatial memory safety. We build upon the formalization of spatial memory safety in SoftBound [Nagarakatte et al. 2009], reuse the same notation, and extend it to support applying spatial memory safety to a subset of memory locations. The formalism can be easily extended to provide

Atomic types	a	$::= \text{int} \mid p^*$
Pointer types	p	$::= a \mid s \mid f \mid \text{void}$
Struct types	s	$::= \text{struct}\{\dots; a_i: id_i; \dots\}$
LHS expressions	lhs	$::= x \mid *lhs \mid lhs.id \mid lhs \rightarrow id$
RHS expressions	rhs	$::= i \mid \&f \mid rhs + rhs \mid lhs \mid \&lhs$ $\mid (a)rhs \mid \text{sizeof}(p) \mid \text{malloc}(rhs)$
Commands	c	$::= c; \mid c \mid lhs = rhs \mid f() \mid (*lhs)()$

Figure 4.4 The subset of C. x denotes local statically typed variables; id denotes structure fields; i denotes integers; and f denotes functions from a pre-defined set.

sensitive int	$::= \text{false}$
sensitive void	$::= \text{true}$
sensitive f	$::= \text{true}$
sensitive p^*	$::= \text{sensitive } p$
sensitive s	$::= \bigvee_{i \in \text{fields of } s} \text{sensitive } a_i$

Figure 4.5 The sensitive criterion for protecting types in CPI.

temporal memory safety, directly applying the CETS mechanism [Nagarakatte et al. 2010] to the safe memory region of the model. Figure 4.4 gives the syntax rules of the C subset we consider in this section. All valid programs must also pass type checking as specified by the C standard.

We define the runtime environment E of a program as a triple (S, M_u, M_s) , where S maps variable identifiers to their respective atomic types and addresses, a regular memory M_u maps addresses to values (denoted v and called regular values), and a safe memory M_s maps addresses to values with bounds information (denoted $v_{(b,e)}$ and called safe values) or a special marker `none`. The bounds information specifies the lowest (b) and the highest (e) address of the corresponding memory object. M_u and M_s use the same addressing but might contain distinct values for the same address. Some locations (e.g., of `void*` type) can store either safe or regular value and are resolved to either M_s or M_u at runtime.

The runtime provides the usual set of memory operations for M_u and M_s , as summarized in Table 4.1. M_u models standard memory, whereas M_s stores values

Table 4.1 Memory Operations in CPI

Operation	Semantics
$\text{read}_u M_u l$	return $M_u[l]$
$\text{write}_u M_u l v$	set $M_u[l] = v$
$\text{read}_s M_s l$	return $M_s[l]$ if l is allocated; return none otherwise
$\text{write}_s M_s l v_{(b,e)}$	set $M_s[l] = v_{(b,e)}$ if l is allocated; do nothing otherwise
$\text{write}_s M_s l \text{none}$	set $M_s[l] = \text{none}$ if l is allocated; do nothing otherwise
$\text{malloc } E i$	allocate a memory object of size i in both $E.M_u$ and $E.M_s$ (at the same address); fail when out of memory

with bounds and has a special marker for “absent” locations, similarly to the memory in SoftBound’s [Nagarakatte et al. 2009] formalization. We assume the memory operations follow the standard behavior of read/write/malloc operations in all other respects, e.g., read returns the value previously written to the same location and malloc allocates a region of memory that is disjoint with any other allocated region.

Enforcing the CPI property with low performance overhead requires placing most variables in M_u , while still ensuring that all pointers that require protection at runtime according to the CPI property are placed in M_s . In this formalization, we rely on type-based static analysis as defined by the `sensitive` criterion, shown in Figure 4.5. We say a type p is sensitive iff `sensitive p = true`. Setting `sensitive` to true for all types would make the CPI operational semantics equivalent to the one provided by SoftBound and would ensure full spatial memory safety of all memory operations in a program.

The classification provided by the `sensitive` criterion is static and only determines which operations in a program to instrument. Expressions of sensitive types could evaluate to both safe or regular values at runtime, whereas expressions of regular types always evaluate to regular values. In particular, according to Figure 4.5, `void*` is sensitive and, hence, in agreement with the C specification, values of that type can hold any pointer value at runtime, either safe or regular.

We extend the SoftBound definition of the result of an operation to differentiate between safe and regular values and left-hand-side locations:

Results $r ::= v_{(b,e)} \mid v \mid l_s \mid l_u \mid \text{OK} \mid \text{OutOfMem} \mid \text{Abort}$

where $v_{(b,e)}$ and v are the safe (with bounds information) and, respectively, regular values that result from a right-hand-side expression, l_u and l_s are locations that result from a safe and regular left-hand-side expression, OK is a result of a successful command, and OutOfMem and Abort are error codes. We assume that all operational semantics rules of the language propagate these error codes unchanged up to the end of the program.

Using the above definitions, we now formalize the operational semantics of CPI through three classes of rules. The $(E, lhs) \Rightarrow_l l_s : a$ and $(E, lhs) \Rightarrow_l l_u : a$ rules specify how left-hand-side expressions are evaluated to safe or regular locations, respectively. The $(E, rhs) \Rightarrow_r (v_{(b,e)}, E')$ and $(E, rhs) \Rightarrow_r (v, E')$ rules specify how right-hand-side expressions are evaluated to safe values with bounds or regular values, respectively, possibly modifying the environment through memory allocation (turning it from E to E'). Finally, the $(E, c) \Rightarrow_c (r, E')$ rules specify how commands are executed, possibly modifying the environment, where r can be either OK or an error code. We only present the rules that are most important for the CPI semantics, omitting rules that simply represent the standard semantics of the C language.

Bounds information is initially assigned when allocating a memory object or when taking a function's address (both operations always return safe values):

$$\frac{(E, rhs) = i}{\text{address}(f) = l} \quad \frac{\text{address}(f) = l}{(E, \&f) \Rightarrow_r (l_{(l,l)})} \quad \frac{(E, rhs) = i}{\text{malloc } E \ i = (l, E')} \quad \frac{(E, rhs) = i}{(E, \text{malloc}(i)) \Rightarrow_r (l_{(l,l+i)}, E')}$$

Taking the address of a variable from S if its type is sensitive is analogous. Structure field access operations either narrow bounds information accordingly or strip it if the type of the accessed field is regular.

Type casting results in a safe value iff a safe value is cast to a sensitive type:

$$\frac{\text{sensitive } a'}{(E, rhs) \Rightarrow_l v_{(b,e)} : a} \quad \frac{\neg \text{sensitive } a'}{(E, rhs) \Rightarrow_l v_{(b,e)} : a} \quad \frac{(E, rhs) \Rightarrow_l v : a}{(E, (a')rhs) \Rightarrow_r (v_{(b,e)}, E)} \quad \frac{(E, rhs) \Rightarrow_l v_{(b,e)} : a}{(E, (a')rhs) \Rightarrow_r (v, E)} \quad \frac{(E, rhs) \Rightarrow_l v : a}{(E, (a')rhs) \Rightarrow_r (v, E)}$$

The next set of rules describes memory operations (pointer dereference and assignment) on sensitive types and safe values:

$\frac{\text{sensitive } a}{(E, lhs) \Rightarrow_l l_s : a*}$	$\frac{\text{sensitive } a}{(E, lhs) \Rightarrow_l l_s : a*}$	$\frac{\text{sensitive } a}{(E, lhs) \Rightarrow_l l_s : a}$
$\frac{\text{read}_s(E.M_s)l_s = \text{some } l'_{(b,e)}}{l' \in [b, e - \text{sizeof}(a)]}$	$\frac{\text{read}_s(E.M_s)l_s = \text{some } l'_{(b,e)}}{l' \notin [b, e - \text{sizeof}(a)]}$	$\frac{(E, rhs) \Rightarrow_r v_{(b,e)} : a}{E'.M_s = \text{write}_s(E.M_s)l_s v_{(b,e)}}$
$(E, *lhs) \Rightarrow_l l'_s : a$	$(E, *lhs) \Rightarrow_l \text{Abort}$	$(E, lhs = rhs) \Rightarrow_c (\text{OK}, E')$

These rules are identical to the corresponding rules of SoftBound [Nagarakatte et al. 2009] and ensure full spatial memory safety of all memory objects in the safe memory. Only operations matching those rules are allowed to access safe memory M_s . In particular, any attempts to access values of sensitive types through regular lvalues cause aborts:

$$\frac{\text{sensitive } a}{(E, lhs) \Rightarrow_l l_u : a*} \quad \frac{\text{sensitive } a}{(E, lhs) \Rightarrow_l l_u : a}$$

$$(E, *lhs) \Rightarrow_l \text{Abort} \quad (E, lhs = rhs) \Rightarrow_c (\text{Abort}, E)$$

Note that these rules can only be invoked if the value of the sensitive type was obtained by casting from a regular type using a corresponding type-casting rule. Levee relaxes the casting rules to allow propagation of bounds information through certain right-hand-side expressions of regular types. This relaxation handles most common cases of unsafe type casting; it affects performance (inducing more instrumentation) but not correctness.

Some sensitive types (only `void*` in our simplified version of C) can hold regular values at runtime. For example, a variable of `void*` type can first be used to store a function pointer and subsequently reused to store an `int*` value. The following rules handle such cases:

$$\frac{\text{sensitive } a}{(E, lhs) \Rightarrow_l l_s : a*} \quad \frac{\text{sensitive } a}{(E, lhs) \Rightarrow_l l_s : a}$$

$$(E, rhs) \Rightarrow_r v : a$$

$$\frac{\text{read}_s(E.M_s)l = \text{none}}{\text{read}_u(E.M_u)l = l'} \quad \frac{E'.M_u = \text{write}_u(E.M_u)l v}{E'.M_s = \text{write}_s(E.M_s)l \text{none}}$$

$$(E, *lhs) \Rightarrow_l l'_u : a \quad (E, lhs = rhs) \Rightarrow_c (\text{OK}, E')$$

Memory operations on regular types always access regular memory, without any additional runtime checks, following the unsafe memory semantics of C.

$$\frac{\neg \text{sensitive } a}{(E, lhs) \Rightarrow_l l : a*} \quad \frac{\neg \text{sensitive } a}{(E, lhs) \Rightarrow_l l : a}$$

$$(E, rhs) \Rightarrow_r v : a$$

$$\frac{\text{read}_u(E.M_u)l = l'}{\text{read}_u(E.M_u)l = l'} \quad \frac{E'.M_u = \text{write}_u(E.M_u)l v}{E'.M_s = \text{write}_s(E.M_s)l \text{none}}$$

$$(E, *lhs) \Rightarrow_l l'_u : a \quad (E, lhs = rhs) \Rightarrow_c (\text{OK}, E')$$

These accesses to regular memory can go out of bounds, but given that read_u and write_u operations can only modify regular memory M_u , they do not violate memory safety of the safe memory.

Finally, indirect calls abort if the function pointer being called is not safe:

$$\frac{(E, lhs) \Rightarrow_r l_s : f*}{(E, (*lhs)()) \Rightarrow_c (OK, E')} \quad \frac{(E, lhs) \Rightarrow_r l_u : f*}{(E, (*lhs)()) \Rightarrow_c (Abort, E)}$$

Note that the operational rules for values that are safe at runtime are fully equivalent to the corresponding SoftBound rules [Nagarakatte et al. 2009]: the rules expressions are equal assuming `sensitive a` is true and, depending on the rule, either `reads` is not none or the right-hand-side value is sensitive. Therefore, under these conditions, these rules satisfy the SoftBound safety invariant, which, as proven in Nagarakatte et al. [2009], ensures memory safety for such values. According to the `sensitive` criterion and the safe location dereference and indirect function call rules above, all dereferences of pointers that require protection according to the CPI property are always safe at runtime, or the program aborts. Therefore, the operational semantics defined above indeed ensure the CPI property as defined in Section 4.4.1.

4.6 Implementation

We describe a CPI/CPS enforcement tool for C/C++, called Levee, that was implemented on top of the LLVM 3.3 compiler infrastructure [LLUM 2017], with modifications to LLVM libraries, the `clang` compiler, and the `compiler-rt` runtime. To use Levee, one just needs to pass additional flags to the compiler to enable CPI (`-fcpi`), CPS (`-fcps`), or safe stack protection (`-fstack-protector-safe`). Levee works on unmodified programs and supports Linux, FreeBSD, and Mac OS X in both 32-bit and 64-bit modes.

Levee can be downloaded from the project homepage <http://levee.epfl.ch>. The SafeStack component of Levee is already integrated upstream into the Clang compiler [SafeStack 2017], and there is an ongoing effort to upstream the rest of CPI/CPS in the future as well.

4.6.1 Analysis and Instrumentation Passes

CPI and CPS Instrumentation Passes. We implemented the static analysis and instrumentation for CPI as two LLVM passes, directly following the design from Section 4.4.4. The LLVM passes operate on the LLVM intermediate representation (IR), which is a low-level, strongly typed, language-independent program representation tailored for static analyses and optimization purposes. The LLVM IR is generated from the C/C++ source code by `clang`, which preserves most of the type information that is required by our analysis, with a few corner cases. For example, in certain cases, `clang` does not preserve the original types of pointers that are cast to `void*`

when passing them as an argument to `memset` or similar functions, which is required for the `memset`-related optimizations discussed in Section 4.4.4. The IR also does not distinguish between `void*` and `char*` (it represents both as `i8*`), but this information is required for our string pointers detection heuristic. We augmented `clang` to always preserve such type information as LLVM metadata.

Safe Stack Instrumentation Pass. The safe stack instrumentation targets functions that contain on-stack memory objects that cannot be put on the safe stack. For such functions, it allocates a stack frame on the unsafe stack and relocates corresponding variables to that frame.

Given that most of the functions do not need an unsafe stack, Levee uses the usual stack pointer (`rsp` register on x86-64) as the safe stack pointer, and stores the unsafe stack pointer in the thread control block, which is accessible directly through one of the segment registers. When needed, the unsafe stack pointer is loaded into an IR local value, and Levee relies on the LLVM register allocator to pick the register for the unsafe stack pointer. Levee explicitly encodes unsafe stack operations as IR instructions that manipulate an unsafe stack pointer; it leaves all operations that use a safe stack intact, letting the LLVM code generator manage them. Levee performs these changes as a last step before code generation (directly replacing LLVM's stack-cookie protection pass), thus ensuring that it operates on the final stack layout.

Certain low-level functions modify the stack pointer directly. These functions include `setjmp/longjmp` and exception-handling functions, which store/load the stack pointer, and thread create/destroy functions, which allocate/free stacks for threads. On FreeBSD we provide full-system CPI, so we directly modified these functions to support the dual stacks. On Linux, our instrumentation pass finds `setjmp/longjmp` and exception-handling functions in the program and inserts required instrumentation at their call sites, while thread create/destroy functions are intercepted and handled by the Levee runtime.

4.6.2 Runtime support library

Most of the instrumentation by the passes discussed in Section 4.6.1 are added as intrinsic function calls, such as `cpi_ptr_store()` or `cpi_memcpy()`, which are implemented by Levee's runtime support library (a part of `compiler-rt`). This design cleanly separates the safe pointer store implementation from the instrumentation pass. In order to avoid the overhead associated with extra function calls, we ensure that some of the runtime support functions are always inlined. We compile these functions into LLVM bitcode and instruct `clang` to link this bitcode into

Table 4.2 Security Guarantees and Performance Overhead of Various Implementations of CPI/CPS

	Security ^a		Overhead ^b	
	CPI	CPS	CPI	CPS
Hardware segmentation	precise		8.4%	1.9%
Software fault isolation	precise		13.8%	7.0%
Information hiding				
hash table	16.6	20.7	9.7%	2.2%
lookup table	15	17	8.9%	2.0%
simple table	5	7	8.4%	1.9%

a. Either precise or number of entropy bits.

b. Average on SPEC2006.

every object file it compiles. Functions that are called rarely (e.g., `cpi_abort()`, called when a CPI violation is detected) are never inlined, in order to reduce the instruction cache footprint of the instrumentation.

We implemented and benchmarked several versions of the safe pointer store map in our runtime support library: a simple array, a two-level lookup table, and a hash table. The array implementation relies on the sparse address space support of the underlying OS. Initially, we found it to perform poorly on Linux due to many page faults (especially at start-up) and additional TLB pressure. Switching to superpages (2 MB on Linux) made this simple table the fastest implementation of the three. Note that due to the large virtual size of the simple table, the implementation based on it cannot be used in conjunction with randomization-based safe region isolation.

4.6.3 Safe Region Isolation

We implemented multiple mechanisms that efficiently enforce instruction-level isolation as required to protect the safe memory region: using hardware-enforced segmentation, software fault isolation, or randomization and information hiding. The security guarantees and performance implications of these mechanisms are summarized in Table 4.2; we discuss them in detail below. We focus on design choices behind each mechanism and ignore potential non-design bugs in the prototypes we released.

Hardware-Enforced Segmentation-Based Implementation. On architectures that support hardware-enforced segmentation (e.g., x86-32 and some x86-64), CPI uses this feature directly to enforce instruction-level isolation. In such implementations, CPI dedicates a segment register to point to the safe memory region, and it enforces, at compile time, that only instructions instrumented with memory safety checks use this segment register. CPI configures all other segment registers, which are used by non-instrumented instructions, to prevent all accesses to the safe region through these segment registers on the hardware level.

Hardware-enforced segmentation is supported on x86-32 CPUs but also on some x86-64 CPUs (see the Long Mode Segment Limit Enable flag), which demonstrates that adding segmentation to x86-64 CPUs is feasible, provided the techniques that could benefit from it indeed prove to be valuable.

This implementation of CPI is precise and imposes zero performance overhead on instructions that do not access sensitive pointers.

Software Fault Isolation-Based Implementation. On architectures where hardware-enforced segmentation is not available (e.g., ARM and most of the x86-64 CPUs), the instruction-level isolation can be enforced using lightweight software fault isolation (SFI). In our implementation, we align the safe region in memory so that enforcing a pointer not to alias with it can be done with a single bitmask operation (unlike more heavyweight SFI solutions, which typically add extra memory accesses and/or branches for each memory access in a program). Furthermore, accesses to the safe stack need not be instrumented, as they are guaranteed to be safe [Kuznetsov et al. 2014a].

Our SFI-based implementation of CPI is precise, and SFI increases the overhead by less than 5% relative to hardware-enforced segmentation.

Information-Hiding-Based Implementation. Another way to implement instruction-level isolation is based on randomization and information hiding. Such implementations exploit the guarantee of the CPI instrumentation that, in a CPI-instrumented program, no pointers into the safe region are ever stored outside the safe region itself. When the base location of the safe region is randomized, the above guarantee implies that the attacker has to resort to random guessing in order to find the safe region, even in the presence of an arbitrary memory read vulnerability. On 64-bit architectures, most of the address space is unmapped and so most of the failed guesses result in a crash. Such crashes, if frequent enough, can be detected by other means.

The actual expected number of crashes required to find the location of the safe region by random guessing is determined by the size of the safe region and the size of the address space. Today’s mainstream x86-64 CPUs provide 2^{48} bytes of address space (while the architecture itself envisions future extensions up to 2^{64} bytes). Half of the address space is usually occupied by the OS kernel, which leaves 2^{47} bytes for applications.

As explained above, the safe region stores a map that, for each sensitive pointer, maps the location that the pointer would occupy in the memory of a non-instrumented program to a tuple of the pointer value and its metadata. On 64-bit CPUs, each entry of this map occupies 32 bytes and, due to pointer alignment requirements, represents 8 bytes of program memory. The expected number of entries depends on the program memory usage, the fraction of sensitive pointers, and the data structure that is used to store this map.

We released three versions of our information-hiding-based CPI implementation that use either a hash table, a two-level lookup table, or a simple table to organize the safe region [Kuznetsov et al. 2014a]. Although all these safe region organizations are compatible with hardware-enforced segmentation and software-based fault isolation implementations, the choice of the organization has the highest impact on the information-hiding-based implementation. We analyze this impact. We estimate the size of the safe region and the expected number of crashes required to find its location for each of the versions below. For the purpose of this estimation, we assume a program uses 1GB of memory, 8% of which stores sensitive pointers (consistent with the experimental evaluation in [Kuznetsov et al. 2014a]), which amounts to $1 \text{ GB} \times 8\%/8 \text{ bytes} \approx 2^{23.4}$ sensitive pointers in total.

Hash table. This implementation is based on a linearly probed lookup table with a bitmask-and-shift-based hash function, which, due to sparsity of sensitive pointers in program memory, performs well with load factors of up to 0.5. Conservatively assuming a load factor of 0.25, the hash table would occupy $2^{23.4}/0.25 \times 32 = 2^{30.4}$ bytes of memory. Randomizing the hash table location can provide up to $47 - 30.4 = 16.6$ bits of entropy, requiring $2^{15.6} \approx 51,000$ crashes on average to guess it. In most systems, that many crashes can be detected externally, making the attack infeasible.

Two-level lookup table. This implementation organizes the safe region similar to page tables, using the higher 23 bits of the address as an index in the directory and the lower 22 bits as an index in a subtable (the lowest 3 bits are zero due to alignment). Each subtable takes 32×2^{22} bytes and describes an 8×2^{22} -byte region of the address space. Assuming sensitive pointers are uni-

formly distributed across the 1GB of continuous program memory, CPI will allocate $1\text{ GB}/(8 \times 2^{22}) \times 32 \times 2^{22} = 2^{32}$ bytes for the subtables. Randomizing the subtable locations gives $47 - 32 = 15$ bits of entropy, requiring 2^{14} crashes on average to guess. Note that the attacker will find a random one among multiple subtables, and finding usable code pointers in it requires further guessing. This attack is thus also infeasible in many practical cases.

Simple table. This simple implementation allocates a fixed-size region of 2^{42} bytes for the safe region that maps addresses linearly. This implementation would give only $47 - 42 = 5$ bits of entropy. The location of the simple table in this implementation can be guessed while causing only 16 crashes on average. This level of protection is not sufficient for most practical cases.

Code-Pointer Separation, unlike full CPI, does not require any metadata and has fewer sensitive pointers (by $8.5\times$ on average [Kuznetsov et al. 2014a]). This increases the number of expected crashes by $17\times$ for the hash table-based implementation, and by $4\times$ for the other two implementations.

The information-hiding-based implementation of CPI that uses a hash table to organize the safe region provides probabilistic security guarantees with $2^{16.6}$ bits of entropy (or $2^{20.7}$ for CPS). We believe that, in certain practical use cases, this number of crashes, especially given the uniform pattern of these crashes, can be detected automatically by external means.

In our evaluations, all three versions of information-hiding-based implementation have performance overhead comparable to the hardware-enforced segmentation.

4.6.4 Discussion

Binary-Level Functionality. Some code pointers in binaries are generated by the compiler and/or linker, and cannot be protected on the IR level. Such pointers include the ones in jump tables, exception handler tables, and the global offset table. Bounds checks for the jump tables and the exception handler tables are already generated by LLVM anyway, and the tables themselves are placed in read-only memory, hence cannot be overwritten. We rely on the standard loader’s support for read-only global offset tables, using the existing `RTLD_NOW` flag.

Limitations. The CPI design described in Section 4.4 includes both spatial and temporal memory safety enforcement for sensitive pointers; however, our current

prototype implements spatial memory safety only. It can be easily extended to enforce temporal safety by directly applying the technique described in [Nagarakatte et al. 2010] for sensitive pointers.

Levee currently supports Linux, FreeBSD, and Mac OS user-space applications. We believe Levee can be ported to protect OS kernels as well. Related technical challenges include integration with the kernel memory management subsystem and handling of inline assembly.

CPI and CPS require instrumenting all code that manipulates sensitive pointers; non-instrumented code can cause unnecessary aborts. Non-instrumented code could come from external libraries compiled without Levee, inline assembly, or dynamically generated code. Levee can be configured to simultaneously store sensitive pointers in both the safe and the regular regions, in which case non-instrumented code works fine as long as it only reads sensitive pointers but doesn't write them.

Inline assembly and dynamically generated code can still update sensitive pointers if instrumented with appropriate calls to the Levee runtime, either manually by a programmer or directly by the code generator.

Dynamically generated code (e.g., for JIT compilation) poses an additional problem: running the generated code requires making writable pages executable, which violates our threat model (this is a common problem for most control-flow integrity mechanisms). One solution is to use hardware or software isolation mechanisms to isolate the code generator from the code it generates.

Sensitive Data Protection. Even though the main focus of CPI is control-flow hijack protection, the same technique can be applied to protect other types of sensitive data. Levee can treat programmer-annotated data types as sensitive and protect them just like code pointers. CPI could also selectively protect individual program variables (as opposed to types), however, it would require replacing the type-based static analysis described in Section 4.4.2 with data-based points-to analysis, such as DSA [Lattner and Adve 2005, Lattner et al. 2007].

Future MPX-Based Implementation. Intel recently introduced a hardware extension, Intel MPX, to be used for hardware-enforced memory safety [Intel 2013].

We believe MPX (or similar) hardware can be re-purposed to enforce CPI with lower performance overhead than our existing software-only implementation. MPX provides special registers to store bounds along with instructions to check them, and a hardware-based implementation of a pointer metadata store (analogous to

the safe pointer store in our design), organized as a two-level lookup table. Our implementation can be adapted to use these facilities once MPX-enabled hardware becomes available. We believe that a hardware-based CPI implementation can reduce the overhead of a software-only CPI in much the same way as HardBound [Devietti et al. 2008] or Watchdog [Nagarakatte et al. 2012] reduced the overhead of SoftBound.

Adopting MPX for CPI might require implementing metadata loading logic in software. Like CPI, MPX also stores the pointer value together with the metadata. However, being a testing tool, MPX chooses compatibility with non-instrumented code over security guarantees: it uses the stored pointer value to check whether the original pointer was modified by non-instrumented code and, if yes, resets the bounds to $[0, \infty]$. In contrast, CPI's guarantees depend on preventing any non-instrumented code from ever modifying sensitive pointer values.

4.7 Evaluation

In this section we evaluate Levee's effectiveness, efficiency, and practicality. We experimentally show that both CPI and CPS are 100% effective on RIPE, the most recent attack benchmark we are aware of (Section 4.7.1). We evaluate the efficiency of CPI, CPS, and the safe stack on SPEC CPU2006 and find average overheads of 8.4%, 1.9%, and 0%, respectively (Section 4.7.2). To demonstrate practicality, we recompile with CPI/CPS/safe stack the base FreeBSD plus over 100 packages and report results on several benchmarks (Section 4.7.3).

We ran our experiments on an Intel Xeon E5-2697 with 24 cores running at 2.7 GHz in 64-bit mode with 512 GB RAM. The SPEC benchmarks ran on an Ubuntu Precise Pangolin (12.04 LTS) and the FreeBSD benchmarks in a KVM-based VM on this same system.

4.7.1 Effectiveness on the RIPE Benchmark

We described in Section 4.4 the security guarantees provided by CPI, CPS, and the safe stack based on their design; to experimentally evaluate their effectiveness, we use the RIPE [Wilander et al. 2011] benchmark. This is a program with many different security vulnerabilities and a set of 850 exploits that attempt to perform control-flow hijack attacks on the program using various techniques.

Levee deterministically prevents all attacks, both in CPS and CPI mode; when using only the safe stack, it prevents all stack-based attacks. On vanilla Ubuntu 6.06, which has no built-in defense mechanisms, 833–848 exploits succeed when

Levee is not used (some succeed probabilistically, hence the range). On newer systems, fewer exploits succeed, due to built-in protection mechanisms, changes in the runtime layout, and compatibility issues with the RIPE benchmark. On vanilla Ubuntu 13.10, with all protections (DEP, ASLR, stack cookies) disabled, 197–205 exploits succeed. With all protections enabled, 43–49 succeed. With CPS or CPI, none do.

The RIPE benchmark only evaluates the effectiveness of preventing existing attacks; as we argued in Section 4.4 and according to the proof outlined in Section 4.5, CPI renders all (known and unknown) memory corruption-based control-flow hijack attacks impossible.

4.7.2 Efficiency on SPEC CPU2006 Benchmarks

In this section we evaluate the runtime overhead of CPI, CPS, and the safe stack. We report numbers on all SPEC CPU2006 benchmarks written in C and C++ (our prototype does not handle Fortran). The results are summarized in Table 4.3 and presented in detail in Figure 4.6. We also compare Levee to two related approaches, SoftBound [Nagarakatte et al. 2009] and control-flow integrity [Abadi et al. 2005a, Zhang and Sekar 2013, Zhang et al. 2013, Niu and Tan 2014a].

CPI performs well for most C benchmarks; however, it can incur higher overhead for programs written in C++. This overhead is caused by an abundant use of pointers to C++ objects that contain virtual function tables—such pointers are sensitive for CPI, so all operations on them are instrumented. The same reason holds for gcc: it embeds function pointers in some of its data structures and then uses pointers to these structures frequently.

The next most important sources of overhead are libc memory manipulation functions, like `memset` and `memcpy`. When our static analysis cannot prove that a

Table 4.3 Summary of SPEC CPU2006 Performance Overheads

	Safe Stack	CPS	CPI
Average (C/C++)	0.0%	1.9%	8.4%
Median (C/C++)	0.0%	0.4%	0.4%
Maximum (C/C++)	4.1%	17.2%	44.2%
Average (C only)	−0.4%	1.2%	2.9%
Median (C only)	−0.3%	0.5%	0.7%
Maximum (C only)	4.1%	13.3%	16.3%

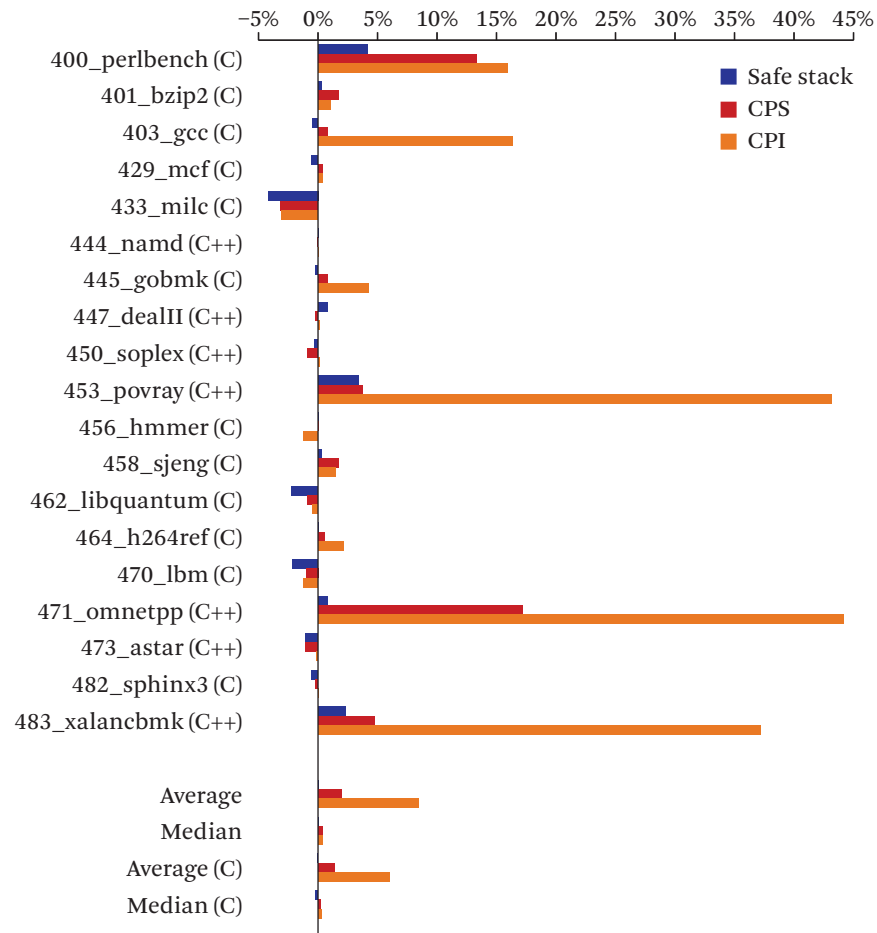


Figure 4.6 Levee performance for SPEC CPU2006, under three configurations: full CPI, CPS only, and safe stack only.

call to such a function uses as arguments only pointers to non-sensitive data, Levee replaces the call with one to a custom version of an equivalent function that checks the safe pointer store for each updated/copied word, which introduces overhead. We expect to remove some of this overhead using improved static analysis and heuristics.

CPS averages 1.2–1.8% overhead, and exceeds 5% on only two benchmarks, `omnetpp` and `perlbench`. The former is due to the large number of virtual function calls occurring at runtime, while the latter is caused by a specific way in which `perl`

implements its opcode dispatch: it internally represents a program as a sequence of function pointers to opcode handlers, and its main execution loop calls these function pointers one after the other. Most other interpreters use a `switch` for opcode dispatch.

Safe stack provided a surprise: in 9 cases (out of 19), it improves performance instead of hurting it; in one case (`namd`), the improvement is as high as 4.2%, more than the overhead incurred by CPI and CPS. This is because objects that end up being moved to the regular (unsafe) stack are usually large arrays or variables that are used through multiple stack frames. Moving such objects away from the safe stack increases the locality of frequently accessed values on the stack, such as CPU register values temporarily stored on the stack, return addresses, and small local variables.

The safe stack overhead exceeds 1% in only three cases: `perlbench`, `xalancbmk`, and `povray`. We studied the disassembly of the most frequently executed functions that use unsafe stack frames in these programs and found that some of the overhead is caused by inefficient handling of the unsafe stack pointer by LLVM's register allocator. Instead of keeping this pointer in a single register and using it as a base for all unsafe stack accesses, the program keeps moving the unsafe stack pointer between different registers and often spills it to the (safe) stack. We believe this can be resolved by making the register allocator algorithm aware of the unsafe stack pointer.

In contrast to the safe stack, stack cookies deployed today have an overhead of up to 5% and offer strictly weaker protection than our safe stack implementation.

The data structures used for the safe stack and the safe memory region result in memory overhead compared to a program without protection. We measure the memory overhead when using either a simple array or a hash table. For SPEC CPU2006 the median memory overhead for the safe stack is 0.1%; for CPS the overhead is 2.1% for the hash table and 5.6% for the array; and for CPI the overhead is 13.9% for the hash table and 105% for the array. We did not optimize the memory overhead and believe it can be improved in future prototypes.

In Table 4.4 we show compilation statistics for Levee. The first column shows that only a small fraction of all functions require an unsafe stack frame, confirming our hypothesis from Section 4.4.3. The other two columns confirm the key premises behind our approach, namely, that CPI requires much less instrumentation than full memory safety and CPS needs much less instrumentation than CPI. The numbers also correlate with Figure 4.6.

Comparison to SoftBound. We compare with SoftBound [Nagarakatte et al. 2009] on the SPEC benchmarks. We cannot fairly reuse the numbers from that paper because

Table 4.4 Compilation Statistics for Levee

Benchmark	FN _{UStack} ^a	MO _{CPS} ^b	MO _{CPI} ^b
400_perlbench	15.0%	1.0%	13.8%
401_bzip2	27.2%	1.3%	1.9%
403_gcc	19.9%	0.3%	6.0%
429_mcf	50.0%	0.5%	0.7%
433_milc	50.9%	0.1%	0.7%
444_namd	75.8%	0.6%	1.1%
445_gobmk	10.3%	0.1%	0.4%
447_dealII	12.3%	6.6%	13.3%
450_soplex	9.5%	4.0%	2.5%
453_povray	26.8%	0.8%	4.7%
456_hmmer	13.6%	0.2%	2.0%
458_sjeng	50.0%	0.1%	0.1%
462_libquantum	28.5%	0.4%	2.3%
464_h264ref	20.5%	1.5%	2.8%
470_lbm	16.6%	0.6%	1.5%
471_omnetpp	6.9%	10.5%	36.6%
473_astar	9.0%	0.1%	3.2%
482_sphinx3	19.7%	0.1%	4.6%
483_xalancbmk	17.5%	17.5%	27.1%

a. FN_{UStack} lists what fraction of functions need an unsafe stack frame.

b. MO_{CPS} and MO_{CPI} show the fraction of memory operations instrumented for CPS and CPI, respectively.

they are based on an older version of SPEC. In Table 4.5 we report numbers for the four C/C++ SPEC benchmarks that can compile with the current version of SoftBound. This comparison confirms our hypothesis that CPI requires significantly lower overhead compared to full memory safety.

Theoretically, CPI suffers from the same compatibility issues (e.g., handling unsafe pointer casts) as pointer-based memory safety. In practice, such issues arise much less frequently for CPI because CPI instruments far fewer pointers. Many of the SPEC benchmarks either do not compile or terminate with an error when instrumented by SoftBound, which illustrates the practical impact of this difference.

Table 4.5 Overhead of Levee and SoftBound on SPEC Programs That Compile and Run Free of Errors with SoftBound

Benchmark	Safe Stack	CPS	CPI	SoftBound
401_bzip2	0.3%	1.2%	2.8%	90.2%
447_dealII	0.8%	-0.2%	3.7%	60.2%
458_sjeng	0.3%	1.8%	2.6%	79.0%
464_h264ref	0.9%	5.5%	5.8%	249.4%

Comparison to Control-Flow Integrity (CFI). The average overhead for compiler-enforced CFI is 21% for a subset of the SPEC CPU2000 benchmarks [Abadi et al. 2005a] and 5–6% for MCFI [Niu and Tan 2014a] (without stack pointer integrity). CCFIR [Zhang et al. 2013] reports an overhead of 3.6%, and binCFI [Zhang and Sekar 2013] reports 8.54% for SPEC CPU2006 to enforce a weak CFI property with globally merged target sets. WIT [Akritidis et al. 2008], a source-based mechanism that enforces both CFI and write integrity protection, has 10% overhead.

At less than 2%, CPS has the lowest overhead among all existing CFI solutions, while providing stronger protection guarantees. Also, CPI’s overhead is bested only by CCFIR. However, unlike any CFI mechanism, CPI guarantees the impossibility of any control-flow hijack attack based on memory corruptions. In contrast, there exist successful attacks against CFI [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014]. While neither of these attacks are possible against CPI by construction, we found that, in practice, neither of them would work against CPS either. We further discuss conceptual differences between CFI and CPI in Section 4.2.

4.7.3 Case Study: A Safe FreeBSD Distribution

Having shown that Levee is both effective and efficient, we now evaluate the feasibility of using Levee to protect an entire operating system distribution, namely, FreeBSD 10. We rebuilt the base system—base libraries, development tools, and services like `bind` and `openssh`—plus more than 100 packages (including `apache`, `postgresql`, `php`, and `python`) in four configurations: CPI, CPS, Safe Stack, and vanilla. FreeBSD 10 uses LLVM/`clang` as its default compiler, while some core components of Linux (e.g., `glibc`) cannot be built with `clang` yet. We integrated the CPI runtime directly into the C library and the threading library. We have not yet ported the runtime to kernel space, so the OS kernel remained uninstrumented.

We evaluated the performance of the system using the Phoronix test suite [Phoronix 2017], a widely used comprehensive benchmarking platform for operat-

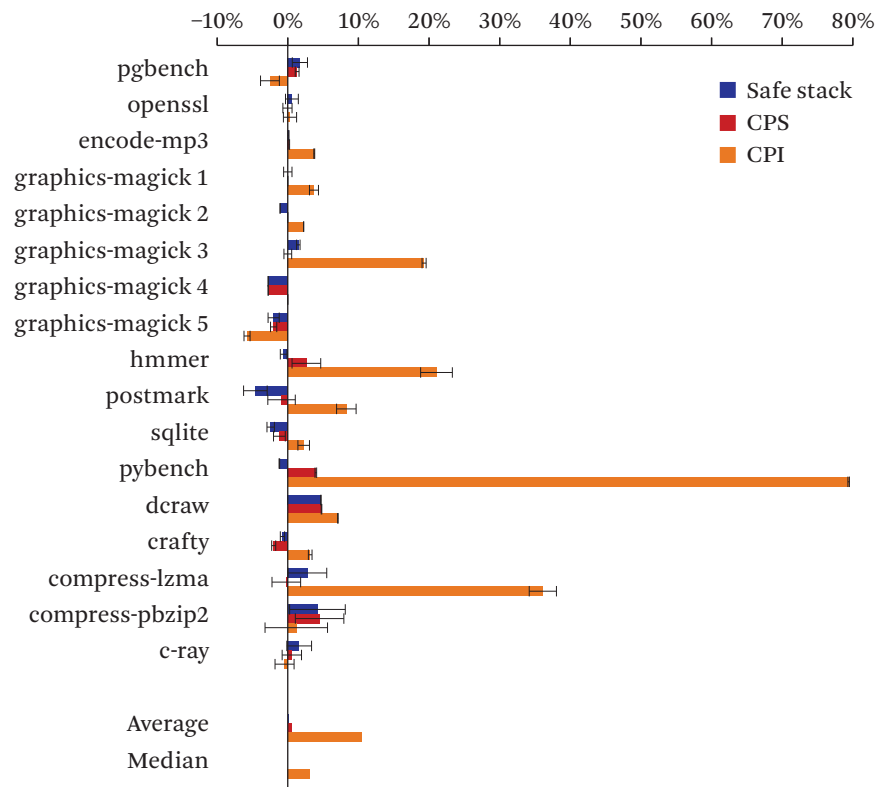


Figure 4.7 Performance overhead on FreeBSD (Phoronix).

ing systems. We chose the “server” setting and excluded benchmarks marked as unsupported or that do not compile or run on recent FreeBSD versions. All benchmarks that compiled and worked on vanilla FreeBSD also compiled and worked in the CPI, CPS, and Safe Stack versions.

Figure 4.7 shows the overhead of CPI, CPS, and the safe stack versions compared to the vanilla version. The results are consistent with the SPEC results presented in Section 4.7.2. The Phoronix benchmarks exercise large parts of the system and some of them are multi-threaded, which introduces significant variance in the results, especially when run on modern hardware. As Figure 4.7 shows, for many benchmarks the overhead of CPS and the safe stack are within the measurement error.

We also evaluated a realistic usage model of the FreeBSD system as a web server. We installed Mezzanine, a content management system based on Django, which

Table 4.6 Throughput Benchmark for Web Server Stack (FreeBSD + Apache + SQLite + mod_wsgi + Python + Django)

Benchmark	Safe Stack	CPS	CPI
Static page	1.7%	8.9%	16.9%
Wsgi test page	1.0%	4.0%	15.3%
Dynamic page	1.4%	15.9%	138.8%

uses Python, SQLite, Apache, and mod_wsgi. We used the Apache ab tool to benchmark the throughput of the web server. The results are summarized in Table 4.6.

The CPI overhead for a dynamic page generated by Python code is much larger than we expected but consistent with suspiciously high overhead of the pybench benchmark in Figure 4.7. We think it might be caused by the use of some C constructs in the Python interpreter that are not yet handled well by our optimization heuristics, e.g., emulating C++ inheritance in C. We believe the performance might be improved in this case by extending the heuristics to recognize such C constructs.

4.8 Conclusion

This chapter describes *code-pointer integrity* (CPI), a way to protect systems against all control-flow hijacks that exploit memory bugs, and *code-pointer separation* (CPS), a relaxed form of CPI that still provides strong guarantees. The key idea is to selectively provide full memory safety for just a subset of a program's pointers, namely, code pointers. The CPI/CPS implementation and evaluation shows that it is effective, efficient, and practical. Given its advantageous security-to-overhead ratio, the authors believe this approach marks a step toward deterministically secure systems that are fully immune to control-flow hijack attacks.

Evaluating Control-Flow Restricting Defenses

Enes Göktas, Elias Athanasopoulos, Herbert Bos,
Georgios Portokalidis

5.1 Introduction

Control-flow hijacking and arbitrary code-execution attacks remain a significant problem for software written in memory-unsafe languages like C and C++ [Joly 2013, Serna 2012, Evans 2013], as sophisticated exploits are able to bypass existing security mechanisms, such as address space layout randomization, stack smashing protection, and data execution protection. This is accomplished through information leakage and guessing attacks [Snow et al. 2013, Strackx et al. 2009], and code-reuse techniques, such as Return-Oriented Programming (ROP) [Roemer et al. 2012, Dai Zovi 2010], Jump-Oriented Programming (JOP) [Checkoway et al. 2010, Bletsch et al. 2011], and return-to-libc [Tran et al. 2011].

The rise of code-reuse attacks, as the next step in the arms race between attackers and defenders, has renewed the interest of the academic community in defenses based on restricting the control flow of a program. Such defenses attempt to limit the execution paths that are available to attackers after they manage to perform a control-flow hijacking attack. Of particular interest are approaches that exhibit low overhead and protect unmodified binaries, as they could be immediately applied to current production systems. One such group of approaches is based on Control-Flow Integrity (CFI) [Abadi et al. 2005a]. Specifically, recent work on practical CFI [Zhang and Sekar 2013, Zhang et al. 2013] is applicable on binaries, but it enforces a *looser* notion of control-flow integrity. Another group [Cheng et al. 2014, Pappas et al. 2013] focuses on characteristics of popular code-reuse attacks, such as the ones using ROP. Particularly, this group builds on the observation that ROP

exploits use long sequences of small gadgets to implement their functionality, and it defines heuristics to efficiently detect and prevent such gadget sequences.

Both of these groups of control-flow restricting defenses have to make compromises because of the limited information available within binaries and for achieving low overhead. *But what are the implications to security of such concessions?* Can these defenses still prevent code-reuse attacks? If not, how is that possible? The security evaluation of these defenses, in the past, included existing ROP-based exploits to demonstrate their ability to stop attacks. Is this the right strategy?

We attempt to provide an answer to these questions by evaluating both coarse-grained CFI solutions and heuristics-based code-reuse defenses. We examine two modern versions of CFI, CCFIR by [Zhang et al. \[2013\]](#) and binCFI by [Zhang and Sekar \[2013\]](#), and two heuristics-based approaches, kBouncer [[Pappas et al. 2013](#)] and ROPecker [[Cheng et al. 2014](#)]. These heuristic approaches utilize a novel hardware feature, known as the Last Branch Record (LBR), which is available in modern Intel CPUs. LBR introduces Machine-Specific Registers (MSR) that can be used to record the last 16 indirect branches taken by a program. kBouncer and ROPecker use the LBR to check the validity of the last 16 indirect branches. We identify the control-flow restrictions they impose and compose a *conservative* model combining their most restrictive aspects, which we coin CFR.

We then proceed to develop a methodology for constructing code-reuse attacks under this most restrictive model. We follow the approach that an attacker would take against systems protected with these technologies. First, we identify what kind of control-flow transitions are permissible and collect the corresponding gadgets that could be executed through them. Then we identify the different ways with which we can chain these gadgets and, finally, produce a proof-of-concept exploit (PoC) using well-known memory corruption bugs and the gadgets we have identified. We show that *despite the huge reduction in the number of available gadgets*, the *quality* of the remaining gadgets enables the creation of powerful code-reuse payloads. This highlights the importance of correctly evaluating the security of proposed defenses to avoid misconceptions on the offered guarantees.

The remainder of this chapter is organized as follows. Section 5.2 presents some background information on the control-flow restricting defenses we model and evaluate. Section 5.3 presents our analysis of the defenses and discusses their weaknesses. In Section 5.4, we perform a quantitative analysis of popular software, scanning them for types of gadgets that remain usable even when applying the evaluated defenses. We show that their number is not insignificant. We present the process of creating a PoC exploit that can circumvent loose control-flow restricting defenses in Section 5.5. We close in Section 5.6 by summarizing our findings.

Table 5.1 Allowable Transfers in Evaluated Defenses

Indirect Transfer Instruction	CCFIR (3-ID CFI)	bin-CFI (2-ID CFI)	kBouncer (heuristics)	ROPecker (heuristics)	CFR (3-ID CFI)
Return	calls: in non-sensitive functions	calls: all; functions: address taken, exception handlers	instructions: any call preceded	instructions: only valid	calls: in non-sensitive functions
Return in sensitive function	calls: all	calls: all; functions: address taken, exception handlers	instructions: any call preceded	instructions: only valid	calls: all
Indirect jump	functions: address taken, exported ^a	functions: address taken, exception handlers; calls: all	instructions: any	instructions: only valid	functions: address taken
Indirect call	functions: address taken, exported ^a	functions: address taken, exported ^a	instructions: any	instructions: only valid	functions: address taken, exported ^a

a. Exported by shared libraries (or imported to Windows).

5.2 Control-Flow Restricting Defenses

Here we explain in detail the control-flow restricting defenses that we will evaluate. First, we look at CFI approaches CCFIR [Zhang et al. 2013] and bin-CFI [Zhang and Sekar 2013] and then at kBouncer [Pappas et al. 2013] and ROPecker [Cheng et al. 2014], which employ heuristics. All the defenses we explore are applicable on commercial-off-the-shelf (COTS) binaries. See Table 5.1 for an overview of the characteristics of the defenses.

5.2.1 CCFIR

CCFIR [Zhang et al. 2013] applies CFI on Windows binaries through binary rewriting enabled by symbol relocation information in the targeted binaries. It disassembles binaries to identify indirect control-flow transfer instructions and code locations that could be targeted by these instructions. Control-flow instructions include forward-edge instructions, like indirect calls and jumps, and backward-edge instructions, like function returns. Possible targets include functions (*function entry points*), which either have their address taken or are imported by a binary, and instructions following a call (*call sites*).

To enforce control flow, CCFIR generates a set of trampolines that are associated with indirect control-flow instructions and their targets. There are two classes of trampolines: one associated with indirect calls and jumps to function

EPs and one associated with returns to call sites. The latter class is further separated to distinguish between call sites in *sensitive* functions (e.g., process-creating and memory-permission-modifying functions) and in other functions. These three types of trampolines are aligned differently and stored in a memory region called the *Springboard*. Control-flow checks are performed by rewriting indirect forward- and backward-edge instructions to go through the trampolines and introducing alignment checks. Essentially, CCFIR enforces CFI through the following checks:

- Forward-edge instructions can target only function entry-point trampolines.
- Backward-edge instructions in sensitive functions can target any call-site trampoline.
- Backward-edge instructions in non-sensitive functions can target only trampolines for call sites in non-sensitive functions.

5.2.2 bin-CFI

bin-CFI applies CFI on Linux binaries and also relies on disassembly and binary rewriting. Control flow is restricted with similar rules to CCFIR with the following differences:

- There is no notion of sensitive functions, so return instructions can target any call site.
- Indirect jumps are allowed to target call sites because the compiler sometimes replaces returns with the instruction sequence `pop; jmp`.
- Returns can target address-taken functions because they are sometimes used for function dispatching in Linux.
- Indirect jumps and returns are allowed to target exception handlers, which can be called during exception-related stack unwinding.

Enforcement is performed by duplicating application code and modifying one of the copies. In the modified copy, which is the one actually executing, indirect control-flow instructions are rewritten to target a check-and-translate trampoline. This trampoline receives a code pointer to the original code segment as an argument, checks that it points to an allowable target, and translates it to a pointer in the executing copy.

5.2.3 kBouncer

kBouncer also restricts the control flow of a program, focusing on code-reuse attacks (CRAs) that employ ROP. Unlike the previous approaches, kBouncer does

not require complete and correct disassembly of the protected binary. Because of this, it only checks that returns target instructions that are *call preceded*, that is, the preceding instruction is a `call`. This includes targets that may not correspond to an actual instruction emitted by the compiler but the preceding bytes still translate to a valid `call` instruction. However, in the core of kBouncer is a heuristic for identifying and preventing CRAs. CRA exploits usually consist of short sequences of instructions that end in an indirect control-flow transfer, called *gadgets*, which are chained together to perform arbitrary computations. A common characteristic of such *gadget chains* is that they involve the execution of *many small* gadgets. kBouncer builds on this observation to detect and prevent such attacks. It monitors the execution of a program at runtime, classifying any sequence of T_C or more of gadget-like code sequences, containing T_G or less instructions, as an attack. A depiction of kBouncer's operation is shown in Figure 5.1.

To achieve low overhead, kBouncer builds on an Intel CPU feature, the *Last Branch Record* (LBR), which can store the outcomes of the last 16 indirect branches. kBouncer checks the LBR every time there is a system call due to a security-sensitive function, such as the ones used for changing memory permissions and launching a new process. The values in the LBR are checked, first to ensure that returns transfer control to a valid target and, second, to ensure that there is no gadget chain with $T_C = 8$ or more gadgets of at most $T_G = 20$ instructions. These thresholds are experimentally determined.

5.2.4 ROPecker

Like kBouncer, ROPecker protects from CRAs by preventing long sequences of short gadgets utilizing the LBR to achieve low overhead. The main difference from kBouncer lies in how frequently the LBR is checked. Like kBouncer, ROPecker performs checks when risky system calls, e.g., like memory management or process creation system calls, are performed. However, in addition to those checks, checks are also performed during execution through the use of an execution sliding window, which is used to trigger interrupts. Briefly, all code except a small number of pages (a recommended two to four pages) are marked as non-executable. Every time a code page outside this window is executed, a page fault is generated, which is intercepted by ROPecker to check the LBR and slide the window of executable code. Because checks are performed continuously, ROPecker needs to set the thresholds T_C and T_G to more permissive values than kBouncer, in particular, $T_C = 11$ gadgets and $T_G = 6$ instructions. As an additional heuristic, ROPecker also looks at the gadgets that will execute in the future, e.g., by examining the frames in the stack, and uses them to apply the detection heuristic.

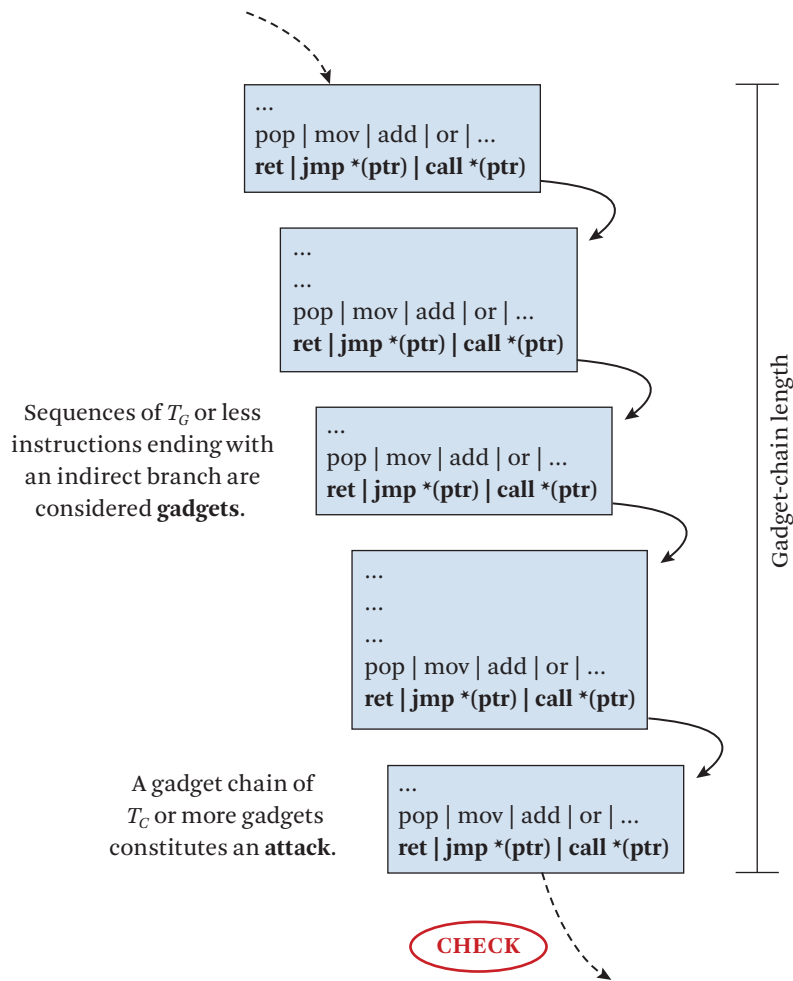


Figure 5.1 Example of a gadget chaining pattern used to identify code-reuse attacks. (From Göktaş et al. [2014b])

5.3 Security Analysis

In “traditional” CRAs, attackers have a large arsenal of gadgets available to choose from since every location in the code can be targeted by indirect control-flow transfer instructions. By applying control-flow restricting defenses, the locations that can be targeted are significantly reduced, which reduces the number of gadgets that can be used. As a result, attackers are left with fewer and more complex gadgets, e.g., gadgets with side effects including conditional branches.

Table 5.2 Gadget Definitions and Detection Thresholds in Heuristic-Based Defenses

Characteristic	kBouncer	ROPecker	CFR
Gadgets terminate with		ret, icall, ijmp	
Direct jumps in gadgets supported	Yes	No	Yes
Max gadget length (T_G)	20	6	20
Detection threshold (T_C)	8	11	8
Check frequency	System calls	System calls, pseudo-continuous	Continuous

However, this fact alone does not imply that attacks are not possible. To determine if they are, we consider an ideal Control-Flow Restricting (CFR) defense that combines the strictest aspects from all the evaluated approaches. Table 5.1 compares the indirect control-flow transfer restrictions imposed by the various defenses that we examine in this chapter and an ideal CFR defense combining all of them. CFR also assumes a runtime heuristic, similar to the one applied by kBouncer and ROPecker, is in place. We compare its properties with kBouncer and ROPecker in Table 5.2.

Note that CFR may actually not be viable in real systems, as it can be too restrictive, but it presents a good higher bound, which we use here to do a security analysis of control-flow restricting defenses.

5.3.1 Remaining Gadgets

With CFR in place, the types of gadgets available in code-reuse attacks changes from any potential code sequence that terminates in an indirect transfer, such as in Fig 5.2a, to a smaller set of gadgets that adhere to CFR's policy. We classify these into two major groups: function *Entry-Point (EP)* gadgets and *Call-Site (CS)* gadgets, which are depicted in Figure 5.2b. CS gadgets can start right after call instructions, while EP gadgets can start at the first instruction of an address-taken or imported (exported on Linux) function. Both types of gadgets end with an indirect control-flow instruction. Moreover, based on the allowed transfers listed in Table 5.1, CS gadgets can be split into two subclasses: the ones that are part of a sensitive function, which we denote by CS' , and the ones that are not, denoted by CS. Similarly, EP gadgets can be categorized as those that correspond to an address-taken function, denoted by EP, and those that correspond to imported/exported functions, denoted by EP' .

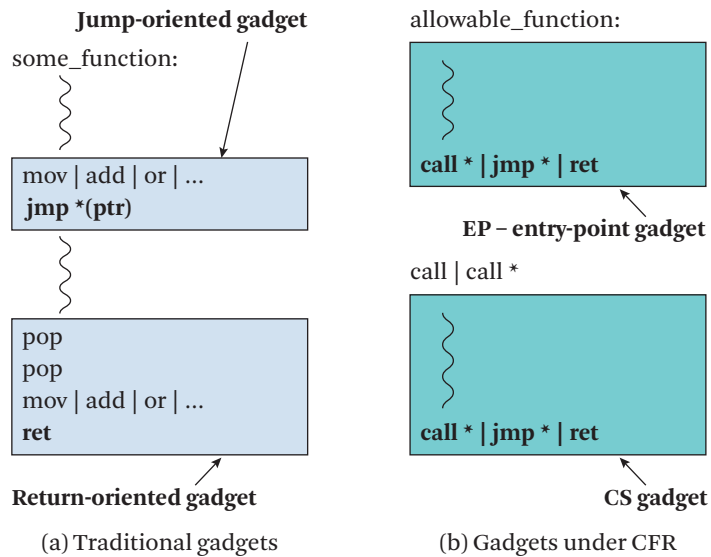


Figure 5.2 Types of gadgets used in code-reuse attacks. (Based on Göktaş et al. [2014a])

5.3.2 Chaining Gadgets

CRAs execute multiple gadgets in a chain to perform the desired computation/attack. With CFR, not only are less gadgets available but arbitrary chains of gadgets are also forbidden. Here we describe the various ways attackers could chain these gadgets under CFR.

In ROP attacks, chaining occurs by controlling a return instruction (e.g., through corrupting the return address in the stack) and placing the addresses of the gadgets in the chain in the stack. This type of chaining, shown in Figure 5.3a, is also possible under CFR by using CS gadgets, with the additional limitation that a CS gadget cannot transfer control to a CS' gadget.

In recent exploits [Pelletier 2012, Joly 2013], the attacker initially receives control of the operand of an indirect call or jump. For example, this includes exploits that hijack C++ virtual calls. In such cases, EP gadgets can be chained using indirect jumps, while both EP and EP' gadgets can be chained using indirect calls, like in Figure 5.3b. Note that, generally, chaining EP gadgets is much harder than chaining CS gadgets, as calls push data into the stack instead of popping data like in the case of returns.

Exactly because EP-gadget chaining is hard, switching from EP to CS-gadget chaining is an important capability for attackers. To achieve this, an EP gadget

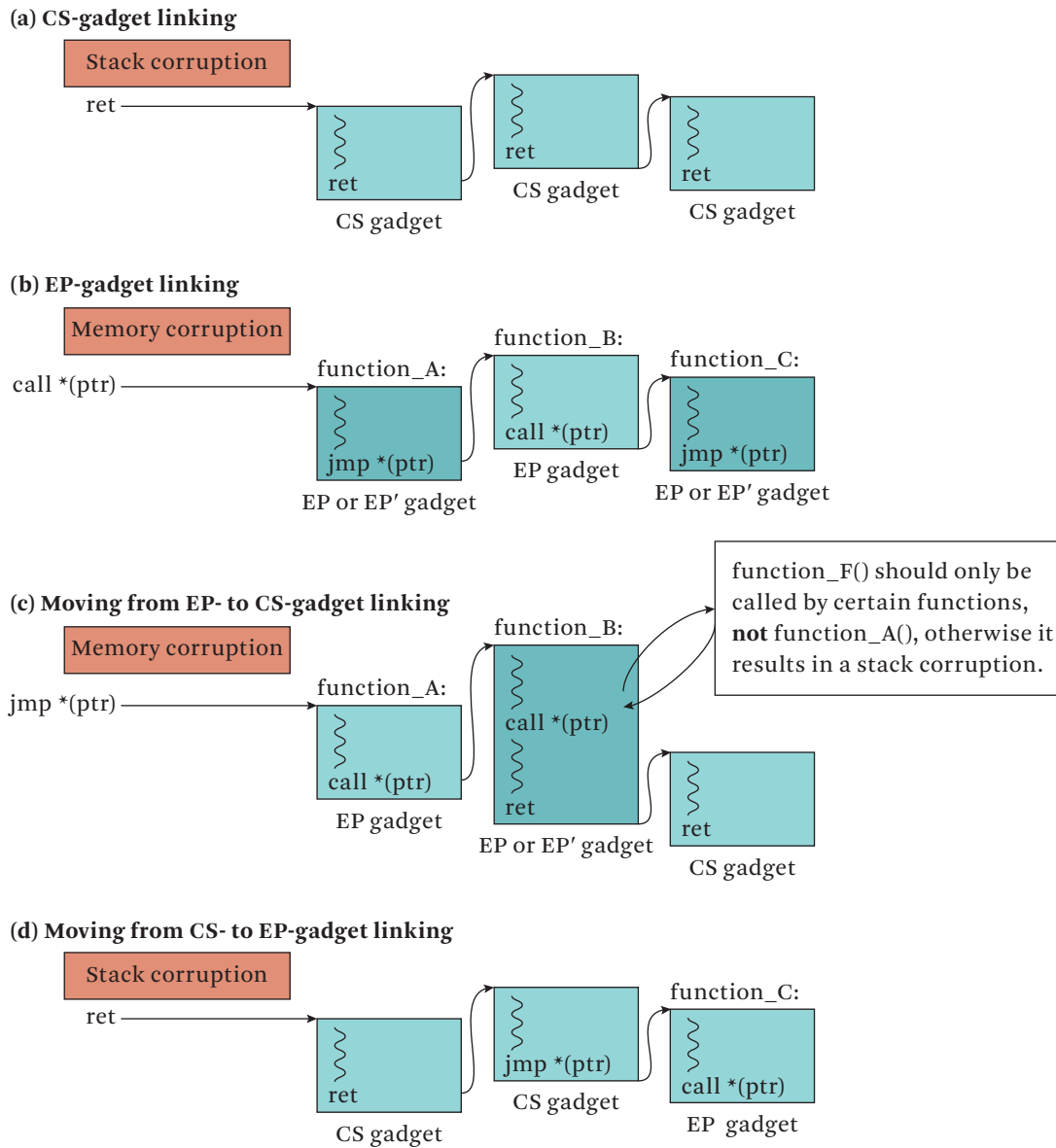


Figure 5.3 Different ways of linking gadgets. (Based on Göktaş et al. [2014a])

needs to be able to corrupt the return address stored in the stack and receive control of the return instruction that will use it. This is the only way for control not to return to the caller (i.e., the previous EP gadget). An example of this is shown in Figure 5.3c. The corruption of the stack does not need to be caused by a program bug; it is sufficient to call a function that performs a memory write and control that write. For example, one can call the `memcpy()` function using the stack as the destination buffer and the desired CS gadget address as source data.

In rare cases, it may be desirable to use a particular EP gadget or gadgets even when doing ROP, for instance, because EP gadgets are more prevalent or an EP gadget performs some rare functionality. Switching from CS- to EP-gadget chaining is possible by using a CS gadget that terminates with an indirect call, which we can control (e.g., its operand is not set within the gadget). Such an example is shown in Figure 5.3d.

Avoiding Heuristic-Based Detection

Table 5.2 shows the runtime restrictions that apply in chaining gadgets. In particular, there are restrictions in the length of a gadget chain that is allowed to execute. CFR adopts the strictest parameters for this, considering any sequence of 20 or more instructions ending in an indirect control-flow transfer a gadget and forbidding the execution of 8 or more consecutive gadgets. We also assume that these thresholds are checked continuously, so a gadget chain can at no point exceed them.

An approach for circumventing these checks and using longer gadget chains involves interleaving longer gadgets within the chain. Gadgets longer than 20 instructions are not considered gadgets by the heuristics, breaking the chain into smaller chains that do not trigger the heuristic, as shown in Figure 5.4. These longer gadgets could be *null gadgets*, that is, they may not perform any useful computation,

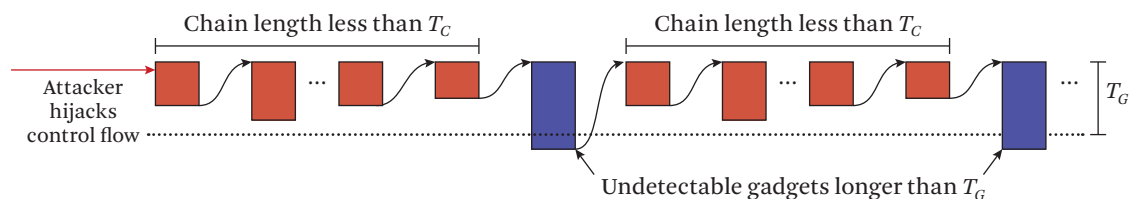


Figure 5.4 Mixing shorter and longer gadgets to avoid detection. Gadgets larger than T_G instructions are not considered gadgets by both kBouncer and ROPecker. The latter also ignores gadgets that contain direct branch instructions. (Based on Göktaş et al. [2014a])

or “productive” gadgets that are carefully inserted in the chain. Of course, using longer gadgets is not necessarily straightforward, as they may include instructions with side effects, e.g., accessing memory or conditional control flow.

5.3.3 Invoking System Calls and Functions

Chaining gadgets to perform arbitrary computations is not enough; attackers also need to be able to execute system calls to change memory permissions, spawn other processes, perform networking operations, etc. CRAs perform system calls by first preparing their arguments in the stack (e.g., on 32-bit architectures) or chaining gadgets to prepare them in registers (e.g., on 64-bit architectures). Subsequently, control is diverted either to a gadget that includes a system call instruction (i.e., `int 0x80` or `syscall`) or by calling a wrapper function (e.g., Linux’s `read()` function) that will perform the system call. Here we discuss how the invocation of function and system calls is affected by CFR.

The preparation of arguments for performing a call is only affected by the availability of gadgets and the chaining considerations we have already presented. Using gadgets that include system call instructions becomes significantly harder, though. That is because most such instructions are within wrapper functions, which are deemed sensitive and cannot be targeted by returns in non-sensitive functions. The best avenue for performing system calls when CFR is in play is calling a wrapper function by chaining a gadget that includes a call instruction. Both CS and EP gadgets can be utilized for this purpose; they need to either include a direct call or terminate in an indirect call instruction.

5.4 Quantifying Gadget Availability in CFR

Previous works [Zhang et al. 2013, Zhang and Sekar 2013, Pappas et al. 2013, Cheng et al. 2014] attempted to evaluate the effectiveness of CFR by calculating the reduction in possible control-flow edges. By blocking transfers to arbitrary bytes, the reduction is significant, reaching 98%; however, as we highlighted in Section 5.3, some gadgets remain. In this section, we present the results of our measurements that count the number of gadgets that remain under CFR defenses in various popular Windows applications. We also present statistics on gadget size, which affects heuristic-based detection.

5.4.1 Number of Available Gadgets

We analyze seven popular 32-bit applications on Windows 7 to identify and count the gadgets that can still be used by attackers. Table 5.3 describes the naming

Table 5.3 Gadget Names: Based on Type, Function-Calling Capability, and Exit Instruction

Type	Naming	Exit Instruction	Performs Function Call
Call-site gadgets	CS-R	return	—
	CS-IJ	indirect jump	—
	CS-IC	indirect call	—
	CS-IC-R	return	indirectly
	CS-C-R	return	directly
Entry-point gadgets	EP-R	return	—
	EP-IJ	indirect jump	—
	EP-IC	indirect call	—
	EP-IC-R	return	indirectly
	EP-C-R	return	directly

scheme we use for different types of gadgets, and Table 5.4 lists the results of our analysis. For each application, we count the number of gadgets contained in the entire application image in memory, including the main binary and all loaded libraries, as well as in selected popular libraries (DLLs) separately. The last row corresponds to the number of gadgets in the subset of libraries that are shared by all seven applications. We focus on CS and EP gadgets, ignoring CS' and EP' , for brevity and because the former are by far easier to chain.

The gadgets were collected by first launching the binaries to collect the set of libraries each one uses at runtime. Then, for each application binary and library, we used the popular disassembler IDA Pro [Hex-Rays 2017] to disassemble and identify the gadgets available under CFR. We assume that attackers favor smaller and less complex gadgets, so we limit our gadget collection to gadgets that do not include conditional branches or more than 30 instructions. *We notice that complex applications include a significant number of gadgets that can be used even under CFR.*

Let us use an example to explain the gadget collection process in more detail. While disassembling a function within a binary and locating a call instruction, we define a CS gadget starting directly after that instruction, unless we are at the end of the function, which may happen in the case of never-returning function calls like `exit()`. We keep disassembling until we reach a direct call to a function that has a resolved function name, or an indirect transfer, such as a return, or indirect call or jump. If we have stopped at an indirect transfer, we classify that gadget as CS-R, CS-IC, or CS-IJ. In the special case, where we have stopped at a call instruction, we

Table 5.4 Number of Different Types of Gadgets Found in Various 32-bit Applications on Windows 7^a

Application	PE File Name	Entry-Point Gadgets						Call-Site Gadgets					
		EP-R	EP-IC-R	EP-C-R	EP-J	EP-IC	EP-J	CS-R	CS-IC-R	CS-C-R	CS-J	CS-IC	
Internet Explorer 9	all	7,043	2,353	4,498	1,183	2,838	179,098	12,400	44,728	456	59,252		
	mshtml.dll	1,748	652	126	912	759	38,559	5,917	4,865	52	8,638		
	ieframe.dll	654	201	363	6	127	18,252	1,326	5,333	31	5,684		
Adobe Reader XI	all	18,303	1,772	8,447	1,082	2,085	166,175	5,641	62,500	1,480	40,091		
	AcroRd32.dll	13,106	650	3,099	778	372	58,027	1,156	25,276	1,213	12,587		
Firefox 24	all	9,773	2,611	6,316	635	2,785	233,183	8,951	52,883	465	45,949		
	xul.dll	3,281	1,349	1,781	172	900	101,475	4,812	16,617	97	23,181		
Word 2013	all	13,955	3,764	9,563	780	3,798	352,170	11,749	74,498	839	129,671		
	WWLIB.dll	962	413	690	33	284	60,289	1,665	3,426	34	31,913		
PowerPoint 2013	all	15,425	3,922	10,214	893	3,835	351,969	10,942	90,826	987	114,909		
	PPCore.dll	1,842	460	1,144	111	236	50,625	704	16,758	161	12,309		
Excel 2013	all	14,026	3,526	9,249	837	3,638	340,511	11,320	74,226	859	119,016		
	Excel.exe	1,313	214	357	102	198	54,414	1,274	4,005	60	21,698		
Microsoft Office 2013	MSO.dll	4,376	934	2,858	124	520	82,006	2,200	14,612	169	24,819		
Shared	all	2,271	731	2,682	171	1,395	67,209	2,506	23,203	204	15,521		
	shell32.dll	671	303	690	7	370	21,864	1,237	10,192	54	9,127		

a. We count only gadgets without conditional branches and with 30 instructions or less.
(From Göktaş et al. [2014a])

resume scanning until we find a return or other control-flow instruction. If we find a return, we count the gadget as a CS-IC-R or CS-C-R.

5.4.2 Gadget Size

Figure 5.5 shows the frequency of gadget sizes in Internet Explorer 9. We take a separate count of the gadgets that contain conditional branches and the ones that do not. We make two major observations. First, we see that there is a significant number of smaller gadgets, indicating that CRAs under CFR may be no more complex than conventional attacks. Second, we notice that there is also a non-negligible number of long gadgets, especially if we look at gadgets that contain branches, which can be used to break long gadget chains that would be detected by the gadget-

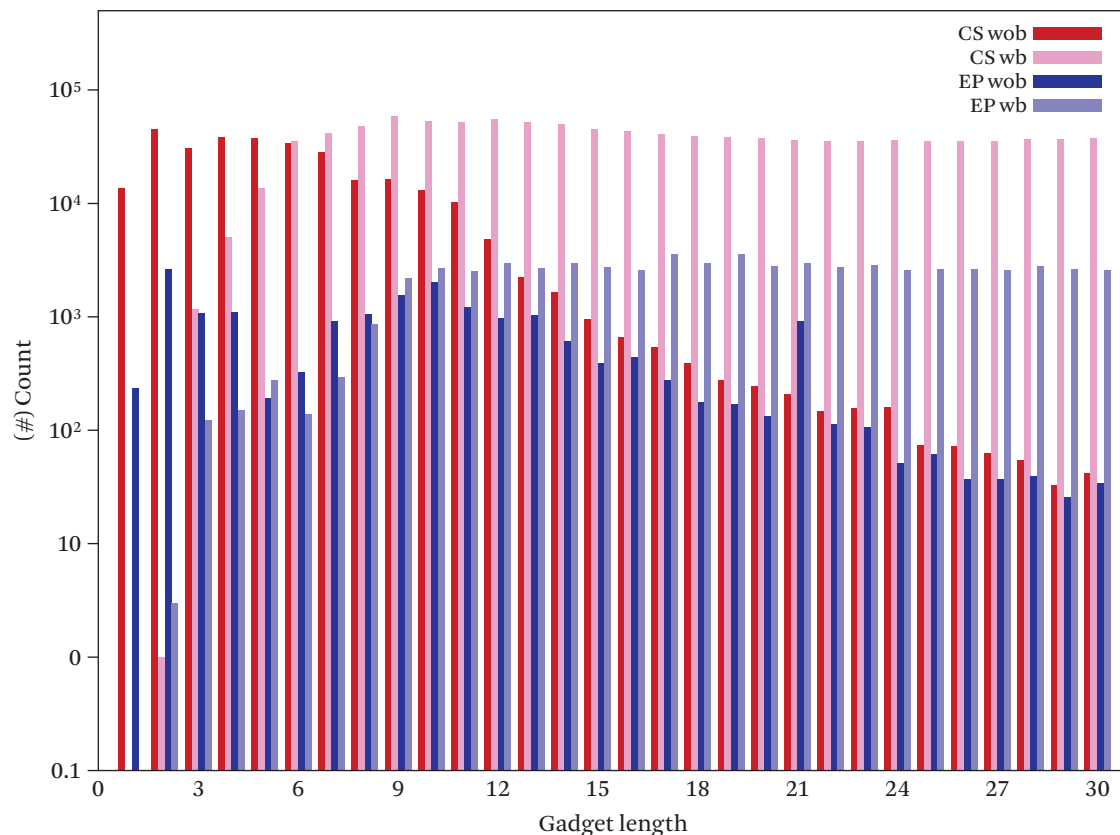


Figure 5.5 Frequency of gadgets in IE9 based on their length (instruction count). CS and EP *wob* refers to gadgets without branches and *wb* refers to gadgets with branches.

chain length heuristic of CFR. Of course, we cannot make any assumptions on how usable all these gadgets are; however, their number is a strong indication that there is a significant pool of gadgets to choose from. Another interesting observation is that there is a peak for gadgets with 21 instructions, which, after investigating, we determined occurs due to a block in the `ole32.dll` library that has 1,021 pointers (JMPs) pointing to it.

5.5 Proof-of-Concept Exploit against CFR

To determine the security of CFR, we take an offensive security approach and attempt to create a proof-of-concept (PoC) exploit that is effective despite the deployed defenses. Our PoC exploit is based on a well-known heap overflow-based exploit against 32-bit Internet Explorer 8 [Pelletier 2012] on Windows 7. This exploit has the interesting property that it can be used to both launch an information disclosure attack to defeat ASLR and perform a control-flow hijacking attack by taking control of the target address used in an indirect jump instruction.

Our goal is to use the security analysis and gadget collection described in the previous sections to guide us in modifying the original exploit so that it can be effective against the CFR model, which combines the most restrictive aspects of CCFIR, bin-CFI, kBouncer, and ROPecker. We follow a common practice of known code-reuse attacks, where the end goal is to inject arbitrary executable code within the vulnerable process.

Based on our security analysis, the requirements for the PoC exploit are the following:

Use and chain only gadgets that are available under CFR. Because of the restrictions on control-flow transitions, we need to restrict ourselves to only use available gadgets, as described in Sections 5.3.1 and 5.3.2.

The payload must evade heuristic-based defenses. Our gadget chain cannot include seven or more gadgets of 20 or fewer instructions in a row. For this, we need to include *heuristic breaker* (HB) gadgets that are longer than 20 instructions, whenever we are close to using seven shorter gadgets. This way our payload “flies below the radar” at run time.

The injected code must be placed on an allowable address. Past exploits performed arbitrary code injection and execution by using a ROP payload to execute `VirtualProtect()` on Windows or `mmap()` on Linux to make an attacker-controlled buffer executable and then transfer control to it. However, with CFR, control can only be transferred to particular addresses, the

ones that host CS and EP gadgets. To inject code now, we need to make existing code *writable*, use code-reuse to inject our code at the address of a CS or EP gadget in the now writable area, and, finally, chain the injected code.

In the remainder of this section, we describe how we construct a PoC exploit in more detail.

5.5.1 Exploit Preparation

The exploit consists of two parts: the first part deals with disclosing information to bypass ASLR and reveal the addresses of the gadgets, while the second part takes control of the target address of an indirect jump instruction. While in reality we begin from the second phase, by first constructing a code-reuse payload and then creating the part of the exploit for leaking the locations of the libraries containing those gadgets that we use, we describe the exploit in the same order the phases execute. As such, we start by describing how we locate two libraries used by Internet Explorer 8: `mshtml.dll` and `ieframe.dll`, which include all the gadgets required for our payload as well as the user-controlled buffer that contains our code-reuse and code-injection payloads.

Breaking ASLR

We trigger the vulnerability by accessing the `span` and `width` attributes of an HTML table's column through JavaScript (JS). We use it to overwrite the `size` attribute of a string object, which consequently allows the `substring()` method of the string class to read data beyond the boundary of the string object, as long as we know the relative offset of that data from the string object. A great feature of the vulnerability is that it can be triggered repeatedly from JS to leak memory from different locations, hence serving as a memory disclosure interface.

To use this memory leak to locate the two DLLs containing our gadgets, we first use heap Feng Shui [Sotirov 2007] to position the vulnerable buffer and the string and button objects in the right order, so that we can overflow in the string object without concurrently hijacking the control flow (i.e., receiving control of the indirect jump). By means of the exploited string object, we can read the pointer from the button object to the `mshtml.dll` library. Because the target of this pointer is at a fixed offset within the library, it reveals its base address.

Next we need to discover the base address of `ieframe.dll`. Fortunately, `mshtml.dll` has pointers to it in its Delayed Import Address Table, but to read them, we first need to compute the offset from the string object to `mshtml.dll` because we can only read memory by knowing this offset. If we learn the address

of the string object itself, we can infer the offset to any location in the memory. To learn this address, we use a pointer in the button object to a buffer at a constant distance from the string object.

Finally, we need to determine the location of the buffer we control, which contains our payloads. We use heap spraying [DarkReading 2009] to create many copies of our buffer in the process's memory, which has the effect of placing one of the copies at an address that can be reliably determined. Note that while heap spraying is not a foolproof method, it works consistently in this particular case.

Control-Flow Hijacking

After breaking ASLR, we use JS to patch the learned addresses in our payload and finally trigger the overflow again to overwrite the virtual function table (VFT) pointer within a button object. Later, when we access the button object from within our carefully prepared JS code, the program will operate on the overwritten data and will eventually grant us control over an indirect jump instruction, which we direct to the first gadget of our code-reuse payload.

5.5.2 Payload Execution

The first part of the payload will execute a chain of gadgets to inject shellcode in the executable and transfer control to it. Since this is only a PoC, the shellcode will do no more than launch a third program, such as the calculator. An attacker, on the other hand, could perform arbitrary actions, like downloading and executing a malicious binary.

Our code-reuse payload consists of the gadget chain depicted in Figure 5.6. To ensure that the chain is not detected by defenses such as kBouncer and ROPecker, we use as many HB gadgets as possible, which results in a gadget chain where 6 out of 11 gadgets are longer HB gadgets. This is even more important if we consider that program execution before we receive control already includes code sequences that are treated as gadgets, as shown in the figure. In detail, our chain performs the following steps:

1. **Go from EP to CS gadgets.** The vulnerability gives us control of an indirect jump instruction, which due to CFR only allows us to transfer control to address-taken EP' gadgets. Because CS gadgets are significantly easier to use, we decided to immediately switch from EP to CS-gadget chaining. As we describe in Section 5.3.2, to perform this switch we need our chain to corrupt the stack so the next ret instruction that executes is under our control. This whole process is accomplished by using four gadgets. We use two EP

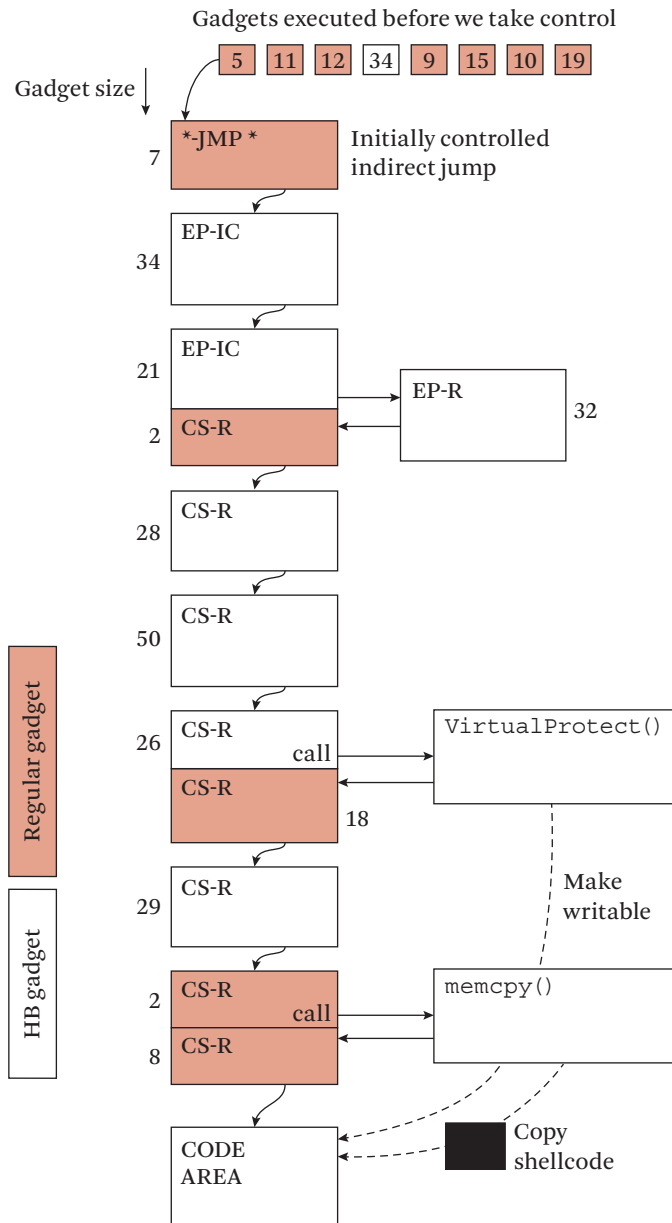


Figure 5.6 Overview of PoC exploit that bypasses the stricter CFR model. All the gadgets used are intended call-site (CS) or entry-point (EP) gadgets. (From Göktas et al. [2014a])

gadgets with 34 and 21 instructions, respectively, to prepare for the code that will corrupt the stack. An EP gadget of 32 instructions is the one actually corrupting the stack so that we control the next return instruction, which is located in a short two-instruction long CS gadget.

2. **Perform a stack pivot.** To continue executing CS gadgets, the stack pointer has to point to the fake stack in the buffer we control. The process of changing the stack pointer for this purpose is called *stack pivoting*. As shown in Figure 5.6, we use two CS gadgets with 2 and 28 instructions, respectively, to perform the stack pivoting. The first gadget loads the address of the fake stack in the frame pointer, the EBP register, and the second gadget moves the value from the frame pointer to the stack pointer, the ESP register, through the `leave` instruction.
3. **Make existing code writable.** Because we eventually want a `ret` instruction to direct control flow to our shellcode, we must inject it at a valid CS location. Hence, we need to invoke `VirtualProtect()` to make a selected CS location writable first. We locate and chain a CS gadget with 50 instructions that we use to prepare the necessary registers for calling `VirtualProtect()` and another one for making a direct call to the function.
4. **Copy over shellcode.** To copy the shellcode, we invoke `memcpy()`. To achieve this, we use another CS gadget with 29 instructions to prepare the necessary registers and call the function through another CS gadget with a direct call to it.
5. **Transfer control flow to shellcode.** Finally, we use the CS gadget executed after `memcpy()` to *legitimately* redirect execution to the shellcode we just copied.

This PoC exploit manages to inject arbitrary code into a process that is protected by CFR, hence demonstrating that attacks are still possible under such defenses. To summarize, the code-reuse payload we constructed consists of EP and CS gadgets alone that are available under CFR. It also consists of 11 gadgets, 6 of which are longer than 20 instructions, so the longest gadget chain detected by heuristics has a length of 4 and, in fact, that chain is part of the program's regular control flow.

5.6 Summary

With increasing dependency on indirect control transfers (ICT) on the one hand, and ever more sophisticated code-reuse attacks on the other, the need for new and better defenses is more urgent than ever. However, what is “better”? Without a

meaningful metric to measure the strength of a defense, it is often impossible to determine how much additional security it provides.

Control-flow restricting defenses in particular are both promising and problematic: Promising because in a program that cannot easily stray from its intended control-flow graph, attackers can only use the set of legitimate targets of indirect control transfers to construct a code-reuse payload; the tighter the set, the fewer the options for attackers. Problematic because as long as we do not know how many and what kind of options are left, we do not know whether it is difficult or easy to construct such payloads.

All previous attempts to define suitable metrics proved insufficient. For instance, a well-known metric is to estimate the reduction in the number of available gadgets after applying control-flow restrictions [Zhang and Sekar 2013], but this is not very helpful if the remaining gadgets trivially allow for the construction of malicious payloads [Göktas et al. 2014a]. An alternative approach to evaluate defense mechanisms is to measure how many *useful* gadgets are left after applying the control-flow restrictions [Salwan 2011, Schwartz et al. 2011, Corelan 2011]. Unfortunately, determining what is useful and what is not is difficult and goes well beyond defining a small set of simple templates a priori. Finally, research papers frequently evaluate a new defense by testing it against a small number of existing real attacks [Zhang and Sekar 2013, Zhang et al. 2013]. Again, such evaluations do not necessarily say much about how the security mechanism fares against targeted exploits, tailored explicitly to bypass it.

Evaluation of new work on control-flow restriction comprises two aspects: the optimality of the target set reduction and the usefulness of the target set. The optimality of the target set reduction measures to what extent it is possible to further minimize the set of possible targets for indirect control transfers. For instance, a CFI solution may reduce the number of possible targets of an indirect call from tens of thousands to just ten, but this is not the optimal set if the indirect call has only two possible targets. The usefulness of the target set assesses what an attacker can still do with the remaining targets.

Some research has been successful in achieving optimality of the target set reduction. For instance, ShrinkWrap [Haller et al. 2015] protects virtual function calls in C++ programs by limiting their targets to appropriate methods in the class hierarchy. By exhaustively creating all possible inheritance and call-site scenarios, the researchers prove that the target sets that their solution generates do not break any C++ semantics while also not including unneeded targets. Such exhaustive testing to assess the optimality of the target set may be possible for (some) other defenses also. Unfortunately, as discussed above, the community has been less

successful in determining the usefulness of the remaining target sets, and currently no best practices exist.

In fact, the community's track record in developing suitable control-flow restrictions appears to be less than stellar: all practical defenses to date are vulnerable to one clever new bypass or another. At the same time, we keep raising the bar for attackers. There is no doubt that restricting the target sets of indirect control transfers complicates the construction of malicious payloads out of the remaining gadgets—we are just struggling to determine exactly how much.

A tentative conjecture that we draw out of the plethora of research on control-flow restrictions in recent years is that while static approaches are relatively weak [Göktas et al. 2014a, Carlini and Wagner 2014, Evans et al. 2015], adding context sensitivity significantly raises the bar [van der Veen et al. 2015, Dang et al. 2015]. While some research applies context sensitivity on the forward edge also [van der Veen et al. 2015], a simpler but still very powerful addition to static techniques to restrict a program's control flow is to add a shadow stack [Carlini et al. 2015e].

Unfortunately, the inability to assess the security of new defenses and the strong claims often found in research papers have led to a perverse cat-and-mouse game between researchers, whereby a defense introduced at one conference is bypassed at the next. As the net result is not always positive, we believe that developing better ways to evaluate security mechanisms based on control-flow restriction is an urgent (albeit challenging) research topic.



Attacking Dynamic Code

Felix Schuster, Thorsten Holz

Typically, code-reuse attacks exhibit unique characteristics in the control flow (and the data flow) that allow for generic protections, regardless of the language an application was programmed in. For example, if one can afford to monitor all return instructions in an application while maintaining a full shadow call stack, even advanced ROP-based attacks [Göktas et al. 2014a, Carlini and Wagner 2014, Davi et al. 2014, Göktaş et al. 2014b, Schuster et al. 2014] cannot be mounted [Frantzen and Shuey 2001, Abadi et al. 2005a, Davi et al. 2011]. In this chapter, we present a form of code-reuse attack we call *Counterfeit Object-Oriented Programming* (COOP) that is different: COOP exploits the fact that each C++ virtual function is address taken, which means that a static code pointer exists to it. Accordingly, C++ applications usually contain a high ratio of address-taken functions, typically a significantly higher one compared to C applications. If, for example, an imprecise CFI solution does not consider C++ semantics, then these functions are all likely valid indirect call targets [Abadi et al. 2005a] and can thus be abused.

COOP exclusively relies on C++ virtual functions that are invoked through corresponding calling sites as gadgets. Hence, without deeper knowledge of the semantics of an application developed in C++, COOP's control flow cannot reasonably be distinguished from a benign one. Another important difference to existing code-reuse attacks is that in COOP conceptually no code pointers (e.g., return addresses or function pointers) are injected or manipulated. As such, COOP is immune to defenses that protect the integrity and authenticity of code pointers. Moreover, in COOP, gadgets do not work relative to the stack pointer. Instead, gadgets are invoked relative to the `this`-ptr on a set of adversary-defined *counterfeit objects*. Note that in C++ the `this`-ptr allows an object to access its own address. Addressing relative to the `this`-ptr implies that COOP cannot be mitigated by defenses that prevent the stack pointer to point to the program's heap [Fratric 2012], which is typically

the case for ROP-based attacks launched through a heap-based memory corruption vulnerability.

The counterfeit objects used in a COOP attack typically overlap such that data can be passed from one gadget to another. Even in a simple COOP program, positioning counterfeit objects manually can become complicated. Hence, we implemented a programming framework that leverages the Z3 SMT solver [de Moura and Bjørner 2008] to derive the object layout of a COOP program automatically.

The main results of this chapter were published in a previous conference paper [Schuster et al. 2015] and in a Ph.D. dissertation [Schuster 2015]. In this chapter, we streamline the presentation and highlight the challenges when attempting to protect against COOP-like attacks. Certain details are omitted and interested readers are referred to closely related publications on this topic.

6.1 Goals and Attacker Model

COOP is a code-reuse attack approach targeting applications developed in C++ or possibly other object-oriented languages. We note that many of today’s notoriously attacked applications are written in C++ (or contain major parts written in C++); examples include Microsoft Internet Explorer, Google Chromium, Adobe Reader, Microsoft Office, LibreOffice, and OpenJDK.

In the following, we state our design goals and our attacker model for COOP before we describe the actual building blocks of a COOP attack in the next section. For brevity, the rest of this chapter mainly focuses on Microsoft Windows and the x86-64 architecture as the runtime environment. The COOP concept is generally applicable to C++ applications running on any operating system; it also extends to other programming languages such as Objective-C [Lettner et al. 2016] and to other instruction set architectures. Interestingly, mounting a COOP attack on a RISC architecture can be even simpler than one mounted on a CISC architecture because calling conventions that pass function arguments through registers rather than over the stack facilitate COOP (further discussed in Section 6.2.7).

6.1.1 Goals

With COOP, we aim to demonstrate powerful code-reuse attacks that do not exhibit the revealing characteristics of existing attack approaches. Even advanced existing variants of return-into-libc, ROP, JOP, or COP [Göktas et al. 2014a, Carlini and Wagner 2014, Davi et al. 2014, Göktas et al. 2014b, Schuster et al. 2014, Bittau et al. 2014, Bosman and Bos 2014, Tran et al. 2011] rely on control-flow and data-flow

patterns that are rarely or never encountered for regular code; among these are typically one or more of the following:

- C-1. Indirect *calls/jumps* to non-address-taken locations
- C-2. *Returns* not in compliance with the call stack
- C-3. Excessive use of indirect branches
- C-4. Pivoting of the stack pointer (possibly temporarily)
- C-5. Injection of new code pointers or manipulation of existing ones

These characteristics still allow for the implementation of effective, low-level, and programming-language-agnostic protections. For instance, maintaining a full shadow call stack [Frantzen and Shuey 2001, Abadi et al. 2005a, Davi et al. 2011] suffices to fend off many types of ROP-based attacks.

With COOP we demonstrate that it is not sufficient to generally rely on the characteristics C-1 to C-5 for the design of code-reuse defenses; we define the following goals for COOP accordingly:

- G-1. Do not expose the characteristics C-1–C-5.
- G-2. Exhibit control flow and data flow similar to those of benign C++ code execution.
- G-3. Be widely applicable to C++ applications.
- G-4. Achieve Turing-completeness under realistic conditions.

6.1.2 Attacker Model

In general, code-reuse attacks against C++ applications oftentimes start by hijacking a C++ object and its `vptr`. Attackers achieve this by exploiting a spatial or temporal memory corruption vulnerability, such as an overflow in a buffer adjacent to a C++ object or a use-after-free condition. When the application subsequently invokes a virtual function on the hijacked object, the attacker-controlled `vptr` is dereferenced and a `vfptr` is loaded from a memory location of the attacker's choice. At this point, the attacker effectively controls the *program counter* (`rip` in x86-64) of the corresponding thread in the target application. Generally, for code-reuse attacks, controlling the program counter is one of the two basic requirements. The other one is gaining (partial) knowledge of the layout of the target application's address space. Depending on the context, there may exist different techniques to achieve this [Snow et al. 2013, Hund et al. 2013, Bittau et al. 2014, Seibert et al. 2014].

For COOP we assume that attackers control a C++ object with a `vptr` and that they can infer the base address of this object or another auxiliary buffer of sufficient size under their control. Further, they need to be able to infer the base addresses of a set of C++ modules whose binary layouts are (partly) known to them. For instance, in practice, knowledge on the base address of a single publicly available C++ library in the target address space can be sufficient.

These assumptions conform to the attacker settings of most defenses against code-reuse attacks. In fact, many of these defenses assume far more powerful adversaries that are, e.g., able to read and write large (or all) parts of an application's address space with respect to page permissions. As such, our attack is applicable to many defenses; we discuss the implications of COOP in Section 6.6.

6.2 Counterfeit Object-Oriented Programming

Every COOP attack starts by hijacking one of the target application's C++ objects. We call this the *initial object*. Up to the point where the attacker controls the program counter, a COOP attack does not deviate much from other code-reuse attacks: in a conventional ROP attack, the attackers typically exploit their control over the program counter to first manipulate the stack pointer and to subsequently execute a chain of short, return-terminated gadgets. In contrast, in COOP, virtual functions existing in an application are repeatedly invoked on counterfeit C++ objects carefully arranged by the attackers.

6.2.1 Counterfeit Objects

Typically, a counterfeit object carries an attacker-chosen `vptr` and a few attacker-chosen data fields. Counterfeit objects are *not* created by the target application, but are injected in bulk by the attacker. Whereas the *payload* in a ROP-based attack is typically composed of fake return addresses interleaved with additional data, in a COOP attack, the payload consists of counterfeit objects and possibly additional data. Similar to a conventional ROP payload, the COOP payload containing all counterfeit objects is typically written as one coherent chunk to a single attacker-controlled memory location.

6.2.2 Vfgadgets

We call the virtual functions used in a COOP attack *vfgadgets*. As for other code-reuse attacks, the attacker identifies useful vfgadgets in an application prior to the actual attack through source code analysis or reverse engineering of binary code. Even when source code is available, it is necessary to determine the actual object

Table 6.1 Overview of COOP vfgadget Types That Operate on Object Fields or Arguments

vfgadget Type	Purpose
General-Purpose Types	
ML-G	The main loop; iterate over container of pointers to counterfeit object and invoke a virtual function on each such object.
ARITH-G	Perform arithmetic or logical operation.
W-G	Write to chosen address.
R-G	Read from chosen address.
INV-G	Invoke C-style function pointer.
W-COND-G	Conditionally write to chosen address. Used to implement conditional branching.
Auxiliary Types	
ML-ARG-G	Execute vfgadgets in a loop and pass a field of the <i>initial object</i> to each as an argument.
W-SA-G	Write to address pointed to by first argument. Used to write to <i>scratch area</i> .
MOVE-SP-G	Decrease/increase stack pointer.
LOAD-R64-G	Load x86-64 argument register <code>rdx</code> , <code>r8</code> , or <code>r9</code> with value.

layout of a vfgadget's class on binary level as the compiler may remove or pad certain fields. Only then is the attacker able to inject compatible counterfeit objects.

We identified a set of vfgadget types that allows implementation of expressive (and Turing-complete) COOP attacks in x86-32 and x86-64 environments. These types are listed in Table 6.1. In the following, we gradually motivate our choice of vfgadget types based on typical code examples. These examples revolve around the simple C++ classes `Student`, `Course`, and `Exam` shown in Figure 6.1, which reflect *some* common code patterns that we found to induce useful vfgadgets; in practice, there are many other ways that C++ code can produce vfgadgets. We first walk through the creation of a COOP attack code that writes to a dynamically calculated address; along the way, we introduce COOP's integral concepts of the main loop, counterfeit `vptrs`, and overlapping counterfeit objects. Afterward, we explain extended concepts in COOP for passing arguments to vfgadgets, calling `api` functions, and implementing conditional branches and loops.

The reader might be surprised to find more C++ code listings than actual assembly code in the following. This is owed to the fact that most of our vfgadget types


```

class Student {
public:
    virtual void incCourseCount() = 0;
    virtual void decCourseCount() = 0;
};

class Course {
private:
    Student **students;
    size_t nStudents;
public:
    /* ... */
    virtual ~Course() {
        for (size_t i = 0; i < nStudents; i++)
            students[i]->decCourseCount();    ML-G
        delete students;
    }
};

```

Figure 6.1 Example for ML-G: the virtual destructor of the class `Course` invokes a virtual function on each object pointer in the array `students`.

are solely defined by their high-level C++ semantics rather than by the side effects of their low-level assembly code. These types of vfgadgets are thus likely to survive compiler changes or even the transition to a different operating system or architecture. In the cases where assembly code is given, it is the output of the Microsoft Visual C++ compiler (MSVC) version 18.00.30501 that is shipped with Microsoft Visual Studio 2013.

6.2.3 The Main Loop

To repeatedly invoke virtual functions without violating goals [G-1](#) and [G-2](#), every COOP program essentially relies on a special *main loop vfgadget* (ML-G). The definition of an ML-G is as follows:

Definition 6.1 A virtual function that iterates over a container (e.g., a C-style array or a vector) of pointers to C++ objects and invokes a virtual function on each of these objects.

Virtual functions that qualify as ML-G are common in C++ applications. Consider, for example, the code in [Figure 6.1](#): the class `Course` has a field `students` that points to a C-style array of pointers to objects of the abstract base class `Stu-`

dent. When a `Course` object is destroyed (e.g., via `delete`), the virtual destructor `Course::~~Course` is executed and each `Student` object is informed via its virtual function `decCourseCount()` that one of the courses it was subscribed to does not exist anymore. Note that it is common practice to declare a virtual destructor when a C++ class has virtual functions.

6.2.4 Layout of the Initial Object

The attacker shapes the initial object to resemble an object of the class of the ML-G. For our example ML-G `Course::~~Course`, the initial object should look as depicted in Figure 6.2: its `vptr` is set to point into an existing vtable that contains a reference to the ML-G such that the first vcall under attacker control leads to the ML-G. In contrast, in a ROP-based attack, this first vcall under attacker control typically leads to a gadget moving the stack pointer to attacker-controlled memory. The initial

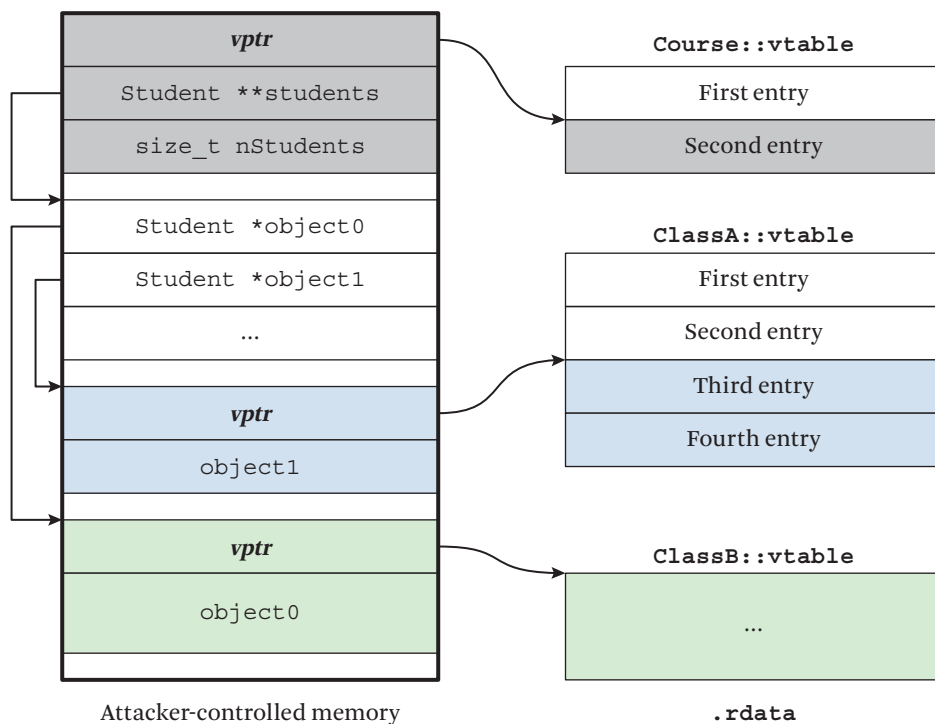


Figure 6.2 Basic layout of attacker-controlled memory (left) in a COOP attack using the example ML-G `Course::~~Course`. The initial object (dark gray, top left) contains two fields from the class `Course`. Arrows indicate a *points-to* relation.

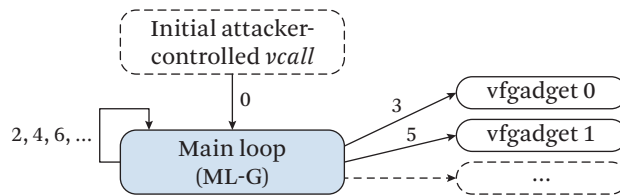


Figure 6.3 Schematic control flow in a COOP attack; transitions are labeled according to the order they are executed.

object contains a subset of the fields of the class of the ML-G; i.e., all data fields required to make the ML-G work as intended. For our example ML-G, the initial object contains the fields `students` and `nStudents` of the class `Course`; the field `students` is set to point to a C-style array of pointers to counterfeit objects (*object0* and *object1* in Figure 6.2), and `nStudents` is set to the total number of counterfeit objects. This makes the `Course::~Course` ML-G invoke a `vfgadget` of the attacker’s choice for each counterfeit object. Note how the attacker controls the `vptr` of each counterfeit object. Figure 6.3 schematically depicts the control-flow transitions in a COOP attack.

6.2.5 Counterfeit Vptrs

The control flow and data flow in a COOP attack should resemble those of a regular C++ program. Hence, we avoid introducing fake vtables and reuse existing ones instead. Ideally, the `vptrs` of all counterfeit objects should point to the *beginning* of existing vtables. Depending on the target application, though, it can be difficult to find vtables with a useful entry at the offset that is fixed for a given `vcall` site. Consider, for example, our ML-G from Figure 6.1: counterfeit objects are treated as instances of the abstract class `Student`. For each counterfeit object, the second entry—corresponding to `decCourseCount()`—in the supplied vtable is invoked. (The first entry corresponds to `incCourseCount()`.) Here, a COOP attack would ideally only use `vfgadgets` that are the second entry in an existing vtable. Naturally, this significantly shrinks the set of available `vfgadgets`.

This constraint can be sidestepped by relaxing goal G-2 and letting `vptrs` of counterfeit objects point not necessarily to the exact beginning of existing vtables but to certain positive or negative offsets, as shown for *object1* in Figure 6.2. When such *counterfeit vptrs* are used, any available virtual function can be invoked from a given ML-G. For example, to invoke the fourth entry in a certain vtable under the given ML-G, the attacker makes a counterfeit object’s `vptr` point to the third entry

of that vtable, as Figure 6.2 depicts for *object1* and `ClassA::vtable`. The vcall in the ML-G then interprets the fourth entry of that vtable as the second entry of a Student vtable.

6.2.6 Overlapping Counterfeit Objects

So far we have shown how, given an ML-G, an arbitrary number of virtual functions (vfgadgets) can be invoked while control flow and data flow resemble that of the execution of benign C++ code.

Two exemplary vfgadgets of types ARITH-G (arithmetic) and W-G (writing to memory) are given in Figure 6.4: in `Exam::updateAbsoluteScore()` the field `score` is set to the sum of three other fields; in `SimpleString::set()` the field `buffer` is used as a destination pointer in a write operation. In conjunction, these

```

class Exam {
private:
    size_t scoreA, scoreB, scoreC;
public:
    /* ... */
    char *topic;
    size_t score;
    virtual void updateAbsoluteScore() {
        score = scoreA + scoreB + scoreC;
    }
    virtual float getWeightedScore() {
        return (float)(scoreA*5+scoreB*3+scoreC*2) / 10;
    }
};

struct SimpleString {
    char* buffer;
    size_t len;
    /* ... */
    virtual void set(char* s) {
        strncpy(buffer, s, len);
    }
};

```

Figure 6.4 Examples for ARITH-G, LOAD-R64-G, and W-G; for simplification, the native integer type `size_t` is used.

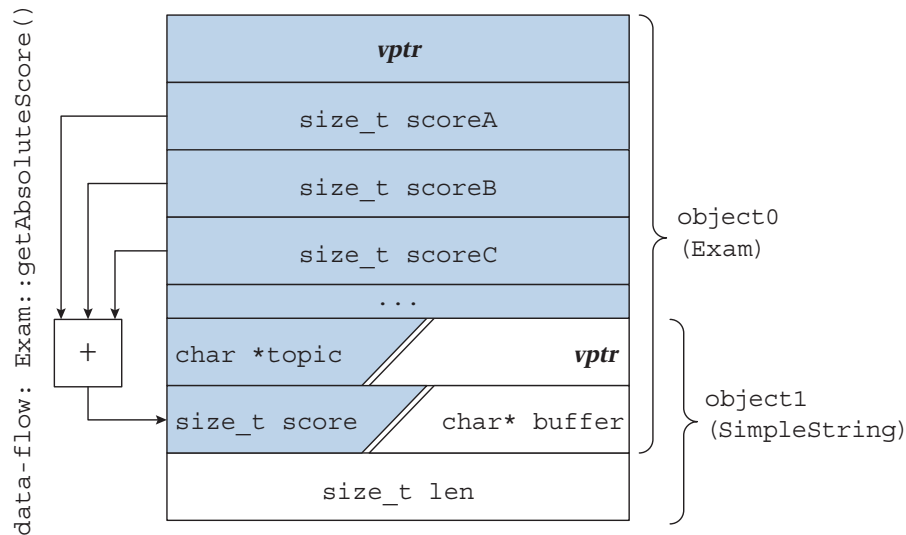


Figure 6.5 Overlapping counterfeit objects of types Exam and SimpleString

two vfgadgets can be used to write attacker-chosen data to a dynamically calculated memory address. For this, two *overlapping* counterfeit objects are needed, and their alignment is shown in Figure 6.5.

The key idea here is that the fields `score` in `object0` and `buffer` in `object1` share the same memory. This way, the result of the summation of the fields of `object0` in `Exam::updateAbsoluteScore()` is written to the field `buffer` of `object1`. For example, `object0.scoreA` could hold a previously determined base pointer (*base_ptr*) to a memory region, `object0.scoreB` could hold a fixed offset into that region, and `object0.scoreC` would simply be set to 0. The write operation in `SimpleString::set()` would then use

$$\text{object0.scoreA} + \text{object0.scoreB} + \text{object0.scoreC} = \text{base_ptr} + \text{offset}$$

as the destination pointer in `strncpy()`. Note how here, technically, also `object0.topic` and `object1.vptr` overlap. As the attackers do not use `object0.topic`, this is not a problem and they can simply make the shared field carry `object1.vptr`.

Of course, in our example, the attackers would likely wish to control not only the *destination address* of the write operation through `object1.buffer` but also the *source address*. For this, they need to be able to set the argument for the vfgadget `SimpleString::set()`. How this can be achieved in COOP is described next.

6.2.7 Passing Arguments to Vfgadgets

The overlapping of counterfeit objects is an important concept in COOP. It allows for data to flow between vfgadgets through object fields regardless of compiler settings or calling conventions. Unfortunately, we found that useful vfgadgets that operate exclusively on object fields are rare in practice. In fact, most vfgadgets we use in our real-world exploits (see Section 6.5) operate on both fields and arguments as is the case for `SimpleString::set()`.

Due to divergent default calling conventions, we describe different techniques for passing arguments to vfgadgets for x86-64 and x86-32 in the following. We begin with x86-64 and not with x86-32 as the technique for the former is simpler.

Approach for Windows x86-64. In the default x86-64 calling convention on Windows, the first four (non-floating-point) arguments to a function are passed through the registers `rcx`, `rdx`, `r8`, and `r9` [Microsoft Developer Network 2017]. In case there are more than four arguments, the additional arguments are passed over the stack. For C++ code, the `this`-ptr is passed through `rcx` as the first argument. All four argument registers are defined to be caller saved, regardless of the actual number of arguments a callee takes. Accordingly, virtual functions often use `rdx`, `r8`, and `r9` as scratch registers and do not restore or clear them on returning. This circumstance makes passing arguments to vfgadgets simple on x86-64: first, a vfgadget is executed that loads one of the corresponding counterfeit object's fields into `rdx`, `r8`, or `r9`. Next, a vfgadget is executed that interprets the contents of these registers as arguments.

We refer to vfgadgets that can be used to load argument registers as LOAD-R64-G. For the x86-64 argument passing concept to work, an ML-G is required that itself does not pass arguments to the invoked virtual functions/vfgadgets. Of course, the ML-G must also not modify the registers `rdx`, `r8`, and `r9` between such invocations. In our example, the attackers can control the source pointer `s` of the write operation (namely, `strncpy()`) by invoking a LOAD-R64-G that loads `rdx` before `SimpleString::set()`.

As an example for a LOAD-R64-G, consider `Exam::getWeightedScore()` from Figure 6.4; MSVC compiles this function to the assembly code shown in Listing 6.1.

In condensed form, this LOAD-R64-G provides the following useful semantics to the attackers:

$$\begin{aligned} \text{rdx} &\leftarrow 3 \cdot [\text{this} + 10h] \\ \text{r8} &\leftarrow [\text{this} + 18h] \\ \text{r9} &\leftarrow 3 \cdot [\text{this} + 18h] + 2 \cdot [\text{this} + 10h] \end{aligned}$$

```

mov     rax, qword ptr [rcx+10h]
mov     r8, qword ptr [rcx+18h]
xorps  xmm0, xmm0
lea     rdx, [rax+rax*2]
mov     rax, qword ptr [rcx+8]
lea     rcx, [rax+rax*4]
lea     r9, [rdx+r8*2]
add     r9, rcx
cvtsi2ss  xmm0, r9
addss  xmm0, dword ptr [__real0]
divss  xmm0, dword ptr [__real1]
ret

```

Listing 6.1 x86-64 assembly code produced by MSVC for `Exam::getWeightedScore()` (example for a LOAD-R64-G).

Thus, by carefully choosing the fields at offsets `10h` and `18h` from the `this-ptr` of the corresponding counterfeit object, the attackers can write arbitrary values to the registers `rdx`, `r8`, and `r9`. Note that the attackers here also control the registers `rax` and `rcx`. However, this is of no value to them as `rax` is not an argument register and is thus virtually never read without having been initialized before in a function; and `rcx` is necessarily always updated by the ML-G to point to the next counterfeit object when a `vfgadget` returns.

In summary, to control the source pointer in the writing operation in `SimpleString::set()`, the attackers would first invoke `Exam::getWeightedScore()` for a counterfeit object carrying the desired source address divided by 3 at offset `10h`. This would load the desired source address to `rdx`, which would next be interpreted as the argument `s` in the `vfgadget SimpleString::set()`.

Other 64-bit Platforms. In the default x86-64 C++ calling convention used by GCC [Matz et al. 2013], e.g., on Linux, the first six arguments to a function are passed through registers, instead of only the first four arguments. In theory, this should make COOP attacks simpler to create on Linux x86-64 than on Windows x86-64, as two additional registers can be used to pass data between `vfgadgets`. In practice, during the creation of our example exploits (see Section 6.5), we did not experience big differences between the two platforms.

Although we did not conduct experiments on RISC platforms, such as ARM or MIPS, we expect that our x86-64 approach directly extends to these because in RISC calling convention arguments are also primarily passed through registers.

```

class Student2 {
private:
    std::list<Exam> exams;
public:
    /* ... */
    virtual void subscribeCourse(int id) { /* ... */ }
    virtual void unsubscribeCourse(int id) { /* ... */ }

    virtual bool getLatestExam(Exam &e) {
        if (exams.empty()) return false;           W-SA-G
        e = exams.back();
        return true;                               W-COND-G
    }
};

class Course2 {
private:
    Student2 **students;
    size_t nStudents;
    int id;
public:
    /* ... */

    virtual ~Course2() {
        for (size_t i = 0; i < nStudents; i++)
            students[i]->unsubscribeCourse(id);
        delete students;
    }                                           ML-ARG-G
};

```

Figure 6.6 Examples for W-SA-G, W-COND-G, and ML-ARG-G.

Approach for Windows x86-32. The standard x86-32 C++ calling convention on Windows is *thiscall* [Microsoft Developer Network 2017]: all regular arguments are passed over the stack whereas the this-ptr is passed in the register ecx; the callee is responsible for removing arguments from the stack. Thus, the described approach for x86-64 does not work for x86-32. In our approach for Windows x86-32, contrary to x86-64, we rely on a *main loop* (ML-G) that *passes* arguments to vfgadgets. More precisely, a 32-bit ML-G should pass one field of the *initial object* as an argument to each vfgadget. In practice, any number of arguments may work; for brevity reasons, we only discuss the simplest case of one argument here. We call this field the *argument field* and refer to this variant of ML-G as ML-ARG-G. For an example of an ML-ARG-G, consider the virtual destructor of the class Course2 in Figure 6.6: the field id is passed as an argument to each invoked virtual function.


```

1  push  ebp
2  mov   ebp, esp
3  cmp   dword ptr [ecx+8], 0
4  jne   copyExam
5  xor   al, al
6  pop   ebp
7  ret   4
8  copyExam:
9  mov   eax, dword ptr [ecx+4]
10 mov   ecx, dword ptr [ebp+8]
11 mov   edx, dword ptr [eax+4]
12 mov   eax, dword ptr [edx+0Ch]
13 mov   dword ptr [ecx+4], eax
14 mov   eax, dword ptr [edx+10h]
15 mov   dword ptr [ecx+8], eax
16 mov   eax, dword ptr [edx+14h]
17 mov   dword ptr [ecx+0Ch], eax
18 mov   eax, dword ptr [edx+18h]
19 mov   dword ptr [ecx+10h], eax
20 mov   al, 1
21 pop   ebp
22 retn  4

```

Listing 6.2 Optimized x86-32 assembly code produced by MSVC for `Student2::getLatestExam()`.

Given such an ML-ARG-G, the attacker can employ one of the following two approaches to pass chosen arguments to vfgadgets:

1. Fix the *argument field* to point to a writable *scratch area*.
2. Dynamically rewrite the *argument field*.

In approach 1, the attacker relies on vfgadgets that interpret their first argument not as an immediate value but as a *pointer* to data. Consider, for example, the virtual function `Student2::getLatestExam()` from Figure 6.6 that copies an Exam object; MSVC produces the optimized x86-32 assembly code shown in Listing 6.2 for the function.

In condensed form, lines 9–22 of the assembly code provide the following semantics:

```

[arg0 + 4] ← [[[this + 4] + 4] + Ch]
           ← [[[this + 4] + 4] + 10h]
           ← [[[this + 4] + 4] + 14h]
           ← [[[this + 4] + 4] + 18h]

```

Note that for approach 1, *arg0* always points to the *scratch area*. Accordingly, this vfgadget allows the attacker to copy 16 bytes (corresponding to the four 32-bit fields of Exam) from the attacker-chosen address $[[this + 4] + 4] + Ch$ to the scratch area. We refer to this type of vfgadget that writes attacker-controlled fields to the scratch area as W-SA-G. Using `Student2::getLatestExam()` as W-SA-G in conjunction with an ML-ARG-G allows the attacker, for example, to pass a string of up to 16 characters as the first argument to the vfgadget `SimpleString::set()`.

In approach 2, the argument field of the initial object is not fixed as in approach 1. Instead, it is dynamically rewritten during the execution of a COOP attack. This allows the attacker to pass *arbitrary* arguments to vfgadgets, as opposed to a *pointer to arbitrary data* for approach 1. For this approach, naturally, a usable W-G is required. As stated above, we found vfgadgets working solely with fields to be rare. Hence, the attacker would typically initially follow approach 1 and implement 2-style argument writing on top of that when required.

Passing Multiple Arguments and Balancing the Stack. So far, we have described how a single argument can be passed to each vfgadget using an ML-ARG-G main loop gadget on Windows x86-32. Naturally, it can be desirable or necessary to pass more than one argument to a vfgadget. Doing so is simple: the ML-ARG-G pushes one argument to each vfgadget. In case a vfgadget does not expect any arguments, the pushed argument remains on the top of the stack even after the vfgadget returned. This effectively moves the stack pointer permanently one slot up, as depicted in Figure 6.7(c). This technique allows the attacker to gradually “pile up” arguments on the stack as shown in Figure 6.7(d) before invoking a vfgadget that expects multiple arguments. This technique only works for ML-ARG-Gs that use `ebp` and not `esp` to access local variables on the stack (i.e., no *frame-pointer omission*) as otherwise the stack frame of the ML-ARG-G is destroyed.

Analogously to how vfgadgets without arguments can be used to move the stack pointer *up* under an ML-ARG-G, vfgadgets with more than one argument can be used to move the stack pointer *down*, as shown in Figure 6.7(b). This may be used to compensate for vfgadgets without arguments or to manipulate the stack. We refer to vfgadgets with little or no functionality that expect less or more than one

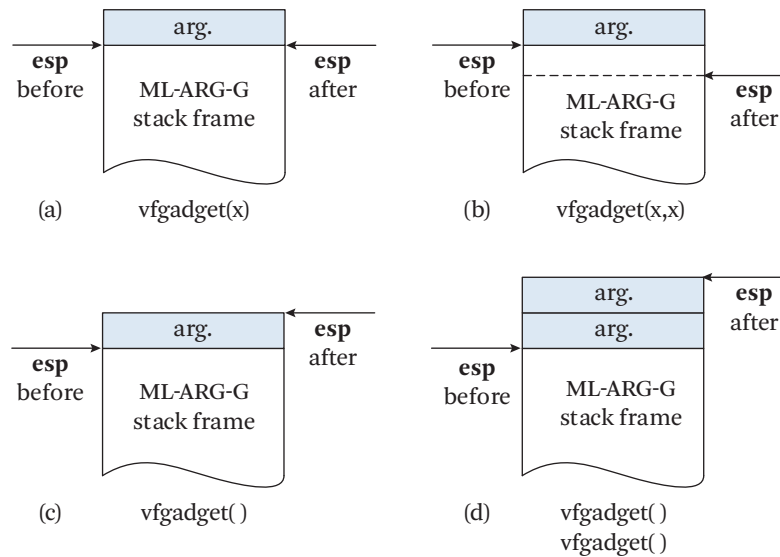


Figure 6.7 Examples for stack layouts *before* and *after* invoking `vfgadgets` under an ML-ARG-G (*thiscall* calling convention). The stack grows upward. (a) `vfgadget` with one argument: the stack is balanced. (b) `vfgadget` with two arguments: esp is moved down. (c) `vfgadget` without arguments: esp is moved up. (d) Two `vfgadgets` without arguments: two arguments are piled up.

argument as MOVE-SP-Gs. Ideally, a MOVE-SP-G is an empty virtual function that just adjusts the stack pointer.

Note that the described technique for passing multiple arguments to `vfgadgets` in 32-bit environments can also be used to pass more than three arguments to `vfgadgets` in 64-bit environments. (Remember that only the first three arguments to a virtual function are passed through registers on x86-64.) For this, the attacker would typically invoke an ML-ARG-G from the main ML-G to write arguments to the stack and invoke the `vfgadget` in question.

Other 32-bit Platforms. The default x86-32 C++ calling convention used by GCC, e.g., on Linux, is not *thiscall* but *cdecl* [Microsoft Developer Network 2017]: all arguments including the `this`-ptr are passed over the stack; instead of the callee, the caller is responsible for cleaning the stack. The described technique of “piling up” arguments thus does not apply to GCC-compiled (and compatible) C++ applications on Linux x86-32 and other POSIX x86-32 platforms. Instead, for these platforms, we propose using ML-ARG-Gs that pass not one but many controllable arguments

to vfgadgets. Conceptually, passing too many arguments to a function does not corrupt the stack in the *cdecl* calling convention. Alternatively, ML-ARG-Gs could be switched during an attack, depending on which arguments to a vfgadget need to be controlled.

6.2.8 Calling API Functions

The ultimate goal of code-reuse attacks is typically to pass attacker-chosen arguments to critical API functions or system calls, e.g., WinAPI functions such as `WinExec()` or `VirtualProtect()`. We identified the following ways to call a WinAPI function in a COOP attack:

1. Use a vfgadget that legitimately calls the WinAPI function of interest.
2. Invoke the WinAPI function like a virtual function from the COOP main loop.
3. Use a vfgadget that calls a C-style function pointer.

While approach [1](#) may be practical in certain scenarios and for certain WinAPI functions, it is unlikely to be feasible in the majority of cases. For example, virtual functions that call `WinExec()` should be close to non-existent.

Approach [2](#) is simple to implement: a counterfeit object can be crafted whose `vptr` does not point to an actual vtable but to the *import table* (IAT) or the *export table* (EAT) [[Russeinovich et al. 2012](#)] of a loaded module such that the ML-G invokes the WinAPI function as a virtual function. Note that IATs, EATs, and vtables are all arrays of function pointers typically lying in read-only memory; they are thus in principle compatible data structures. As simple as it is, the approach has two important drawbacks: (i) it goes counter to our goal [G-2](#), as a C function is called at a vcall site without a legitimate vtable being referenced; and (ii) for x86-64, the `this-ptr` of the corresponding counterfeit object is always passed as the first argument to the WinAPI function due to the given C++ calling convention. This circumstance, for example, effectively prevents the passing of a useful command line to `WinExec()`: the function `WinExec()` expects the pointer to an ASCII command line to be the first argument. In case a `this-ptr` is passed as the first argument, the corresponding `vptr` is interpreted as a command line, which is likely not useful. However, this can be different for other WinAPI functions. For example, calling `VirtualProtect()` with a `this-ptr` as the first argument still allows the attacker to mark the memory of the corresponding counterfeit object as *executable*. Note that `VirtualProtect()` changes the memory access rights for a memory region pointed to by the first argument. Arguments other than the first one can be passed as described in [Section 6.2.7](#)

```

class GuiButton {
private:
    int id;
    void(*callbackClick)(int, int, int);
public:
    void registerCbClick(void(*cb)(int, int, int)) {
        callbackClick = cb;
    }
    virtual void clicked(int posX, int posY) {
        callbackClick(id, posX, posY);
    }
};

```

INV-G

Figure 6.8 Example for an INV-G: `clicked` invokes a field of `GuiButton` as a C-style function pointer.

for x86-64. For x86-32, *all* arguments can be passed using the technique from Section 6.2.7.

For approach 3, a special type of vfgadget is required: a virtual function that calls a C-style function pointer with non-constant arguments. We refer to this type of vfgadget as an INV-G. An example is given in Figure 6.8: the virtual function `GuiButton::clicked()` invokes the field `GuiButton::callbackClick` as a C-style function pointer. This particular vfgadget allows for the invocation of arbitrary WinAPI functions with at least three attacker-chosen arguments. Note that, depending on the actual assembly code of the INV-G, a fourth argument could possibly be passed through `r9` for x86-64. Additional stack-bound arguments for x86-32 and x86-64 may also be controllable depending on the actual layout of the stack.

Calling WinAPI functions through INV-Gs should generally be the technique of choice, as this is more flexible than approach 1 and stealthier than 2. An INV-G also enables seemingly legit transfers from C++ to C code (e.g., to `libc`) in general. On the downside, we found INV-Gs to be relatively rare overall. For our real-world example exploits discussed in Section 6.5, though, we could always select from multiple suitable ones.

6.2.9 Implementing Conditional Branches and Loops

Up to this point, we have described all the building blocks required to practically mount COOP code-reuse attacks. Since we aim for COOP not only to be stealthy but

also to be *Turing-complete* under realistic conditions (goal G-4), we now describe the implementation of *conditional branches* and *loops* in COOP.

In COOP, the *program counter* is the index into the container of counterfeit object pointers. The program counter is incremented for each iteration in the ML-G's main loop. The program counter may be a plain integer index, as in our exemplary ML-G `Course::~Course`, or a more complex data structure, such as an iterator object for a C++ linked list. Implementing a conditional branch in COOP is generally possible in two ways: through (i) a conditional increment/decrement of the program counter or (ii) a conditional manipulation of the next-in-line counterfeit object pointers in the container. Both can be implemented given a conditional write vfgadget, which we refer to as W-COND-G. An example for this vfgadget type is again `Student2::getLatestExam()` from Figure 6.6. As can be seen in lines 3–7 of the function's assembly code in Listing 6.2, the controllable write operation is only executed in case $[this - ptr + 8] \neq 0$. With these semantics, the attacker can rewrite the COOP program counter or upcoming pointers to counterfeit objects under the condition that a certain value is not null. In case the program counter is stored on the stack (e.g., in the stack frame of the ML-G) and the address of the stack is unknown, the technique for moving the stack pointer described in Section 6.2.7 can be used to rewrite it.

Given the ability to conditionally rewrite the program counter, implementing loops with an exit condition also becomes possible.

6.3 Loopless Counterfeit Object-Oriented Programming

The basic COOP code-reuse attack technique described in Section 6.2 inherently relies on a *main loop* vfgadget (ML-G). Accordingly, one can think of different possible (partial) defenses against COOP that make ML-Gs unavailable to an attacker or at least complicated to misuse. Yet our observation is that the COOP concept is not necessarily bound to ML-Gs. In the following, we describe two refined versions of COOP that do not require ML-Gs and emulate the original *main loop* through *recursion* and *loop unrolling*. For brevity, we only discuss the x86-64 platform.

Generally, all semantics that can be expressed programmatically through loops can also be expressed through recursive functions. This naturally also applies to COOP's main loop. We identified a certain code pattern that is commonly found in virtual functions and is especially common within virtual destructors. This code pattern can be misused to emulate an ML-G by means of recursion. We refer to a virtual function that exhibits this pattern as a *REC-G* (short for *recursion vfgadget*).

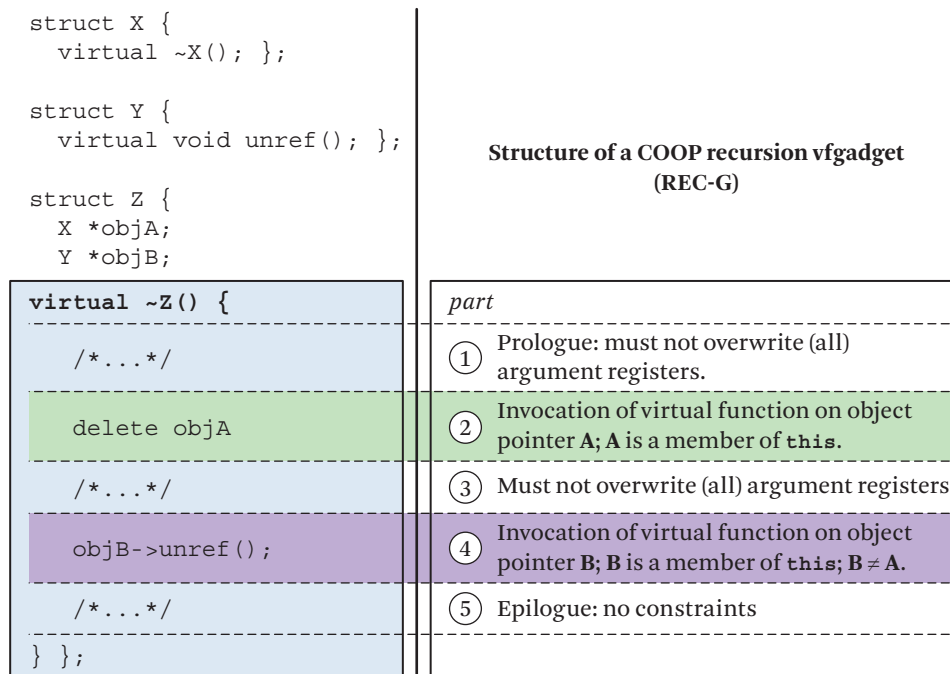


Figure 6.9 Example code (left) and general structure (right) of a REC-G

For an example of a REC-G, consider the C++ code in Figure 6.9: `Z::~~Z()` is a typical (virtual) destructor. It deletes the object `objA` and removes a reference to `objB`. Consequently, a virtual function is invoked on both `objA` and `objB`. In case `Z::~~Z()` is invoked on an adversary-controlled counterfeit object, the adversary effectively controls the pointers `*objA` and `*objB`. The adversary can make these pointers point to injected counterfeit objects.

Accordingly, `Z::~~Z()` can be misused by an adversary to make two consecutive COOP-style vfgadget invocations. This, in turn, effectively enables the adversary to invoke an *arbitrary number* of vfgadgets, if the counterfeit object `objB` is shaped such that `Z::~~Z()` is recursively invoked. The left side of Figure 6.10 schematically depicts the counterfeit object layouts that are required for this: for each regular counterfeit object, one additional *auxiliary counterfeit object* is required that resembles an object of class `Z`. Each auxiliary counterfeit object's `*objB` points to the next auxiliary counterfeit object (pointers ② and ④ in Figure 6.10), whereas each `*objA` points to a regular counterfeit object that corresponds to a certain vfgadget (point-

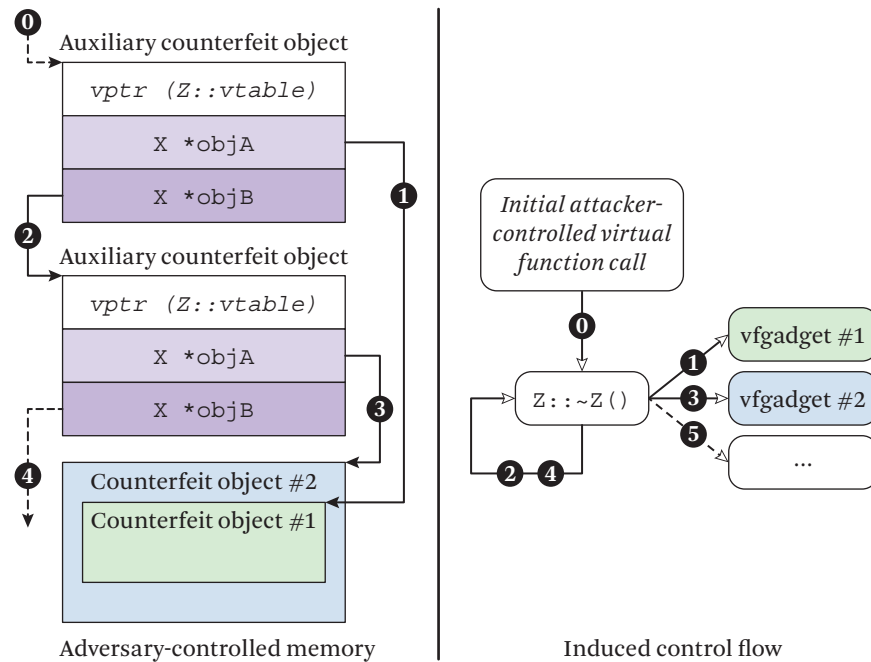


Figure 6.10 Schematic layout of adversary-controlled memory with pointers (left) and control-flow transitions (right) in a recursion-based COOP attack using `Z::~Z()` as a REC-G.

ers ① and ③). The right side of Figure 6.10 shows the resulting adversary-induced control flow.

We note that not only destructors but any virtual function may qualify as a REC-G. The required abstract structure of a REC-G is shown on the right side of Figure 6.9: as discussed, at least two invocations of virtual functions on distinct and adversary-controlled object pointers are required (parts ② and ④); the code before these invocations (parts ① and ③) must not write to registers that are required for passing arguments to vfgadgets. As per definition, C++ destructors do not receive any explicit arguments, parts ① and ③ are particularly likely not to write to argument registers if parts ② and ④ both comprise a `delete` statement.

Unrolled COOP. Given a virtual function with not only *two* consecutive virtual function invocations (like a REC-G) but *many*, it is also possible to mount an *unrolled* COOP attack that does not rely on a loop or recursion. This COOP variant is detailed in a paper by Crane et al. [2015].

6.4 A Framework for Counterfeit Object-Oriented Programming

Implementing a COOP attack against a given application is a three-step process: (i) identification of vfgadgets, (ii) implementation of attack semantics using the identified vfgadgets, and (iii) arrangement of possibly overlapping counterfeit objects in a buffer. Since the individual steps are cumbersome and hard to perform by hand, we created a framework in the Python scripting language that automates steps (i) and (iii). This framework greatly facilitated the development of our example exploits for Internet Explorer and Chromium (see Section 6.5). In the following, we provide an overview of our implementation.

6.4.1 Finding Vfgadgets Using Basic Symbolic Execution

For the identification of useful vfgadgets in an application, our *vfgadget searcher* relies on binary code only and optionally debug symbols. Binary x86-32 C++ modules are disassembled using the popular *Interactive Disassembler* (IDA). Each virtual function in a C++ module is considered a potential vfgadget. The searcher statically identifies all vttables in a C++ module using debug symbols or, if these are not available, a set of simple but effective heuristics is applied. Akin to other work [Prakash et al. 2015, Zhang et al. 2015], our heuristics consider each address-taken array of function pointers a potential vttable. The searcher examines all identified virtual functions whose number of basic blocks does not exceed a certain limit. In practice, we found it sufficient and convenient to generally only consider virtual functions with one or three basic blocks as potential vfgadgets; the only exception being ML-Gs and ML-ARG-Gs, which due to the required loop often consist of more basic blocks. Using short vfgadgets is favorable as their semantics are easier to evaluate automatically and they typically exhibit fewer unwanted side effects. Including long vfgadgets can, however, be necessary to fool heuristics-based code-reuse attack detection approaches (see Section 6.6).

The searcher summarizes the semantics of each basic block in a vfgadget in *single static assignment* (SSA) form. These summaries reflect the I/O behavior of a basic block in a compact and easy-to-analyze form. The searcher relies on the *backtracking* feature of the METASM binary code analysis toolkit [Guillot and Gazet 2010], which performs symbolic execution on the basic-block level. An example of a basic-block summary as used by our searcher is provided in the listed semantics for the second basic block of `Exam::getWeightedScore()` in Section 6.2.7. To identify useful vfgadgets, the searcher applies filters on the SSA representation of the potential vfgadgets' basic blocks. For example, the filter "*left side of assignment must dereference any argument register; right side must dereference the this-ptr*" is

useful for identifying 64-bit W-Gs; the filter “*indirect call independent of [this]*” is useful for finding INV-Gs; and the filter “*looped basic block with an indirect call dependent on [this] and a non-constant write to [esp-4]*” can in turn be used to find 32-bit ML-ARG-Gs.

6.4.2 Aligning Overlapping Objects Using an SMT Solver

Each COOP “program” is defined by the order and positioning of its counterfeit objects, each of which corresponds to a certain vfgadget. As described in Section 6.2.6, the overlapping of counterfeit objects is an integral concept of COOP; it enables immediate data flows between vfgadgets through fields of counterfeit objects. Manually obtaining the alignment of overlapping counterfeit objects right on the binary level is a time-consuming and error-prone task. Hence, we created a COOP *programming environment* that automatically, if possible, correctly aligns all given counterfeit objects in a fixed-size buffer. In our programming environment, the “programmer” defines counterfeit objects and labels. A label may be assigned to any byte within a counterfeit object. When bytes within different objects are assigned the same label, the programming environment takes care that these bytes are mapped to the same location in the final buffer, while assuring that bytes with different labels are mapped to distinct locations. Fields without labels are in turn guaranteed never to overlap. These constraints are often satisfiable, as actual data within counterfeit objects is typically sparse.

For example, the counterfeit object *A* may only contain its *vptr* (at relative offset +0), an integer at the relative offset +16, and have the label *X* for its relative offset +136; the counterfeit object *B* may only contain its *vptr* and have the same label *X* for its relative offset +8. Here, the object *B* fits comfortably and without conflict inside *A* such that *B* +8 maps to the same byte as *A* +136.

Our programming environment relies on the Z3 SMT solver [de Moura and Bjørner 2008] to determine the alignment of all counterfeit objects within the fixed-size buffer such that, if possible, all label-related constraints are satisfied. At the baseline, we model the fixed-size buffer as an *array* mapping integer indexes to integers in Z3. To prevent unwanted overlap, for each byte in each field, we add a *select* constraint [de Moura and Bjørner 2009] in Z3 of the form

$$\text{select}(\text{offset-obj} + \text{reloffset-byte}) = \text{id-field}$$

where *offset-obj* is an integer variable to be determined by Z3 and *reloffset-byte* and *id-field* are constant integers that together uniquely identify each byte. For each desired overlap (e.g., between objects *A* and *B* using label *X*), we add a constraint

of the form

$$\text{offset-objA} + \text{reoffset}(A, X) = \text{offset-objB} + \text{reoffset}(B, X)$$

where *offset-objA* and *offset-objB* are integers to be determined by Z3 and *reoffset(A, X) = 136* and *reoffset(B, X) = 8* are constants.

In the programming environment, for convenience, symbolic pointers to labels can be added to counterfeit objects. Symbolic pointers are automatically replaced with concrete values once the offsets of all labels are determined by Z3. This way, multiple levels of indirection can be implemented conveniently.

An example of a vfgadget that reads attacker-controlled data through multiple levels of indirection was provided in the `W-SA-G Student2::getLatestExam()` whose semantics are given in Section 6.2.7. The programming environment also contains templates for common object-pointer container formats used in ML-Gs. For these common formats, the counterfeit object-pointer container can be created automatically. The programming environment outputs a buffer that contains all counterfeit objects and is ready to be injected in a COOP attack.

6.5 Proof-of-Concept Exploits

To demonstrate the practical viability of our approach, we implemented exemplary COOP attacks for Microsoft Internet Explorer 10 (32-bit and 64-bit) and Google Chromium 41 for Linux x86-64. In the following, we discuss different aspects of our attack codes that we find interesting. We used our framework for the development of all three attack codes. Each of them fits into 1,024 bytes or less.

For our Internet Explorer 10 examples, we used a publicly documented vulnerability related to an integer signedness error in Internet Explorer 10 [Joly 2013] as our foundation. The vulnerability allows a malicious website to perform arbitrary reads at any address and arbitrary writes within a range of approximately 64 pages on the respective heap using JavaScript code. This gives the attackers many options for hijacking C++ objects residing on the heap and injecting their own buffer of counterfeit objects; it also enables the attackers to gain extensive knowledge on the respective address space layout. We successfully tested our COOP-based exploits for Internet Explorer 10 32-bit and 64-bit on Windows 7. Note that our choice of Windows 7 as the target platform is only for practical reasons; the described techniques also apply to Windows 8. To demonstrate the flexibility of COOP, we implemented different attack codes for 32-bit and 64-bit. Both attack codes could be ported to the respective other environment without restrictions.

6.5.1 Internet Explorer 10 64-Bit

Our COOP attack code for 64-bit only relies on vfgadgets contained in mshtml.dll that can be found in every Internet Explorer process; it implements the following functionality:

- Read pointer to kernel32.dll from IAT.
- Calculate pointer to `WinExec()` in kernel32.dll.
- Read the current tick count from the `KUSER_SHARED_DATA` data structure.
- If tick count is odd, launch `calc.exe` using `WinExec()`; otherwise, execute alternate execution path and launch `mspaint.exe`.

The attack code consists of 17 counterfeit objects with counterfeit vptrs and 4 counterfeit objects that are pure data containers. Overall, eight different vfgadgets are used, including one LOAD-R64-G for loading `rdx` through the dereferencing of a field that is used five times. The attack code is based on an ML-G similar to our exemplary one given in Figure 6.1 that iterates over a plain array of object pointers. With four basic blocks, the ML-G is the largest of the eight vfgadgets. The conditional branch depending on the current tick count is implemented by overwriting the next-in-line object pointer such that the ML-G is recursively invoked for an alternate array of counterfeit object pointers. In summary, the attack code contains eight overlapping counterfeit objects, and we used 15 different labels to create it in our programming environment. All vfgadgets used in this attack code are listed in Table 6.2.

Attack Variant Using Only Vptrs Pointing to the Beginning of Vtables. The described 64-bit attack code relies on counterfeit vptrs that do not necessarily point to the beginning of existing vtables but to positive or negative offsets from them. As a proof of concept, we developed a stealthier variant of the attack code above that *only* uses vptrs that point to the beginning of existing vtables. Accordingly, at each vcall site, we were restricted to the set of virtual functions compatible with the respective fixed vtable index. Under this constraint, our exploit for the given vulnerability is still able to launch `calc.exe` through an invocation of `WinExec()`. The attack code consists of only five counterfeit objects, corresponding to four different vfgadgets (including the main ML-G) from mshtml.dll. Corresponding to the given vulnerability, the used main ML-G can be found as the fourth entry in an existing vtable, whereas, corresponding to the vcall site of the ML-G, the other three vfgadgets can be found as third entries in existing vtables. The task of calculating the address of `WinExec` is done in JavaScript code beforehand. All vfgadgets used in this attack code are listed in Table 6.3.

Table 6.2 Vfgadgets in mshtml.dll 10.0.9200.16521 Used in Internet Explorer 10 64-Bit Exploit^a

Symbol Name of vfgadget	Number	Type	Purpose
CExtendedTagNameSpace::Passivate	1, 9b	ML-G	array-based main loop
CCircularPositionFormatFieldIterator::Next	2, 5, 7, 9a, 10b	LOAD-R64-G	load rdx from dereferenced field
XHDC::SetHighQualityScalingAllowed	3	ARITH-G	store rdx&1
CWigglyShape::OffsetShape	4	LOAD-R64-G	load r9 from field
CStyleSheetArrayVarEnumerator::MoveNextInternal	6	LOAD-R64-G	load r8 from field
CDataCache<class CBoxShadow>::InitData	8	W-COND-G	write r8 to [rdx] if r9 is not zero
CRectShape::OffsetShape	10a, 11b	ARITH-G	add [rdx] to field
PtIs6::CLsBlockObject::Display	11a, 12b	INV-G	invoke field as function pointer

a. Execution splits into paths *a* and *b* after index 8.

Table 6.3 Vfgadgets in mshtml.dll 10.0.9200.16521 Used in Exemplary Internet Explorer 10 64-Bit Exploit^a

Symbol Name of vfgadget	Number	Type	Purpose
CExtendedTagNameSpace::Passivate	1	ML-G	array-based main loop
CMarkupPageLayout::IsTopLayoutDirty	2, 4	LOAD-R64-G	load edx from field
HtmlLayout::GridBoxLayoutTrackCollection::GetRangeTrackNumber	3	ARITH-G	$r8 = 2 \cdot rdx$
CAnimatedCacheEntryTyped<float>::UpdateValue	4	INV-G	invoke field from argument as function pointer

a. Only uses vptrs pointing to the beginning of existing vttables.

6.5.2 Internet Explorer 10 32-Bit

Our 32-bit attack code implements the following functionality: (1) read pointer to `kernel32.dll` from IAT; (2) calculate pointer to `WinExec()` in `kernel32.dll`; (3) enter loop that launches `calc.exe` using `WinExec()` n times; (4) finally, enter an infinite waiting loop such that the browser does not crash.

The attack code does not rely on an array-based ML-ARG-G (recall that in 32-bit ML-ARG-Gs are used instead of ML-Gs); instead, it uses a more complex ML-ARG-G that traverses a linked list of object pointers using a C++ iterator. We discovered this ML-ARG-G in `jscript9.dll` that is available in every Internet Explorer process. The ML-ARG-G consists of four basic blocks and invokes the function `SListBase::Iterator::Next()` to get the next object pointer from a linked list in a loop. The assembly code of the ML-ARG-G is given in Listing 6.3.

Figure 6.11 depicts the layout of the linked list: each item in the linked list consists of one pointer to the next item and another pointer to the actual object. This layout allows for the low-overhead implementation of conditional branches and loops. For example, to implement the loop in our attack code, we simply made parts of the linked list circular, as shown in Figure 6.11. Inside the loop in our attack code, a counter within a counterfeit object is incremented for each iteration. Once the counter overflows, a W-COND-G rewrites the *backward* pointer such that the loop is left and execution proceeds along another linked list.

Our attack code consists of 11 counterfeit objects, and 11 linked-list items of which two point to the same counterfeit object. Four counterfeit objects overlap and one counterfeit object overlaps with a linked-list item to implement the conditional rewriting of a *next* pointer. The actual vfgadgets used in our attack code are listed in Table 6.4. This example highlights how powerful linked-list-based ML-Gs/ML-ARG-Gs are in general.

6.5.3 Chromium 41 for Linux x86-64

To demonstrate the wide applicability of COOP, we also created an attack code for a modified version of Chromium 41 for Linux x86-64. This specific version was compiled with LLVM and was altered to contain the critical vulnerability CVE-2014-3176, which had been identified and patched in an earlier version of Chromium. Our COOP attack code here reads a pointer to `libc.so` from the *global offset table* (GOT) and calculates the address of `system()` from that in order to finally invoke `system("/bin/sh")`.

```

mov     edi, edi
push   ebp
mov     ebp, esp
push   ecx
push   ecx
push   esi
mov     esi, ecx
lea     eax, [esi+3ACh]
; -- inlined constructor of iterator --
mov     [ebp+iterator.end], eax
mov     [ebp+iterator.current], eax
; --

loop:
lea     ecx, [ebp+iterator]
call   SListBase::Iterator::Next()
test   al, al
jnz    end

mov     eax, [ebp+iterator.current]
push   [esi+140h] ; push argument field
mov     ecx, [eax+4] ; read object pointer from iterator
mov     eax, [ecx]
call   [eax+4] ; call second virtual function
jmp    loop

end:
pop    esi
mov    esp, ebp
pop    ebp
ret

```

Listing 6.3 Assembly code of ML-ARG-G in jsript9.dll version 10.0.9200.16521 used in exemplary Internet Explorer 10 32-bit exploit: A linked list of object pointers is traversed; a virtual function with one argument is invoked on each object.

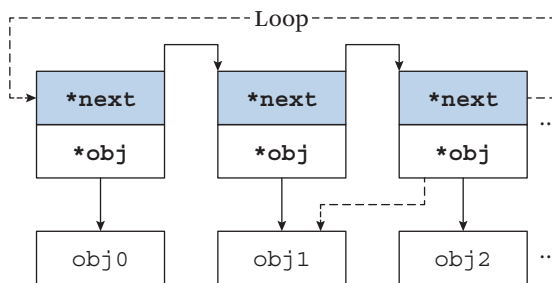


Figure 6.11 Schematic layout of the linked list of object pointers the ML-ARG-G traverses in the Internet Explorer 10 32-bit exploit; dashed arrows are examples of dynamic pointer rewrites for the implementation of conditional branches.

Table 6.4 Vfgadgets Used in Internet Explorer 10 32-Bit Exploit^a

Symbol Name of vfgadget	Number	Type	Purpose
jscript9!ThreadContext::ResolveExternalWeakReferencedObjects	1	ML-ARG-G	linked-list-based main loop
CDataTransfer::Proxy	2	W-SA-G	write dereference field to scratch area
CDCompSwapChainLayer::SetDesiredSize	3	R-G	load field from scratch area
CDCompSurfaceTargetSurface::GetOrigin	4	ARITH-G / W-SA-G	write summation of two fields to scratch area
CDComplayerManager::SetAnimationCurveToken	5	R-G	load field from scratch area
HtmlLayout::SvgBoxBuilder::PrepareBoxForDisplay	<i>loop_entry</i> : 6, 11	W-G	rewrite <i>argument field</i>
CDXTargetSurface::OnEndDraw	7, 8	MOVE-SP-G	move stack pointer up
ieframe!Microsoft::WRL::Callback::ComObject::Invoke	9	INV-G	invoke function pointer with two arguments
CMarkupPageLayout::AddLayoutTaskOwnerRef	10	ARITH-G	increment field
PtIs6::CLsDnodeNonTextObject::SetDurFmtCore	12	W-COND-G	conditionally write argument to field; rewrites linked list; resumes at <i>loop_entry</i> or <i>loop_exit</i>
CDispRecalcContext::OnBeforeDestroyInitialIntersectionEntry	<i>loop_exit</i>	NOP	nop; loops to self

a. vfgadgets taken from mshtml.dll (if not marked differently), jscript9.dll, or ieframe.dll version 10.0.9200.16521.

The attack code comprises six counterfeit objects (of which two overlap) corresponding to six different vfgadgets from Chromium's main executable module. The vfgadgets are listed in detail in Table 6.5.

We also created a *loopless* variant of this COOP attack code that, instead of an ML-G, uses a REC-G from Chromium 41, which is depicted in Listing 6.4.

In this REC-G, `fShaderA->contextSize()` constitutes part ② and `fShaderB->contextSize()` part ④, as depicted earlier in Figure 6.9.

Table 6.5 Vfgadgets Used in Chromium 41 64-Bit Linux Exploit

Symbol Name of vfgadget	Number	Type	Purpose
icu_52::PatternMap::~~PatternMap	1	ML-G	array-based main loop
SkBlockMemoryStream::~rewind	2	R-G/W-G	read pointer to libc and write it to field
TraceBufferRingBuffer:: ClonedTraceBuffer::NextChunk	3	LOAD-R64-G	load rsi with offset of system()
net::AeadBaseEncrypter:: GetCiphertextSize	4	ARITH-G	add field to rsi
TtsControllerImpl:: SetPlatformImpl	5	W-G	store rsi
browser_sync::AddDBThread ObserverTask::RunOnDBThread	6	INV-G	invoke function pointer from field and pass field as argument

```

size_t SkComposeShader::contextSize() const {
    return sizeof(ComposeShaderContext)
        + fShaderA->contextSize() + fShaderB->contextSize();
}

```

Listing 6.4 Example of a REC-G in Chromium 41 (C++).

6.6 Discussion

We now analyze the properties of COOP, discuss different defense concepts against it, and review our design goals [G-1–G-4](#) from Section [6.1.1](#). The effectiveness against COOP of several existing defenses is discussed in Section [6.7](#).

6.6.1 Preventing COOP

We observe that characteristics [C-1–C-5](#) of existing code-reuse-attack approaches cannot be relied on to defend against COOP (goal [G-1](#)): in COOP, control flow is only dispatched to existing and address-taken functions within an application through existing indirect calls. In addition, COOP neither injects new nor alters existing return addresses or other code pointers directly. Instead, only existing vptrs (i.e., pointers to code pointers) are manipulated or injected. Technically, however, depending on the choice of vfgadgets, a COOP attack may execute a high ratio of indirect branches and thus exhibit characteristic [C-3](#). But we note that ML-Gs (which are used in each COOP attack as central dispatchers) are legitimate C++ virtual

functions whose original purpose is to invoke many (different) virtual functions in a loop. Any heuristics attempting to detect COOP based on the frequency of indirect calls will thus inevitably face the problem of high numbers of false positive detections. Furthermore, similar to existing attacks against behavioral-based heuristics [Göktaş et al. 2014b, Davi et al. 2014], it is straightforward to mix-in long “dummy” vfgadgets to decrease the ratio of indirect branches.

As a result, COOP cannot be effectively prevented by (i) CFI that does not consider C++ semantics or (ii) detection heuristics relying on the frequency of executed indirect branches and is unaffected by (iii) shadow call stacks that prevent rogue returns and (iv) the plain protection of code pointers.

On the other hand, a COOP attack can only be mounted under the preconditions given in Section 6.1.2. Accordingly, COOP is conceptually thwarted by defense techniques that prevent the hijacking or injection of C++ objects or conceal necessary information from the attacker, e.g., by applying ASLR and preventing information leaks.

6.6.2 Generic Defense Techniques

We now discuss the effectiveness of several other possible defensive approaches against COOP that do not require knowledge of precise C++ semantics and can thus likely be deployed without analyzing or recompiling an application’s source code.

Restricting the Set of Legitimate API Invocation Sites. A straightforward approach to tame COOP attacks is to restrict the set of code locations that may invoke certain sensitive library functions. For example, by means of binary rewriting it is possible to ensure that certain WinAPI functions may only be invoked through constant indirect branches that read from a module’s IAT (see CCFIR [Zhang et al. 2013]). In the best case, this approach could effectively prevent the API calling techniques 2 and 3 described in Section 6.2.8. However, it is also common for benign code to invoke repeatedly used or dynamically resolved WinAPI functions through non-constant indirect branches like `call rsi`. Accordingly, in practice, it can be difficult to precisely identify the set of a module’s legitimate invocation sites for a given WinAPI function. We also remark that even without immediate access to WinAPI functions or system calls, COOP is still potentially dangerous because, for example, it could be used to manipulate or leak critical data.

Monitoring of the Stack Pointer. In 64-bit COOP, the stack pointer is virtually never moved in an irregular or unusual manner. For the 32-bit *thiscall* calling convention,

though, this can be hard to avoid when calling a series of vfgadgets each taking a fixed, but different, number of arguments. This is a potential weakness that can reveal a COOP attack on Windows x86-32 to a C++-unaware defender that closely observes the stack pointer. However, we note that it may be difficult to always distinguish this behavior from the benign invocation of functions in the *cdecl* calling convention.

6.6.3 Fine-Grained Code Randomization

COOP is conceptually resilient against the fine-grained randomization of locations of binary code, e.g., on function, basic-block, or instruction level. This is because in a COOP attack (as opposed to, for example, a ROP attack), knowing the exact locations of certain instruction sequences is not necessary but rather only the locations of certain vtables. Moreover, in COOP, the attacker mostly misuses the *actual* high-level semantics of existing code. Most vfgadget types, other than ROP gadgets, are thus likely to be unaffected by semantics-preserving rewriting of binary code. Only LOAD-R64-Gs that are used to load x86-64 argument registers can be broken by such means. However, the attacker could probably oftentimes fall back to x86-32-style ML-ARG-G-based COOP in such a case.

C++ Semantics-Aware Defense Techniques

We observe that the control flow and data flow in a COOP attack are similar to those of benign C++ code (goal G-2). However, there are certain deviations that can be observed by C++-aware defenders. We now discuss several corresponding defenses.

Verification of Vptrs. In basic COOP, vptrs of counterfeit objects point to existing vtables but not necessarily to their beginning. This allows for the implementation of viable defenses against COOP when all legitimate vcall sites and vtables in an application are known and, accordingly, each vptr access can be augmented with sanity checks. Such a defense can be implemented without access to source code by means of static binary code rewriting as shown by [Prakash et al. \[2015\]](#). While such a defense significantly shrinks the available vfgadget space, our exploit code from Section 6.5.1 demonstrates that COOP-based attacks are still possible, at least for large C++ target applications.

Ultimately, a defender needs to know the set of allowed vtables for each vcall site in an application to reliably prevent malicious COOP control flow (or at least needs to arrive at an approximation that sufficiently shrinks the vfgadget space).

For this, the defender needs (i) to infer the global hierarchy of C++ classes with virtual functions and (ii) to determine the C++ class (within that hierarchy) that corresponds to each vcall site. Both can easily be achieved when source code is available. Without source code, given only binary code and possibly debug symbols or *Runtime Type Information* (RTTI) metadata, the former can be achieved with reasonable precision while, to the best of our knowledge, the latter is generally considered to be hard for larger applications by means of static analysis [Dewey and Giffin 2012, Gawlik and Holz 2014, Fokin et al. 2011, Prakash et al. 2015]. Note that RTTI metadata is often linked to C++ applications for various purposes and includes the literal names of classes and the precise class hierarchy.

Monitoring of Data Flow. COOP also exhibits a range of data-flow patterns that can be revealing when C++ semantics are considered. Probably foremost, in basic COOP, vfgadgets with varying numbers of arguments are invoked from the same vcall site. This can be detected when the number of arguments expected by each virtual function in an application is known. While trivial with source code, deriving this information from binary code can be challenging [Prakash et al. 2015]. An even stronger (but also likely costlier) protection could be created by considering the actual types of arguments.

In a COOP attack, counterfeit objects are not created and initialized by legitimate C++ constructors, but are injected by the attacker. Further, the concept of overlapping objects creates unusual data flows. To detect this, the defender needs to be aware of the life cycle of C++ objects in an application. This requires knowledge of the whereabouts of (possibly inlined) constructors and destructors of classes with virtual functions.

Fine-Grained Randomization of C++ Data Structures. In COOP, the layout of each counterfeit object needs to be byte-compatible with the semantics of its vfgadget. Accordingly, randomizing C++ object layouts on application start-up, e.g., by inserting randomly sized paddings between the fields of C++ objects, can hamper COOP. Also, the fine-grained randomization of the positions or structures of vtables is a viable defense against COOP. In fact, this approach is presented in detail in a paper by Crane et al. [2015].

We conclude that COOP can be mitigated by a range of means that do not require knowledge of C++ semantics. But we regard it as vital to consider and to enforce C++ semantics to reliably prevent COOP. Doing so by means of static binary analysis and rewriting only is challenging, as the compilation of C++ code is in most cases

a *lossy* process. For example, in binary code, distinguishing the invocation of a virtual function from the invocation of a C-style function pointer that happens to be stored in a read-only table can be difficult. Hence, unambiguously recovering essential high-level C++ semantics afterward can be hard or even impossible. In fact, as we discuss in more detail in Section 6.7, we are not aware of any binary-only CFI solution that considers C++ semantics precisely enough to fully protect against COOP.

6.6.4 Applicability and Turing Completeness

We have shown that COOP is applicable to popular C++ applications on different operating systems and hardware architectures (goal G-3). Naturally, a COOP attack can only be mounted in case at least a minimum set of vfgadgets is available. We did not conduct a quantitative analysis on the general frequency of usable vfgadgets in C++ applications: determining the actual usefulness of potential vfgadgets in an automated way is challenging, and we leave this for future work. In general, we could choose from many useful vfgadgets in the libraries `mshtml.dll` (around 20 MB) and `libxul.so` (around 60 MB), and the basic vfgadget types ARITH-G, W-G, R-G, LOAD-R64-G, and W-SA-G are common even in smaller binaries.

The availability of central dispatcher vfgadgets, such as ML-Gs/ML-ARG-Gs or REC-Gs, is vital to every COOP attack. While ML-Gs/ML-ARG-Gs are generally sparser than the more basic types, we found usable dispatcher vfgadgets, e.g., in Microsoft's standard C/C++ runtime libraries `msvcr120.dll` and `msvcp120.dll` (both smaller than 1 MB; dynamically linked to many C and C++ applications on Windows): the virtual function `SchedulerBase::CancelAllContexts()` with five basic blocks in `msvcr120.dll` is a linked-list-based ML-G. In `msvcr120.dll`, we also found the INV-G `CancellationTokenRegistration_TaskProc::_Exec()` that consists of one basic block and is suitable for x86-32 and x86-64 COOP.

The virtual function `propagator_block::unlink_sources()` with eight basic blocks in `msvcp120.dll` is an array-based ML-ARG-G. Interestingly, this particular ML-ARG-G is also defined in Visual Studio's standard header file `agents.h`. The virtual destructor of the class `Concurrency::_Order_node_base<enum Concurrency::agent_status>` with seven basic blocks in `msvcp120.dll` is a REC-G.

Given the vfgadget types defined in Table 6.1, COOP has the same expressiveness as unrestricted ROP [Shacham 2007]. Hence, it allows for the implementation of a Turing machine (goal G-4) based on memory load/store, arithmetic, and branches. In particular, the COOP examples in Section 6.5 show that complex semantics, like loops, can be implemented under realistic conditions.

6.7 Security Assessment of Existing Defenses

Based on the discussions in the previous section, we now assess a selection of contemporary defenses against code-reuse attacks and discuss whether they are vulnerable to COOP in our adversary model. A summary of our assessment is given in Table 6.6.

Table 6.6 Overview of the Effectiveness of a Selection of Code-Reuse Defenses and Memory Safety Techniques against COOP

Category	Scheme	Realization	Effective? ^a
Generic CFI	Original CFI + shadow call stack [Abadi et al. 2005a]	Binary + debug symbols	X
	Lockdown [Payer et al. 2015c]	Binary + debug symbols	X
	CFI for COT [Zhang and Sekar 2013]	Binary	X
	CCFIR [Zhang et al. 2013]	Binary	X
	O-CFI [Mohan et al. 2015]	Binary	X
	MIP [Niu and Tan 2013]	Source code	X
	SW-HW Co-Design [Davi et al. 2014]	Source code + CPU features	X
	Windows 10 CFG	Source code	X
	LLVM IFCC [Tice et al. 2014]	Source code	?
C++-aware CFI	various [Tice et al. 2014, Jang et al. 2014, Akritidis et al. 2008]	Source code	✓✓✓
	T-VIP [Gawlik and Holz 2014]	Binary	X
	VTint [Zhang et al. 2015]	Binary	X
	vfGuard [Prakash et al. 2015]	Binary	?
Heuristics-based detection	various [Pappas et al. 2013, Cheng et al. 2014, Xia et al. 2012, Zhou et al. 2014]	Binary + CPU features	XXX
	Microsoft EMET 5 [Microsoft Corp 2014, Fratric 2012]	Binary	X
Code hiding, shuffling, or rewriting	STIR [Wartell et al. 2012]	Binary	X
	G-Free [Onarlioglu et al. 2010]	Source code	X
	Readactor [Crane et al. 2015]	Source code + CPU features	X
	XnR [Backes et al. 2014]	Binary/source code + CPU features	?
	Readactor++ [Crane et al. 2015]	Source code + CPU features	✓
Memory safety	various [Akritidis et al. 2009, Nagarakatte et al. 2010, Akritidis et al. 2008, Akritidis 2010, Serebryany et al. 2012, Chen et al. 2015]	Mostly source code	✓✓✓
	CPI/CPS [Kuznetsov et al. 2014a]	Source code	✓/X

a. ✓ indicates effective protection and X indicates vulnerability; ? indicates at least partial protection.

6.7.1 Generic CFI

We first discuss CFI approaches that do not consider C++ semantics for the derivation of the CFG that should be enforced. We observe that all of them are vulnerable to COOP.

The basic implementation of the original CFI work by [Abadi et al. \[2005a\]](#) instruments binary code such that indirect calls may only go to address-taken functions (imprecise CFI). This scheme and a closely related one [[Zhang and Sekar 2013](#)] have recently been shown to be vulnerable to advanced ROP-based attacks [[Göktas et al. 2014a](#), [Davi et al. 2014](#)]. Abadi et al. also proposed to combine their basic implementation with a shadow call stack that prevents call/return mismatches. This extension effectively mitigates these advanced ROP-based attacks while, as discussed in Section 6.6, it does not prevent COOP. The same also applies in general to the recently proposed Lockdown system [[Payer et al. 2015c](#)]. However, besides a shadow call stack and standard imprecise CFI policies, Lockdown additionally enforces that across modules only mutually *imported/exported* functions may be invoked indirectly. Accordingly, a COOP attack would, for instance, be limited to those functions from `kernel32.dll` or `libc` that are actually used by the target application. We remark that this import/export policy probably cannot generally be applied to C++ virtual functions without the risk of high rates of false positives. This is because it is not uncommon for a C++ module to unknowingly access a vtable defined in another module when dynamically dispatching a virtual function call. In such a case, a virtual function that is neither *exported* nor *imported* is legitimately invoked across module boundaries.

[Davi et al. \[2014\]](#) described a hardware-assisted CFI solution for embedded systems that incorporates a shadow call stack and a certain set of runtime heuristics. However, the indirect call policy only validates whether an indirect call targets a valid function start. As COOP only invokes entire functions, it can bypass this hardware-based CFI mechanism.

CCFIR [[Zhang et al. 2013](#)], a CFI approach for Windows x86-32 binaries, uses a randomly arranged “springboard” to dispatch all indirect branches within a code module. On the baseline, CCFIR allows indirect calls and jumps to target all address-taken locations in a binary and restricts returns to certain call-preceded locations. One of CCFIR’s core assumptions is that the attackers are unable to “selectively reveal springboard stub addresses of their choice” [[Zhang et al. 2013](#)]. [Göktas et al.](#) recently showed that ROP-based bypasses for CCFIR are possible given an up-front information leak from the springboard [[Göktas et al. 2014a](#)]. In contrast, COOP breaks CCFIR without violating its assumptions: the springboard technique is ineffective against COOP as we do not inject code pointers but only `vptrs` (point-

ers to code pointers). CCFIR, though, also ensures that sensitive WinAPI functions (e.g., `CreateFile()` or `WinExec()`) can only be invoked through constant indirect branches. However, as examined in Section 6.6.2, this measure does not prevent dangerous attacks and can probably also be sidestepped in practice. In any case, COOP can be used in the first stage of an attack to selectively read out the springboard.

In *Monitor Integrity Protection* (MIP) [Niu and Tan 2013], applications are compiled such that they are composed of variable-sized chunks: single instructions, basic blocks, or functions that do not include calls (*leaf functions*). Indirect branches are instrumented in such a way that they can only lead to the beginning of chunks. It is claimed that MIP can “prevent arbitrary code execution” by an attacker who is able to read/write arbitrary data in an application’s address space but is unable to directly write to the processor’s registers. Since COOP only invokes legitimate virtual functions, it will never trigger an alarm in MIP.

Many system modules in Microsoft Windows 10 are compiled with *Control Flow Guard* (CFG), a simple form of CFI. In summary, Microsoft CFG ensures that protected indirect calls may only go to a certain set of targets. This set is specified in a module’s PE header [Russeinovich et al. 2012]. If multiple CFG-enabled modules reside in a process, their sets are merged. At least all functions contained in a DLL’s EAT are contained in the set. For C++ modules like `mshtml.dll`, additionally, all virtual functions are contained in the set and can thus be invoked from any indirect call site. Accordingly, Microsoft CFG in its current form does not prevent COOP and is also unlikely to stop advanced ROP-based attacks like the one by Göktaş et al. [2014a].

Tice et al. [2014] recently described two variants of *Forward-Edge CFI* for the GCC and LLVM compiler suites that solely aim at constraining indirect calls and jumps but not returns. As such, taken for itself, forward-edge CFI does not prevent ROP in any way. One of the proposed variants is the C++-aware *Virtual Table Verification* (VTV) technique for GCC. It tightly restricts the targets of each `vcall` site according to the C++ class hierarchy and thus prevents COOP. VTV is available in mainline GCC since version 4.9.0. However, the variant for LLVM called *Indirect Function-Call Checks* (IFCC) “does not depend on the details of C++ or other high-level languages” [Tice et al. 2014]. Instead, each indirect call site is associated with a set of valid target functions. A target is valid if (i) it is address taken and (ii) its signature is *compatible* with the call site. Tice et al. discuss two definitions for the *compatibility* of function signatures for IFCC: (i) all signatures are compatible or (ii) signatures with the same number of arguments are compatible. We observe that the former configuration does not prevent COOP, whereas the latter can still allow for powerful COOP-based attacks in practice, as discussed in Section 6.6.3.

6.7.2 C++-Aware CFI

As discussed in Section 6.6, COOP's control flow can be reliably prevented when precise C++ semantics are considered from source code. Accordingly, various source-code-based CFI solutions exist that prevent COOP, e.g., GCC VTV as described above, SafeDispatch [Jang et al. 2014], and WIT [Akritidis et al. 2008].

Recently, three C++-aware CFI approaches for legacy binary code were proposed: T-VIP [Gawlik and Holz 2014], vfGuard [Prakash et al. 2015], and VTint [Zhang et al. 2015]. They follow a similar basic approach:

- Identification of vcall sites and vttables (only vfGuard and VTint) using heuristics and static data-flow analysis
- Instrumentation of vcall sites to restrict the set of allowed vttables

T-VIP ensures at each instrumented vcall site that the vp_{tr} points to read-only memory. Optionally, it also checks if a random entry in the respective vttable points to read-only memory. Similarly, VTint copies all identified vttables into a new read-only section and instruments each vcall site to check if the vp_{tr} points into that section. Both effectively prevent attacks based on the injection of fake vttables, but since a COOP attack only references actual vttables, they do not prevent COOP. VfGuard instruments vcall sites to check if the vp_{tr} points to the *beginning* of any known vttable. As discussed in Section 6.6.3, such a policy restricts the set of available vfgadgets significantly but still cannot reliably prevent COOP. VfGuard also checks the compatibility of calling conventions and consistency of the this_{p_{tr}} at vcall sites, but this does not affect COOP. Nonetheless, we consider vfGuard to be one of the strongest available binary-only defenses against COOP. VfGuard significantly constrains attackers, and we expect it to be a reliable defense in at least some attack scenarios, e.g., for small- to medium-sized x86-32 applications that are considerably smaller than Internet Explorer.

6.7.3 Heuristics-Based Detection

Microsoft EMET [Microsoft Corp 2014] is probably the most widely deployed exploit mitigation tool. Among others, it implements different heuristics-based strategies for the detection of ROP [Fratric 2012]. Additionally, several related heuristics-based defenses have been proposed that utilize certain debugging features available in modern x86-64 processors [Pappas et al. 2013, Cheng et al. 2014, Xia et al. 2012]. All of these defenses have recently been shown to be unable to detect more advanced ROP-based attacks [Carlini and Wagner 2014, Davi et al. 2014, Göktaş

et al. 2014b, Schuster et al. 2014]. Similarly, the HDROP [Zhou et al. 2014] defense utilizes the *performance monitoring counters* of modern x86-64 processors to detect ROP-based attacks. The approach relies on the observation that a processor’s internal branch prediction typically fails in abnormal ways during the execution of common code-reuse attacks.

As discussed in Section 6.6.1, such heuristics are unlikely to be practically applicable to COOP, and we can in fact confirm that our Internet Explorer exploits are not detected by EMET version 5.

6.7.4 Code Hiding, Shuffling, or Rewriting

STIR [Wartell et al. 2012] is a binary-only defense approach that randomly reorders basic blocks in an application on each start-up to make the whereabouts of gadgets unknown to attackers—even if they have access to the exact same binary. As discussed in Section 6.6.2, approaches like this do not conceptually affect our attack, as COOP only uses entire functions as vfgadgets and only knowledge on the whereabouts of vtables is required. This applies also to the recently proposed O-CFI approach [Mohan et al. 2015] that combines the STIR concept with coarse-grained CFI.

G-Free [Onarlioglu et al. 2010] is an extension to the GCC compiler. G-Free produces x86-32 native code that (largely) does not contain *unaligned* indirect branches. Additionally, it aims to prevent attackers from misusing *aligned* indirect branches: return addresses on the stack are encrypted/decrypted on a function’s entry/exit, and a “cookie” mechanism is used to ensure that indirect jump/call instructions may only be reached through their respective function’s entry. While effective even against many advanced ROP-based attacks [Göktas et al. 2014a, Carlini and Wagner 2014, Davi et al. 2014, Göktas et al. 2014b, Schuster et al. 2014], G-Free does not affect COOP.

The Execute-no-Read (XnR) concept [Backes et al. 2014] prevents an application’s code pages from being read at runtime in order to hamper so-called *JIT-ROP* attacks [Snow et al. 2013]. We note that, depending on the concrete scenario, a corresponding JIT-COOP attack could not always be thwarted by such measures, as an adversary can read out vtables and possibly RTTI metadata (which contains the literal names of classes) from data sections and apply pattern matching to identify the addresses of the vtables of interest. XnR is meant to be implemented in hardware as a complementary feature to the execute-disable/NX bit already available in modern processors.

The Readactor system [Crane et al. 2015] leverages the *Extended Page Tables* (EPT) [Intel Corp 2013] feature of modern x86-64 processors to place an application’s code in *execute-only* memory¹ at runtime. In Readactor, a C/C++ application is compiled such that (i) all its actual code pointers are hard-coded inside *trampolines* in execute-only memory, (ii) only pointers to those trampolines but no actual code pointers—including return addresses—are stored in readable memory at runtime, and (iii) the binary code layout is randomized in a fine-grained manner. Consequently, the whereabouts of all an application’s code—except for trampolines—are concealed at runtime even from attackers that can read the entire address space with respect to page permissions. Crane et al. claim that Readactor “provides protection against all known variants of ROP attacks” [Crane et al. 2015]. However, we observe that the Readactor concept does not conceptually hinder COOP because Readactor neither hides vtables in any special way nor randomizes their layouts. Vptrs also receive no special treatment from Readactor.

Finally, Readactor++ [Crane et al. 2015] is an extension of the Readactor concept that was specifically designed to tackle COOP and RILC. Readactor++ applies all the defensive measures of Readactor and also pseudo-randomly changes the structure layout of vtables (and other function pointer tables) as outlined in Section 6.6.3. In order to exacerbate attempts at guessing useful vtable entries, Readactor++ also adds so-called “booby trap” entries to randomized vtables that on execution terminate the protected application. It is argued that a minimal COOP attack, which requires at least the execution of three vfgadgets from distinct vtables, would be hindered by Readactor++ with a chance of at least 99.97%. Readactor++’s average overhead is low, and it can be considered one of the most cost-effective strong defenses against COOP.

6.7.5 Memory Safety

Systems that provide forms of memory safety for C/C++ applications [Serebryany et al. 2012, Akritidis et al. 2009, Nagarakatte et al. 2010, Akritidis et al. 2008, Kuznetsov et al. 2014a, Akritidis 2010, Chen et al. 2015] can constitute strong defenses against control-flow hijacking attacks in general. As our adversary model explicitly foresees an initial memory corruption and information leak (see Sec-

1. Across different contemporary processor architectures, if memory is *executable*, then typically it is implicitly also *readable*.

tion 6.1.2), we do not explore the defensive strengths of these systems in detail. Instead, we exemplarily discuss two recent approaches.

Kuznetsov et al. [2014a] proposed *Code-Pointer Integrity* (CPI) as a low-overhead control-flow hijacking protection for C/C++. On the baseline, CPI guarantees the spatial and temporal integrity of code pointers and, recursively, that of pointers to code pointers. Since C++ applications typically have many code pointers (essentially each object’s vptr), CPI can impose significant overhead there. As a consequence, Kuznetsov et al. also proposed *Code-Pointer Separation* (CPS) as a less expensive variant of CPI that specifically targets C++. In CPS, sensitive pointers are not protected recursively, but it is still enforced that “(i) code pointers can only be stored to or modified in memory by code pointer store instructions, and (ii) code pointers can only be loaded by code pointer load instructions from memory locations to which previously a code pointer store instruction stored a value” [Kuznetsov et al. 2014a], where *code pointer load/store instructions* are fixed at compile time. Kuznetsov et al. argue that the protection offered by CPS could be sufficient in practice as it conceptually prevents recent advanced ROP-based attacks [Carlini and Wagner 2014, Davi et al. 2014, Göktaş et al. 2014b]. We observe that CPS does not prevent our attack because COOP does not require the injection or manipulation of code pointers. In the presence of CPS, though, it is likely hard to invoke library functions not imported by an application. But we note that almost all applications import critical functions. The invocation of library functions through an INV-G could also be complicated or impossible in the presence of CPS. However, this is not a hurdle because, as CPS does not consider C++ semantics, imported library functions can always easily be called without taking the detour through an INV-G, as described in Section 6.2.8 in approach 2.

CPS can straightforwardly be made resilient to COOP by extending the protection of immediate code pointers to C++ vptrs. In fact, recent implementations of CPS incorporate this tweak [Kuznetsov et al. 2014b].

6.8 Conclusion

In this chapter, we discussed counterfeit object-oriented programming (COOP), a novel code-reuse attack technique to bypass almost all CFI solutions and many other defenses that do not consider object-oriented C++ semantics. We explained the specifics of object-oriented programming and the technical details behind COOP. We believe that our results contribute to the ongoing research on designing

practical and secure defenses against control-flow hijacking attacks, a severe threat that has been around for more than two decades. Our basic insight that higher-level programming-language-specific semantics need to be taken into account is a valuable guide for the design and implementation of future defenses. In particular, our results demand a rethinking of the assessment of defenses that rely solely on binary code.



Hardware Control Flow Integrity

Yier Jin, Dean Sullivan, Orlando Arias, Ahmad-Reza Sadeghi,
Lucas Davi

Control-Flow Integrity (CFI) is a promising and general defense against control-flow hijacking with formal underpinnings. A key insight from the extensive research on CFI is that its effectiveness depends on the precision and coverage of a program's Control-Flow Graph (CFG). Since precise CFG generation is highly challenging and often difficult, many CFI schemes rely on brittle heuristics and imprecise, coarse-grained CFGs. Furthermore, comprehensive, fine-grained CFI defenses implemented purely in software incur overheads that are unacceptably high.

In this chapter, we first specify a CFI model that captures many known CFI techniques, including stateless and stateful approaches as well as fine-grained and coarse-grained CFI policies. We then design and implement a novel hardware-enhanced CFI. Key to this approach is a set of dedicated CFI instructions that can losslessly enforce any CFG and diverse CFI policies within our model. Moreover, we fully support multi-tasking and shared libraries, prevent various forms of code-reuse attacks, and allow code protected with CFI to interoperate with unprotected legacy code. Our prototype implementation on the SPARC LEON3 is highly efficient with a performance overhead of 1.75% on average when applied to several SPECInt2006 benchmarks and 0.5% when applied to EEMBC's CoreMark benchmark.

7.1 Introduction

Control-flow integrity has been proposed as a general defense technique against control-flow hijacking attacks [Abadi et al. 2005a, Abadi et al. 2009]. In particular, it defends against modern code-reuse attacks, such as Return-Oriented Programming (ROP) [Roemer et al. 2012]. These attacks are prevalent, Turing-complete,

and are repeatedly leveraged to compromise commonly used applications such as web browsers [Marschalek 2014] and document viewers [jduck 2010]. CFI mitigates these attacks by ensuring that an application follows a legitimate control-flow path. The legitimate paths are manifested in the application's control-flow graph derived during an offline static analysis phase. Whenever an attacker attempts to subvert the execution to follow an illegal control-flow path, CFI detects this malicious control flow and immediately terminates the process. In addition, CFI is not vulnerable to memory disclosure and side channel attacks [Snow et al. 2013, Bittau et al. 2014, Seibert et al. 2014], and allows verifiable security [Abadi et al. 2005b].

A number of CFI schemes have been proposed that aim at introducing practical CFI enforcement incurring almost no overhead [Zhang and Sekar 2013, Zhang et al. 2013, Pappas et al. 2013, Cheng et al. 2014]. On the other hand, these schemes enforce coarse-grained CFI policies that an attacker can bypass [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014, Göktas et al. 2014b, Schuster et al. 2014]. In parallel to the development of practical CFI schemes, a number of defenses have been proposed that focus on a CFI subclass, i.e., only protecting indirect virtual calls to C++ virtual methods [Gawlik and Holz 2014, Zhang et al. 2015, Prakash et al. 2015]. However, Schuster et al. [2015] recently demonstrated that modern programs offer a large number of valid virtual methods. Hence, an attacker can exploit the available virtual methods to launch a code-reuse attack. Further, Google recently released a CFI compiler extension for virtual calls [Tice et al. 2014] that resists the latest attacks on virtual method exploitation [Schuster et al. 2015], but can be circumvented by means of stack attacks [Liebchen et al. 2015]. Last, Carlini et al. [2015e] question the overall benefit of CFI, since even fine-grained CFI protection still offers a large code base of valid CFG nodes and edges that an attacker can exploit.

The continued success of code-reuse attacks has several reasons. First, many CFI defenses evaluate their effectiveness based on existing exploits, which naturally do not align to any given CFI policy. These exploits are typically more sophisticated and can be rewritten to align to the CFI-enforced CFG [Carlini et al. 2015e, Liebchen et al. 2015, Carlini and Wagner 2014, Davi et al. 2014, Göktas et al. 2014a]. Second, CFI defenses leverage unreliable metrics, such as gadget reduction, gadget length, or average indirect branch reduction (AIR), to measure CFI precision [Zhang and Sekar 2013, Kayaalp et al. 2012, Niu and Tan 2014a, Mohan et al. 2015, Arias et al. 2015, Tice et al. 2014]. These metrics have frequently over-estimated the provided security and have been shown to be bypassable [Carlini et al. 2015e, Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014].

In this chapter, we first address the mismatch between a sound CFI policy and various insecure implementations by revisiting CFI to provide a comprehensive model covering many CFI policies proposed today. We then develop a precise, stateful CFI policy that enables us to address the granularity of a given CFI scheme and make informed design decisions regarding protection and cost of coverage. We then introduce a general-purpose, hardware-enhanced CFI platform that scales to the coverage provided by any CFG, enables highly efficient enforcement of diverse CFI policies, and losslessly enforces any provided CFG.

We also evaluate runtime attacks and CFI vulnerabilities using hardware-enhanced CFI by first evaluating its effectiveness based on the most current code-reuse attacks [Carlini et al. 2015e, Schuster et al. 2015, Carlini and Wagner 2014, Göktas et al. 2014a, Davi et al. 2014, Checkoway et al. 2010, Tran et al. 2011, Liebchen et al. 2015]. These attacks are able to perform malicious actions while adhering to the restrictions imposed by a CFI-protected system. We additionally address attacks targeting both C and C++ applications as well as JIT-compiled programs. Our evaluation focuses on fundamental attacks that manipulate or otherwise violate our CFI policy, assuming we are provided with a precise CFG.

We take a hardware-based approach for several reasons. First, as we will show, CFI in hardware along with dedicated CFI instructions scales for any CFG and various CFI policies from very coarse-grained to highly precise control-flow checks. Second, a hardware-based approach allows us to instantiate a CFI processor module that highly improves the efficiency of CFI while offering strong, precise CFI protection. Third, hardware-based CFI enables precise stateful CFI policy enforcement. Fourth, a CFI processor module can be associated to an on-chip, dedicated memory that securely isolates CFI data (e.g., CFG information).

We conducted a performance evaluation of our approach using SPEC2006 benchmarks and CoreMark micro-benchmarks on the SPARC LEON3 processor [Gaisler Research 2017]. Our system is highly efficient, incurring almost no performance overhead; on average only 1.75% for SPEC and 0.5% for CoreMark. Our hardware-enhanced CFI area overhead is negligible and can be clocked up to 3 GHz using a 32/28 nm process.

In summary, our core contributions of this chapter are as follows.

CFI model. We revisit CFI to reason about the protection offered by various CFI implementations and policies that have been presented thus far, and present *precise, stateful CFI*.

Scalable, precise CFI enforcement. We present a design that scales to any CFG provided and losslessly enforces the provided CFG.

Comprehensive prevention. Our CFI hardware platform prevents many known code-reuse attacks: traditional ROP [Shacham 2007], ROP without returns [Checkoway et al. 2010, Davi et al. 2014, Carlini and Wagner 2014] dynamic ROP [Snow et al. 2013], JOP [Checkoway et al. 2010], and full-function reuse [Schuster et al. 2015, Tran et al. 2011].

CFI hardware platform. We present the design, implementation, and evaluation of a scalable and highly efficient hardware-enhanced CFI implementation for the open source SPARC LEON3 hardware platform. Our hardware platform features new CFI instructions that support precise enforcement at diverse CFG granularities.

We stress that the goal of this chapter is the introduction and design of a hardware CFI framework that can enforce CFI policies of different precision, including coarse- and fine-grained variants. Our work is explicitly not about sophisticated static analysis of source code or advanced binary analysis to extract fine-grained control-flow graphs. Generation of CFGs for real-world software remains an open research problem. However, issues in CFG generation are orthogonal to the challenges we address: making CFI enforcement efficient by adding dedicated instructions and supporting hardware.

7.2 Threat Model and Assumptions

Our threat model follows the traditional CFI threat model. We assume an adversary who has arbitrary read and write access to data memory, and read access to code memory. As a consequence, the CFI threat model tolerates memory disclosure attacks, i.e., it allows information leakage but still protects applications against memory corruption attacks. The attacker can be either a local or remote attacker. However, the attacker only has access to user applications, as kernel exploits can undermine any security mechanism implemented for user-space applications.

CFI aims at defending against runtime exploits that violate the integrity of the program's control flow to perform malicious actions. That said, we target benign applications that an attacker attempts to compromise, but do not protect against applications that are inherently malicious. This includes cases where the attacker modifies the binary either in disk or memory. Further, we focus on code-reuse attacks but not code injection attacks, which today are prevented by means of Data Execution Prevention (DEP) [Andersen and Abella 2004].

It is important to note that CFI does not defend against the so-called non-control-data attacks [Chen et al. 2005]. These attacks do not modify any code

pointer but non-control data, such as an authentication variable. Recently proposed hybrid attacks, called Control-Flow Bending (CFB), include both exploitation of non-control data and control data [Carlini et al. 2015e]. Our threat model focuses on the control-data part of the attack.

Return-oriented programming is a generic attack instantiation of code-reuse attacks: it combines short instruction sequences (gadgets) from various functions to generate a new malicious program [Roemer et al. 2012]. Typically, these sequences end with a return instruction to transfer control to the subsequent sequence [Shacham 2007]. That said, these attacks exploit *backward edges* (returns) of a program's control-flow graph. However, an attacker can also leverage sequences that terminate with an indirect call or jump instruction [Checkoway et al. 2010], that is, code-reuse attacks that exploit *forward edges* in the CFG. Sometimes these attacks are referred to as Jump-Oriented Programming (JOP) [Checkoway et al. 2010]. Both attack variants, ROP and JOP, have been shown to be Turing-complete, meaning that the identified code sequences form a Turing-complete language. We aim at defending against these attacks based on control-flow integrity in hardware.

Another code-reuse attack variant is a function-reuse attack that only invokes a chain of library functions [Nergal 2001, Schuster et al. 2015, Tran et al. 2011]. Existing CFI schemes rarely provide protection against these attacks. In fact, preventing these attacks is highly challenging: Consider a program that legitimately invokes a critical function, e.g., `open()`, via an indirect call. As a consequence, the critical function is considered a legitimate control-flow target in CFI. Protection of these code-reuse attacks are within the scope of our threat model.

7.3 Requirements

The requirements that satisfy the goals of a lossless, scalable, and highly efficient hardware-enhanced CFI framework are given below.

Precision. We must losslessly enforce any CFG with which we are provided. In general, it may be impossible to resolve a precise CFG either because source code is unavailable or the analysis is imprecise. In any case we must strictly enforce what we are given.

Scalability. The effectiveness of any CFI approach depends on the CFG precision. Hence, we require that our CFI scheme scales to any level of CFG precision. Given a CFG, we should be capable of enforcing precise CFI. Our system should also be capable of enforcing coarse-grained CFI if no precise CFG is available.

Efficiency. One of the main limitations of software-based CFI approaches are their significant performance overhead. As a consequence, we require negligible performance overhead for our CFI scheme.

Stateful. We require stateful CFI since stateless CFI is vulnerable to stitching gadgets [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014, Göktas et al. 2014b, Schuster et al. 2014] and control-flow bending attacks [Carlini et al. 2015e].

Compatibility. A CFI scheme needs to co-exist with legacy programs that are not instrumented with CFI.

Security. Based on a precise CFG, we require the CFI scheme to cover all of the existing code-reuse attacks including traditional return-oriented programming [Shacham 2007], jump-oriented programming [Checkoway et al. 2010], just-in-time code-reuse attacks [Snow et al. 2013], and whole function-reuse attacks [Schuster et al. 2015].

7.4 Modeling CFI

It has been more than a decade since Abadi et al. [2005a] introduced the idea of control-flow integrity. Since then, various CFI implementations have been proposed, each with different performance and security metrics. We have noticed that there is a division in the community regarding what encompasses a “fine-grained” or “coarse-grained” policy and the kind of security each provides. In the remainder of this section, we present an abstract description of CFI and then use it to state the requirements of a theoretical CFI policy that can provide as much protection as possible. We start by defining a control-flow graph. Subsequently, we extend this mechanism to include the notion of execution state in a process. Using these definitions, we develop a CFI policy that can yield the maximum protection possible under the framework of what is computable and decidable. Our definitions can be shown to be equivalent to those presented in Abadi et al. [2005b], while presenting extensions to incorporate missing elements when needed.

7.4.1 Control-Flow Graph

To introduce our definition of a control-flow graph, we need to define its components. Let \mathbb{C} be the set of control-flow instructions and \mathbb{I} be the set of non-control-flow instructions. Then we say that a node \mathbb{N}_i consists of a sequential set of non-control-flow instructions. That is, $\mathbb{N}_i = \{I_1, I_2, I_3, \dots, I_z\}, I_1, \dots, z \in \mathbb{I}$. An edge E_j is given by an instruction $I_C \in \mathbb{C}$, which transfers control (\rightarrow) to a new node \mathbb{N}_j

(e.g., a call to a new function or jump to a case statement within the same function) or to the same node \mathbb{N}_{i-1} (e.g., a for loop). That is, $E_j : \mathbb{N}_{i-1} \rightarrow (\mathbb{N}_i \vee \mathbb{N}_{i-1})$. Using these definitions, a *precise control-flow graph* for an arbitrary program P is characterized by the set of 2-tuples $\mathbb{CFG} = \{CFG_{(0,0)}, CFG_{(0,1)}, \dots, CFG_{(m,n)}\}$, where $CFG_{(i,j)} = (\mathbb{N}_i, E_j)$. We should note that not all combinations (i, j) need to exist in a CFG. Furthermore, if a program has no dead code (unreachable code), it can be shown that the CFG is connected.

7.4.2 Control-Flow Integrity Policy

A control flow integrity policy must ensure that a program follows the intended execution path given by its CFG. Accordingly, a CFI policy defines a model for program execution such that whenever a control-flow instruction executes, it targets a valid destination in its CFG. An ideal control-flow graph will provide *all* valid target destinations for an arbitrary program.

CFI policies are therefore constrained to enforcing all possible benign paths in an arbitrary CFG. Any CFI policy can be evaluated by its ability to completely enforce the intended execution path of a program or the extent to which it completely reflects a program's CFG. Therefore, a CFI policy's precision is a fundamental metric of its coverage and/or protection. The CFI policy must losslessly enforce only valid CFG paths. The precision with which a CFI policy completely reflects a CFG is given by granularity. The granularity G of a CFI policy K is determined by how closely it reflects the precise CFG of a process P .

7.4.2.1 Precise Static CFI

We consider the granularity G of a CFI policy to be either precise or coarse. Consider the portion of a CFG for a process shown in Figure 7.1. The set of 2-tuples $\mathbb{CFG} = \{(\mathbb{N}_1, E_1), (\mathbb{N}_2, E_3), (\mathbb{N}_4, E_5), (\mathbb{N}_5, E_2), (\mathbb{N}_5, E_4), (\mathbb{N}_5, E_6)\}$ represent the static CFG. A precise static CFI approach contains a representation and enforcement of *only* these node-edge 2-tuples.

Definition 7.1 Precise static CFI. The *precise static CFI policy* K for a process P is given by strict enforcement of its \mathbb{CFG} , that is, $K : CFI_P \rightarrow \mathbb{CFG}$.

Figure 7.1 reflects a portion of a static CFG for a particular process, where shaded areas represent functions. Edges E_1 , E_3 , and E_5 represent function calls and edges E_2 , E_4 , and E_6 function returns. Although the CFG in Figure 7.1 depicts control flow for the process, there is insufficient information to determine the proper behavior of a return path, or backward edge. Although the CFG depicts the path $\mathbb{N}_4 \rightarrow \mathbb{N}_5 \rightarrow \mathbb{N}_3$ through edges E_5 and E_4 as viable, it is logically incorrect, as the

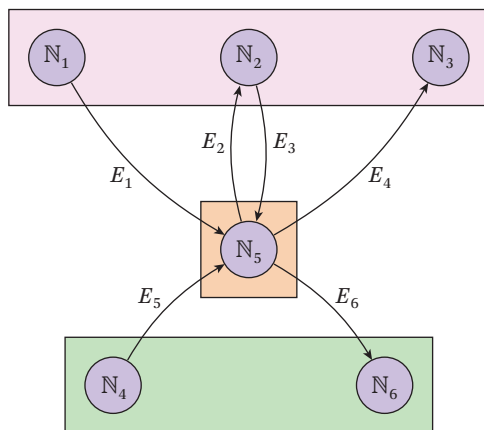


Figure 7.1 Portion of a CFG.

return target should be N_6 . As such, a CFI policy that enforces this CFG without any extra information is inherently incomplete, as backward edges are loosely handled. This is exactly the point of weakness that has been exploited in recent CFI [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014, Göktas et al. 2014b, Schuster et al. 2014] and control-flow bending attacks [Carlini et al. 2015e].

7.4.2.2 Precise, Stateful CFI

Given this limitation of the CFG, it is crucial to introduce the concept of *state* and add it to the CFG.

Definition 7.2 CFG state. A *CFG state* is a set $S_k = \{E_0, E_1, E_2, \dots, E_p\}$ of valid non-jump forward edges on a CFG for a process P .

We do not include jump edges in our CFG state definition because there is no state to be recovered by the transition, i.e., they do not store a return address on the stack. Furthermore, we allow backward edges to remove elements from the state set in an orderly fashion. We combine this concept of state with the CFG to make a *precise, stateful CFG*. We define a stateful CFG to be a set of 3-tuples, $CFG_S = \{CFG_{(0,0,0)}, CFG_{(0,1,1)}, \dots, CFG_{(m,n,o)}\}$, where $CFG_{(i,j,k)} = (N_i, E_j, S_k)$.

Figure 7.2 reflects the CFG with states added. Here, backward-edge paths are only taken if the proper state is preserved. As such, execution path $N_4 \rightarrow N_5 \rightarrow N_3$ through edges E_5 and E_4 is now illegal because the state is not correctly preserved in execution, i.e., $S_3 \neq S_2$ in the stateful CFG. We call a CFI policy capable of enforcing a stateful CFG a *precise, stateful CFI policy*.

Definition 7.3 Precise, stateful CFI. The *precise, stateful CFI* policy K for a process P is given by strict enforcement of its stateful CFG_S , that is, $K: \text{CFI}_P \rightarrow \text{CFG}_S$.

7.4.2.3 Coarse-Grained CFI

A coarse-grained CFI policy is any policy that does not meet the requirements of precise, stateful CFI. Consider again the execution path $N_4 \rightarrow N_5 \rightarrow N_3$ through edges E_5 and E_4 shown in Figure 7.2. This execution path is illegal as it does not maintain the execution state imposed by the stateful CFG. As such, a CFI policy that allows for this execution path to exist contains erroneous edges. We define \mathbb{E} to be the set of erroneous edges included in the CFI policy enforcement and consider this policy to be *coarse-grained*.

Definition 7.4 Coarse-grained CFI. The *coarse-grained CFI* policy K for a process P is given by $K: \text{CFI}_P \rightarrow \text{CFG}' = \text{CFG}_S \cup \mathbb{E}$, where \mathbb{E} is the set of unintended edges.

Corollary 7.1 Granularity of a policy. The *granularity* G of the policy K is said to increase as $|\mathbb{E}|$ increases.

At this point, it is noteworthy to mention that all CFI schemes known to the authors add some factor \mathbb{E} . Even the original CFI implementation for x86 considers two destinations as equivalent when the CFG contains edges from the same set of sources [Abadi et al. 2009].

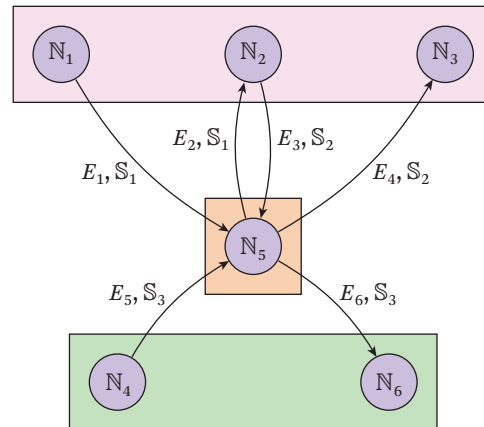


Figure 7.2 Portion of a stateful CFG.

7.5 Constructing a Precise Stateful CFI Policy

We use Figure 7.2 throughout this section to define both a precise forward-edge and backward-edge stateful CFI policy.

7.5.1 Precise Forward-Edge Stateful CFI Considerations

A precise forward-edge stateful CFI policy must be capable of strictly enforcing the intended execution path of a process according to its stateful CFG. That is, it must reflect forward-edge transitions and it must not introduce granularity by allowing erroneous edges in an execution path.

Consider the stateful CFG depicted in Figure 7.2. We define $\not\prec(N_i)$ to be the set of valid targets $\{N_{i+1}, N_{i+2}, \dots\}$ for node N_i . For example, $\not\prec(N_5) = \{N_2, N_3, N_6\}$. Since this node has multiple branch targets ($|\not\prec(N_5)| \geq 2$), we call it a *divergent node*. If the edges leaving a divergent node are caused by indirect jumps, such as those in a jump table, or an indirect call targeting multiple functions, the stateful CFG is unable to fully predict the behavior of the branches as this requires taking into account user input. Full computation of a process behavior under these circumstances reduces to the halting problem. This results in a lower bound in the coarseness of a forward-edge stateful CFI policy. For divergent nodes, the most precision that can be obtained in a forward-edge CFI policy is by checking the branch target of a node against the members of its $\not\prec$ -function.

Consider again the CFG depicted in Figure 7.2. We define $\preceq(N_i)$ to be the set of valid nodes $\{N_{i-1}, N_{i-2}, \dots\}$ that can target node N_i . For example, $\preceq(N_5) = \{N_1, N_2, N_4\}$. Since this node has multiple branch sources ($|\preceq(N_5)| \geq 2$), we call it a *convergent node*. Function entries that are targeted from multiple indirect call instructions exhibit this behavior. A stateful CFG must then be able to encode transition information in such a way that the source of the transition can be differentiated and validated. For example, the CFI policy must reflect that N_5 is targeted by N_1 along edge E_1 , as opposed to N_4 along edge E_5 . Failure to do so results in a coarse-grained CFI policy.

7.5.2 Constructing a Precise Forward-Edge Stateful CFI Policy

Eliminating the granularity due to divergent and convergent nodes is therefore necessary to ensure that a forward-edge CFI policy is precise. We separate our precise forward-edge stateful CFI policy into two categories: (1) indirect jumps and (2) indirect calls and indirect tail call jumps.

7.5.2.1 Indirect Jumps

An indirect jump is constrained to targeting a valid destination, as given by the stateful CFG. In compiled code, indirect jumps, with the exception of indirect tail call jumps, will always target constructs within function bounds. We require that indirect jumps may only target a member of the source node's \succ -function as given by the stateful CFG.

7.5.2.2 Indirect Calls and Indirect Tail Call Jumps

Indirect calls and tail call jumps are constrained to targeting valid destinations, as given by the stateful CFG. In portable, standards-compliant code, these destinations are function entries. We require that indirect calls and tail call jumps may only target a member of the source node's \succ -function as given by the stateful CFG. Furthermore, any additional state information about this transition must be recorded by the policy.

7.5.3 Precise Backward-Edge Stateful CFI Considerations

A precise backward-edge CFI policy must be capable of exactly enforcing the intended execution path of a process according to its stateful CFG. It must not introduce granularity by allowing erroneous edges in an execution path.

Consider the \succ -function for node \mathbb{N}_5 in Figure 7.2, where $\succ(\mathbb{N}_5) = \{\mathbb{N}_2, \mathbb{N}_3, \mathbb{N}_6\}$ and the corresponding \preceq -function $\preceq(\mathbb{N}_5) = \{\mathbb{N}_1, \mathbb{N}_2, \mathbb{N}_4\}$. In the case that node \mathbb{N}_5 is the epilogue of a function, the precise stateful CFI policy must be able to use state information to identify the valid return path. It must be able to utilize state information given by the forward-edge transition from a member of $\preceq(\mathbb{N}_5)$ to validate the backward-edge transition into a member of $\succ(\mathbb{N}_5)$. For example, if the path is given by $\mathbb{N}_4 \rightarrow \mathbb{N}_5$, then the state information provided is \mathbb{S}_3 . Only a member of $\succ(\mathbb{N}_5)$ with state \mathbb{S}_3 may be targeted, in this case \mathbb{N}_6 .

7.5.4 Constructing a Precise Backward-Edge Stateful CFI Policy

Eliminating erroneous backward edges caused by divergent nodes in a CFG is therefore necessary for any precise backward-edge stateful CFI policy. We can enforce this policy by accurately depicting the execution path based on a program's forward-edge behavior. Resolving a valid transition for a backward edge is only a matter of restoring to the previous state in the execution path. More precisely, a precise backward-edge stateful CFI policy must only allow returns to the *most recent forward-edge transition*. As such, a precise backward-edge stateful CFI policy maintains a representation of these transitions.

7.5.4.1 Return Instructions

Return instructions are constrained to follow only the edges with a matching state as described in the stateful CFG, i.e., the code location following the call instruction that resulted in the execution of the returning function. For example, in Figure 7.2 the CFI policy must enforce the backward edge E_6 if node \mathbb{N}_5 was accessed using the forward edge E_5 , as this maintains the state \mathbb{S}_3 .

7.6 Hardware-Enhanced CFI: Design and Implementation

7.6.1 Overview

To restrain the execution of a program to its stateful CFG, a precise stateful CFI policy must enforce the policies outlined in Sections 7.5.2 and 7.5.4. However, a general challenge in designing a system capable of enforcing a precise, stateful CFI policy is how to encode and record the backward- and forward-edge state of a process and how to ensure efficient enforcement.

To solve these problems, we extend the instruction set of an architecture and add dedicated hardware. The Instruction Set Architecture (ISA) extensions enable dynamic creation of a stateful CFG, which in turn allows us to encode, record, and enforce precise, stateful CFI. The execution-path behavior of the program is *encoded* in our ISA extensions, where dedicated hardware is instructed to validate the forward- and backward-edge state of the program. In particular, we track both forward and backward edges by means of CFI instructions each processing a label: `cfi ins lbl`. Forward-edge state is encoded by a CFI instruction, where the label (`lbl`) is a valid target determined by the CFG and recorded in a *label state register*. Backward-edge state is encoded by the execution path's forward-edge behavior as an `cfi ins lbl`, where the label (`lbl`) is recorded in a *label state stack*.

A Label State Stack (LSS) is used to record backward edges to tightly couple caller/callee pairs and ensure only the most recently executed forward edge is returned to. A Label State Register (LSR) is used to record forward edges because there are inherent program semantics that prevent it from being coupled with the label state stack, such as fall-through in a case statement (see Section 7.6.4). We *enforce* precise, stateful CFI using a simple state machine supervising execution. If a violation of the stateful CFI policy is detected, a fault is triggered, resulting in the termination of the process.

The ensuing subsections describe the semantics of the ISA extensions and their interaction with the hardware subsystem. Figure 7.3 illustrates a stateful CFG for a snippet of code and accompanies Figure 7.4, which depicts the code snippet beginning at a function entry.

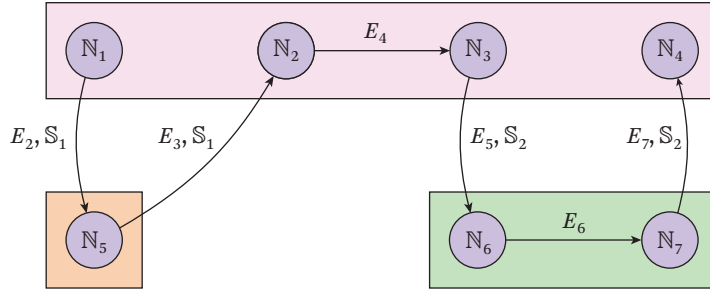


Figure 7.3 Stateful control-flow graph.

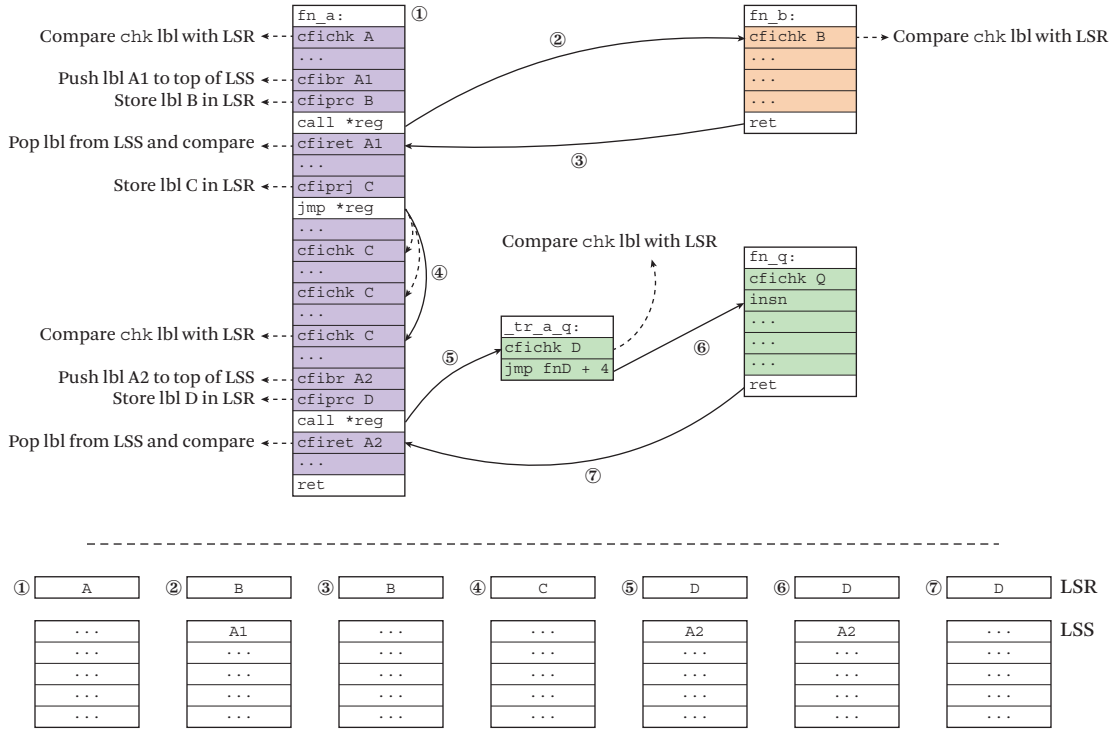


Figure 7.4 Stateful dynamic control-flow graph creation.

7.6.2 CFI Instruction Semantics and Instrumentation

Alongside Figure 7.3, we use Figure 7.4 as an example to highlight the stateful CFI instruction semantics and their interaction with the LSR and LSS. In the code snippet, the process begins execution at the prologue of `fn_a`, labeled ①. Control

Table 7.1 Additions to the Instruction Set Architecture

Instruction	Syntax	Semantics
<code>cfibr</code>	<code>cfibr lbl</code>	Push <code>lbl</code> to top of LSS, flagging a call site as currently active. Unique <code>cfibr lbl</code> issued per call.
<code>cfiret</code>	<code>cfiret lbl</code>	Pop and compare <code>lbl</code> with label at the top of the LSS (returns only). Must be issued on valid return sites.
<code>cfiprj</code>	<code>cfiprj lbl</code>	Store <code>lbl</code> in LSR, flagging intended jump target for subsequent check. Must precede indirect jump instruction.
<code>cfiprc</code>	<code>cfiprc lbl</code>	Store <code>lbl</code> in LSR, flagging intended call target for subsequent check. Must precede call instructions and indirect tail call jump instructions.
<code>cfichk</code>	<code>cfichk lbl</code>	Compare <code>lbl</code> with value stored in LSR. A mismatch results in a control-flow violation, triggering a fault. Must be issued in targets of indirect jumps or function entries.

flow is transferred to `fn_b` along ② and returns to `fn_a` along ③. An indirect jump into a jump table is then made in `fn_a` along ④. Control flow is then transferred to `fn_q` through a *trampoline* along ⑤ and ⑥, which returns to `fn_a` through ⑦. Aside from the trampoline, the execution path is one that may be normally encountered in an arbitrary program.

Both forward and backward edges on a stateful CFG must be checked by the CFI policy during code execution. To aid this process while reducing execution overhead, we introduce five instructions extending the ISA. Table 7.1 lists our newly added instructions. Integral to the functionality of the system are the placement and semantics of the CFI instructions, as these aid in the construction and encoding of the stateful CFG.

cfibr Instruction. The `cfibr` instruction is issued before every call. Predicated with a label, the instruction pushes its label to the top of the LSS, thereby flagging the call site as active and adding a *new state* in execution to the stateful CFG. For instance, in Figure 7.4, the `cfibr` instructions in function `fn_a` push their accompanying labels onto the LSS. This is illustrated prior to calling `fn_b`, where `cfibr A1` pushes the label `A1` onto the stack. In the program's CFG shown in Figure 7.3, this is equivalent to encoding state S_1 . On any call, a new label is added

into the LSS, flagging a new call site as active and setting the new execution *state*. For a recursive function, if the last pushed label matches the label of the currently executing `cfibr`, a per label counter is incremented instead of pushing a new label into the LSS. This aids with reducing hardware overhead in the LSS memory.

cfiret Instruction. The `cfiret` instruction is issued after every call site and is predicated with a label. This label matches the label given by the `cfibr` preceding the indirect call instruction. In Figure 7.4, the call to `fn_b` is instrumented with a `cfibr A1/cfiret A1`, which encodes state S_1 in Figure 7.3. When a `cfiret` instruction is executed, the accompanying label is checked against the value on the top of the LSS. This evaluates the backward edge of the function, ensuring that the state of execution has been maintained during the return.

Instrumenting each instruction after every call site with a *unique label* eliminates granularity by removing erroneous edges \mathbb{E} . For instance, the state S_2 in Figure 7.3 is not a valid state for \mathbb{N}_5 , or equivalently, `cfiret A2` is not a valid return target for `fn_b`. Furthermore, the hardware CFI subsystem enforces that a backward edge must target a `cfiret` instruction. Our design allows us to limit the number of return targets to only the last active call site. After a return target label has been validated, it is popped from the top of the LSS.

The functions `setjmp()` and `longjmp()` are cases using non-local gotos and would raise a false positive if instrumented using `cfibr lbl/cfiret lbl` instruction pairs. We did not include specific support for these functions; however, we could easily design two separate CFI instructions and simply introduce a new CFI register. One of these instructions would store the current LSS pointer into the newly added CFI register. This instruction would be issued as part of the `setjmp()` function. During execution of the `longjmp()` instruction, the register would be written to the LSS pointer using the other instruction, thus unwinding the LSS.

cfiprc and cfiprj Instructions. The `cfiprc` and `cfiprj` instructions are issued before any call or indirect jump instruction, including tail call jumps. The instruction is predicated with a label representing the valid branch target. This label is stored in the LSR and subsequently checked after branching to a `cfichk` instruction. This ensures that only valid members of the node's \mathcal{N}_i can be targeted. Only valid targets as determined by the CFG are encoded with matching labels. Following the example in Figure 7.4, prior to calling function `fn_b`, the `cfiprc B` instruction stores label B in the LSR. A check is performed once the branch executes and reaches the `cfichk B` instruction. The jump table in `fn_a` is validated in a similar fashion, with the `cfiprj C` saving the label in the LSR and subsequent

jump targets containing the corresponding `cfchk` C instruction. A mismatch in labels or the presence of any instruction other than `cfchk` results in a violation of control flow, and a fault is triggered.

cfchk Instruction. The `cfchk` instruction is issued at every function entry or indirect jump target. Predicated with a label, it checks the value stored in the LSR and performs a comparison with its predicate. This validates forward edges, which are restricted to targeting `cfchk` instructions. For instance, in Figure 7.4, when function `fn_a` calls `fn_b`, its forward-edge state is encoded as label *B* and captured by the LSR. The `cfchk B` encoding maps `fn_b` as a valid target for \mathbb{N}_1 , where $\not\prec(\mathbb{N}_1) = \mathbb{N}_5$. Upon executing the `cfchk B` instruction, its label is matched against the current label stored in the LSR. Subsequent `cfchk` instructions are similarly handled.

Trampolines. A challenge with `cfprc` and `cfchk` instructions is differentiating edges in divergent nodes that point to a convergent node. Consider the case where two divergent nodes resulting from indirect calls \mathbb{N}_a and \mathbb{N}_b with different $\not\prec$ -functions target one common converging node \mathbb{N}_c . That is to say, $\mathbb{N}_c \in \not\prec(\mathbb{N}_a) \cap \not\prec(\mathbb{N}_b)$ and $\not\prec(\mathbb{N}_a) \neq \not\prec(\mathbb{N}_b)$. Instrumenting code alone with matching labels in `cfprc/cfchk` to verify the edge would necessarily require instrumenting all target nodes in $\not\prec(\mathbb{N}_a) \cup \not\prec(\mathbb{N}_b)$ to share the same label. This results in breaking the inequality between both $\not\prec$ -functions, effectively introducing erroneous edges in the CFG and therefore granularity. To preserve precision in the stateful CFG, trampolines are added to serve as unique bridges between these converging edges. Trampolines are instrumented with a `cfchk` instruction and a direct jump into the target function's body, bypassing the function's `cfchk` instruction. This instrumentation is illustrated in Figure 7.4, where function `fn_q` is assumed to be the target of multiple indirect calls. A trampoline `_tr_a_q` is added to serve as the indirect call target, thus precisely validating the function call.

7.6.3 Runtime Environment

A modified runtime environment is needed to support both CFI-instrumented and non-CFI-instrumented software. As such, a mechanism is needed to track all CFI-instrumented processes and inform the hardware. Figure 7.5 shows a high-level overview of the software stack.

As the figure illustrates, the operating system kernel is capable of handling both CFI and non-CFI processes. This is accomplished by modifying the process control block in the operating system kernel to track CFI processes and signal

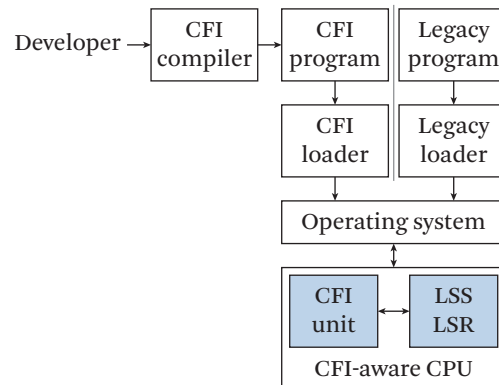


Figure 7.5 Software stack.

the underlying hardware when a CFI process is scheduled. A custom CFI loader is added and utilized by CFI-instrumented software, which utilizes the kernel’s syscall interface to activate CFI protection for the software in question.

7.6.4 CFI Hardware Infrastructure

As depicted in Figure 7.4, when the CFI-instrumented program executes, the newly added instructions control read/write operations on the LSR and LSS. However, to check that the CFI semantics are being precisely followed, we must supervise their execution. We propose a CFI finite-state machine (FSM) that supervises execution using the CFI instructions as input.

The primary design requirement for our dedicated CFI hardware infrastructure is to losslessly enforce any CFG with which we are provided and to efficiently enforce diverse CFI policies. Our platform does not constrict execution to a particular CFG or CFI policy. Instead, we propose a scalable, transparent subsystem capable of enforcing various CFG granularities, from precise to coarse. In this way, the vendor may choose the level of protection as determined by the security requirements of the application. Our hardware performs the necessary checks regardless of CFG coverage. The hardware will build a stateful CFG dynamically and enforce it based on the information provided during execution. It does so by recording the valid members for forward edges on the *label state register* and the valid backward edges on the *label state stack*.

It is necessary to maintain an LSS to tightly couple caller/callee pairs in order to ensure only the most recently executed forward edge is returned to. The LSR records forward edges separate from the label state stack to handle false positives.

For example, consider the jump table instrumented with precise, stateful CFI instructions in Figure 7.4, where the indirect jump stores its label C on the top of the label state stack. Upon reaching a valid `cfchk C` instruction, the label at the top of the LSS would be popped and matched. However, if fall-through were to occur, the next `cfchk C` instruction executed would similarly pop the label stored on the top of the LSS to be checked. This would of course trigger an error in our system, as the labels being checked would not match. It is therefore necessary to maintain separate forward- and backward-edge label state storage elements.

Label State Register. The LSR is a dedicated n -bit register accessible only by `cfiprc/j` and `cfchk` instructions. A `cfiprc/j lbl` triggers a write to the LSR. A `cfchk lbl` triggers a read from the LSR. The instruction label encodes valid targets as determined by the CFG for forward edges. Depicted in Figure 7.3, the valid forward-edge members are $\not\prec(\mathbb{N}_1) = \mathbb{N}_5$, $\not\prec(\mathbb{N}_2) = \mathbb{N}_3$, $\not\prec(\mathbb{N}_3) = \mathbb{N}_6$, and $\not\prec(\mathbb{N}_6) = \mathbb{N}_7$. These correspond to transitions ②, ④, ⑤, and ⑥, respectively, in Figure 7.4.

Label State Stack. The LSS is a dedicated $n \times m$ last-in-first-out buffer accessible only by `cfibr` and `cfiret` instructions. A `cfibr lbl` pushes the label to the top of the LSS. A `cfiret lbl` pops the label from the top of the LSS. The `cfibr lbl/cfiret lbl` pair encodes and checks stateful backward-edge targets. Backward edges are restricted to targeting valid members of the $\not\prec(\mathbb{N}_i)$ function based on the state obtained from a member of the $\preceq(\mathbb{N}_j)$ function. Depicted in Figure 7.3, these states are S_1 and S_2 . These correspond to transitions ③ and ⑦, respectively, in Figure 7.4. The depth of the LSS should be chosen to limit the occurrence of overflowing the LSS when encountering nested functions. If an LSS overflow is detected, the contents may be written to a protected region of memory.

CFI Finite-State Machine. The CFI finite-state machine (FSM) is shown in Figure 7.6 and executes in parallel to the instruction commit stage of the processor. Placement in the pipeline at the commit stage ensures that the FSM follows the precise CPU state, i.e., all earlier exceptions/interrupts have been handled before performing CFI operations. Each transition in the FSM requires a single cycle, so the FSM state is synchronized with in-order program execution.

The initial state of the FSM assumes an arbitrary point in program execution. If the program is CFI enabled, then transitions in the FSM will occur upon encountering CFI instructions only after being notified by the OS if the process is CFI enabled. Otherwise, the FSM will remain in the initial state for the process's lifetime. Non-CFI instructions return the current FSM state to its initial state. If either `cfibr` or `cfiprc/j` are executed, then the FSM transitions to state LSS or state LSR, respectively. This is functionally equivalent to a write because the semantics of both

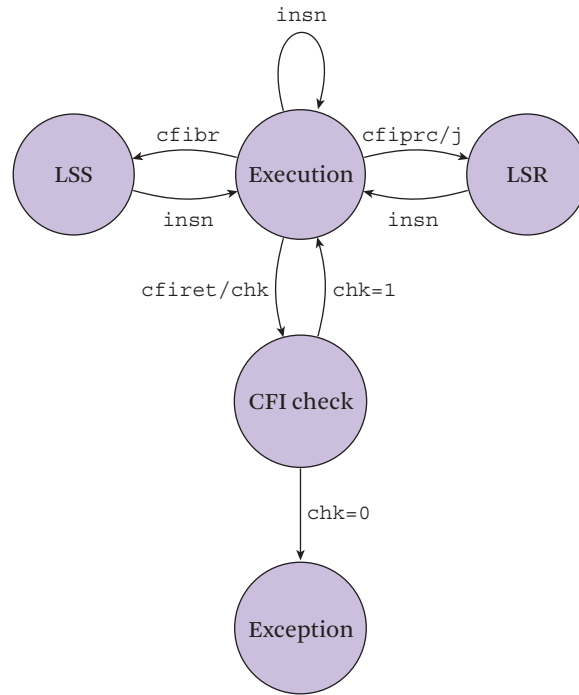


Figure 7.6 CFI FSM.

`cfibr` and `cfiprc/j` are to update the current label state. The state transitions to CFI check if either `cfiret` and `cfichk` instructions are executed. This is functionally equivalent to a read and compare because the semantics of both `cfiret` and `cfichk` is to check the current label state. If the label state check is validated, then the FSM state returns to the execution state; otherwise an exception is triggered.

Note that if the vendor provides any CFG then we losslessly enforce it. If in Figure 7.4 all `cfibr` labels match, or the labels at `cfichk` for `fn_b` and `fn_q` are grouped into an equivalence class, then the CFI FSM will not trigger an exception because the label state check will pass.

The CFI FSM generates control signals for reading from and writing to the LSS or LSR. It also monitors valid CFI transitions as determined by the stateful CFI semantics at runtime against the dynamically built CFG. Violations are detected if the intended execution flow, as given by the CFG, is not precisely executed. We group these violations into *execution-flow* and *logic-flow* violations. Any invalid transition in the FSM is considered a violation of `execution flow`. If a call/jump is executed,

then a `cfichk` must be targeted. Similarly, a return must target a `cfiret`. In addition, every call/jump instruction must be prefixed with a `cfiprc/j` instruction, and every call with `cfibr`. Logic-flow violations occur when an invalid label is encountered in either the LSS or LSR. For example, in Figure 7.4, if transition ② targets `cfichk Q` rather than `cfichk B`, then a logic-flow violation will be triggered.

7.7 Security Evaluation

7.7.1 Security Objectives and Requirements

The main goal of our hardware-enhanced CFI platform is to prevent code-reuse attacks. We must prevent runtime exploits that leverage either invalid backward edges, forward edges, or full functions. These include attacks that corrupt return addresses [Shacham 2007, Davi et al. 2014, Göktaş et al. 2014a], corrupt code pointers used in indirect calls/jumps [Checkoway et al. 2010, Checkoway and Shacham 2010, Carlini and Wagner 2014], or reuse entire functions [Schuster et al. 2015, Tran et al. 2011]. Finally, we must prevent runtime attacks that bypass CFI while adhering to its policies [Carlini et al. 2015e].

For our security discussion, we consider the adversary model and assumptions mentioned in Section 7.2. In particular, we assume that the application under protection has been provided with a precise CFG, and that the application has been instrumented with precise, stateful CFI instructions, as described in Section 7.6.2. We do not address the security of our hardware-enhanced CFI if given a *coarse* CFG.

7.7.2 Backward-Edge Code-Reuse Attacks

Conventional return-oriented programming attacks use backward edges (returns) to combine code sequences (gadgets) residing in the executable address space of an application to perform malicious actions. Traditionally, a memory write vulnerability is exploited allowing the attacker to inject a ROP payload, which is typically a number of return addresses each pointing to a gadget terminating in a return instruction [Shacham 2007]. Gadgets can be located at any arbitrary location in the application’s program space. Recent ROP attacks [Davi et al. 2014, Göktaş et al. 2014a] target only call-preceded code sequences, where a call-preceded code sequence is any instruction following a call.

Our hardware-enhanced CFI prevents backward-edge runtime attacks as described above, and in general, because they require redirection to invalid call-preceded instructions or arbitrary code locations. This is in direct violation of precise state preservation. Each call instruction is instrumented with a unique label that encodes the execution path’s state information with a `cfibr lbl/cfiret lbl`

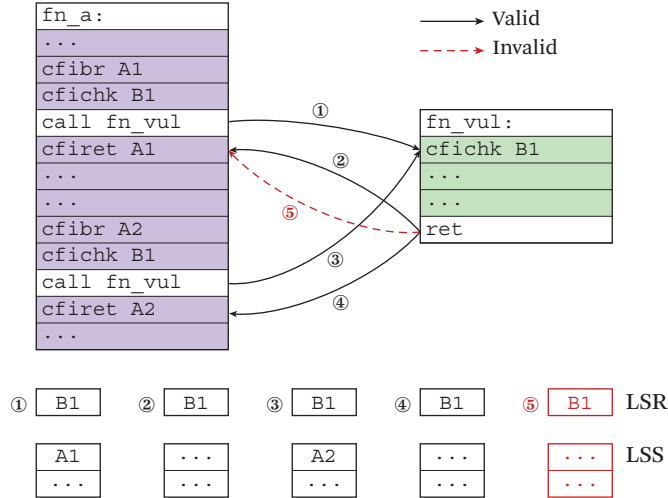


Figure 7.7 Illustrative backward-edge code-reuse attack.

instruction pair. A return instruction is only allowed to target a cfiret instruction if it is the most recent in the execution path history, i.e., it is a valid state. This is determined by checking the label at the top of the LSS against the cfiret lbl at the return target. Only cfiret instructions may be targeted by returns.

We use Figure 7.7 to discuss how our security requirements are fulfilled for runtime exploits that leverage these invalid backward edges. In Figure 7.7 we depict a function fn_a that consists of two direct function calls to fn_vul. The function fn_vul suffers from a memory corruption vulnerability that an attacker can exploit to corrupt a return address. Without CFI enforcement, the attacker can manipulate the return address to target any other instruction inside the program memory. However, in our CFI model, the return instruction must target the original caller. Our CFI state model also prevents an attacker from redirecting the control flow to a valid but currently inactive return place for fn_vul. As an example, assume an attacker attempts to redirect the control flow on edge ⑤. Since the valid return target, transition ④, is given by the precise state of control flow, an attacker is unable to exploit this backward edge. As described in Section 7.5.4, a return needs to target the *most recent forward-edge transition*. In our hardware-enhanced CFI, we encode the most recent forward-edge transition as a cfibr A2/cfiret A2 instruction pair. We enforce precise, stateful CFI by pushing the label A2 to the top of the LSS prior to any call and constraining the callee to returning to a cfiret instruction with a matching label.

7.7.3 Forward-Edge Code-Reuse Attacks

There are several variants of forward-edge runtime attacks, which typically use a corrupted code pointer to redirect control flow when dereferenced by an indirect call/jump. Jump-oriented programming attacks [Checkoway et al. 2010] use a dispatcher gadget, which acts as a virtual program counter (PC), to advance control flow through a dispatch table containing attacker-controlled addresses pointing to gadgets. These gadgets, rather than terminating with a return instruction, terminate with either an indirect call or jump instruction. Control is redirected back into the dispatcher to branch to the next gadget. ROP-without-return attacks [Checkoway and Shacham 2010] require a trampoline gadget that acts as a virtual PC to redirect control flow into gadgets terminating in an indirect call/jump. (Note that “trampoline” in Checkoway and Shacham [2010] has a different meaning than our usage.) Each terminating indirect call/jump instruction is used to point back into a trampoline wherein control flow can again be maliciously redirected. Variants of forward-edge runtime attacks exclusively target function entries [Göktas et al. 2014a, Carlini and Wagner 2014]. Typically, these are code sequences beginning at a function entry and terminating with an indirect call/jump.

Our hardware-enhanced CFI prevents forward-edge runtime attacks as described above, and in general, because they rely on either redirecting control flow to an invalid member or an arbitrary code location. Only valid indirect call/jump targets are allowed as given by their CFG members, per Section 7.5.2. This prevents the attacker from redirecting control flow to arbitrary locations in the application’s program space. Each benign call/jump target is instrumented with a `cfipr* lbl/cfichk lbl` pair that encodes its intended, benign target members. Redirection to an invalid member is prevented because its `cfichk lbl` state label encoding will not match the `cfipr* lbl` label in the LSR. Redirection to an arbitrary location in the application’s code space will not target `cfichk` instructions.

We use Figure 7.8 to discuss how our security requirements are fulfilled for runtime exploits that leverage invalid forward edges. Within Figure 7.8 we depict a vulnerable function `fn_vuln` attempting to exploit a corrupted code pointer to redirect control flow, ③ and ④. The vulnerable function suffers from a memory vulnerability that allows the attacker to exploit a corrupted code pointer. Without CFI enforcement, the attacker can manipulate the code pointer to target either `fn_d` or an arbitrary location in `fn_e`. Our hardware-enhanced CFI prevents attackers from targeting invalid code locations via indirect calls/jumps. In our system, the valid targets for the indirect call/jump instructions are given by its valid members \mathcal{N}_i per its precise CFG. For example, valid members for the indirect call in the vulnerable function are encoded with the `cfiprc B1/cfichk B1` instruction pair.

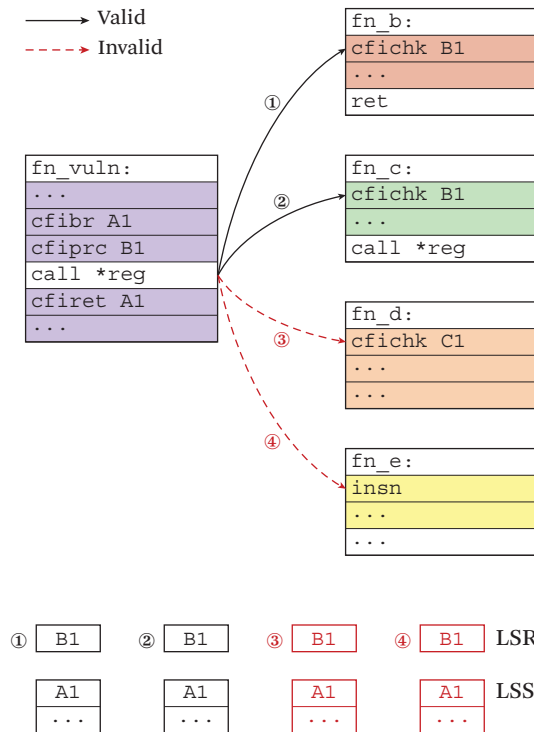


Figure 7.8 Illustrative forward-edge code-reuse attack.

We enforce precise, stateful CFI by storing the label B1 to the LSR prior to executing the indirect call and checking that its target `cfichk B1` label matches. Only valid members, as given by the CFG, are encoded with matching labels.

Note that our hardware-enhanced CFI architecture also includes protection against dynamic code-reuse attacks, i.e., attacks such as JIT-ROP that dynamically determine gadgets on executable memory pages [Snow et al. 2013]. These attacks exploit backward and forward edges that we instrument with CFI checks based on the program’s CFG.

7.7.4 Full-Function Code-Reuse Attacks

Conventional full-function reuse attacks use corrupted code pointers, along with attacker-controlled function arguments, to redirect control flow through a chain of existing `libc` functions [Solar Designer 1997a, Wojtczuk 1998, Solar Designer 1997b, Nergal 2001].

In a recent paper, [Tran et al. \[2011\]](#) were able to extend conventional Return-into-Libc (RILC) by demonstrating Turing-Complete RILC. Variants of full-function reuse attacks create counterfeit objects and fake virtual table pointers to redirect control flow to existing virtual methods in C++ programs [[Schuster et al. 2015](#)].

Our hardware-enhanced CFI prevents full-function reuse attacks because they rely on redirecting control flow to invalid indirect call/jump targets. Valid branch targets are instrumented with `cfiprc lbl/cfichk lbl` pairs. Only benign control-flow targets are encoded with matching labels. Redirection to an invalid control-flow target is prevented by checking that the label currently in the LSR matches the `cfichk lbl` label.

We use [Figure 7.8](#) again to discuss how our security requirements are fulfilled for runtime exploits that leverage full functions. Within [Figure 7.8](#) we depict a vulnerable function where the attacker may corrupt a code pointer and use it to redirect control flow to a function entry `fn_d`, ③. Without CFI enforcement, the attacker can freely manipulate the code pointer vulnerability to target any function entry in the executable address space of the application. Our hardware-enhanced CFI prevents attackers from targeting invalid functions. Valid functions are given by the precise CFG as a set of valid members \mathcal{N}_i . For example, valid members for the indirect call in `fn_vuln` are encoded with the `cfiprc B1/cfichk B1` instruction pair. We enforce precise, stateful CFI by storing the label `B1` to the LSR prior to executing the indirect call. We check that its targeted `cfichk lbl` label matches what is currently stored in the LSR. Only valid members, as given by the CFG, are encoded with matching labels.

Note that COOP attacks can be prevented in our design if the class hierarchy is correctly and precisely covered in the CFG. For instance, Google compiler extensions can be leveraged to extract such precise CFG information [[Tice et al. 2014](#)].

7.7.5 Control-Flow Bending

A recent attack [[Carlini et al. 2015e](#)], called Control-Flow Bending (CFB), demonstrates code-reuse attacks are possible while adhering to fully precise static CFI. In a CFB attack, attackers may corrupt a code pointer to call a valid function entry where a vulnerability exists, allowing it to corrupt a return address. They then may use the corrupted return address to return to any call-preceded site. In particular, CFB exploits any function with a vulnerability that can overwrite its own return address and adheres to CFI by returning to any location where this function was called.

Our hardware-enhanced CFI prevents CFB attacks because it requires redirection to any call-preceded slot in a stateless CFI-protected system. We offer

precise, *stateful* CFI so that only the most recently executed forward-edge transition may be returned to. As described above, this is ensured with a unique `cfibr lbl/cfiret lbl` instruction pair. A return instruction is only allowed to target a call-preceded slot if it is the most recent in the execution path history. Using Figure 7.7, the invalid return ⑤ is prevented from returning to `cfiret A1` because it is not the most recent call-preceded slot.

7.7.6 Security of Label State Stack/Register

Even though it is not a strict security requirement, our design only allows CFI instructions to access the LSS and LSR and avoids CFI data being loaded to main memory. Recall the recent CFI attack that corrupts offset pointers referencing a CFI jump table spilled to the program's stack for efficiency reasons [Liebchen et al. 2015]. We prevent this attack, and similar attacks that corrupt or disclose CFI data, by storing CFI-related data, such as labels, in a dedicated memory, LSS, and LSR.

7.8 Performance Evaluation

To evaluate the support of our hardware-enhanced CFI protection, we generated custom build tools, a custom runtime environment, and custom hardware infrastructure. This enabled us to (i) issue newly added CFI instructions in proper code locations, (ii) create unique CFI labels for any arbitrary application, and (iii) support CFI services within a rich OS environment on a hardware platform.

7.8.1 Build Tools

A set of modified build tools are needed in order to issue the necessary CFI instructions in the proper places. As such, in order to test the performance of our system, we developed an instrumented toolchain based on the GNU Compiler Collection (`gcc`) version 4.9.2, the GNU Binary Utilities (`binutils`) version 2.23 and `uClibc` (`uClibc`) version 0.9.33.2.

The compiler, `gcc`, was made to issue the `cfibr` and `cfiprc` instructions before any function call, with a corresponding `cfiret` instruction at the return site. Similarly, a `cfiprj` was issued before indirect jumps with a `cfichk` at function entries and at indirect jump targets. The assembler, `gas`, was modified to recognize these new instructions and emit the necessary machine code. As the C library, `uClibc`, is built by the compiler, only those routines written in assembly need to be manually instrumented; the others can be directly compiled without issue.

For testing and CFG instrumentation purposes, we wrote an IDA Pro plugin that extends the SPARC processor module bundled with the program. This enabled us to

automatically instrument backward edges in our binary. Forward edges for indirect jumps were instrumented by manually extracting jump table information from the binary and feeding this information to the plugin. For indirect calls, a new section with trampolines was added to the binary at compile time using a custom linker script. The trampoline was instrumented with the proper check instruction and an indirect jump to the target function. Using the plugin, indirect calls were rewritten as direct calls to the proper trampoline. As this instrumentation is equivalent to the one presented in section 7.6.2, it is sufficient for performance-testing purposes. We should stress, however, that this chapter does not present a general solution to CFG generation and limits itself to providing a mechanism that can be used with manual analysis to generate an estimate of the CFG of a program.

7.8.2 Hardware Platform

To evaluate the overhead of our CFI implementation on a real system, we integrated it into the open-source LEON3 processor distributed by the European Space Research and Technology Centre [Gaisler Research 2017]. The LEON3 is a 32-bit processor that implements the SPARC V8 ISA [SPARC 2017]. The synthesizable LEON3 core is equipped with a 7-stage pipeline, separate instruction and data caches, memory management unit, hardware floating-point units, AMBA 2.0 AHB bus, and on-chip debug support.

Modifications were made to the processor pipeline to incorporate the CFI FSM, LSR, and LSS in the `iu3.vhd` module. The modified processor is implemented on the Xilinx Spartan-6 FPGA evaluation board. The FSM, LSR, and LSS are placed in parallel to the write-back stage of the LEON3 pipeline. Their operations include read/write access to the LSS/LSR and FSM operations, which are synchronized with the write-back stage so that they do not stall the pipeline. Our CFI instructions are decoded as nop instructions on the pipeline, which ensures single-cycle latency as determined by the SPARC V8 ISA [SPARC 2017].

7.8.3 Hardware-Enhanced CFI Evaluation Results

7.8.3.1 Performance

For the evaluation of our hardware-enhanced CFI, we used the industrial standard EEMBC's CoreMark benchmark suite [EEMBC 2017]. This benchmark suite is designed to test a processor core's functionality, namely, its pipeline, memory access, and functional unit operations. It is made up of small C programs containing read/write, integer, and control operations whose workload models several commonly used algorithms, e.g., matrix manipulation, linked-list manipulation, state-machine operation, and cycle redundancy check [EEMBC 2017]. This suite

covers usage of code pointers and frequent conditional/unconditional branching, which provides a representative class of CFI-instrumented code coverage and performance overhead comparison.

CoreMark programs are instrumented with precise, stateful CFI instructions. We follow the instrumentation described in Section 7.6.2 using our build tools.

We also evaluated several SPECInt2006 benchmarks, namely, `bzip2`, `libquantum`, and `h264ref`. These are representative example programs from the group of business, scientific, and problem-solving workloads. We did not evaluate full SPEC because of resource constraints on the FPGA evaluation board. The FPGA board provides 128 MiB of main memory, whereas full SPEC requires at least 1 GiB. Each of the programs evaluated could be run within the memory constraints imposed by the FPGA platform. Additionally, porting full SPEC2006 would require significant engineering effort in resolving all dependencies. The benchmarks we evaluated offered a reasonable trade-off in build time and coverage.

SPEC benchmarks are also instrumented with precise, stateful CFI instructions. We again follow the instrumentation described in Section 7.6.2 using our build tools.

The results are shown in Figure 7.9, where the performance overhead on average is 1.75%, with a worst-case overhead of 3.5% for SPEC benchmarks and 0.5% for CoreMark. The average code size overhead is 13.5% across both SPEC and CoreMark. We should note that this overhead is directly related to the number of calls and indirect jumps in the binaries. As more calls and indirect jumps are contained in the program, more CFI instructions need to be issued.

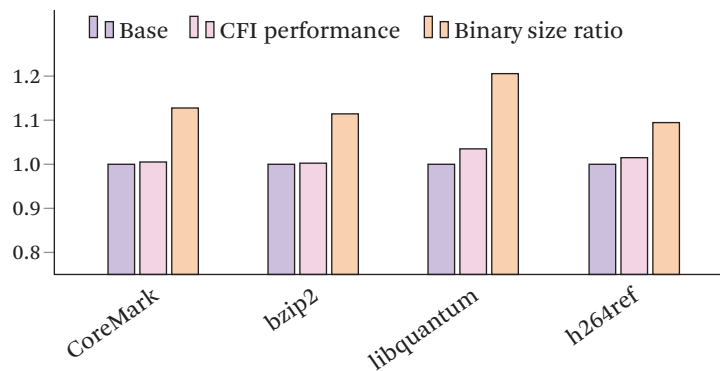


Figure 7.9 Normalized benchmark results.

Table 7.2 Evaluation of Area Overhead with CFI Implemented on a LEON3 Processor

	LEON3	LEON3 CFI	Percent Change
Combinational	8,759.073	8,996.952	2.72
Sequential	16,921.416	17,143.284	1.31
Total	25,680.589	26,140.236	1.78

7.8.3.2 Area and Timing Overhead

We integrated the micro-architectural features to support our hardware-enhanced CFI design elements, as outlined in Section 7.6.4 into the 7-stage pipeline of the LEON3 processor. Our hardware-enhanced CFI LEON3 core was synthesized with Design Compiler H-2013.03-SP5-3 using the Synopsys 32/28 nm generic library, a teaching library created for micro-electronic design education. We evaluated both area overhead and maximum clock rate. In general, smaller area ensures better resource usage and lower cost requirements. A faster clock ensures our hardware will not be on the critical path or violate existing timing constraints. Table 7.2 displays the area overhead caused by extending the pipeline with full CFI protection. The total area overhead seen is negligible at 1.78%. We also evaluated the maximum frequency at which our CFI FSM, LSS, and LSR implementation could be clocked. Our hardware-enhanced CFI could be clocked up to 3 GHz without incurring timing violations.

7.9 Related Work

CFI defenses have been proposed to prevent code-reuse attacks [Abadi et al. 2009, Budiu et al. 2006, Wang and Jiang 2010, Davi et al. 2012, Zhang and Sekar 2013, Zhang et al. 2013, Bletsch et al. 2011, Tice et al. 2014, Arias et al. 2015]. In their seminal work on CFI, Abadi et al. [2009] propose a label-based mechanism. In particular, indirect branch targets are marked with unique labels. Before an indirect branch, CFI validates whether the branch targets a pre-defined label. Whereas the original CFI proposal targeted applications running on an x86-based desktop PC, CFI has been also adapted to mobile applications [Davi et al. 2012] and hypervisor code [Wang and Jiang 2010]. Unfortunately, software-based instrumentation induces too high performance penalties. Even a recent implementation utilizing an optimized shadow stack [Dang et al. 2015] adds significantly more performance overhead than our hardware-based CFI implementation and still leaves the shadow stack unprotected in main memory.

A number of coarse-grained CFI approaches aim at tackling the performance overhead. Several solutions build on behavioral-based heuristics to detect (i) the execution of short instruction sequences [Pappas et al. 2013, Cheng et al. 2014] or (ii) indirect branch counters [Yao et al. 2013, Kayaalp et al. 2013]. Other CFI schemes relax the CFI policies, most notably, they force returns to target *any* call site [Zhang and Sekar 2013, Zhang et al. 2013, Bletsch et al. 2011, Pappas et al. 2013]. However, a number of recent attacks against CFI demonstrate that neither behavioral-based heuristics nor relaxed CFI policies withstand advanced code-reuse attacks [Göktaş et al. 2014b, Davi et al. 2014, Carlini and Wagner 2014, Schuster et al. 2014]. In contrast, our hardware-based CFI scheme allows for finer-grained policies that resist these latest attacks, while being highly efficient.

Architectural fine-grained CFI support, as proposed by Budiu et al. [2006], introduced hardware support for fine-grained CFI protection via integrity checking of control-flow graph encoding. For forward-edge protection, Budiu et al. [2006] leverage a CFI label register similar to our LSR. However, for backward-edge protection, they assume a shadow stack, which incurs more performance overhead compared to our LSS. Similarly, Davi et al. [Davi et al. 2014, Arias et al. 2015] introduce hardware-assisted CFI instructions but focus only on CFI backward edges and bare metal code. In contrast to previous work on hardware-assisted CFI [Budiu et al. 2006, Davi et al. 2014, Arias et al. 2015], we support highly efficient CFI for shared libraries, multitasking, and support of legacy code [Sullivan et al. 2016].

Control-Flow Locking (CFL), as proposed by Bletsch et al. [2011], prevents CRAs by asserting a lock value before executing each indirect control-flow instruction, and de-asserting it upon entry into a valid destination. However, CFL checking is on the critical path, and applications with a larger number of control-flow instructions, such as XML parsers and interpreters, will suffer significant performance degradation. Compared with our approach, we can offer the same protection but in a very efficient way.

Branch regulation [Kayaalp et al. 2012] is a hardware-assisted CFI approach that restrains control-flow behavior dynamically. Indirect branches are forced to target function entries or function bounds, and a return should target a call-preceded instruction. A Secure Call Stack (SCS) is implemented to restrict backward-edge CFI, and each stack entry is augmented with function bounds to support forward-edge CFI. This approach shares many similarities with ours but fails to handle dynamically linked libraries, stack unwinding, tail call optimization, and compatibility with non-CFI-instrumented programs. In contrast, our approach does not need knowledge of function bounds to enforce a policy, as this is determined via the

CFI label. Branch regulation requires the storage of bounds data to be included in the program executable.

[Onarlioglu et al. \[2010\]](#) eliminate unaligned indirect control-flow instructions from a program with the insertion of nop sleds. The remaining indirect control-flow instructions are then secured by enforcing that they can only be executed by means of an aligned entry. Return address encryption is implemented to prevent backward-edge CRAs. In addition, per-function cookies are used to constrain indirect jumps to the function's bounds. This approach reports a large increase in both binary file size and performance overhead.

7.10 Conclusion

Within this chapter we present the formal underpinnings of a precise stateful CFI policy, which enabled the design and implementation of a lossless, scalable, and highly efficient hardware-enhanced CFI platform. The new framework leverages dedicated CFI instructions to losslessly enforce any CFG and diverse CFI policies within our model. Our hardware-enhanced CFI significantly lowers the performance overhead when applied to several SPECInt2006 and CoreMark benchmarks. Further, if provided with a precise CFG we show comprehensive protection from many traditional and recently proposed code-reuse attacks. The goal of our work is the design and implementation of a hardware-enhanced CFI framework that can losslessly support CFI policies with varying precision. Generation of precise CFGs for real-world applications remains an open challenge.

Acknowledgments

This work was partially supported by the U.S. Department of Energy (DE-FOA-0001386), the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP. Orlando Arias is also supported by the National Science Foundation through the Graduate Research Fellowship Program. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Energy or the National Science Foundation.



Multi-Variant Execution Environments

Bart Coppens, Bjorn De Sutter, Stijn Volckaert

Memory corruption vulnerabilities are a common problem in software implemented in C/C++. Attackers can exploit these vulnerabilities to steal sensitive data and to seize or disrupt the system on which the software is executed. Memory safety techniques can, in principle, eliminate these vulnerabilities [Nagarakatte et al. 2009, Nagarakatte et al. 2010] but are prohibitively expensive in terms of runtime overhead [Szekeres et al. 2013].

Instead, modern operating systems and compilers deploy exploit mitigations such as Address Space Layout Randomization (ASLR) [PaX Team 2004a], Data Execution Prevention (DEP, a.k.a. W \oplus X) [PaX Team 2004b], and stack canaries [Cowan et al. 1998]. These exploit mitigations incur minimal performance overhead, but are limited in scope—often only defending against one particular type of exploit—and can be bypassed with only modest effort.

Up-and-coming exploit mitigations, such as control-flow integrity [Abadi et al. 2005a, Tice et al. 2014], require more effort to bypass [Göktas et al. 2014a, Davi et al. 2014, Carlini et al. 2015e, Evans et al. 2015, Schuster et al. 2015], but, similar to the aforementioned defenses, they defend only against attacks of one particular type: code reuse.

The ubiquity of multi-core processors has made Multi-Variant Execution Environments (MVEEs) an increasingly attractive option to provide strong, comprehensive protection against memory corruption exploits, while still incurring only a fraction of the runtime overhead of full memory safety. MVEEs have been shown to successfully defend against several types of attacks, including code reuse [Volckaert et al. 2015], information leakage [Koning et al. 2016], stack buffer overflows [Salamat et al. 2009], and code injection [Cox et al. 2006].

The underlying idea is to run several diversified instances of the same program, often referred to as variants or replicas, side by side on equivalent program inputs. The MVEE's main component, the monitor, feeds all variants these equivalent inputs and monitors the variants' behavior. The diversity techniques used to generate the variants ensure that the variants respond differently to malicious inputs, while leaving the behavior under normal operating conditions unaffected. The MVEE monitor detects the diverging behavior and halts the execution of the variants before they can harm the system. This implies that the variants must, to some extent, be executed in lockstep: potentially harmful operations in a variant are only executed when the consistency with the other variants has been validated.

In recent years, over half a dozen systems have been proposed that match the above description. While most of them show many similarities, some authors have made radically different design choices. In this chapter, we discuss the design of MVEEs and provide implementation details about our own MVEE, the Ghent University Multi-Variant Execution Environment, or GHUMVEE, and its extensions. GHUMVEE has been open sourced and can be downloaded from <http://github.com/stijn-volckaert/ReMon/>.

8.1 General Design of an MVEE

Broadly speaking, there are two key factors that distinguish the high-level designs of existing MVEEs: monitoring granularity and placement in the software stack. In this section, we review these factors, point out their implications, and justify the design choices we made for GHUMVEE.

8.1.1 Monitoring Granularity

Monitoring the variants' behavior can be done at many granularities, ranging from monitoring only explicit I/O operations to system calls, function calls, or even individual instructions. In practice, however, existing MVEEs either monitor at I/O-operation granularity or at system call granularity. Among the MVEEs that monitor at system call granularity, there are some that monitor all system calls, while the others monitor only "sensitive" calls. There is some debate over what the ideal monitoring granularity is. Coarse-grained monitoring yields better performance but might not guarantee the integrity of the system.

Most MVEEs monitor at system call granularity. On modern operating systems that offer page-level memory protection, each application is confined to its own address space. An application must therefore use system calls to interact with the

system in any meaningful way. The same holds for exploits. If the ultimate goal of an attack is to compromise the target system, then the attack's payload must invoke system calls to interact with the system.

It makes little sense to monitor at finer granularity levels for the sole purpose of comparing variants' behavior. The premise of multi-variant execution is that the variants are constructed such that they react differently to malicious input. While a given malicious input might be sufficient to seize control of one specific variant, it will not have the desired effect on other variants. These other variants will either crash or behave differently. As several authors have argued in the literature, both of these outcomes are visible at the system call level [[Salamat et al. 2009](#), [Cox et al. 2006](#)].

8.1.2 Placement in the Software Stack

The placement of the MVEE within the software stack has far-reaching consequences for the MVEE's security and performance properties. This placement is motivated by the conflicting goals of ensuring maximum security and maximum performance. To maximize performance, it is of vital importance to minimize the overhead on the interaction between the variants and the monitor. Since most monitors intervene in each system call invocation, such interactions can occur frequently.

All existing monitors interact synchronously with the variants. When a variant instigates an interaction with the monitor, it must wait until the monitor returns the control flow to the variant before it may resume its execution. To achieve maximum performance, it therefore is of vital importance to minimize this waiting time, which is dominated by the latency on the monitor-variant interaction. If the monitor runs as a separate process (Cross-Process, or CP), then the interaction latency is high because the kernel must perform a context switch to transfer the control from the variant to the monitor. Context switches are notoriously slow as they require a page table and a Translation Lookaside Buffer (TLB) flush [[Belay et al. 2012](#)]. CP monitors can therefore be detrimental for the variants' performance.

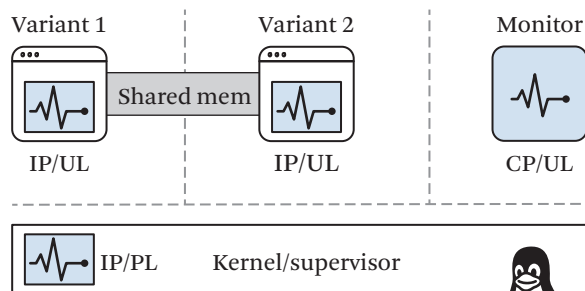
The advantage of CP monitors is that they are portable and easy to implement because they rely solely on the operating system's standardized debugging interfaces, and they are strongly isolated from the variants, since address spaces form a hardware-enforced boundary between processes. Placing the monitor outside the variants' address spaces therefore protects it from misbehaving variants. [Table 8.1](#) illustrates that most authors recognize the importance of such a hardware-enforced boundary. Almost all the existing monitors prioritize security over performance

Table 8.1 Classification of Existing MVEEs Based on Their Position in the Software Stack

	Unprivileged Level (UL)	Privileged Level (PL)
In-Process (IP)	VARAN [Hosek and Cadar 2015]	N-Variant Systems [Cox et al. 2006] RAVEN [Co et al. 2016] MvArmor [Koning et al. 2016]
Cross-Process (CP)	DieHard [Berger and Zorn 2006] Cavallaro [Cavallaro 2007] Orchestra [Salamat et al. 2009] Tachyon [Maurer and Brumley 2012] Mx [Hosek and Cadar 2013] GHUMVEE [Volckaert et al. 2013]	
IP+CP		ReMon [Volckaert et al. 2016]

and run cross-process. These monitors correspond with the label “CP/UL” (Cross-Process/Unprivileged Level) in Figure 8.1.

N-Variant Systems [Cox et al. 2006], RAVEN [Co et al. 2016], and MvArmor [Koning et al. 2016] are notable exceptions. These monitors run within the same address space as the variants (In-Process, or IP) but are protected from misbehaving variants because the monitors operate at a higher privilege level (kernel level for N-Variant Systems and RAVEN, supervisor level for MvArmor). This design is represented by “IP/PL” (In-Process/Privileged Level) in Figure 8.1. This is, at least in principle, the ideal approach. However, it does have the downside of enlarging the Trusted Computing Base (TCB). This is undesirable from a security standpoint [Rushby 1981]. An additional disadvantage is that the programming interfaces that are available at the privileged level are non-standardized and

**Figure 8.1** Possible placements of an MVEE in the software stack.

architecture-specific. IP/PL monitors are therefore significantly harder to port to other platforms than CP monitors.

VARAN finally implements a third design that is represented by “IP/UL” in Figure 8.1 [Hosek and Cadar 2015]. VARAN is a reliability-oriented IP monitor, embedded into the variants. It consists of several components, each of which can communicate directly with the variant in which it is embedded. VARAN primarily intends to increase the reliability of software, e.g., by running two variants, one with and one without a new patch applied to them, to test that the patch does not introduce unintended side effects. It therefore uses a less secure design than the aforementioned “CP/UL” and “IP/PL” MVEEs. VARAN’s authors also recognize this fact.

GHUMVEE is a security-oriented MVEE and is therefore implemented as a CP/UL MVEE. In Section 8.5 we also describe a hybrid design called ReMon [Volckaert et al. 2016]. ReMon is based on GHUMVEE, but it also includes an in-process component and a small kernel component, which makes ReMon a hybrid CP+IP/UL+PL MVEE.

8.1.3 Monitor-Variant and Monitor-Monitor Interaction

The MVEE’s monitor and the variants interact whenever the variants trigger an event that is subject to monitoring. These events typically include executing a system call and raising a processor exception (e.g., by executing a privileged instruction or causing a segmentation fault). Each interaction requires transferring the control flow from the variant to the monitor and back, and may require copying the register context or memory contents of the variant to the monitor. Several mechanisms exist to fulfill each of these tasks. The placement of the monitor in the software stack defines which mechanisms are available.

8.1.3.1 Control-Flow Transfer

The most trivial way to transfer control from the variant to the monitor and back is to invoke the monitor directly using a branch instruction. This is only possible for IP/UL monitors, which operate in the same address space and at the same privilege level as the variants. This control-flow transfer method is efficient but not very secure, as the variants typically invoke the monitor at their own discretion. Compromised variants could, for example, easily execute a system call without invoking the monitor first.

IP/PL monitors, which operate in the same address space but at a higher privilege level than the variants, cannot be invoked directly. Instead, these monitors

must be invoked by the kernel or supervisor's system call and trap handlers, either by patching these handlers or by installing hooks. When a variant executes a system call or triggers an exception, the processor transfers control to the kernel/supervisor's system call handler or exception handler, and the handler must then invoke the monitor. This interaction method is fully secure. Since the monitor invocation is handled by the kernel/supervisor, compromised variants are not able to escape the monitoring mechanism.

CP/UL monitors, which operate in a separate address space, cannot be invoked directly either. Instead, they rely on the operating system's debugging interface to "attach" to the variant, thus establishing a debugger-debuggee relationship between the monitor on one side and the variants on the other side. With such a relationship in place, the operating system will suspend the execution of the debuggee (variant) whenever it triggers an event that requires the attention of the debugger (monitor). The operating system will then schedule the monitor and make information about the event available to the monitor. This interaction method is also fully secure but incurs significant runtime overhead, since synchronous interaction between two separate processes requires context and TLB flushes.

ReMon, the hybrid design we describe in Section 8.5, consists of both an IP monitor and a CP monitor. ReMon has a system call broker component that intercepts system calls in kernel space and invokes the appropriate monitor based on a user-defined policy. The system call broker can invoke either the CP monitor, using the operating system's debugging interface, or the IP monitor, by pointing the user-space program counter at the IP monitor's known entry point before exiting kernel space. ReMon's system call handling mechanism is fully secure and more efficient than the one used by CP/UL monitors.

8.1.3.2 Register Context and Data Transfer

The MVEE's monitor requires access to the variant's register context, e.g., to read the system call number, and to the variant's virtual memory, e.g., to read system call arguments.

The variant's virtual memory can be accessed directly by all IP/UL and IP/PL monitors, as these monitors share their address space with the variant. IP/UL monitors also share their register contexts with the variant and can therefore access this context directly. IP/PL monitors do not share their register contexts with the variant. Instead, they must access a copy of this context. The processor stores this copy at a pre-defined location whenever it changes the privilege level. The performance overhead incurred by having to access the copy of the register context is negligible.

CP monitors can access neither the register context nor the variant's memory directly. Instead, they rely on the OS's debugging interfaces to transfer this information to the monitor's address space. Such transfers incur significant runtime overhead.

8.1.3.3 Inter-monitor Communication

Some MVEE designs, particularly the IP/UL ones, use multiple monitor instances, each embedded into or assigned to just one variant. The instances frequently communicate with each other, e.g., to verify if all variants execute the same system call. While most operating systems offer a variety of options for inter-process communication, all MVEEs that fall into this category use a ring buffer backed by a shared memory region for inter-monitor communication.

8.1.3.4 Performance Implications

[Koning et al. \[2016\]](#) conducted the most comprehensive study to compare monitor-variant interaction mechanisms. Through a series of micro-benchmarks, they showed that intercepting system calls and invoking the monitor in kernel space, similar to N-Variant Systems' IP/PL monitor [[Cox et al. 2006](#)], generally yields the highest performance. Using hardware virtualization features to intercept system calls and running the monitor in supervisor mode, similar to MvArmor's IP/PL monitor [[Koning et al. 2016](#)], is marginally faster than a kernel-based design if the monitor can emulate the system call, and up to $7.59\times$ slower if the call cannot be emulated. Intercepting system calls using the OS's debugging interface, as is done in all existing CP/UL monitors, is up to two orders of magnitude slower than the mechanisms used in IP/PL monitors.

Prior to this study, [Volckaert et al. \[2013\]](#) compared data transfer mechanisms used in CP/UL monitors and showed that GHUMVEE's `ptrace` extension yields significantly faster monitor-variant data transfers than Orchestra's shared-memory-based mechanism [[Salamat et al. 2009](#)], while the latter, in turn, is significantly faster than regular `ptrace`-based data transfers.

8.2 Implementation of GHUMVEE

GHUMVEE is a CP/UL monitor and thus relies on the OS's debugging interface to set up and communicate with the variants. GHUMVEE launches the variants by forking them off its main thread and by executing a `sys_execve` system call in the context of the forked-off processes. Prior to this call, the newly created variant processes establish a link between GHUMVEE's monitor and themselves by requesting to be placed under a monitor's supervision after which they raise a SIGSTOP signal.

The kernel suspends the variants after they have raised this signal, and it reports their status to the monitor. The monitor can then resume the variants and begin to monitor their execution.

8.2.1 Monitoring System Calls

Like most MVEEs, GHUMVEE monitors the variants' behavior at the system call interface by intervening at the kernel level at the entry and exit of every system call. GHUMVEE leverages the operating system's debugging API to place the variants under the monitor's control, to intercept the variants' system calls, and to run the variants in *lockstep*. The monitor suspends each variant that enters or exits from a system call until all variants have reached the same entry or return point. When this happens, the variants are said to have reached a *RendezVous Point* (RVP) (sometimes referred to as a synchronization point).

The monitor asserts that the variants are in equivalent states whenever they reach such an RVP by comparing the system call arguments. Two sets of system call arguments are considered equivalent if they are identical (in the case of non-pointer arguments) or if the data they refer to is identical (in the case of pointer arguments). [Salamat \[2009\]](#) gives a formal definition of equivalent states.

If the variants are not in equivalent states at an RVP, the monitor raises an alarm and takes the appropriate action. GHUMVEE considers all tasks that share an address space with one of the variants that caused the discrepancy as tainted, and it therefore terminates these tasks. Do note that this does not necessarily stop the entire program. It is becoming a common practice to compartmentalize complex programs, such as web browsers and web servers, into multiple, mostly independent tasks that do not share address spaces. In some modern web browsers, for example, every open tab is backed by a separate process. Should GHUMVEE detect a discrepancy when running multiple variants of such a process, it would only terminate the browser tab that caused the discrepancy.

Reliability-oriented monitors that are, e.g., used to test new software patches may differ from *security-oriented* monitors, such as GHUMVEE, with respect to system call monitoring. For example, VARAN does not enforce lockstep execution [[Hosek and Cadar 2015](#)]. Instead, it lets the master variant run ahead of the slave variants and caches the arguments and results of all the master variant's system calls so that they may be consulted by the slave variants at a later point.

8.2.2 Transparent Execution

Many system calls require special handling to ensure that the multi-variant execution is transparent to the end user. With the exception of runtime overhead,

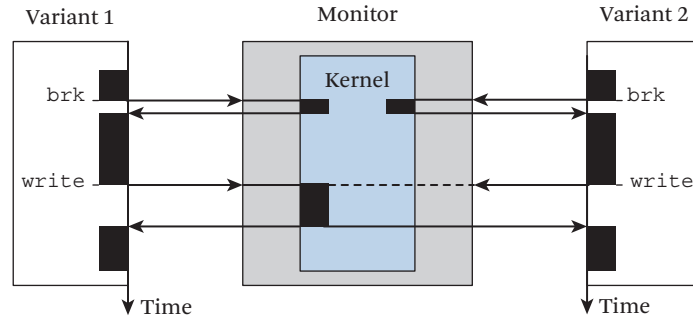


Figure 8.2 Transparently executing I/O-related system calls.

the end user should not be able to notice that more than one variant of the program is running. GHUMVEE therefore uses a master/slave replication model. One of the variants is the designated master variant and the other variants are slaves. GHUMVEE ensures that only the master variant can execute system calls that have visible effects on the rest of the operating system. Specifically, these are the system calls that correspond with I/O operations. Whenever the variants reach an RVP at the start of an I/O-related system call, GHUMVEE verifies that the variants are in equivalent states, and then overwrites the system call number in the slave variants with that of a system call with no visible effects. GHUMVEE currently uses `sys_getpid` for this purpose since it is a trivial and fast system call. When GHUMVEE subsequently resumes all variants, only the master variant executes the intended I/O operation.

At the next RVP, when all variants have returned from their system call, GHUMVEE copies the results of the system call from the address space of the master to the address space of the slave variants. We refer to this mechanism as *master calls*. System calls that do not require special handling, other than consistency checking, and that may therefore be executed by all variants are called *normal calls*. In Figure 8.2, the handling of the normal call `brk` is shown, as well as that of the master call `write`.

8.2.3 Injecting System Calls and Restarting Variants

On top of the above tasks, GHUMVEE can also inject new system calls and, as a result, rewind variants to their initial state. Injecting system calls can be useful to add new functionality to the variants transparently. To inject a system call in a variant, GHUMVEE waits until the variant has reached an RVP. At this point,

GHUMVEE stores a backup of the register context of the variant and overwrites the system call arguments.

Many system calls accept arguments that are stored in data buffers. To inject such arguments, GHUMVEE searches for a writable memory page in the variant that is large enough for the arguments. If the variant is multi-threaded, GHUMVEE searches for the variant's thread-local stack, in order not to corrupt memory that might be used by other tasks that share an address space with the variant.

GHUMVEE then reads and stores the original content of the memory page and writes the arguments into that page; it then resumes the variant and waits until the injected system call returns. At that point, GHUMVEE restores the original contents of the overwritten memory page and restores the original register context, prior to the system call injection.

Restarting variants to their initial state is a trivial extension of this system. To support restarting, GHUMVEE stores the original arguments of the `sys_execve` call that was used to start the variant as well as the environment variables [GNU.org 2017] at the time of the original `sys_execve` invocation. Whenever a variant reaches an RVP, GHUMVEE can restore the original environment variables and inject a new `sys_execve` call with those original arguments using the mechanism described above to restart the variant. GHUMVEE uses this restart mechanism to enforce disjoint code layouts, as we will explain in Section 8.4.

8.3 Inconsistencies and False Positive Detections

MVEEs must feed all variants the same input in order to guarantee that they behave identically under normal operating conditions. For explicit input operations, such as reading an incoming packet from a socket, the monitor can satisfy this requirement by applying the master call mechanism we described in Section 8.2.2 to system calls, such as `sys_read`.

In some cases this is not sufficient, however. Several sources of input can be accessed directly, without invoking any system calls. The variants often behave differently after reading input from such sources. This can lead to false positive detections by the monitor. In this section, we summarize the sources of input that can be accessed directly and describe how we provide consistent input from such sources to all variants.

8.3.1 Shared Memory

All commodity operating system kernels offer a file-mapping API and an Inter-Process Communication (IPC) API to share physical memory pages among multiple processes.

The file-mapping API, which can be accessed through the `sys_mmap` system call on Linux systems, allows programmers to associate individual physical memory pages with regions within a file on the file system. The associated file is often referred to as the *backing file*. When a page fault is triggered on a physical page that is backed by a file, which happens when this page is accessed for the first time, the operating system loads the contents for the page from the associated region in the backing file. The operating system will also write the contents of the page back to the file should the page ever become dirty.

The programmer can specify which region of the backing file each memory page corresponds to and whether or not the changes should be written back to the file. However, even if the programmer requests that changes be written back to the file, the operating system will only do so if the programmer has opened the backing file with read/write access. For some backing files, such as system libraries, the operating system denies any requests made by a non-privileged user to open the file with read/write access and instead allows only read access.

Programmers often use file mapping as an efficient way to access files. A mapped file can be accessed directly, without having to invoke `sys_read` or `sys_write` calls. The file-mapping API is also commonly used to create shared memory pages. A program can create a temporary file with read/write access and map this temporary file into its own address space. Other programs can then map the same file into their address spaces, thus sharing the associated physical memory pages with the program that created the file.

Programmers can also use the IPC API, which can be accessed through the `sys_ipc` or `sys_shmget/sys_shmat` system calls on Linux systems, to create and map shared physical memory pages not associated with a backing file. These pages have a unique identifier. Programs that know this unique identifier can map the associated physical pages into their virtual address spaces.

Shared memory pages often constitute a problem within an MVEE. Variants can read from shared memory pages without invoking a system call and, consequently, are not subject to the lockstep execution mechanism we discussed in Section 8.2.1 when doing so. The MVEE's monitor therefore cannot guarantee that the variants will read the same input from shared memory pages that are being written to by an external process. Similarly, the variants could also write to the pages directly, which prevents the MVEE's monitor from asserting that the variants write the same data to the pages.

A possible solution to this problem is to revoke the variants' access rights to all shared memory pages. Each read from or write to the shared pages would then result in a page fault. The operating system would translate this page fault into a `SIGSEGV` signal, which is normally passed down to the program so it can

invoke its signal handler. When a debugger is attached, however, a notification is sent to the debugger first and the actual signal is not passed to the program until the debugger has approved it. In an MVEE, this mechanism could be used to intercept all accesses to shared memory. For each SIGSEGV signal that results from a read operation on a shared memory page, the monitor could perform the read operation itself and replicate the results to all variants. For write operations, the monitor could perform the write itself. The monitor could then prevent the SIGSEGV signal from being delivered, thus effectively emulating all accesses to the shared memory pages. Emulating accesses to shared memory is, unfortunately, prohibitively slow [Maebe et al. 2003] and completely negates the performance benefits of using shared memory in the first place.

In GHUMVEE, we therefore opted to deny all requests to map shared memory, unless the monitor can assert that the accesses to the shared memory will not result in inconsistencies. Specifically, GHUMVEE denies all requests to map shared memory through the System V IPC API, since any pages mapped through this API can always be written by external processes that know the page identifiers.

For file mappings, on the other hand, GHUMVEE does allow read-only shared mappings that are backed by files to which the user does not have write access. Such mappings have content that is completely static (i.e., the pages cannot be written to by either the variants or any external process that runs at the same privilege level). The monitor can therefore still guarantee that the variants will receive the same input. Allowing read-only shared mappings is necessary to support dynamically linked programs since the program interpreter's preferred method of loading shared libraries is by mapping them using the file-mapping API.¹

GHUMVEE does not allow read/write shared mappings. The monitor generally returns an EPERM error when a variant attempts to establish such a mapping, thus indicating that the mapping is not allowed. In specific cases, however, read/write shared mappings are used not to communicate with external processes but instead simply as an efficient way to access files. To handle these cases, we implemented a *mapping-type-override method*. With this method, GHUMVEE changes the mapping type from shared to private by overriding the arguments of the `sys_mmap` call that is used to set up the mapping. Private mappings are implemented using Copy-On-Write (COW) paging. The operating system will therefore create a private copy of the privately mapped page when a variant attempts to write to it for the first time. From that point onward, external processes can no longer influence the contents of

1. The program interpreter is a user-space OS component responsible for loading programs and setting up their initial virtual address space.

the privately mapped page, which eliminates the need for the monitor to replicate the contents of the pages to all variants. The monitor does, however, still verify whether the variants all write the same contents to the privately mapped pages by comparing the page contents when they are unmapped. If the contents of the pages do not match, the monitor raise an alarm. If they do match, however, the monitor writes the contents back to the backing file.

GHUMVEE's handling of shared memory is similar to Cavallaro's MVEE [Cavallaro 2007] but is more advanced than other security-oriented MVEEs because those do not support the mapping-type-override method.

8.3.2 Timing Information

Interactive and real-time applications frequently need to measure the length of a time interval to guarantee that they function correctly. Media players, for example, need to know exactly when to start rendering a frame. For such applications, the timing information must be accurate, precise, and accessible with minimal overhead. Both processor vendors and kernel programmers therefore offer an interface to access timing information with minimal overhead.

All x86 processors since the original Pentium support the `RD TSC` instruction, which reads the value of a special-purpose register that counts the number of clock cycles since the processor was powered on [Intel 2014]. This number can be divided by the clock frequency to accurately measure the length of a time interval.

The 64-bit x86 version of the Linux kernel, as well as recent versions of the 32-bit x86 kernel, implement the Virtual Dynamic Shared Object (VDSO) [Linux Programmer's Manual 2017a]. The VDSO is a small, dynamically linked library that is mapped into every running program's virtual address space. It consists of two memory pages: an executable memory page that contains code and a read-only memory page that contains timing information. The VDSO implements virtual system call functions. Each virtual system call is an optimized version of one of the system calls that is exposed by the kernel. Unlike the system call they correspond to, however, the virtual system calls execute entirely in user space, thus avoiding the often costly mode and/or context switches that come with the execution of a normal system call. Linux currently offers virtual system calls for each API that exposes timing information.

Both the `RD TSC` instruction and the VDSO are therefore sources of timing information that can be accessed without invoking an actual system call. Once again, an MVEE's monitor cannot guarantee that the variants that access this information receive consistent input.

GHUMVEE implements work-arounds for both problems. GHUMVEE's monitor sets the Time Stamp Disable (TSD) flag in the CR4 register of the processor within the context of each running variant [Intel 2014]. Setting this flag discards the variants' privileges to execute the RDTSC instruction. Whenever the variants try to execute an RDTSC instruction, the processor raises a general protection fault. The operating system translates this fault into a SIGSEGV signal and notifies the monitor accordingly. Whenever the monitor receives such a notification, it disassembles the instruction that caused the fault. If the instruction is indeed an RDTSC, GHUMVEE executes the instruction on the variants' behalf and replicates the results.

To eliminate the inconsistencies caused by the VDSO, GHUMVEE overrides the arguments of each `sys_execve` system call. This call is used to execute a program. GHUMVEE changes the name of the program that must be executed into the name of a small loader program we have created. This small loader program, which we aptly call the GHUMVEE Program Loader (GPL),² deletes the ELF auxiliary vector entry argument that specifies the location of the VDSO [Linux Programmer's Manual 2017b]. Afterward, GPL manually maps the original program into the virtual address space, sets up the initial stack exactly as it would have been set up had GHUMVEE not overridden the arguments of the `sys_execve` call, and passes the control to the original program. A program never invokes the VDSO directly but instead uses the wrappers provided by the C standard library (`libc`). If the ELF auxiliary vector entry for the VDSO is missing, however, `libc` falls back to using the original system call that each virtual system call corresponds to. These original system calls are intercepted by GHUMVEE's monitor. An alternative solution could be to replace the VDSO with a custom library that leverages GHUMVEE's USRVP replication infrastructure (see Section 8.3.7) to replicate the master variant's system call results to all slave variants.

To the best of our knowledge, GHUMVEE is the only existing MVEE that handles the RDTSC instruction correctly. Along with Hosek and Cadar, who independently proposed a solution of their own, we were also the first to handle system calls in the VDSO correctly [Hosek and Cadar 2015]. Our proposed solutions have a minimal performance impact on the many applications we tested. The RDTSC instruction is typically only used during the start-up and shutdown of a program, e.g., to measure the runtime of individual threads. Our proposed solution for the VDSO does significantly impact the latency on executing individual timing-related system calls. In Section 8.5, we propose a new monitor design that reduces this impact to a bare minimum.

2. GPL is available under the BSD 3-clause license at http://github.com/stijn-volckaert/ReMon/tree/master/MVEE_LD_Loader.

8.3.3 File Operations

Multi-variant execution should be transparent to the variants and to external observers. GHUMVEE therefore uses the master call mechanism to ensure that I/O operations are only performed once. With this mechanism, only the master variant performs the actual I/O operations, and the monitor replicates the results to the slave variants. Intuitively, it might also make sense to use master calls for system calls that open, modify, or close file descriptors. Regular files, however, might be mapped into the variants' address spaces using the file-mapping API. If we apply the master call mechanism to such files, any subsequent file-mapping request would fail in all slave variants. GHUMVEE therefore allows slave variants to open, modify, and close file descriptors for regular files.

The master call mechanism must still be used to open, modify, and close other file descriptors, such as sockets, however. Certain system calls, such as `sys_accept`, operate only on file descriptors associated with sockets that are in listening state. Since only one socket can listen on each port, GHUMVEE uses master calls for all socket operations.

Since some file descriptors are opened only in the master variant and some are opened in all variants, the same file descriptor might have different values in the different variants. As GHUMVEE must ensure that the multi-variant execution is transparent to the variants, the monitor replicates the same file descriptor values to all variants, regardless of whether or not they have actually opened the file. Whenever the variants perform a normal system call that they must all execute, GHUMVEE maps the replicated file descriptor value back to the original file descriptor value at the system call entrance site. When the call returns, GHUMVEE maps the original file descriptor value back to the replicated file descriptor value.

Although few details on how other MVEEs handle file descriptors are available, we assume that most of them use a similar solution to ours. One notable exception is Orchestra. Orchestra's monitor performs most I/O operations on behalf of the variants. Variants running in Orchestra therefore do not open any file descriptors other than those for regular files.

8.3.4 Signal Handling

UNIX systems use signals as a general-purpose mechanism to send processes notifications [[Linux Programmer's Manual 2017c](#)]. Each notification has a signal number associated with it, and the signal number generally defines the notification's meaning. For example, when a program performs an invalid memory access, the kernel sends it a SIGSEGV signal.

Broadly speaking, we can distinguish between two kinds of signals. *Control-flow signals*, such as SIGSEGV, are sent as a direct consequence of a program's normal control flow. The program cannot continue executing until the kernel has handled the control-flow signal. If a control-flow signal is not blocked and the program has registered a signal handler function for the signal in the handler table, the signal is delivered synchronously. *Asynchronous signals*, on the other hand, originate from an external source, and the program may continue executing while the kernel is handling the delivery of an asynchronous signal.

Supporting control-flow signals in an MVEE is generally straightforward, as they occur at the same point in each variant's execution. Supporting asynchronous signals sent to the variants is extremely challenging, however. These signals can easily trigger behavioral divergences in the variants if their delivery is not meticulously controlled by the MVEE's monitor. Since the monitor generally only intervenes in the variants when they execute a system call, the variants may execute freely for the most part. Consequently, one variant can easily run ahead of the others, and they are generally not in equivalent states until they reach an RVP.

If the behavior of a signal handler used to handle an asynchronous signal depends on the state of the variant in any way, delivering these asynchronous signals directly all but guarantees behavioral divergence and thus a false positive alarm. MVEEs that support asynchronous signals therefore attempt to defer their delivery until the variants reach an RVP. At an RVP, the variants are in equivalent states, and the asynchronous signals can be delivered synchronously without inducing a divergence.

While the general principle of deferred synchronous signal delivery is simple, its implementation is not. To the best of our knowledge, GHUMVEE is the only MVEE that overcomes all the intricacies of asynchronous signal delivery and implements deferred synchronous signal delivery correctly. We refer to Volckaert's PhD dissertation for a full overview of challenges that need to be overcome when implementing deferred signal delivery in an MVEE [Volckaert 2015]. These challenges include, but are not limited to, correct handling of blocked and ignored signals, support for per-thread signal masks, support for system call interruption, support for master call interruption, and support for `sys_sigsuspend` and `sys_rt_sigsuspend`.

GHUMVEE's mechanism for handling asynchronous signal delivery is optimized for correctness rather than performance. During our evaluation, we concluded that almost every program that relies on signal handling still functions correctly inside GHUMVEE. The only exception is the `john-the-ripper` program in the `phoronix 4.8.3` benchmark suite. This program waits for signals to be de-

livered in a busy loop, in which no system calls are used. Therefore, if GHUMVEE intercepts a signal that is delivered to the variants, it indefinitely defers the delivery of the signal because the variants never reach another system call RVP. One solution could be to start a timer when a signal is intercepted and to force the delivery of the signal when the timer expires.

Orchestra’s mechanism for signal handling is optimized for performance rather than correctness [Salamat et al. 2009]. Orchestra uses a heuristic to determine whether a signal can be safely delivered, even if its variants have not reached a system call RVP yet. However, Orchestra does not handle signals that interrupt system calls correctly.

VARAN’s signal-handling mechanism is ideal with respect to performance and correctness [Hosek and Cadar 2015]. VARAN is an IP monitor and therefore does not rely on the `ptrace` API. Furthermore, VARAN forces its follower variants not to invoke system calls at all. Instead, the follower variants just wait for the results of the leader variants’ system calls. In VARAN, the leader variant accepts and processes incoming signals without delay. While the follower variants generally do not receive signals at all, the leader variant logs the metadata associated with the signal into the event streaming buffer. This metadata provides the follower variants with sufficient information to replay the invocation of the signal handler truthfully.

8.3.5 Address-Sensitive Behavior

Most of the sources of inconsistencies in the behavior of single-threaded variants can be eliminated or mitigated by the monitor itself. The one notable exception is *address sensitivity*, a problem frequently encountered in real-world software. The monitored behavior of address-sensitive programs depends on their address space layout. Any form of code, data, or address space layout diversification we use in the variants can therefore lead to false positive detections by the monitor. We have identified three problematic idioms that lead to address sensitivity, and we discuss them now.

Address-Sensitive Data Structures. We have frequently encountered programs that use data structures whose runtime layout and shape depends on numerical pointer values. This practice is especially common among programs that rely on `glib`, the base library of the GNOME desktop suite. `glib` exposes interfaces that C programs may use to create, manage, and access hash tables and binary trees. The default behavior of these `glib` data structures is to insert new elements based on their location in memory: the (hash) keys used to select buckets and to order elements are

based on the numerical pointer values. The problem in general is that the address-sensitive behavior induces changes in the shape of allocated data structures, i.e., in the way they are linked via pointer chains.

Applying diversification techniques that result in diversified address space layouts and shapes eventually yields divergences in the variants' system call and synchronization behavior. For example, in address-sensitive hash tables an insertion of the same object can trigger a hash table collision and a subsequent memory allocation request (e.g., through a `sys_mmap` call) to resize the table in some variants but not in others.

While it might seem sensible to tolerate small variations in the system call behavior, we typically cannot allow variations in the memory allocation behavior of the variants, which we are bound to see in programs with address-sensitive data structures. Variations in the memory allocation behavior cause a ripple effect in multi-threaded variants: tolerating a minor discrepancy early on leads to bigger and bigger discrepancies in the synchronization behavior and, consequently, in the system call of the variants, to the point where we can no longer distinguish between benign discrepancies and compromised variants.

Dynamic memory allocators are the instigators of this ripple effect. For example, GNU libc's `ptmalloc` attempts to satisfy any memory allocation request by reserving memory in one of its arenas. All accesses to the allocator's internal bookkeeping structures must be thread-safe. It therefore relies on thread synchronization to ensure safety. As we discuss in the next section, GHUMVEE replicates the master variant's synchronization operations in the slave variants. Thus, if the variants behave differently with respect to memory allocations, the replicated synchronization information might be misinterpreted by other variants because it does not match their actual behavior. From that point onward, such variants will no longer replay synchronization operations in the same order as the master and will therefore typically diverge from the master with respect to the system call behavior.

Allocation of Aligned Memory Regions. An additional problem we identified in `ptmalloc` is its requirement that any memory region it allocates through `sys_mmap` is aligned to a large boundary of, e.g., 1 MiB. The operating system only guarantees that newly allocated memory regions are aligned to a boundary equal to the size of a physical memory page. To bridge this gap, `ptmalloc` always allocates twice the memory it needs and subsequently deallocates the region before and the region after the boundary. When running multiple variants that use this memory allocator, the sizes of the deallocated upper and lower regions might differ. Worse yet, in some cases the newly allocated memory might already be aligned to the desired

boundary and `ptmalloc` therefore only deallocates the upper region. This might trigger false positive detections in MVEEs that execute their variants in lockstep, since some variants may deallocate the lower and the upper region while others only deallocate the upper region.

Writing Output That Contains Pointers. Some programs output numerical pointer values. Unlike the previous problematic idioms, writing out pointers often leads to only minor differences in the system call behavior, and we have not encountered any cases where writing out pointers triggers a ripple effect. It is therefore sensible to tolerate minor differences in the program output.

One problem to deal with, however, is that pointers are not always easily recognizable in a program's output. Some programs encode pointers, e.g., by storing them as an offset relative to a global variable or object. Encoded pointers are often smaller than the size of a memory word.

Similarly, many programs and libraries use partially uninitialized structures as arguments for a system call. The uninitialized portions of these structures may contain leftovers of previous allocations. These leftovers often include pointers. While it can often be considered a bug to pass uninitialized structures to the kernel, there are cases where the programmer and the compiler are not to blame. An optimizing compiler aligns members of a data structure to their natural boundary. If necessary, padding bytes are inserted between the members. These padding bytes are never used, and it is therefore acceptable that they are not initialized. If that is the case, and they overlap with remainders of previously allocated objects, this can once again lead to minor variations in the output behavior.

All of the above idioms lead to discrepancies in the variants' system call behavior and/or synchronization behavior. Small variations in the system call behavior can in some cases be tolerated, especially if the variations are limited to the arguments of a single system call. Variations in the synchronization behavior cannot be tolerated, however, as we argue in the next section.

8.3.6 Nondeterminism in Parallel Programs

Except for the few cases we discussed in the previous sections, single-threaded variants produce the same outputs when given the same program inputs. The same is not true of multi-threaded variants, in which threads may communicate directly through shared memory, without using system calls. In these variants, the output also depends on the runtime thread interleaving. Security-oriented MVEEs, which run variants in strict lockstep, must therefore control the thread interleaving such that each variant makes the same system calls with equivalent arguments.

Two lines of research address exactly this challenge. On the one hand, there are the Deterministic Multi-Threading (DMT) systems, which repeat the same thread interleaving when given the same program inputs [Basile et al. 2002, Reiser et al. 2006, Berger et al. 2009, Liu et al. 2011, Merrifield and Eriksson 2013, Cui et al. 2013, Olszewski et al. 2009, Lu et al. 2014, Devietti et al. 2009, Bergan et al. 2010, Zhou et al. 2012]. On the other hand, there are online Record/Replay (R/R) systems, which capture the thread interleaving in one running instance of a program and impose this captured schedule in a concurrently running instance [Basile et al. 2006, Lee et al. 2010, Basu et al. 2011].

DMT systems are an ill fit in the context of MVEEs, as such systems are likely to impose a different thread interleaving whenever the program code or code layout changes. Thus, running diversified variants in which the code layout differs with near certainty and with DMT on top of an MVEE will still result in different thread interleavings and, consequently, divergent behavior. We refer to our earlier work for an in-depth reasoning to support this argument [Volckaert et al. 2017].

In GHUMVEE, we therefore opt for the second option. R/R systems usually capture the thread interleaving at the granularity of synchronization operations (e.g., pthread mutex operations). The underlying thought is that in data-race-free programs, any inter-thread communication must, by definition, be protected by critical sections. Imposing an equivalent synchronization operation schedule in every execution of the program thus trivially leads to a thread interleaving that is equivalent for each run. Since synchronization operations are not likely to differ between diversified variants, R/R systems are a much better fit than DMT in the context of an MVEE.

8.3.7 User-Space Rendezvous Points

In order to maintain equivalent system call behavior, even in parallel programs or in programs that feature address-sensitive behavior, we introduce *user-space rendezvous points* (USRVPs). Conceptually, we add these USRVPs to any operation in the variants' code if (i) the operation may affect the variants' system call behavior and (ii) the operation may produce different results in each variant. At each USRVP, we insert calls to a replication agent. This replication agent is a shared library we forcefully load into each variant's address space. As shown in Figure 8.3, the replication agent captures the results of the instrumented operation in the master variant and stores them in a circular buffer that is visible to all variants. The agent then forces the slave variants to overwrite the results of their own instrumented operations with the results produced by the master variant.

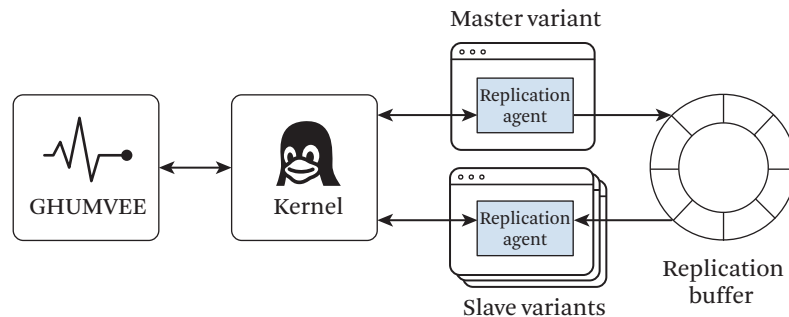


Figure 8.3 Using replication agents to replicate nondeterministic program behavior.

We developed three components that help a developer with the implementation of USRVs and replication agents:³

The GHUMVEE replication API. The GHUMVEE replication API can be used to generate the replication agents and the USRVs. The API consists of a set of preprocessor macros that expand into C functions. These C functions implement the recording and forwarding logic of the replication agent. In the master variant, the generated function can retrieve the results of the instrumented operation by calling a programmer-specified function. It can then record the input into a circular buffer. In the slave variants, the generated function retrieves the results from the circular buffer. The replication API further allows the programmer to specify whether or not the slaves should also execute the instrumented operation. This may be necessary, e.g., if the instrumented operation has side effects that may affect future system call and synchronization behavior.

The Lazy hooker. The generated USRV functions can be embedded in the variant by registering them with a shared library, which we call “the lazy hooker.” This library monitors the dynamic loading process of the variants and determines whether or not a USRV generated with the above API should be installed. At the time of the registration, the lazy hooker may insert the USRV function immediately, if the specified library has already been loaded, or it can defer the insertion until the program loads the library.

3. We refer interested readers to Volckaert’s Ph.D. thesis for an extensive discussion that includes usage examples [Volckaert 2015].

The LinuxDetours library. We insert the USRVP functions using `LinuxDetours`, a runtime code-patching library we developed for use in GHUMVEE. The library is named after Microsoft's `Detours` library and implements a subset of the official `Detours` API [Hunt and Brubacher 1999]. `LinuxDetours` can redirect calls to functions and generate trampolines that may be used to call the original function, without interception.

8.3.7.1 USRVP Applications

GHUMVEE currently relies on USRVPS for two purposes. First, we add USRVPS to all thread synchronization operations in order to embed our R/R system into the variants. Thanks to our replication APIs and infrastructure, we were able to construct an R/R system that captures the order of thread synchronization operations in the master variant and replays an equivalent order in the slave variants. Contrary to most existing R/R systems, which capture only high-level synchronization operations, such as `pthread` mutex operations, we capture the order of *all* thread synchronization operations, including individual atomic instructions.

Second, we add USRVPS to operations on address-dependent data structures, such as the ones described in Section 8.3.5. At these USRVPS, we capture values such as pointer hashes and pointer comparison results used by sorting algorithms, and we force all the variants to use the same values. Thanks to our USRVPS, GHUMVEE can ensure that address-dependent data structures potentially allocated at different memory locations have the same shape in all variants.

It is important to note that USRVPS are application specific. Although we believe that the identification of USRVP insertion points can be automated to some extent, some help from the application developer will always be required. We refer to our earlier work for details on USRVP insertion point identification, the construction of efficient replication agents and R/R systems, and automation opportunities [Volckaert et al. 2013, Volckaert et al. 2017].

Evaluation and Comparison with other MVEEs

We applied our replication API and infrastructure to run a variety of applications, including the SPLASH-2x and PARSEC 2.1 parallel benchmark suites and two popular, though now outdated, desktop programs: the Firefox 3.6 browser and the LibreOffice 4.5 office suite.

To run the parallel benchmark suites, we constructed a replication agent that contains an R/R system and embedded this agent into `glibc`. This R/R replication agent is called from the more than 1,000 USRVPS we added to thread synchroniza-

tion operations in the `glibc`, `libgomp`, `libpthread`, and `libstdc++` libraries, as well as a few of the program binaries.

To run Firefox and LibreOffice, we constructed five different replication agents, each one supporting a particular address-sensitive data structure.

To the best of our knowledge, GHUMVEE is the only MVEE to date that has any provisions to eliminate inconsistencies resulting from address-sensitive behavior and user-space synchronization operations.

8.4 Comprehensive Protection against Code-Reuse Attacks

In 2007 Shacham presented the first Return-Oriented Programming (ROP) attacks for the x86 architecture [Shacham 2007]. He demonstrated that ROP attacks, unlike return-to-libc attacks, can be crafted to perform arbitrary computations, provided that the attacked application is sufficiently large. ROP attacks were later generalized to architectures such as SPARC [Buchanan et al. 2008], ARM [Kornau 2010], and many others.

8.4.1 Disjoint Code Layouts

As an alternative protection against user-space ROP attacks, we present Disjoint Code Layouts (DCL). With this diversification technique, GHUMVEE ensures that no code segments in the variants' address spaces overlap. Lacking overlapping code segments, no code gadgets co-exist in the different variants to be executed during ROP attacks. Hence no ROP attacks can alter the behavior of all variants consistently. Our design and implementation of DCL offers many advantages over existing solutions:

- DCL offers complete immunity against user-space ROP attacks rather than just raising the bar for attackers.
- The execution slowdown incurred by this form of diversification is minimal.
- A single version of the application binary suffices to protect against ROP attacks. Optionally, our monitor supports the execution and replication of multiple diversified binaries of an application to protect against other types of exploits as well.
- DCL is compatible with existing compilers and existing solutions such as stack canaries [Cowan et al. 1998] and control-flow integrity [Abadi et al. 2005a, Tice et al. 2014].
- Unlike some existing techniques for memory layout diversification in MVEEs, DCL causes only marginal memory footprint overhead within the protected

application's address space. Thus, DCL can protect programs that flirt with address space boundaries on, e.g., 32-bit systems. Systemwide, DCL does of course still cause considerable memory overhead due to its duplication of process-local data regions, such as the heap and writable pages. Still, GHUMVEE with DCL outperforms memory-checking tools in this regard.

Our technique of DCL is implemented mostly inside GHUMVEE's monitor. Its support for DCL is based on the following Linux features:

- In general, any memory page that might at some point contain executable code is mapped through a `sys_mmap2` call. When the program interpreter (e.g., `ld-linux`) or the standard C library (e.g., `glibc`) load an executable or shared library, the initial `sys_mmap2` requests that the entire image be mapped with `PROT_EXEC` rights. Subsequent `sys_mmap2` and `sys_mprotect` calls then adjust the alignment and protection flags for non-executable parts of the image. (The few exceptions are discussed later.)
- Even with ASLR enabled, Linux allows for mapping pages at a fixed address by specifying the desired address in the `addr` argument of the `sys_mmap2` call.
- When a variant enters a system call, this constitutes an RVP for GHUMVEE, at which point GHUMVEE can modify the system call arguments before the system call is passed on to the OS. Consequently, GHUMVEE can modify the `addr` arguments of all `sys_mmap2` calls to control the variant's address space.

As shared libraries are loaded into memory from user space (i.e., by the program interpreter component to which the kernel transfers control when returning from the `sys_execve` system call used to launch a new process), GHUMVEE can fully control the location of all loaded shared libraries. It suffices to replace the arguments of any `sys_mmap2` call invoked with `PROT_EXEC` protection flags and originating from within the interpreter. Some simple bookkeeping in the monitor then suffices to enforce that the code mapped in the different variants does not overlap, i.e., that whenever one variant maps code onto some address in its address space, the other ones do not map code there.

8.4.2 Mapping Segments Normally Mapped by the Kernel

Some code regions require special handling, however. Under normal circumstances the kernel maps the program image (i.e., the main binary's segments), the program interpreter, and the VDSO. When ASLR is enabled, it maps them at randomized addresses. But randomized addresses in all the variants do not suffice

to guarantee disjoint code layout. Because GHUMVEE cannot intervene in decision processes in kernel space, it therefore needs to prevent the kernel from mapping them and instead have them mapped from user space, i.e., by the program interpreter. GHUMVEE can then again intercept the mapping system calls and enforce non-overlapping mappings of code regions.

Disjoint Program Images. The standard way to launch new applications is to fork off a running process and invoke a `sys_execve` system call. For example, to read a directory's contents with the `ls` tool, the shell forks and invokes

```
sys_execve("/bin/ls", {"ls", ...}, ...);
```

The kernel then clears the virtual address space of the forked process and maps the mentioned components and a main process stack into its now empty address space.

Mapping the program image from within user space is rather trivial. It suffices to load a program indirectly, rather than directly, with a slightly altered system call

```
sys_execve("/lib/ld-linux.so.2", {"ld-linux.so.2", "/bin/ls",
                                argv[1], ...}, NULL);
```

If a program is loaded indirectly, the kernel maps only the program interpreter, the VDSO, and the initial stack into memory. The remainder of the loading process is handled by the interpreter from within user space. Through indirect invocation, GHUMVEE can override the `sys_mmap2` request in the interpreter that maps the program image. In order to allow GHUMVEE to choose a different address for the program image in each variant, the program needs to be compiled into a Position-Independent Executable (PIE). Recent versions of GCC and LLVM can do so without introducing significant overheads.

At this point, it is important to point out that GHUMVEE does not itself launch applications through this altered system call. Instead, GHUMVEE lets the original, just forked-off processes invoke the standard system call, after which GHUMVEE intercepts that system call and overrides its arguments before passing them to the kernel. This way, GHUMVEE can control the layout of the variants' processes it spawns itself as well as the layout of all the processes subsequently spawned within the variants. This is an essential feature to provide complete protection in the case of multi-process applications, such as applications that are launched through shell scripts.

Program Interpreter. Even with the above indirect program invocation, we cannot prevent the kernel itself from mapping the program interpreter. Hence the indirect invocation does not suffice to ensure that no code regions overlap in the variants.

In Linux, the interpreter is only mapped when the kernel loads a dynamically linked program. To prevent that loading even when launching dynamically linked programs, we developed a statically linked loader program, hereafter referred to as the GHUMVEE Program Loader (GPL). Whenever an application is launched under the control of GHUMVEE, it is launched by launching GPL and having GPL load the actual application. Launching GPL is again done by intercepting the original `sys_execve` calls in GHUMVEE and rewriting their arguments.

VDSO. In each variant launched by GHUMVEE, the copy of GPL is started under GHUMVEE's control. At GPL's entry point, GHUMVEE first checks whether the VDSOs, which were allocated randomly in each variant with ASLR, are disjoint. If they are not, GHUMVEE restarts new variants until a layout is obtained in which the VDSOs are disjoint. Until recently, the Linux kernel mapped the VDSO anywhere between 1 and 1,023 pages below the stack on the i386 platform. It was therefore not uncommon that GHUMVEE had to restart one or more variants. However, a single restart takes less than 4 ms on our system, so the overall performance overhead is negligible.

After ensuring that the VDSOs are disjoint, GPL manually maps the program interpreter through `sys_mmap2` calls. This way, GHUMVEE can override the base addresses of the variants' interpreters to map them onto regions that contain no code in the other variants.

Program Stack. Next, GPL sets up an initial stack in each variant with the exact same layout as when the interpreter would have been loaded by the kernel. Setting up this stack requires several modifications to the stack that the kernel had set up for GPL itself, but this is rather simple to implement.

GPL then transfers control to GHUMVEE through a pseudo system call. GHUMVEE intercepts this call and modifies the call number and arguments such that the kernel unmaps GPL in each variant. Upon return from the call to GHUMVEE, it transfers control to the program interpreter. When the variants then resume, they have fully disjoint code layouts.

Original Program and Shared Libraries. In each variant, the interpreter then continues to load and map the original program and the shared libraries, all of which will be subject to DCL as GHUMVEE intercepts the invoked system calls. Afterward, the

interpreter passes control to the program in each of the variants, which are then all ready to start executing the actual programs.

Assuming that the original program stack is protected by $W \oplus X$, the summarized loading and mapping process is rather complicated, but from the user's perspective this completely transparent launching process allows us to control, in user space, the exact base address of every region that might contain executable code during the execution of the actual program launched by the user.

The end result is two or more variants with completely disjoint code regions. Any divergence in I/O behavior caused by a ROP attack successfully attacking one variant will be detected and aborted by the monitor.

8.4.3 Disjoint Code Layout vs. Address Space Partitioning

Cox et al. and Cavallaro independently proposed to combat memory exploits with essentially identical techniques they called Address Space Partitioning (ASP) [Cox et al. 2006] and Non-Overlapping Address Spaces [Cavallaro 2007], respectively. We will refer to both as ASP.

ASP ensures that addresses of program code (and data) are unique to each variant, i.e., that no virtual address is ever valid for more than one variant. ASP does this by effectively dividing the amount of available virtual memory by N , with N being the number of variants running inside the system. We relaxed this requirement. In DCL, only code addresses must be unique among the variants, but data addresses can occur in multiple variants. So for real-life programs, DCL reduces the amount of available virtual memory by a much smaller fraction than N .

Another significant difference between both of the proposed ASP techniques and DCL is that both implementations of ASP require modifications to either the kernel or the program loader. Cox's N-Variant Systems was fully implemented in kernel space. This way, N-Variant Systems can easily determine where each memory block should be mapped. Cavallaro's ASP implementation requires a patched program loader (`ld-linux.so.2`) to remap the initial stack and override future mapping requests. By contrast, GHUMVEE and DCL do not rely on any changes to the standard loader, standard libraries, or kernel installed on a system. As such, DCL can much more easily be deployed selectively, i.e., for part of the software stack running on a machine, similar to how PIE is used for selected programs on current Linux distributions. As is the case with the relaxed monitoring policies we describe in Section 8.5, by refraining from modifying core system libraries, GHUMVEE offers the end user a great degree of flexibility in when, how, and where its security features should be used.

Finally, whereas DCL relies on PIE [Murphy 2012] to achieve non-overlapping code regions in the variants, both presented forms of ASP rely on standard, non-PIE ELF binaries, despite the fact that PIE support was added to the GCC/binutils toolchain in 2003, well before ASP was proposed. Those non-PIE binaries cannot be relocated at load time. Enabling ASP is therefore only possible by compiling multiple versions of the same ELF executable, each at a different fixed address. ASP tackles this problem by deploying multiple linker scripts for generating the necessary versions of the executable. Unlike regular ELF executables, PIE executables can be relocated at load time. So our DCL solution requires only one, PIE enabled, version of each executable. This feature can again facilitate widespread adoption of DCL.

8.4.4 Compatibility Considerations

Programs that use self-modifying or dynamically compiled, decrypted, or downloaded code may require special treatment when run with DCL. Particularly, GHUMVEE needs to ensure that these programs cannot violate the DCL guarantees. We therefore clarify how GHUMVEE interacts with the program variants in a number of scenarios.

Changing the protection flags of memory pages that were not initially mapped as executable is not allowed. GHUMVEE keeps track of the initial protection flags for each memory page. If the initial protection flags do not include the `PROT_EXEC` flag, the memory page was not subject to DCL at the time it was mapped and GHUMVEE therefore refuses any requests to make the page executable by returning the `EPERM` error from the `sys_mprotect` call that is used to request the change. This inevitably prevents some JIT engines from working out of the box. However, adapting the JIT engine to restore compatibility is trivial. It suffices to request that any JIT region be executable at the time it is initially mapped.

Changing the protection flags of memory pages that were initially mapped as executable is allowed without restrictions. GHUMVEE does not deny any `sys_mprotect` requests to change the protection flags of such pages.

Programs that use the infamous “double-mmap method” to generate code that is immediately executable do not work in GHUMVEE. With the double-mmap method, JIT regions are mapped twice—once with read/write access and once with read/execute access [Moser 2006, Drepper 2006]. The code is generated by writing into the read/write region and can then be executed from the read/execute region. On Linux, a physical page can only be mapped at two distinct locations with two distinct sets of protection flags through the use of one of the APIs for shared memory. As we discussed in Section 8.2, GHUMVEE does not allow the

use of shared memory. Applications that use the double-mmap method therefore would not work. That being said, in this particular case we do not consider our lack of support for bi-directional shared memory as a limitation. Any attacker with sufficient knowledge of such a program's address space layout would be able to write executable code directly, which renders protection mechanisms such as W \oplus X useless. The double-mmap method is therefore nothing short of a recipe for disaster. In practice, we only witnessed this method being used once, in the `vtablefactory` of LibreOffice.

8.4.5 Protection Effectiveness

We cannot provide a formal proof of the effectiveness of DCL. Informally, we can argue that by intercepting all system calls, GHUMVEE can ensure that not a single region in the virtual memory address space has its protections set to `PROT_EXEC` in more than one variant. Furthermore, GHUMVEE's replication ensures that all variants receive exactly the same input. This is the case for input provided through system calls and through signals. Combined, these features ensure that when an attacker passes an absolute address to the application by means of a memory corruption exploit, the code at that address is executable in no more than one variant. The operating system's memory protection makes the variants crash as soon as they try to execute code in their non-executable or missing page at the same virtual address.

We should point out, however, that this protection only works against external attacks, i.e., attacks triggered by external inputs that feed addresses to the program. Artificial ROP attacks set up from within a program itself, as is done in the runtime intrusion prevention evaluator (RIPE) [Wilander et al. 2011], are not necessarily prevented, because in such attacks return addresses are computed within the programs themselves. For those return addresses, different values are hence computed within the different variants, rather than being replicated and intercepted by the replication engine.

To validate the above claimed effectiveness of GHUMVEE with DCL to some extent, we constructed four ROP attacks against high-profile targets. The attacks are available at <http://github.com/stijn-volckaert/ReMon/>.

Our first attack is based on the Braille tool by Bittau et al. [2014]. It exploits a stack buffer overflow vulnerability (CVE-2013-2028) in the nginx web server. The attack first uses stack reading to leak the stack canary and the return address at the bottom of the vulnerable function's stack frame. From this address, it calculates the base address of the nginx binary and uses prior knowledge of the nginx binary to set up a ROP chain. The ROP program itself grants the attacker a remote shell. We

tested this attack by compiling `nginx` with GCC 4.8, with both PIE and stack canaries enabled. The attack succeeds when `nginx` is run natively with ASLR enabled and when `nginx` is run inside GHUMVEE with only one variant. If we run the attack on two variants, however, it fails to leak the stack canary. While attempting to leak the stack canary, at least one variant crashes for every attempt. Whenever a variant crashes, GHUMVEE assumes that the program is under attack and shuts down all other variants in the same logical process. Despite the repeatedly crashing worker processes, the master process manages to restart workers quickly enough to keep the server available throughout the attack.

While GHUMVEE with DCL blocks this attack, the attack probably would not have worked even with DCL disabled: with more than one variant, the attack's stack-reading step can only succeed if every variant uses the same value for its stack canary and the same base address for the `nginx` binary. To prove that DCL does indeed stop ROP attacks, we therefore constructed three other attacks against programs that do not use stack canaries and for which we read the memory layout directly from the `/proc` interface rather than through stack reading.

Our second attack exploits a stack buffer overflow (CVE-2010-4221) in the `proftpd` FTP server. The attack scans the `proftpd` binary and the `libc` library for gadgets for the ROP chain, and reads the load addresses of `proftpd` and `libc` from `/proc/pid/maps` to determine the absolute addresses of the gadgets. The gadgets are combined in a ROP chain that loads and transfers control to an arbitrary payload. In our proof of concept, this payload ends with an `execve` system call used to copy a file. The buffer containing the ROP chain is sent to the application over an unauthenticated FTP connection. The attack succeeds when `proftpd` is run natively with ASLR enabled and also when run inside GHUMVEE with only one variant. When run with two variants, GHUMVEE detects that one variant crashes while the other attempts to perform a `sys_execve` call. GHUMVEE therefore assumes that an attack is in progress, and it shuts down all variants in the same logical process. During the attack, `proftpd`'s master process manages to restart worker processes quickly enough to keep the server available throughout the attack.

Our third attack exploits a stack-based buffer overflow (CVE-2012-4409) in `mcrypt`, an encryption program intended to replace `crypt`. The attack loads addresses of the `mcrypt` binary and the `libc` library from the `/proc` interface to construct a ROP chain, which is sent to the `mcrypt` application over a pipe. The attack succeeds when `mcrypt` is run natively with ASLR enabled and also when run inside GHUMVEE with only one variant. When run with two variants, GHUMVEE detects a crash in one variant and an attempt to perform a system call in the other. It therefore shuts down the program to prevent any damage to the system.

Our fourth attack exploits a stack-based buffer overflow vulnerability (CVE-2014-0749) in the TORQUE resource manager server. The attack reads the load address of the `pbs_server` process, constructs a ROP chain to load and execute an arbitrary payload from found gadgets, and sends it to the server over an unauthenticated network connection. The attack succeeds when TORQUE is run natively with ASLR enabled and also when run inside GHUMVEE with only one variant. When run with two variants, GHUMVEE detects a crash in one variant and an attempt to perform a system call in the other. It therefore shuts down the program to prevent any damage to the system.

8.5 Relaxed Monitoring

Most of the security-oriented MVEEs that preceded GHUMVEE incur non-negligible runtime performance overhead. This overhead can be attributed to two key design decisions: the strict lockstep synchronization model for system calls and the CP/UL operation of the MVEE’s monitor.

Both of these design decisions aggressively favor security over performance. In this section, we revisit these key decisions and present a new MVEE design called ReMon. ReMon incurs significantly lower runtime overhead than other CP/UL MVEEs, while maintaining a high level of security.

Our design is motivated by the fact that a security policy of monitoring *all* system calls is overly conservative [Garfinkel et al. 2004, Provos 2003]. Many system calls cannot affect any state outside of the process making the system call. Only a small set of *sensitive* system calls are potentially useful to an attacker. Thanks to its IP-MON component discussed below, ReMon supports configurable *relaxation* policies that allow non-sensitive calls to execute without being cross-checked against other variants.

8.5.1 ReMon Design and Implementation

Like GHUMVEE, ReMon supervises the execution of multiple diversified program variants that run in parallel. ReMon’s main goals are (i) to monitor all security-sensitive system calls—hereafter referred to as “monitored calls”—issued by these variants; (ii) to force monitored calls to execute in lockstep; (iii) to disable monitoring and lockstepping for non-security-sensitive system calls—hereafter referred to as “unmonitored calls”—thus allowing the variants to execute these calls as efficiently as possible while still providing them with consistent system call results; and (iv) to support configurable monitoring relaxation policies that define which

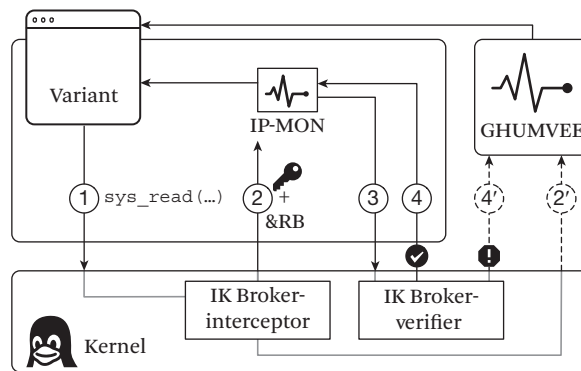


Figure 8.4 ReMon's major components and interactions.

subset of all system calls is considered non-security-sensitive, and therefore should not be monitored. ReMon uses three main components to attain these goals:

GHUMVEE is the security-oriented CP monitor implemented as discussed in the preceding sections. Although GHUMVEE can be used in stand-alone fashion, it only handles monitored calls when used as part of ReMon.

IP-MON is an in-process monitor loaded into each variant as a shared library. IP-MON provides the application with the necessary functionality to replicate the results of unmonitored calls.

IK-B is a small in-kernel broker that forwards unmonitored calls to IP-MON and monitored calls to GHUMVEE. IK-B also enforces security restrictions on IP-MON and provides auxiliary functionality that cannot be implemented in user space. The broker is aware of the system calls that IP-MON handles and of the relaxation policy that is in effect.

These three components interact whenever a variant executes a system call, as shown in Figure 8.4. Our kernel-space system call broker, IK-B, intercepts the system call ① and either forwards it to IP-MON ② or to GHUMVEE ②'. The call is forwarded to IP-MON only if the variant has loaded an IP-MON that can replicate the results of the call, and if the active relaxation policy allows the invoked call to be executed as an unmonitored call. If these two criteria are not met, IK-B uses the standard `ptrace` facilities to forward the call to GHUMVEE instead, which handles it exactly as a regular CP-MVEE.

In the former case, IK-B forwards the call by overwriting the program counter so that the system call returns to a known “system call entry point” in IP-MON's

executable code. While doing so, IK-B gives IP-MON a one-time authorization to complete the execution of the call without having the call reported to GHUMVEE. The broker grants this authorization by passing a random 64-bit token ② as an implicit argument to the forwarded call. IP-MON then performs a series of security checks and eventually completes the execution of the forwarded call by restarting it ③. IP-MON can choose to restart the call with or without the authorization token still intact. If the token is intact upon reentering the kernel, IK-B allows the execution of the system call to complete and returns the call's results to IP-MON ④. If the token is not intact, or if IP-MON executes a different system call, or if the first system call executed after a token has been granted does not originate from within IP-MON itself, IK-B revokes the token and forces the call to be forwarded to GHUMVEE ④.

IP-MON generally executes unmonitored system calls only in the master variant and replicates the results of the system call to the slave variants through the Replication Buffer (RB) discussed in Section 8.5.1.2. The slaves wait for the master to complete its system call and copy the replicated results from the RB when they become available.

Although IP-MON allows the master variant to run ahead of the slaves, it still checks if the variants have diverged. To do so, the master's IP-MON deep-copies all its system call arguments into the RB, and the slaves' IP-MONs compare their own arguments with the recorded ones when they invoke IP-MON. This measure minimizes opportunities for asymmetrical attacks (cf. Section 8.5.2).

8.5.1.1 Securing the Design

The IK-B verifier only allows variants to complete the execution of unmonitored system calls if those calls originate from within an IP-MON instance having a valid one-time authorization token. As only the IK-B interceptor can generate valid tokens, this mechanism *forces every unmonitored system call to go through IK-B*. At the same time, it also ensures that IP-MON can *only execute unmonitored system calls if it is invoked by IK-B and it is invoked through its intended entry point*. This mechanism is, in essence, a form of control-flow integrity [Abadi et al. 2005a]. It also allows us to hide the location of the RB, thereby preventing the RB from being accessed from outside IP-MON. Protecting the RB is of critical importance to the security of our MVEE, as we discuss in Section 8.5.2. To fully hide the location of the RB while still allowing benign accesses, we ensure that the pointer to the RB is only stored in kernel memory.

IK-B loads the RB pointer and the token into designated processor registers whenever it forwards a call to IP-MON, and IP-MON is designed and implemented

such that it does not leak these sensitive values into user-space-accessible memory. First, we compile IP-MON using `gcc` and use the `-ffixed-reg` option to remove the RB pointer and authorization token's designated registers from `gcc`'s register allocator. This ensures that the sensitive values never leak to the stack nor to any other register. Second, we carefully craft specialized accessor functions to access the RB. These functions may temporarily load the RB pointer into other registers, e.g., to calculate a pointer to a specific element in the RB, but they restore these registers to their former values upon returning. We also force IP-MON to destroy the RB pointer and authorization token registers themselves upon returning to the system call site. Finally, we use inlining to avoid indirect control-flow instructions from IP-MON's system call entry point. This ensures that IP-MON's control flow cannot be diverted to a malicious function that could leak the RB pointer or authorization token.

ReMon further prevents discovery of the RB through the `/proc/maps` interface: it forcibly forwards all system calls accessing the maps file to GHUMVEE and it filters the data read from the file. This requires marking the maps file as a special file, as described in Section 8.5.1.6.

To prevent IP-MON itself from being tampered with, we also force all system calls that could adversely affect IP-MON to be forwarded to GHUMVEE. These calls (e.g., `sys_mprotect` and `sys_mremap`) are then subject to the default lockstep synchronization mechanism.

8.5.1.2 The IP-MON Replication Buffer

Like the replication agents discussed in Section 8.3.7, IP-MON must be embedded into all variants, so it consists of multiple independent copies, one per variant. These copies must cooperate, which requires an efficient communication channel. Although a socket or FIFO could be used, we opted for an RB stored in a memory segment and shared by all the variants.

To increase the scalability of our design, we opted not to use a true circular buffer. Instead, we use a linear RB. When our RB overflows, we signal GHUMVEE using a system call. GHUMVEE then waits for all variants to synchronize, resets the buffer to its initial state, and resumes the variants. Involving GHUMVEE as an arbiter avoids costly read/write sharing on RB variables that keep track of where data starts and ends in the RB. Instead, each variant thread only reads and writes its own RB position. The implementation of the IP-MON RB is nearly identical to that of the RBs GHUMVEE uses to support USRVs (see Section 8.3.7).

```

/* read(int fd, void * buf, size_t count) */
MAYBE_CHECKED(read) {
    // check if our current policy allows us to dispatch read
    // calls on this file as unmonitored calls
    return !can_read(ARG1);
}

CALCSIZE(read) {
    // reserve space for 3 register arguments
    COUNTREG(ARG);
    COUNTREG(ARG);
    COUNTREG(ARG);
    // one buffer whose maximum size is in argument 3 of syscall
    COUNTBUFFER(RET, ARG3);
}

PRECALL(read) {
    // compare the args each variant passed to the call.
    // if they match, we allow only the master to complete the call,
    // while the slaves wait for the master's results.
    CHECKREG(ARG1);
    CHECKPOINTER(ARG2);
    CHECKREG(ARG3);
    return MASTERCALL | MAYBE_BLOCKING(ARG1);
}

POSTCALL(read) {
    // replicate the results
    REPLICATEBUFFER(ARG2, ret);
}

```

Listing 8.1 Replicating the read system call in IP-MON.

8.5.1.3 Adding System Call Support

ReMon currently supports well over 200 system calls. To provide a fast path, IP-MON supports a subset of 67 system calls. However, adding support to IP-MON for a new system call is generally straightforward. IP-MON offers a set of C macros to describe how to handle the replication of the system call and its results.

As an example, Listing 8.1 shows IP-MON's code for the read system call. The code is split across four handler functions that each implement one step in the handling of a system call using the C macros provided by IP-MON.

First, the `MAYBE_CHECKED` function is called to determine if the call should be monitored by GHUMVEE. If the `MAYBE_CHECKED` handler returns true, IP-MON

forces the original system call to be forwarded to GHUMVEE (④) by destroying the authorization token and restarting the call.

IP-MON uses a fixed-size RB to replicate system call arguments, results, and other system call metadata. Prior to restarting the forwarded call, we therefore need to calculate the maximum size this information may occupy in the RB. If the size of the data as calculated by the CALCSIZE handler exceeds the size of the RB, IP-MON forces the original system call to be forwarded to GHUMVEE. If the data size does not exceed the size of the RB, but it is bigger than the available portion of the RB, the master waits for the slaves to consume the data already in the RB, after which it resets the RB.

Next, if IP-MON has decided not to forward the original system call to GHUMVEE, it calls the PRECALL handler. In the context of the master variant, this function logs the forwarded call's arguments, call number, and a small amount of metadata into the RB. This metadata consists of a set of boolean flags that indicate whether or not the master has forwarded the call to GHUMVEE, whether or not the call is expected to block when it is resumed, etc. If the function is called in a slave variant's context, IP-MON performs sanity checking by comparing the slave's arguments with the master's arguments. If they do not match, IP-MON triggers an intentional crash, thereby signaling GHUMVEE through the ptrace mechanism and causing a shutdown of the MVEE.

The return value of the PRECALL handler determines whether the original call should be resumed or aborted. By returning the MASTERCALL constant from the PRECALL handler, for example, IP-MON instructs the master variant to resume the original call and the slave variants to abort the original call. Alternatively, the original call may be resumed or aborted in all variants.

Finally, IP-MON calls the POSTCALL handler. Here, the master variant copies its system call return values into the RB.

The slave variants instead wait for the return values to appear in the RB. Depending on the aforementioned system call metadata, the handler may wait using either a spin-wait loop, if the system call was not expected to block, or a specialized condition variable, whose implementation we describe in Section 8.5.1.7.

8.5.1.4 System Call Monitoring Policies

There are many ways to draw the line between system calls to be monitored by GHUMVEE and system calls to be handled by IP-MON. We propose two concrete monitoring relaxation policies.

The first option is *spatial exemption*, where certain system calls are either unconditionally handled by IP-MON and not monitored by GHUMVEE, or handled by

IP-MON only if their system call arguments meet certain criteria. IP-MON comes with several predefined levels of spatial exemption, which the program developer or administrator can choose from. However, regardless of which level of spatial exemption is selected, GHUMVEE always monitors system calls that relate to allocation and management of process resources and threads, as we consider these system calls dangerous no matter what. These system calls include all signal-handling system calls as well as those that (i) allocate, manage, and close file descriptors (FDs); (ii) map, manage, and unmap memory regions; and (iii) create, control, and kill threads and processes. We refer to our earlier work for a full overview of IP-MON’s spatial exemption policies [Volckaert et al. 2016].

The second option is *temporal exemption*, where IP-MON probabilistically exempts system calls from the monitoring policy if similar calls have been repeatedly approved by the monitor. We observe that many programs, especially those with high system call frequencies, often repeatedly invoke the same sequence of system calls. If a series of system calls is approved by GHUMVEE, then one possible temporal relaxation policy is to stochastically exempt some fraction of the identical system calls that follow within some time window or range. Note that temporal relaxation policies must be highly unpredictable; deterministic policies (e.g., “Exempt system calls X, Y, Z from monitoring after N approvals within an M millisecond time window”) are insecure. In other words, care must be taken to ensure that temporal relaxation does not allow adversaries to coerce the MVEE into a state where potentially dangerous system calls are not monitored.

8.5.1.5 IP-MON Initialization

IK-B does not forward any system calls to IP-MON until IP-MON explicitly registers itself through a new system call we added to the kernel. When this call is invoked, the kernel first attempts to report the call to GHUMVEE, which receives the notification and can decide if it wants to allow IP-MON to register.

The registration system call expects three arguments. The first argument is the set of “unmonitored” calls supported by IP-MON. If the IP-MON registration succeeds, IK-B forwards any system call in this set to IP-MON from that point onward, as we explained earlier. GHUMVEE can modify this set of system calls or, potentially, prevent the registration altogether. The second and third arguments are a pointer to the RB and a pointer to the entry point function that should be invoked when IK-B forwards a call to IP-MON.

The RB pointer must be valid and must point to a writable region. IP-MON must therefore set up an RB that it shares with all the other variants. We use the System V IPC facilities to create, initialize, and map the RB [man-pp. project 2017b].

GHUMVEE arbitrates the RB initialization process to ensure that all the variants attach to the same RB.

8.5.1.6 The IP-MON File Map

GHUMVEE arbitrates all system calls that create/modify/destroy FDs, including sockets. It thus maintains metadata, such as the type of each FD (regular/pipe/socket/poll-fd/special). It also tracks which FDs are in non-blocking mode. System calls that operate on non-blocking FDs always return immediately, regardless of whether or not the corresponding operation succeeds.

Variants can map a read-only copy of this metadata into their address spaces using the same mechanism we use for the RB. We refer to this metadata as the IP-MON file map. We maintain exactly 1 byte of metadata per FD, resulting in a page-sized file map. For some system calls, IP-MON uses the file map to determine if the call is to be monitored or not, as per the monitoring policy.

8.5.1.7 Blocking System Calls

Its file map permits IP-MON to predict whether an unmonitored call can block or not. IP-MON handles blocking calls efficiently. If the master variant knows that a call will block, it instructs the slaves to wait on an optimized and highly scalable IP-MON condition variable (as opposed to a slower spin-read loop) until the results become available. IP-MON uses the futex (7) API to implement wait and wake operations. This allowed us to implement several optimizations.

For each system call invocation, IP-MON allocates a separate structure within the RB. Each individual structure contains a condition variable. Slave variants must wait on only the condition variable associated with the system call results they are interested in. Using separate condition variables for each system call invocation prevents an unnecessary bottleneck that would arise when using just a single variable, because the slave variants might progress at different paces. Furthermore, IP-MON tracks whether or not there are variants waiting for the results of a specific system call invocation. If none are waiting when the master has finished writing its system call results into the buffer, no FUTEX_WAKE operation is needed to resume the slaves. IP-MON does not have to reuse condition variables because a new condition variable is allocated for each system call invocation. Thus, IP-MON does not have to reset condition variables to their initial state after it has used one to signal slave variants.

8.5.2 Security Analysis

Unlike previous MVEEs, ReMon eschews fixed monitoring policies and instead allows security/performance trade-offs to be made on a per-application basis.

With respect to integrity, we already pointed out that a CP MVEE monitor and its environment are protected by (i) running it in an isolated process space protected by a hardware-enforced boundary to prevent user-space tampering with the monitor from within the variants; (ii) enforcing lockstep, consistent, monitored execution of all system calls in all variants to prevent malicious impact of a single compromised variant on the monitor; and (iii) diversity among the variants to increase the likelihood that attacks cause observable divergence, i.e., that they fail to compromise the variants in consistent ways.

With those three properties in place, it becomes exceedingly hard for an attacker to subvert the monitor and to execute arbitrary system calls. Nevertheless, MVEEs do not protect against attacks that exploit incorrect program logic or leak information through side-channel attacks. This is similar to many other code-reuse mitigations such as software diversity, software fault isolation [Wahbe et al. 1993], and control-flow integrity [Abadi et al. 2005a].

In ReMon, monitored system calls are still handled by a CP monitor, so malicious monitored calls are as hard to abuse as they are in existing CP MVEEs. For unmonitored calls, IP-MON relaxes the first two of the above three properties. The master variants can run ahead of the slaves and the system call consistency checks in the slaves' IP-MON, so an attacker could try to hijack the master's control with a malicious input to execute at least one, and possibly multiple, unmonitored calls without verification by a slave's IP-MON. An attacker could also attempt to locate the RB and feed malicious data to the slaves, in order to stall them or to tamper with their consistency checks. This way, the attacker could increase the window of opportunity to execute unmonitored calls in the master.

As long as the attacker executes unmonitored calls only according to a given relaxation policy, those capabilities by definition pose no significant security threat: unmonitored calls are exactly those calls that are defined by the chosen policy to pose either no security threat at all or an acceptable security risk. However, an attacker can also try to bypass IP-MON's policy verification checks on conditionally allowed system calls to let IP-MON pass calls unmonitored that should have been monitored by GHUMVEE according to the policy. Therefore, we now consider several aspects of these attack scenarios.

Unmonitored Execution of System Calls. ReMon ensures that IP-MON can only execute unmonitored system calls if it is invoked by IK-B through its intended system call entry point. When invoked properly, IP-MON performs policy verification checks on conditionally allowed system calls, as well as the security checks a CP monitor normally performs. An attacker that manages to compromise a program variant could jump over these checks in an attempt to execute unmonitored system

calls directly. Such an attack would, however, be ineffective thanks to the authorization mechanism we described in Section 8.5.1.1.

Manipulating the RB. We designed IP-MON so that it never stores a pointer to the RB, nor any pointer derived thereof, in user-space-accessible memory. Instead, IK-B passes an RB pointer to IP-MON, and IP-MON keeps the RB pointer in a fixed register. To access the RB, the attacker must therefore find its location by random guessing or by mounting side-channel attacks.

ReMon's current implementation uses RBs that are 16 MiB and located on different addresses in each variant. This gives the RB pointer 24 bits of entropy per variant, which makes guessing attacks unlikely to succeed.

Furthermore, because neither IP-MON nor the application needs to know the exact location of the RB and because every invocation of IP-MON is routed through IK-B, we could extend IK-B to periodically move the RB to a different virtual address by modifying the variants' page table entries. This would further decrease the chances of a successful guessing attack.

Diversified Variants. Our current implementation of ReMon deploys the combined diversification of ASLR and DCL [Volckaert et al. 2015]. ReMon, however, support all other kinds of automated software diversity techniques as well. We refer to the literature for an overview of such techniques [Larsen et al. 2014]. The security evaluations in the literature, including demonstrations of resilience against concrete attacks, therefore still apply to ReMon.

8.6 Evaluation

We performed both performance and functional correctness evaluations of the different optimization techniques and protection schemes we implemented for GHUMVEE. We consider GHUMVEE as our baseline MVEE, against which we compare the spatial relaxation as implemented by ReMon.

8.6.1 Baseline GHUMVEE

To ensure that the approaches and design decisions taken in the development of GHUMVEE are applicable to a wide range of real-life programs, we have evaluated the functional correctness and the overhead imposed by our baseline MVEE on a wide range of applications. This includes not only the typical computationally heavy benchmarks, such as SPEC CPU2006, but also server applications and graphical applications. Unless otherwise mentioned, we evaluated our baseline MVEE with GHUMVEE monitoring two variants with DCL enabled.

All SPEC benchmarks can successfully run on top of GHUMVEE without the need to apply any patches. One benchmark, `416.gamess`, can trigger a false positive intrusion detection in GHUMVEE because it unintentionally prints a small chunk of uninitialized memory to a file. When ASLR is enabled, the uninitialized data differs from one variant to another. In GHUMVEE, we whitelisted the responsible system call to prevent the false positive. The average overhead on SPEC CPU2006 is 7.2%.

We also tested GHUMVEE on several interactive desktop programs that build on large graphical user interface environments, including GNOME tools such as `gcalctool`, KDE tools such as `kcalc`, and `MPlayer`. None of these programs needed patches to run on top of GHUMVEE.

The method of overriding the mapping type for file-backed shared memory was necessary to support KDE applications. These programs use file-backed shared memory to read and write configuration files efficiently. Our method did not cause noticeable slowdowns when running such programs. This overriding method will likely not work for all programs, however.

Our decision to disallow read/write shared mappings and the use of the System V IPC API in commodity applications does not constitute a big problem either. While shared memory is the preferred method for graphical applications to communicate with the display server, we have not seen a single application that does not have a fallback method in place for when GHUMVEE's monitor rejects the request to map shared memory pages. This fallback method typically yields significantly worse performance but is still acceptable in many situations. `MPlayer`, for example, also relies on shared memory for hardware-accelerated playback of movies. When running `MPlayer` in GHUMVEE, it falls back to software-rendered playback.

Early on in GHUMVEE's development, we also tested Firefox and LibreOffice. For LibreOffice, we tested operations such as opening and saving files, editing various types of documents, running the spell checker, etc. For Firefox, we tested opening several web pages. We repeated tests in which GHUMVEE spawned between one and four variants from the same executable, and tests were conducted with and without ASLR enabled. Although these tests were successful, both LibreOffice and Firefox needed extensive patching to be able to run in the context of an MVEE. These patches were needed to eliminate (benign) data races and address-sensitive behavior (cf. Section 8.3.5) and to embed our implicit-input replication agents and synchronization agents.

We not only verified the functional correctness but also evaluated the usability of interactive and real-time applications. Except for small start-up overheads, no significant usability impact was observed. For example, with two variants and

without hardware support, MPlayer was still able to play 720p HD H.264 movies in real time without dropping a single frame, and 1080p Full HD H.264 movies at a frame drop rate of approximately 1%. Because none of the dropped frames were keyframes, playback was still fluent.

Finally, while we can apply our DCL protection with each variant using the same program binary, we did also successfully run programs for which we created a different diversified program binary for each variant. While we only created program binaries to which we applied code randomization techniques [Larsen et al. 2014], we believe that there are no fundamental limitations to the types of diversity techniques GHUMVEE can support.

8.6.2 Disjoint Code Layouts

We evaluated the DCL protection with GHUMVEE on two machines. The first machine has two Intel Xeon E5-2650L CPUs with 8 physical cores and a 20 MB L3 cache each. It has 128GB of main memory and runs a 64-bit Ubuntu 14.04 LTS OS with a Linux 3.13.9 kernel. The second machine has an Intel Core i7 870 CPU with 4 physical cores and an 8 MB L3 cache. It has 32GB of main memory and runs a 32-bit Ubuntu 14.10 OS with a Linux 3.16.7 kernel. On both machines, we disabled hyper-threading and all dynamic frequency and voltage scaling features. Furthermore, we compiled both kernels with a 1,000 Hz tick rate to minimize the monitor-variant interaction latency.

Execution Time Overhead

To evaluate the execution time overhead of GHUMVEE and DCL on compute-intensive applications, we ran each of the SPEC CPU2006 benchmarks five times on their reference inputs. From each set of five measurements, we eliminated the first one to account for I/O-cache warm-up. On the 64-bit machine, we compiled all benchmarks using GCC 4.8.2. On the 32-bit machine, we used GCC 4.9.1. All benchmarks were compiled at optimization level `-O2` and with the `-fno-aggressive-loop-optimizations` flag. We did not use the `-pie` flag for the native benchmarks. Although running with more than two variants does not improve DCL's protection, we have also included the benchmark results for three and four variants for the sake of completeness.

As shown in Figures 8.5 and 8.6, the runtime overhead of DCL is rather low overall.⁴ On our 32-bit machine, the average overhead across all SPEC benchmarks

4. The 434.zeusmp benchmark maps a very large code section and therefore does not run with more than two variants on our 32-bit machine.

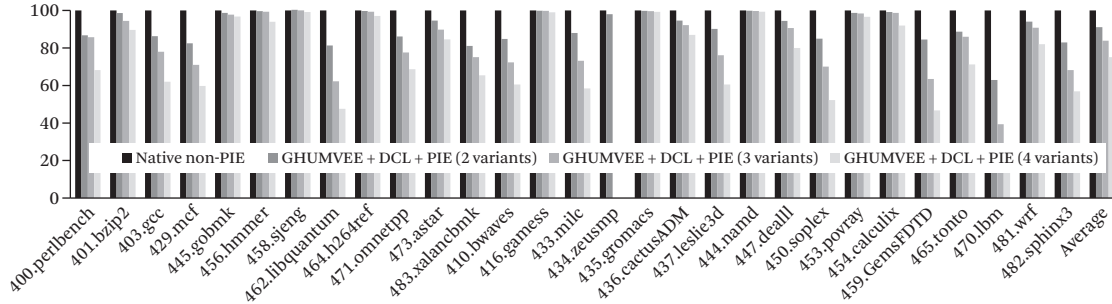


Figure 8.5 Relative performance of 32-bit protected SPEC 2006 benchmarks.

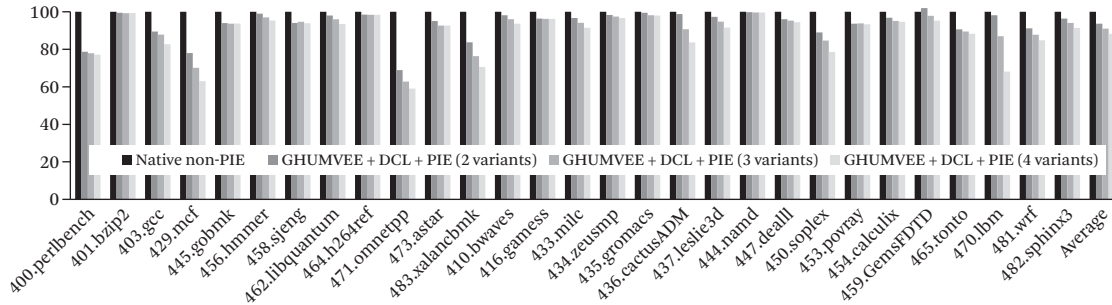


Figure 8.6 Relative performance of 64-bit protected SPEC 2006 benchmarks.

was 8.94%. On our 64-bit machine, which has much larger CPU caches, the average overhead was only 6.37%. That being said, a few benchmarks do stand out in terms of overhead. On i386, we see that `470.lbm` performs remarkably worse than on AMD64. We also see several benchmarks that perform much worse than average on both platforms, including `429.mcf`, `471.omnetpp`, `483.xalancbmk`, and `450.soplex`. For each of these benchmarks, though, our observed performance losses correlate very well with the figures in Jaleel’s cache sensitivity analysis for SPEC [Jaleel 2007].

A second factor that definitely plays its role is PIE itself. While our figures only show the native performance for the original, non-PIE, benchmarks, we did measure the native performance for the PIE version of each benchmark as well. For the most part we did not see significant differences between PIE and non-PIE, except for the `400.perlbenc` and `429.mcf` benchmarks on the AMD64 platform. These benchmarks slow down by 10.98% and 11.93%, respectively, by simply using PIE.

8.6.3 ReMon and IP-MON

We evaluated the performance of IP-MON’s spatial relaxation policy on both synthetic benchmark suites and on a set of server benchmarks. We conducted all of our experiments on a machine with two 8-core Intel Xeon E5-2660 processors each having 20 MB of cache, 64 GB of RAM, and a gigabit Ethernet connection, running the x86_64 version of Ubuntu 14.04.3 LTS. This machine runs the Linux 3.13.11 kernel, to which we applied the IK-B patches. We used the official 2.19 versions of GNU’s `glibc` and `libpthread`s in our experiments, but we did apply a small patch of less than 10 LoC to `glibc` to reinitialize IP-MON’s thread-local storage variables after each fork. As before, we disabled hyper-threading as well as frequency and voltage scaling to maximize reproducibility of our measurements.

Address space layout randomization was enabled in our tests, and we configured ReMon to map IP-MON and its associated buffers at non-overlapping addresses in all variants.

Synthetic Benchmark Suites

We evaluated ReMon on the PARSEC 2.1, SPLASH-2x, and Phoronix benchmark suites⁵. These benchmarks cover a wide range in system call densities and patterns (e.g., bursty vs. spread over time, and mixes of sensitive and non-sensitive calls) as well as various scales and schemes of multi-threading, the most important factors contributing to the overhead of traditional CP-MVEEs that we want to overcome with IP-MON.

We evaluated all five levels of our spatial exemption policy on some of the Phoronix benchmarks, and show the performance of the `NONSOCKET_RW_LEVEL` policy on the other suites. We used the largest available input sets for all benchmarks and ran the multi-threaded benchmarks with four worker threads and used two variants for all benchmarks. We excluded PARSEC’s `canneal` benchmark from our measurements because it purposely causes data races that result in divergent behavior when running multiple variants. This makes the benchmark incompatible with MVEEs. We also excluded SPLASH’s `cholesky` benchmark due to incompatibilities with the version of the `gcc` compiler we used.

The results for these benchmarks are shown in Figures 8.7 and 8.8. The baseline overhead was measured by running ReMon with IP-MON and IK-B disabled. In this configuration, GHUMVEE runs as a stand-alone MVEE.

5. C. Segulja kindly provided his data race patches for PARSEC and SPLASH [Segulja and Abdelrahman 2014].

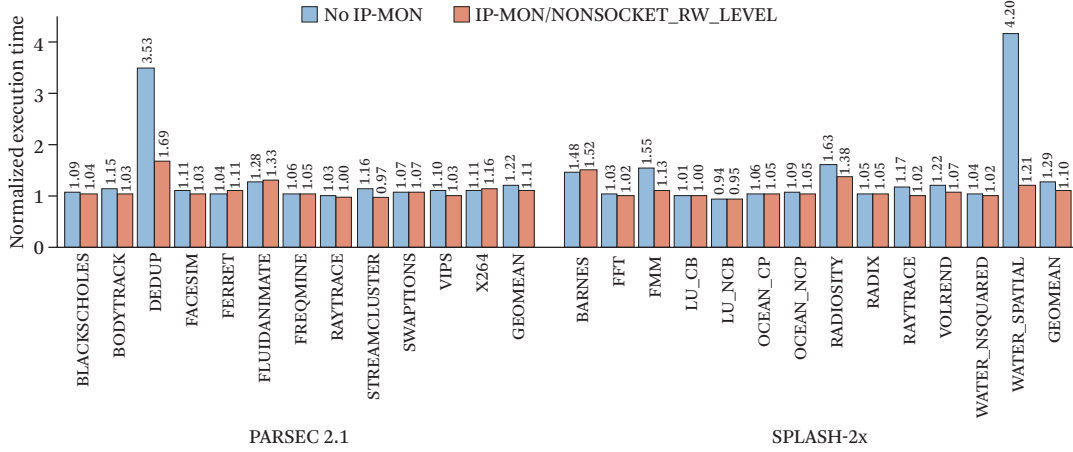


Figure 8.7 Performance overhead for PARSEC 2.1 and SPLASH-2x benchmark suites (two variants).

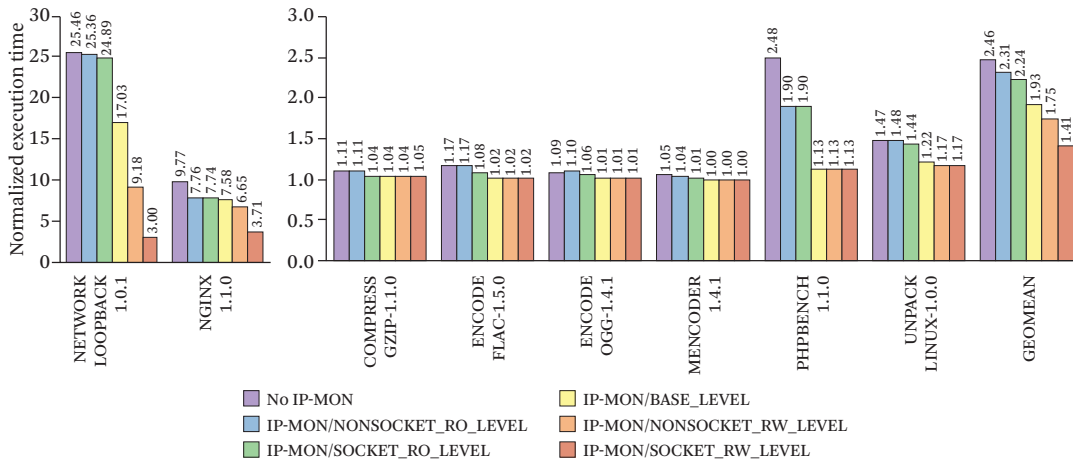


Figure 8.8 Comparison of IP-MON’s spatial relaxation policies in a set of Phoronix benchmarks (two variants).

GHUMVEE generally performs well in these benchmarks. Our machine can run the variants on disjoint CPU cores, which means that only the additional pressure on the memory subsystem and the MVEE itself cause performance degradation compared to the benchmarks’ native performance. Yet, we still see the effect of enabling IP-MON. For PARSEC 2.1, the relative performance overhead decreases

from 21.9% to 11.2%. For SPLASH-2x, the overhead decreases from 29.2% to 10.4%. In Phoronix, the overhead drops from 146.4% to 41.2%. Particularly interesting are the `dedup` (PARSEC 2.1), `water_spatial` (SPLASH-2x) and `network_loopback` (Phoronix) benchmarks, which feature very high system call densities of over 60 K system call invocations per second. In these benchmarks, the overheads drop from 252.9% to 69.4%, from 320% to 20.7%, and from 2446% to 200%, respectively. Furthermore, the Phoronix results clearly show that different policies allow for different security-performance trade-offs.

Server Benchmarks

Server applications are great candidates for execution and monitoring by MVEEs because they are frequently targeted by attackers and they often run on many-core machines with idle CPU cores that can run variants in parallel. In this section, we specifically evaluate our MVEE on applications used to evaluate other MVEEs. These applications include the Apache web server (used to evaluate Orchestra [Salamat et al. 2009]), `thttpd` (ab) and `lighttpd` (ab) (used to evaluate Tachyon [Maurer and Brumley 2012]), `lighttpd` (http_load) (used to evaluate Mx [Hosek and Cadar 2013]), and `beanstalkd`, `lighttpd` (wrk), `memcached`, `nginx` (wrk), and `redis` (used to evaluate VARAN [Hosek and Cadar 2015]). We use the same client and server configurations described by the creators of those MVEEs.

We tested IP-MON by running a benchmark client on a separate machine that was connected to our server via a local gigabit link. We evaluated three scenarios. In the first scenario, we used the gigabit link as is and therefore simulated an unlikely worst-case scenario since the latency on the gigabit link was very low (less than 0.125ms). In the second scenario, we added a small amount of latency (bringing the total average latency to 2ms) to the gigabit link to simulate a realistic worst-case scenario (average network latencies in the U.S. are 24–63 ms [Commission 2014]). In the third scenario, which we only evaluated to allow for comparison with existing MVEEs, we simulated a total average latency of 5ms. We used Linux' built-in `netem` driver to simulate the latency [man-pp. project 2017a].

Figure 8.9 shows the unlikely and the realistic scenarios side by side. For each benchmark, we measured the overhead IP-MON introduces when running between two and seven parallel variants with the spatial exemption policy at the `SOCKET_RW_LEVEL`. We also show the overhead for running two variants with IP-MON disabled. The latter case represents the best-case scenario without IP-MON.

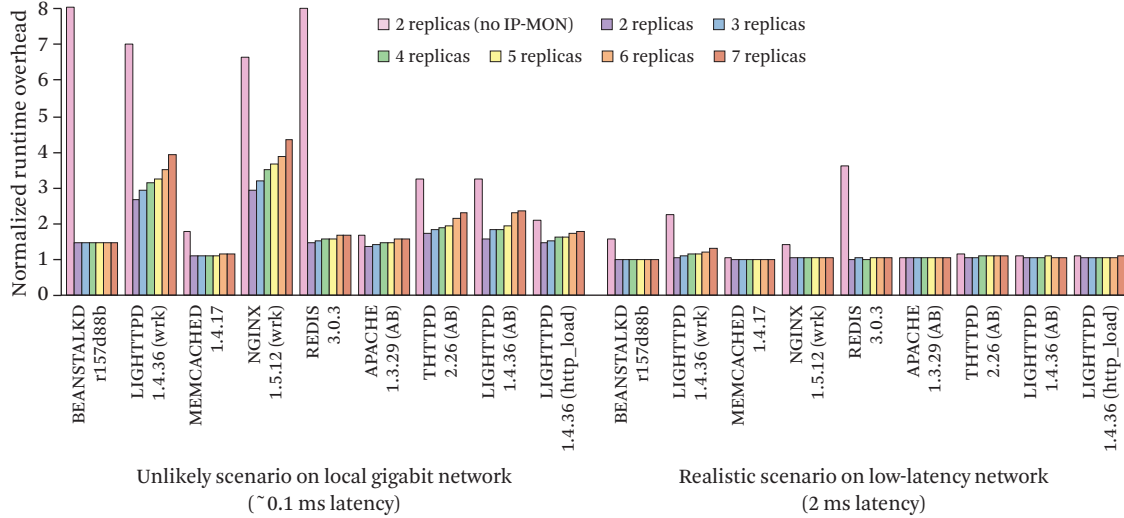


Figure 8.9 Server benchmarks in two network scenarios for two to seven variants with IP-MON and two variants without IP-MON.

8.6.4 Comparison to Existing MVEEs

Table 8.2 compares GHUMVEE’s and ReMon’s performance with the results reported for other MVEEs in the literature [Hosek and Cadar 2013, Hosek and Cadar 2015, Maurer and Brumley 2012, Salamat et al. 2009]. We omitted some MVEEs from this comparison because we could not find (i) enough published performance results for these MVEEs or (ii) sufficient information about the setup in which these MVEEs were evaluated to allow for a meaningful comparison with ReMon and GHUMVEE.

Since each MVEE was evaluated in a different experimental setup, the table also lists two features that have a significant impact on the performance overhead. These are the network latencies, because higher latencies hide server-side overhead, and the CPU cache sizes, as some of the memory-intensive SPEC benchmarks benefit significantly from larger caches, in particular with multiple concurrent variants.

From a performance overhead perspective, the worst-case setup in which Mx and Tachyon were evaluated had the benchmark client running on the same (localhost) machine as the benchmark server. For VARAN, two separate machines resided in the same rack and were hence connected by a very-low-latency gigabit Ethernet.

The worst-case setups in which ReMon and Orchestra were evaluated consist of two separate machines connected by a low-latency gigabit link. In these unlikely worst-case scenarios for servers, the differences in setups hence favor ReMon and Orchestra over VARAN, and VARAN over Tachyon and Mx.

In the best-case setups in which Mx and Tachyon were evaluated, one of the machines was located on the U.S. West Coast, while the other was located in England (Mx) or the U.S. East Coast (Tachyon). In ReMon's best-case setup, we used a gigabit link with a simulated 5 ms latency. So in the more realistic setups and for the server benchmarks, the differences favor Mx and Tachyon over ReMon.

This comparison demonstrates that ReMon outperforms existing non-hardware-assisted security-oriented MVEEs while approaching the efficiency of reliability-oriented MVEEs.

8.7 Conclusion

In this chapter, we presented GHUMVEE, the most efficient non-hardware-assisted security-oriented MVEE to date. GHUMVEE is equipped to support a wide range of realistic programs, including those that contain user-space thread synchronization operations and address-sensitive data structures.

GHUMVEE supports disjoint code layouts, a practical technique to stop code-reuse attacks that rely on payloads containing absolute code addresses. It also supports relaxation policies and selective lockstepping, two techniques that further boost GHUMVEE's efficiency.

Acknowledgments

The authors thank Per Larsen, the Agency for Innovation by Science and Technology in Flanders (IWT), and the Fund for Scientific Research - Flanders.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, and FA8750-10-C-0237, by the National Science Foundation under award number CNS-1513837 as well as gifts from Mozilla, Oracle, and Qualcomm.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, or any other agency of the U.S. Government.

References

- M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005a. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pp. 340–353. DOI: [10.1145/1102120.1102165](https://doi.org/10.1145/1102120.1102165). 12, 25, 38, 39, 62, 82, 86, 95, 97, 110, 114, 117, 139, 141, 173, 174, 181, 186, 211, 233, 243, 249
- M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005b. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering (ICFEM)*. DOI: [10.1007/11576280_9](https://doi.org/10.1007/11576280_9). 182, 186
- M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2009. Control-flow integrity: Principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1). DOI: [10.1145/1609956.1609960](https://doi.org/10.1145/1609956.1609960). 181, 189, 208
- A. Acharya and M. Raje. 2000. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium (SSYM)*, pp. 1–17. 16
- P. Akritidis. 2010. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pp. 177–192. 84, 173, 178
- P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. 2008. Preventing memory error exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, pp. 263–277. DOI: [10.1109/SP.2008.30](https://doi.org/10.1109/SP.2008.30). 8, 58, 82, 84, 114, 173, 176, 178
- P. Akritidis, M. Costa, M. Castro, and S. Hand. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pp. 51–66. 84, 173, 178
- Aleph One. 1996. Smashing the stack for fun and profit. *Phrack*, 7. 11, 17
- A. Alexandrov, P. Kmiec, and K. Schauer. 1999. Consh: Confined execution environment for Internet computations. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.488>. DOI: [10.1.1.57.488](https://doi.org/10.1.1.57.488). 16
- G. Altekar and I. Stoica. 2010. Focus replay debugging effort on the control plane. In *USENIX Workshop on Hot Topics in Dependability*. 89

- S. Andersen and V. Abella. August 2004. Changes to functionality in Windows XP service pack 2—part 3: Memory protection technologies. <http://technet.microsoft.com/en-us/library/bb457155.aspx>. 9, 19, 184
- J. Ansel. March 2014. Personal communication. 53
- J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. Schuff, D. Sehr, C. Biffle, and B. Yee. 2011. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 355–366. DOI: [10.1145/1993316.1993540](https://doi.org/10.1145/1993316.1993540). 58, 59
- O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. 2015. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Design Automation Conference (DAC)*, pp. 74:1–74:6. DOI: [10.1145/2744769.2744847](https://doi.org/10.1145/2744769.2744847). 182, 208, 209
- J.-P. Aumasson and D. J. Bernstein. 2012. SipHash: A fast short-input PRF. In *13th International Conference on Cryptology in India (INDOCRYPT)*. 73
- M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security (CCS)*. DOI: [10.1145/2660267.2660378](https://doi.org/10.1145/2660267.2660378), pp. 1342–1353. 65, 173, 177
- M. Backes and S. Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, pp. 433–447. 64, 66
- A. Balasubramanian, M. S. Baranowski, A. Burtsev, and A. Panda. 2017. System programming in Rust: Beyond safety. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pp. 94–99. DOI: [10.1145/3102980.3103006](https://doi.org/10.1145/3102980.3103006). 79
- C. Basile, Z. Kalbarczyk, and R. Iyer. 2002. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 149–158. DOI: [10.1109/DSN.2003.1209926](https://doi.org/10.1109/DSN.2003.1209926). 230
- C. Basile, Z. Kalbarczyk, and R. K. Iyer. 2006. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 17(5):448–465. DOI: [10.1109/TPDS.2006.56](https://doi.org/10.1109/TPDS.2006.56). 230
- A. Basu, J. Bobba, and M. D. Hill. 2011. Karma: Scalable deterministic record-replay. In *Proceedings of the International Conference on Supercomputing*, pp. 359–368. DOI: [10.1145/1995896.1995950](https://doi.org/10.1145/1995896.1995950). 230
- M. Bauer. 2006. Paranoid penguin: an introduction to Novell AppArmor. *Linux J.*, (148):13. 16
- A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 335–348. 213

- T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. 2010. CoreDet: A compiler and runtime system for deterministic multithreaded execution. *ACM SIGARCH Computer Architecture News*, 38(1):53–64. DOI: [10.1145/1735971.1736029](https://doi.org/10.1145/1735971.1736029). 230
- E. Berger, T. Yang, T. Liu, and G. Novark. 2009. Grace: Safe multithreaded programming for C/C++. *ACM Sigplan Notices*, 44(10):81–96. 230
- E. D. Berger and B. G. Zorn. 2006. DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices*, (6):158–168. DOI: [10.1145/1133255.1134000](https://doi.org/10.1145/1133255.1134000). 214
- E. Bhatkar, D. C. Duvarney, and R. Sekar. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the USENIX Security Symposium (SSYM)*, pp. 105–120. 10
- S. Bhatkar and R. Sekar. 2008. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 1–22. DOI: [10.1007/978-3-540-70542-0_1](https://doi.org/10.1007/978-3-540-70542-0_1). 11, 85
- S. Bhatkar, R. Sekar, and D. C. DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium (SSYM)*, pp. 17–17. <http://dl.acm.org/citation.cfm?id=1251398.1251415>. 10, 95
- D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 268–279. DOI: [10.1145/2810103.2813691](https://doi.org/10.1145/2810103.2813691). 11
- A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. 2014. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 227–242. DOI: [10.1109/SP.2014.22](https://doi.org/10.1109/SP.2014.22). 62, 140, 141, 182, 239
- D. Blazakis. 2010. Interpreter exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, pp. 1–9. 59
- T. Bletsch, X. Jiang, and V. Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 353–362. DOI: [10.1145/2076732.2076783](https://doi.org/10.1145/2076732.2076783). 208, 209
- T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 30–40. DOI: [10.1145/1966913.1966919](https://doi.org/10.1145/1966913.1966919). 20, 81, 82, 117
- E. Bosman and H. Bos. 2014. Framing signals—a return to portable shellcode. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 243–258. DOI: [10.1109/SP.2014.23](https://doi.org/10.1109/SP.2014.23). 31, 140
- K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. 2016. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*. 68

- S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. 2011. Exploit programming: From buffer overflows to “weird machines” and theory of computation. *Usenix ;login*: issue: December 2011, volume 36, number 6. 19
- E. Buchanan, R. Roemer, H. Shacham, and S. Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pp. 27–38. DOI: [10.1145/1455770.1455776](https://doi.org/10.1145/1455770.1455776). 233
- M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pp. 42–51. DOI: [10.1145/1181309.1181316](https://doi.org/10.1145/1181309.1181316). 208, 209
- N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. 2016. Control-flow integrity: Precision, security, and performance. *Computing Research Repository (CoRR)*. 50(1). <http://arxiv.org/abs/1602.04056>. 12, 28, 62, 82
- N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. 2017. Control-flow integrity: precision, security, and performance. *ACM Computing Surveys*. DOI: [10.1145/3054924](https://doi.org/10.1145/3054924). 12
- J. Butler and anonymous. 2004. Bypassing 3rd party Windows buffer overflow protection. *Phrack*, 11. 17
- N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, pp. 161–176. <http://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini>. 15, 20, 21, 59, 81, 82, 97, 137, 182, 183, 185, 186, 188, 200, 204, 211
- N. Carlini and D. Wagner. 2014. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 385–399. <http://dl.acm.org/citation.cfm?id=2671225.2671250>. 15, 53, 82, 84, 86, 97, 114, 137, 139, 140, 176, 177, 179, 182, 183, 184, 186, 188, 200, 202, 209
- M. Castro, M. Costa, and T. Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 147–160. 11, 86
- M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. 2009. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles*, pp. 45–58. DOI: [10.1145/1629575.1629581](https://doi.org/10.1145/1629575.1629581). 86, 93
- L. Cavallaro. 2007. Comprehensive memory error protection via diversity and taint-tracking. PhD thesis, Università Degli Studi Di Milano. 214, 223, 237
- S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pp. 559–572. DOI: [10.1145/1866307.1866370](https://doi.org/10.1145/1866307.1866370). 20, 81, 117, 183, 184, 185, 186, 200, 202

- S. Checkoway and H. Shacham. 2010. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical report CS2010-0954, UC San Diego. <http://cseweb.ucsd.edu/~hovav/dist/noret.pdf>. 200, 202
- P. Chen, Y. Fang, B. Mao, and L. Xie. 2011. JITDefender: A defense against JIT spraying attacks. In *26th IFIP International Information Security Conference*, volume 354, pp. 142–153. 60
- P. Chen, R. Wu, and B. Mao. 2013. JITSafe: A framework against just-in-time spraying attacks. *IET Information Security*, 7(4):283–292. DOI: [10.1049/iet-ifs.2012.0142](https://doi.org/10.1049/iet-ifs.2012.0142). 59, 60
- S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. 2005. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*. <http://dl.acm.org/citation.cfm?id=1251398.1251410>. 21, 184
- X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. 2015. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Symposium on Network and Distributed System Security (NDSS)*. 173, 178
- Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen. 2017. NORAX: Enabling execute-only memory for COTS binaries on AArch64. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 304–319. DOI: [10.1109/SP.2017.30](https://doi.org/10.1109/SP.2017.30). 68
- Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21st Symposium on Network and Distributed System Security (NDSS)*. 117, 118, 119, 127, 173, 176, 182, 209
- M. Co, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong, W. Weimer, J. Burket, G. L. Frazier, T. M. Frazier, and B. Dutertre, et al. 2016. Double Helix and RAVEN: A system for cyber fault tolerance and recovery. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, p. 17. DOI: [10.1145/2897795.2897805](https://doi.org/10.1145/2897795.2897805). 214
- F. B. Cohen. 1993. Operating system protection through program evolution. *Computers & Security*, 12(6): 565–584. DOI: [10.1016/0167-4048\(93\)90054-9](https://doi.org/10.1016/0167-4048(93)90054-9). 62
- Corelan. 2011. Mona: A debugger plugin/exploit development Swiss army knife. <http://redmine.corelan.be/projects/mona>. 136
- C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. 2000. SubDomain: Parsimonious server security. In *Proceedings of the 14th USENIX Conference on System Administration*, pp. 355–368. 16
- C. Cowan, S. Beattie, J. Johansen, and P. Wagle. 2003. Pointguard™: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium (SSYM)*, pp. 7–7. <http://dl.acm.org/citation.cfm?id=1251353.1251360>. 11, 63, 76, 82, 85
- C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pp. 346–355. 61, 63, 82, 95, 211, 233

- B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. 2006. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, 9. 211, 213, 214, 217, 237
- S. Crane, A. Homescu, and P. Larsen. 2016. Code randomization: Haven't we solved this problem yet? In *IEEE Cybersecurity Development (SecDev)*. DOI: [10.1109/SecDev.2016.036](https://doi.org/10.1109/SecDev.2016.036). 66
- S. Crane, P. Larsen, S. Brunthaler, and M. Franz. 2013. Booby trapping software. In *New Security Paradigms Workshop (NSPW)*, pp. 95–106. DOI: [10.1145/2535813.2535824](https://doi.org/10.1145/2535813.2535824). 68
- S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy (S&P)*, pp. 763–780. DOI: [10.1109/SP.2015.52](https://doi.org/10.1109/SP.2015.52). 11, 60, 66, 76, 173, 178
- S. Crane, S. Voleckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. 2015. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 243–255. DOI: [10.1145/2810103.2813682](https://doi.org/10.1145/2810103.2813682). 11, 68, 77, 159, 171, 173, 178
- J. Criswell, N. Dautenhahn, and V. Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 292–307. DOI: [10.1109/SP.2014.26](https://doi.org/10.1109/SP.2014.26). 58
- H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. 2013. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 388–405. DOI: [10.1145/2517349.2522735](https://doi.org/10.1145/2517349.2522735). 230
- D. Dai Zovi. 2010. Practical return-oriented programming. Talk at *SOURCE Boston*, 2010. 117
- L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. 2010. Transactional mutex locks. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, pp. 2–13. 44
- T. H. Y. Dang, P. Maniatis, and D. Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 555–566. DOI: [10.1145/2714576.2714635](https://doi.org/10.1145/2714576.2714635). 10, 137, 208
- DarkReading. November 2009. Heap spraying: Attackers' latest weapon of choice. <http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=221901428>. 133
- L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi. 2012. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. 58, 208

- L. Davi, P. Koeberl, and A.-R. Sadeghi. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference—Special Session: Trusted Mobile Embedded Computing (DAC)*, pp. 1–6. DOI: [10.1145/2593069.2596656](https://doi.org/10.1145/2593069.2596656). 173, 174, 209
- L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 401–416. <http://dl.acm.org/citation.cfm?id=2671225.2671251>. 15, 43, 53, 82, 84, 86, 97, 114, 139, 140, 169, 174, 176, 177, 179, 182, 183, 184, 186, 188, 200, 209, 211
- L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. 2015. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*. DOI: [10.14722/ndss.2015.23262](https://doi.org/10.14722/ndss.2015.23262). 64
- L. Davi, A.-R. Sadeghi, and M. Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 40–51. DOI: [10.1145/1966913.1966920](https://doi.org/10.1145/1966913.1966920). 139, 141
- L. de Moura and N. Bjørner. 2009. Generalized, efficient array decision procedures. In *Formal Methods in Computer Aided Design (FMCAD)*. DOI: [10.1109/FMCAD.2009.5351142](https://doi.org/10.1109/FMCAD.2009.5351142). 161
- L. M. de Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 337–340. 140, 161
- T. de Raadt. 2005. Exploit mitigation techniques. <http://www.openbsd.org/papers/ven05-deraadt/index.html>. 8
- J. Dean, D. Grove, and C. Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, pp. 77–101. 32
- D. Dechev. 2011. The ABA problem in multicore data structures with collaborating operations. In *7th International Conference on Collaborative Computing: Networking, Applications, and Worksharing (CollaborateCom)*, pp. 158–167. DOI: [10.4108/icst.collaboratecom.2011.247161](https://doi.org/10.4108/icst.collaboratecom.2011.247161). 44
- L. Deng, Q. Zeng, and Y. Liu. 2015. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *30th International Conference on ICT Systems Security and Privacy Protection*, pp. 386–400. 41
- L. P. Deutsch and A. M. Schiffman. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 297–302. DOI: [10.1145/800017.800542](https://doi.org/10.1145/800017.800542). 54
- J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. 2008. HardBound: Architectural support for spatial safety of the C programming language. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 103–114. DOI: [10.1145/1353534.1346295](https://doi.org/10.1145/1353534.1346295). 109

- J. Devietti, B. Lucia, L. Ceze, and M. Oskin. 2009. DMP: Deterministic shared memory multiprocessing. *ACM SIGARCH Computer Architecture News*, 37(1):85–96. DOI: [10.1145/1508244.1508255](https://doi.org/10.1145/1508244.1508255). 230
- D. Dewey and J. T. Giffin. 2012. Static detection of C++ vtable escape vulnerabilities in binary code. In *Symposium on Network and Distributed System Security (NDSS)*. 171
- D. Dhurjati, S. Kowshik, and V. Adve. June 2006. SAFECode: Enforcing alias analysis for weakly typed languages. *SIGPLAN Notices*, 41 (6): 144–157. DOI: [10.1145/1133255.1133999](https://doi.org/10.1145/1133255.1133999). 82, 84
- U. Drepper. April 2006. SELinux memory protection tests. <http://www.akkadia.org/drepper/selinux-mem.html>. 238
- V. D'Silva, M. Payer, and D. Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *LangSec'15: Second Workshop on Language-Theoretic Security*. DOI: [10.1109/SPW.2015.33](https://doi.org/10.1109/SPW.2015.33). 16
- T. Durden. 2002. Bypassing PaX ASLR protection. *Phrack*, 11. 10, 17
- EEMBC. The embedded microprocessor benchmark consortium: EEMBC benchmark suite. <http://www.eembc.org>. 206
- Ú Erlingsson, M. Abadi, M. Vrabie, M. Budiu, and G. Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation*, pp. 75–88. 58, 86, 95
- H. Etoh and K. Yoda. June 2000. Protecting from stack-smashing attacks. Technical report, IBM Research Division, Tokyo Research Laboratory. 63
- C. Evans. 2013. Exploiting 64-bit Linux like a boss. <http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html>. 117
- I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. E. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. 2015. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy, (S&P)*, pp. 781–796. DOI: [10.1109/SP.2015.53](https://doi.org/10.1109/SP.2015.53). 11, 62, 87
- I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 901–913. DOI: [10.1145/2810103.2813646](https://doi.org/10.1145/2810103.2813646). 20, 21, 59, 82, 97, 137, 211
- Federal Communications Commission. 2014. Measuring broadband America—2014. <http://www.fcc.gov/reports/measuring-broadband-america-2014>. 256
- C. Fetzer and M. Suesskraut. 2008. SwitchBlade: Enforcing dynamic personalized system call models. In *Proceedings of the 3rd European Conference on Computer Systems*, pp. 273–286. DOI: [10.1145/1357010.1352621](https://doi.org/10.1145/1357010.1352621). 16
- A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. 2011. SmartDec: Approaching C++ decompilation. In *Working Conference on Reverse Engineering (WCRE)*. 171
- B. Ford and R. Cox. 2008. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX ATC*, pp. 293–306. 8, 9

- M. Frantzen and M. Shuey. 2001. StackGhost: Hardware facilitated stack protection. In *USENIX Security Symposium*. 139, 141
- I. Fratric. 2012. Runtime prevention of return-oriented programming attacks. <http://github.com/ivanfratric/ropguard/blob/master/doc/ropguard.pdf>. 139, 173, 176
- Gaisler Research. LEON3 synthesizable processor. <http://www.gaisler.com>. 183, 206
- A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 465–478. DOI: [10.1145/1543135.1542528](https://doi.org/10.1145/1543135.1542528). 50
- T. Garfinkel, B. Pfaff, and M. Rosenblum. 2004. Ostia: A delegating architecture for secure system call interposition. In *Network and Distributed System Security Symposium (NDSS)*. 241
- R. Gawlik and T. Holz. 2014. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, pp. 396–405. 171, 173, 176, 182
- R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*. 68
- X. Ge, M. Payer, and T. Jaeger. 2017. An evil copy: How the loader betrays you. In *Network and Distributed System Security Symposium (NDSS)*. DOI: [10.14722/ndss.2017.23199](https://doi.org/10.14722/ndss.2017.23199). 15
- J. Gionta, W. Enck, and P. Ning. 2015. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 325–336. DOI: [10.1145/2699026.2699107](https://doi.org/10.1145/2699026.2699107). 65
- GNU.org. The GNU C library: Environment access. http://www.gnu.org/software/libc/manual/html_node/Environment-Access.html. 220
- E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. 2014a. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 575–589. DOI: [10.1109/SP.2014.43](https://doi.org/10.1109/SP.2014.43). 15, 53, 82, 84, 86, 97, 114, 124, 125, 126, 129, 134, 136, 137, 139, 140, 174, 175, 177, 182, 183, 186, 188, 200, 202, 211
- E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. 2014b. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium*. <http://dl.acm.org/citation.cfm?id=2671225.2671252>. 122, 139, 140, 169, 177, 179, 182, 186, 188, 209
- E. Göktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. 2016. Undermining information hiding (and what to do about it). In *Proceedings of the 25th USENIX Security Symposium*, pp. 105–119. 11, 68
- I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. 1996. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th USENIX Security Symposium (SSYM)*. 16

- Google Chromium Project. 2013. Undefined behavior sanitizer. <http://www.chromium.org/developers/testing/undefinedbehaviorsanitizer>. 7
- B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. In *Annual Network and Distributed System Security Symposium (NDSS)*. 67
- Y. Guillot and A. Gazet. 2010. Automatic binary deobfuscation. *J. Comput. Virol.* 6(3): pp. 261–276. DOI: [10.1007/s11416-009-0126-4](https://doi.org/10.1007/s11416-009-0126-4). 160
- I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. 2015. ShrinkWrap: VTable protection without loose ends. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 341–350. DOI: [10.1145/2818000.2818025](https://doi.org/10.1145/2818000.2818025). 136
- I. Haller, Y. Jeon, H. Peng, M. Payer, H. Bos, C. Giuffrida, and E. van der Kouwe. 2016. TypeSanitizer: Practical type confusion detection. In *ACM Conference on Computer and Communication Security (CCS)*. DOI: [10.1145/2976749.2978405](https://doi.org/10.1145/2976749.2978405). 7
- N. Hasabnis, A. Misra, and R. Sekar. 2012. Light-weight bounds checking. In *IEEE/ACM Symposium on Code Generation and Optimization*. DOI: [10.1145/2259016.2259034](https://doi.org/10.1145/2259016.2259034). 84
- Hex-Rays. 2017. IDA Pro. <http://www.hex-rays.com/index.shtml>. 128
- M. Hicks. 2014. What is memory safety? <http://www.pl-enthusiast.net/2014/07/21/memory-safety/>. 4
- E. Hiroaki and Y. Kunikazu. 2001. ProPolice: Improved stack-smashing attack detection. *IPSJ SIG Notes*, pp. 181–188. 11
- J. Hiser, A. Nguyen, M. Co, M. Hall, and J. W. Davidson. 2012. ILR: Where’d my gadgets go? In *33rd IEEE Symposium on Security and Privacy (S&P)*, pp. 571–585. DOI: [10.1109/SP.2012.39](https://doi.org/10.1109/SP.2012.39). 11, 66
- J. D. Hiser, D. Williams, A. Filipi, J. W. Davidson, and B. R. Childers. 2006. Evaluating fragment construction policies for SDT systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, pp. 122–132. DOI: [10.1145/1134760.1134778](https://doi.org/10.1145/1134760.1134778). 8, 9
- U. Hölzle, C. Chambers, and D. Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming (ECOOP)*, pp. 21–38. 54
- U. Hölzle, C. Chambers, and D. Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 32–43. DOI: [10.1145/143103.143114](https://doi.org/10.1145/143103.143114). 54
- A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. 2013. Librando: Transparent code randomization for just-in-time compilers. *CCS '13*, pp. 993–1004. DOI: [10.1145/2508859.2516675](https://doi.org/10.1145/2508859.2516675). 58, 59
- A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. 2013. Profile-guided automated software diversity. In *IEEE/ACM Symposium on Code Generation and Optimization*, pp. 1–11. DOI: [10.1109/CGO.2013.6494997](https://doi.org/10.1109/CGO.2013.6494997). 85

- P. Hosek and C. Cadar. 2013. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*, pp. 612–621. DOI: [10.1109/ICSE.2013.6606607](https://doi.org/10.1109/ICSE.2013.6606607). 214, 256, 257
- Petr Hosek and Cristian Cadar. 2015. Varan the unbelievable: An efficient n-version execution framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 339–353. DOI: [10.1145/2694344.2694390](https://doi.org/10.1145/2694344.2694390). 214, 215, 218, 224, 227, 256, 257
- H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. 2015. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, pp. 177–192. <http://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu.21>
- R. Hund, C. Willems, and T. Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 191–205. DOI: [10.1109/SP.2013.23](https://doi.org/10.1109/SP.2013.23). 82, 86, 141
- G. Hunt and D. Brubacher. 1999. Detours: Binary interception of win32 functions. In *Usenix Windows NT Symposium*, pp. 135–143. 232
- Intel. 2013. *Intel Architecture Instruction Set Extensions Programming Reference*. <http://download-software.intel.com/sites/default/files/319433-015.pdf>. 108
- Intel. 2013. Introduction to Intel memory protection extensions. <http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions>. 93
- Intel. 2013. *Intel 64 and IA-32 Architectures Software Developer's Manual—Combined Volumes 1, 2a, 2b, 2c, 3a, 3b, and 3c*. 178
- Intel. 2014. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A–Z*. 2014. 223, 224
- Itanium C++ ABI. <http://mentorembedded.github.io/cxx-abi/abi.html>. 32
- A. Jaleel. 2007. Memory characterization of workloads using instrumentation-driven simulation—a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. *technical report*. <http://www.glue.umd.edu/~ajaleel/workload/>. 253
- D. Jang, Z. Tatlock, and S. Lerner. 2014. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*. 32, 173, 176
- jduck. 2010. The latest Adobe exploit and session upgrading. <http://bugix-security.blogspot.de/2010/03/adobe-pdf-libtiff-working-exploitcve.html>. 182
- T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. 2002. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*. 5, 82, 84, 88, 95
- N. Joly. 2013. Advanced exploitation of Internet Explorer 10/Windows 8 overflow (Pwn2Own 2013). http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php. 117, 124, 162
- M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. 2012. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual*

- International Symposium on Computer Architecture (ISCA)*. <http://dl.acm.org/citation.cfm?id=2337159.2337171>. DOI: [10.1109/ISCA.2012.6237009](https://doi.org/10.1109/ISCA.2012.6237009). 182, 209
- M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh. 2013. Scrap: Architecture for signature-based protection from code reuse attacks. In *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, pp. 258–269. DOI: [10.1109/HPCA.2013.6522324](https://doi.org/10.1109/HPCA.2013.6522324). 209
- C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pp. 339–348. DOI: [10.1109/ACSAC.2006.9](https://doi.org/10.1109/ACSAC.2006.9). 11, 85
- V. Kiriansky, D. Bruening, and S. P. Amarasinghe. 2002. Secure execution via program shepherding. In *Proceedings 11th USENIX Security Symposium*, pp. 191–206. 8, 9
- K. Koning, H. Bos, and C. Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 431–442. DOI: [10.1109/DSN.2016.46](https://doi.org/10.1109/DSN.2016.46). 211, 214, 217
- T. Kornau. 2010. Return oriented programming for the ARM architecture. Ph.D. thesis, Master's thesis, Ruhr-Universität Bochum. 233
- V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014a. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 147–163. 9, 10, 59, 62, 105, 106, 107, 173, 178, 179
- V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014b. Code-Pointer Integrity website. <http://dslab.epfl.ch/proj/cpi/>. 179
- V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song. 2015. Poster: Getting the point (er): On the feasibility of attacks on code-pointer integrity. In *36th IEEE Symposium on Security and Privacy (S&P)*. 87
- P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. 2014. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 276–291. DOI: [10.1109/SP.2014.25](https://doi.org/10.1109/SP.2014.25). 11, 62, 66, 250, 252
- C. Lattner and V. Adve. 2005. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *ACM Conference on Programming Language Design and Implementation*, pp. 129–142. DOI: [10.1145/1064978.1065027](https://doi.org/10.1145/1064978.1065027). 91, 108
- C. Lattner, A. Lenharth, and V. Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM Conference on Programming Language Design and Implementation*, pp. 278–289. DOI: [10.1145/1273442.1250766](https://doi.org/10.1145/1273442.1250766). 91, 108
- B. Lee, C. Song, T. Kim, and W. Lee. 2015. Type casting verification: Stopping an emerging attack vector. In *USENIX Security 15*, pp. 81–96. <http://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lee>. 7

- D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. 2010. Respec: Efficient online multiprocessor replay via speculation and external determinism. *ACM SIGARCH Computer Architecture News*, 38(1):77–90. DOI: [10.1145/1736020.1736031](https://doi.org/10.1145/1736020.1736031). 230
- J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz. 2016. Subversive-C: Abusing and protecting dynamic message dispatch. In *USENIX Annual Technical Conference (ATC)*, pp. 209–221. 70, 140
- E. Levy. 1996. Smashing the stack for fun and profit. *Phrack*, 7. 61
- J. Li, Z. Wang, T. K. Bletsch, D. Srinivasan, M. C. Grace, and X. Jiang. 2011. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security*, 6(4):1404–1417. DOI: [10.1109/TIFS.2011.2159712](https://doi.org/10.1109/TIFS.2011.2159712). 82, 86
- C. Liebchen, M. Negro, P. Larsen, L. Davi, A.-R. Sadeghi, S. Crane, M. Qunaibit, M. Franz, and M. Conti. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security (CCS)*. DOI: [10.1145/2810103.2813671](https://doi.org/10.1145/2810103.2813671). 182, 183, 205
- Linux Man-Pages Project. 2017a. tc-netem(8)—Linux manual page. 256
- Linux Man-Pages Project. 2017b. shmop(2)—Linux manual page. 247
- Linux Programmer's Manual*. 2017a. vdso(7)—Linux manual page. 223
- Linux Programmer's Manual*. 2017b. getauxval(3)—Linux manual page. 224
- Linux Programmer's Manual*. 2017c. signal(7)—Linux manual page. 225
- T. Liu, C. Curtsinger, and E. Berger. 2011. DTHREADS: Efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP)*, pp. 327–336. DOI: [10.1145/2043556.2043587](https://doi.org/10.1145/2043556.2043587). 230
- LLVM. The LLVM compiler infrastructure. <http://llvm.org/>. 102
- K. Lu, X. Zhou, T. Bergan, and X. Wang. 2014. Efficient deterministic multithreading without global barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 287–300. DOI: [10.1145/2555243.2555252](https://doi.org/10.1145/2555243.2555252). 230
- K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. 2015. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *ACM Conference on Computer and Communications Security (CCS)*, pp. 280–291. DOI: [10.1145/2810103.2813694](https://doi.org/10.1145/2810103.2813694). 68
- J. Maebe, M. Ronsse, and K. D. Bosschere. 2003. Instrumenting JVMs at the machine code level. In *3rd PA3CT symposium*, volume 19, pp. 105–107. 222
- G. Maisuradze, M. Backes, and C. Rossow. 2003. What cannot be read, cannot be leveraged? Revisiting assumptions of JIT-ROP defenses. In *USENIX Security Symposium*. 67
- M. Marschalek. 2014. Dig deeper into the IE vulnerability (cve-2014-1776) exploit. <http://www.cyphort.com/dig-deeper-ie-vulnerability-cve-2014-1776-exploit/>. 182
- A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. 2014. Cryptographically enforced control flow integrity. <http://arxiv.org/abs/1408.1451>. 86

- A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. 2015. CCFI: Cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, pp. 941–951. DOI: [10.1145/2810103.2813676](https://doi.org/10.1145/2810103.2813676). 72, 76, 77
- M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. 2013. System V application binary interface: AMD64 architecture processor supplement. <http://x86-64.org/documentation/abi.pdf>. 150
- M. Maurer and D. Brumley. 2012. Tachyon: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*, pp. 617–630. 214, 256, 257
- S. McCamant and G. Morrisett. 2006. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*. 41, 59, 68, 86
- H. Meer. 2010. Memory corruption attacks: The (almost) complete history. In *Proceedings of Blackhat USA*. 62
- T. Merrifield and J. Eriksson. 2013. Conversion: Multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pp. 127–139. DOI: [10.1145/2465351.2465365](https://doi.org/10.1145/2465351.2465365). 230
- Microsoft Corp. November 2014. Enhanced mitigation experience toolkit (EMET) 5.1. <http://technet.microsoft.com/en-us/security/jj653751>. 173, 176
- Microsoft Developer Network. 2017. Argument passing and naming conventions. <http://msdn.microsoft.com/en-us/library/984x0h58.aspx>. 149, 151, 154
- V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. 2015. Opaque control-flow integrity. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*. <http://www.internetsociety.org/doc/opaque-control-flow-integrity>. 173, 177, 182
- J. R. Moser. 2006. Virtual machines and memory protections. <http://lwn.net/Articles/210272/>. 238
- G. Murphy. 2012. Position independent executables—adoption recommendations for packages. <http://people.redhat.com/~gmurphy/files/pie.odt>. 238
- S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *International Symposium on Computer Architecture*, pp. 189–200. DOI: [10.1145/2366231.2337181](https://doi.org/10.1145/2366231.2337181). 109
- S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. 2015. Everything you want to know about pointer-based checking. In *First Summit on Advances in Programming Languages (SNAPL)*. DOI: [10.4230/LIPIcs.SNAPL.2015.190](https://doi.org/10.4230/LIPIcs.SNAPL.2015.190). 5
- S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM Sigplan Notices*, volume 44, pp. 245–258. DOI: [10.1145/1542476.1542504](https://doi.org/10.1145/1542476.1542504). 4, 5, 36, 82, 84, 88, 91, 97, 99, 101, 102, 110, 112, 211
- S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. 2010. CETS: Compiler enforced temporal safety for C. In *ACM Sigplan Notices*, volume 45, pp. 31–40. DOI: [10.1145/1806651.1806657](https://doi.org/10.1145/1806651.1806657). 6, 83, 84, 88, 89, 91, 98, 108, 173, 178, 211

- G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526. DOI: [10.1145/1065887.1065892](https://doi.org/10.1145/1065887.1065892). 5, 82, 84, 88, 95
- Nergal. December 2001. The advanced return-into-lib(c) exploits (PaX case study). *Phrack*, 58 (4): 54. [http://www.phrack.org/archives/58/p58_0x04_Advanced%20return-into-lib\(c\)%20exploits%20\(PaX%20case%20study\)_by_nergal.txt](http://www.phrack.org/archives/58/p58_0x04_Advanced%20return-into-lib(c)%20exploits%20(PaX%20case%20study)_by_nergal.txt). 81, 82, 185, 203
- B. Niu. 2015. Practical control-flow integrity. Ph.D. thesis, Lehigh University. 26, 37, 39, 56, 60
- B. Niu and G. Tan. 2013. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 199–210. DOI: [10.1145/2508859.2516649](https://doi.org/10.1145/2508859.2516649). 39, 82, 86, 173, 175
- B. Niu and G. Tan. 2014a. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. DOI: [10.1145/2594291.2594295](https://doi.org/10.1145/2594291.2594295). 26, 27, 40, 44, 58, 82, 110, 114, 182
- B. Niu and G. Tan. 2014b. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*, pp. 1317–1328. DOI: [10.1145/2660267.2660281](https://doi.org/10.1145/2660267.2660281). 9, 26, 34
- B. Niu and G. Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 914–926. DOI: [10.1145/2810103.2813644](https://doi.org/10.1145/2810103.2813644). 14, 30, 59
- A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. 2016. Poking holes in information hiding. In *25th USENIX Security Symposium*, pp. 121–138. 11, 68, 94
- M. Olszewski, J. Ansel, and S. Amarasinghe. 2009. Kendo: Efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108. DOI: [10.1145/1508244.1508256](https://doi.org/10.1145/1508244.1508256). 230
- K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirida. 2010. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, pp. 49–58. DOI: [10.1145/1920261.1920269](https://doi.org/10.1145/1920261.1920269). 173, 177, 210
- V. Pappas, M. Polychronakis, and A. D. Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, pp. 447–462. 117, 118, 119, 127, 173, 176, 182, 209
- A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. Marx: Uncovering class hierarchies in C++ programs. In *Annual Network and Distributed System Security Symposium (NDSS)*. 67
- PaX Team. 2004a. Address space layout randomization. <http://pax.grsecurity.net/docs/aslr.txt>, 2004a. 82, 85, 211
- PaX Team. 2004b PaX non-executable pp. design & implementation. <http://pax.grsecurity.net/docs/noexec.txt>, 2004b. 8, 211

- M. Payer. 2012. Safe loading and efficient runtime confinement: A foundation for secure execution. Ph.D. thesis, ETH Zurich. <http://nebelwelt.net/publications/12PhD>. DOI: [10.1109/SP.2012.11](https://doi.org/10.1109/SP.2012.11). 8
- M. Payer, A. Barresi, and T. R. Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. DOI: [10.1007/978-3-319-20550-2_8](https://doi.org/10.1007/978-3-319-20550-2_8). 10, 14, 58, 173, 174
- M. Payer and T. R. Gross. 2011. Fine-grained user-space security through virtualization. In *Proceedings of the 7th International Conference on Virtual Execution Environments (VEE)*. DOI: [10.1145/1952682.1952703](https://doi.org/10.1145/1952682.1952703). 8, 9
- A. Pelletier. 2012. Advanced exploitation of Internet Explorer heap overflow (Pwn2Own 2012 exploit). VUPEN Vulnerability Research Team (VRT) blog. http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php. 124, 131
- J. Pewny and T. Holz. 2013. Control-flow restrictor: Compiler-based CFI for iOS. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 309–318. DOI: [10.1145/2523649.2523674](https://doi.org/10.1145/2523649.2523674). 58
- Phoronix. Phoronix test suite. <http://www.phoronix-test-suite.com/>. 114
- A. Prakash, X. Hu, and H. Yin. 2015. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Symposium on Network and Distributed System Security (NDSS)*. 58, 160, 170, 171, 173, 176, 182
- N. Provos. 2003. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium (SSYM)*, volume 12, pp. 18–18. <http://dl.acm.org/citation.cfm?id=1251353.1251371>. 16, 241
- H. P. Reiser, J. Domaschka, F. J. Hauck, R. Kapitza, and W. Schröder-Preikschat. 2006. Consistent replication of multithreaded distributed objects. In *IEEE Symposium on Reliable Distributed Systems*, pp. 257–266. DOI: [10.1109/SRDS.2006.14](https://doi.org/10.1109/SRDS.2006.14). 230
- R. Roemer, E. Buchanan, H. Shacham, and S. Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34. DOI: [10.1145/2133375.2133377](https://doi.org/10.1145/2133375.2133377). 20, 117, 181, 185
- R. Rudd, R. Skowrya, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi. 2017. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *Annual Network and Distributed System Security Symposium (NDSS)*. 69
- J. M. Rushby. 1981. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 12–21. DOI: [10.1145/800216.806586](https://doi.org/10.1145/800216.806586). 214
- M. Russinovich, D. A. Solomon, and A. Ionescu. 2012. *Windows Internals, Part 1*. Microsoft Press, 6th edition. ISBN 978-0-7356-4873-9. 155, 175
- SafeStack. Clang documentation: Safestack. <http://clang.llvm.org/docs/SafeStack.html>. 102

- B. Salamat. 2009. Multi-variant execution: Run-time defense against malicious code injection attacks. Ph.D. thesis, University of California at Irvine. 218
- B. Salamat, T. Jackson, A. Gal, and M. Franz. 2009. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, pp. 33–46. DOI: [10.1145/1519065.1519071](https://doi.org/10.1145/1519065.1519071). 211, 213, 214, 217, 227, 256, 257
- J. Salwan. 2011. ROPGadget. <http://shell-storm.org/project/ROPgadget/>. 136
- F. Schuster. July 2015. *Securing Application Software in Modern Adversarial Settings*. Ph.D. thesis, Katholieke Universiteit Leuven. 140
- F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy (S&P)*, pp. 745–762. DOI: [10.1109/SP.2015.51](https://doi.org/10.1109/SP.2015.51). 15, 20, 67, 70, 97, 140, 182, 183, 184, 185, 186, 200, 204, 211
- F. Schuster, T. Tendyck, J. Powny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. 2014. Evaluating the effectiveness of current anti-ROP defenses. In *Research in Attacks, Intrusions, and Defenses*, volume 8688 of *Lecture Notes in Computer Science*. DOI: [10.1007/978-3-319-11379-1_5](https://doi.org/10.1007/978-3-319-11379-1_5). 139, 140, 177, 182, 186, 188, 209
- E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*. DOI: [10.1109/SP.2010.26](https://doi.org/10.1109/SP.2010.26). 86
- E. J. Schwartz, T. Avgerinos, and D. Brumley. 2011. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security (SEC)*, pp. 25–25. 136
- C. Segulja and T. S. Abdelrahman. 2014. What is the cost of weak determinism? In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 99–112. DOI: [10.1145/2628071.2628099](https://doi.org/10.1145/2628071.2628099). 254
- J. Seibert, H. Okhravi, and E. Söderström. 2014. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pp. 54–65. DOI: [10.1145/2660267.2660309](https://doi.org/10.1145/2660267.2660309). 141, 182
- K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pp. 309–318. 82, 84, 173, 178
- F. J. Serna. 2012. CVE-2012-0769, the case of the perfect info leak. http://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf. 63, 117
- H. Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, pp. 552–561. DOI: [10.1145/1315245.1315313](https://doi.org/10.1145/1315245.1315313). 62, 172, 184, 185, 186, 200, 233

- H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pp. 298–307. DOI: [10.1145/1030083.1030124](https://doi.org/10.1145/1030083.1030124). 62
- N. Shavit and D. Touitou. 1995. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 204–213. 28
- K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy (S&P)*, pp. 574–588. DOI: [10.1109/SP.2013.45](https://doi.org/10.1109/SP.2013.45). 10, 20, 49, 63, 82, 86, 117, 141, 177, 182, 184, 186, 203
- K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. 2016. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *37th IEEE Symposium on Security and Privacy (S&P)*, pp. 954–968. DOI: [10.1109/SP.2016.61](https://doi.org/10.1109/SP.2016.61). 70
- Solar Designer. 1997a. “return-to-libc” attack. Bugtraq. 203
- Solar Designer. 1997b. lpr LIBC RETURN exploit. <http://insecure.org/splotts/linux.libc.return.lpr.splott.html>. 203
- C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. 2015. Exploiting and protecting dynamic code generation. In *Network and Distributed System Security Symposium (NDSS)*. 49, 58, 60
- A. Sotirov. 2007. Heap feng shui in JavaScript. In *Proceedings of Black Hat Europe*. 132
- E. H. Spafford. January 1989. The internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19 (1): 17–57. ISSN 0146-4833. DOI: [10.1145/66093.66095](https://doi.org/10.1145/66093.66095). 61
- SPARC. SPARC V8 processor. <http://www.sparc.org>. 206
- R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. 2009. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security (EUROSEC)*, pp. 1–8. DOI: [10.1145/1519144.1519145](https://doi.org/10.1145/1519144.1519145). 63, 117
- D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin. 2016. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 83.2:1–6. DOI: [10.1145/2897937.2898098](https://doi.org/10.1145/2897937.2898098). 209
- L. Szekeres, M. Payer, T. Wei, and D. Song. 2013. SoK: Eternal war in memory. In *Proceedings International Symposium on Security and Privacy (S&P)*. DOI: [10.1109/SP.2013.13](https://doi.org/10.1109/SP.2013.13). 2, 61, 82, 85, 211
- L. Szekeres, M. Payer, L. Wei, D. Song, and R. Sekar. 2014. Eternal war in memory. *IEEE Security and Privacy Magazine*. DOI: [10.1109/MSP.2013.47](https://doi.org/10.1109/MSP.2013.47). 2
- A. Tang, S. Sethumadhavan, and S. Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM Conference on Computer and Communications Security (CCS)*, pp. 256–267. DOI: [10.1145/2810103.2813685](https://doi.org/10.1145/2810103.2813685). 70

- C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*. <http://dl.acm.org/citation.cfm?id=2671225.2671285>. 58, 86, 173, 175, 182, 204, 208, 211, 233
- M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. 2011. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, pp. 121–141. DOI: [10.1007/978-3-642-23644-0_7](https://doi.org/10.1007/978-3-642-23644-0_7). 117, 140, 183, 184, 185, 200, 204
- A. van de Ven. August 2004. New security enhancements in Red Hat Enterprise Linux v.3, update 3. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf. 9, 82
- V. van der Veen, D. Andriess, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. 2015. PathArmor: Practical ROP protection using context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 927–940. DOI: [10.1145/2810103.2813673](https://doi.org/10.1145/2810103.2813673). 14, 137
- V. van der Veen, N. D. Sharma, L. Cavallaro, and H. Bos. 2012. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, pp. 86–106. DOI: [10.1007/978-3-642-33338-5_5](https://doi.org/10.1007/978-3-642-33338-5_5). 61
- S. Volckaert. 2015. Advanced Techniques for multi-variant execution. Ph.D. thesis, Ghent University. 226, 231
- S. Volckaert, B. Coppens, and B. De Sutter. 2015. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Trans. on Dependable and Secure Computing*, 13 (4): 437–450. DOI: [10.1109/TDSC.2015.2411254](https://doi.org/10.1109/TDSC.2015.2411254). 211, 250
- S. Volckaert, B. Coppens, B. De Sutter, K. De Bosschere, P. Larsen, and M. Franz. 2017. Taming parallelism in a multi-variant execution environment. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pp. 270–285. DOI: [10.1145/3064176.3064178](https://doi.org/10.1145/3064176.3064178). 230, 232
- S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz. 2016. Secure and efficient application monitoring and replication. In *USENIX Annual Technical Conference (ATC)*, pp. 167–179. 214, 215, 247
- S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere. 2013. GHUMVEE: Efficient, effective, and flexible replication. In *5th International Symposium on Foundations and Practice of Security (FPS)*, pp. 261–277. 214, 217, 232
- R. Wahbe, S. Lucco, T. Anderson, and S. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pp. 203–216. DOI: [10.1145/168619.168635](https://doi.org/10.1145/168619.168635). 8, 9, 41, 68, 249
- Z. Wang and X. Jiang. 2010. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 380–395. DOI: [10.1109/SP.2010.30](https://doi.org/10.1109/SP.2010.30). 58, 208

- R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 157–168. DOI: [10.1145/2382196.2382216](https://doi.org/10.1145/2382196.2382216). 11, 173, 177
- R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. 2010. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium*, pp. 29–46. 16
- T. Wei, T. Wang, L. Duan, and J. Luo. 2011. INSERT: Protect dynamic code generation against spraying. In *International Conference on Information Science and Technology (ICIST)*, pp. 323–328. DOI: [10.1109/ICIST.2011.5765261](https://doi.org/10.1109/ICIST.2011.5765261). 59
- J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis. 2016. No-execute-after-read: Preventing code disclosure in commodity software. In *11th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 35–46. DOI: [10.1145/2897845.2897891](https://doi.org/10.1145/2897845.2897891). 70
- J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. 2011. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 41–50. DOI: [10.1145/2076732.2076739](https://doi.org/10.1145/2076732.2076739). 109, 239
- R. Wojtczuk. 1998. Defeating Solar Designer's non-executable stack patch. <http://insecure.org/sploits/non-executable.stack.problems.html>. 20, 81, 82, 203
- C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. 2002. Linux security modules: General security support for the Linux kernel. In *Proceedings 11th USENIX Security Symposium*. 16
- R. Wu, P. Chen, B. Mao, and L. Xie. 2012. RIM: A method to defend from JIT spraying attack. In *7th International Conference on Availability, Reliability, and Security (ARES)*, pp. 143–148. DOI: [10.1109/ARES.2012.11](https://doi.org/10.1109/ARES.2012.11). 59
- Y. Xia, Y. Liu, H. Chen, and B. Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, pp. 1–12. DOI: [10.1109/DSN.2012.6263958](https://doi.org/10.1109/DSN.2012.6263958). 173, 176
- F. Yao, J. Chen, and G. Venkataramani. 2013. JOP-alarm: Detecting jump-oriented programming-based anomalies in applications. In *IEEE 31st International Conference on Computer Design (ICCD)*, pp. 467–470. DOI: [10.1109/ICCD.2013.6657084](https://doi.org/10.1109/ICCD.2013.6657084). 209
- B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy (S&P)*, pp. 79–93. DOI: [10.1109/SP.2009.25](https://doi.org/10.1109/SP.2009.25). 8, 39, 86
- B. Zeng, G. Tan, and Ú. Erlingsson. 2013. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security Symposium*, pp. 369–382. 58, 86
- B. Zeng, G. Tan, and G. Morrisett. 2011. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 29–40. DOI: [10.1145/2046707.2046713](https://doi.org/10.1145/2046707.2046713). 58, 59, 86

- C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. 2015. VTint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)*. DOI: [10.14722/ndss.2015.23099](https://doi.org/10.14722/ndss.2015.23099) . 160, 173, 176, 182
- C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. 2013. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy (S&P)*, pp. 559–573. DOI: [10.1109/SP.2013.44](https://doi.org/10.1109/SP.2013.44). 38, 82, 86, 97, 110, 114, 117, 118, 119, 127, 136, 169, 173, 174, 182, 208, 209
- M. Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, pp. 337–352. <http://dl.acm.org/citation.cfm?id=2534766.2534796>. 38, 82, 86, 97, 110, 114, 117, 118, 119, 127, 136, 173, 174, 182, 208, 209
- H. W. Zhou, X. Wu, W. C. Shi, J. H. Yuan, and B. Liang. 2014. HDROP: Detecting ROP attacks using performance monitoring counters. In *Information Security Practice and Experience*, pp. 172–186. Springer International Publishing. DOI: [10.1007/978-3-319-06320-1_14](https://doi.org/10.1007/978-3-319-06320-1_14). 173, 177
- X. Zhou, K. Lu, X. Wang, and . Li. 2012. Exploiting parallelism in deterministic shared memory multiprocessing. *Journal of Parallel and Distributed Computing*, 72(5):716–727. DOI: [10.1016/j.jpdc.2012.02.008](https://doi.org/10.1016/j.jpdc.2012.02.008). 230

Contributor Biographies

Editors

Per Larsen is trying his hand as an entrepreneur and co-founded an information security startup—Immunant, Inc.—specializing in exploit mitigation. Previously, he worked for four years as a postdoctoral scholar at the University of California, Irvine. He graduated with a Ph.D. from the Technical University of Denmark in 2011.

Per co-organized the 2015 Dagstuhl Seminar upon which this book is based and has served as program committee member for several academic conferences including USENIX Security, USENIX WOOT, ICDCS, and AsiaCCS. In 2015, he was recognized as a DARPA Riser.

Ahmad Sadeghi is a full professor of Computer Science at TU Darmstadt, Germany, and the Director of the Intel Collaborative Research Institute for Secure Computing (ICRI-SC) at TU Darmstadt. He holds a Ph.D. in computer science from the University of Saarland in Saarbrücken, Germany. Prior to academia, he worked in research and development for telecommunications enterprises, amongst others Ericsson Telecommunications. He is editor-in-chief of *IEEE Security and Privacy Magazine*, and on the editorial board of ACM Books. He served five years on the editorial board of the *ACM Transactions on Information and System Security* (TISSEC), and was guest editor of the *IEEE Transactions on Computer-Aided Design (Special Issue on Hardware Security and Trust)*.

Authors

Orlando Arias is pursuing his Ph.D. in computer engineering from the University of Central Florida under the advisement of Dr. Yier Jin. His research interests include secure computer architectures, network security, IP core design, and integration and cryptosystems.

Elias Athanasopoulos is an assistant professor with the Computer Science Department at the University of Cyprus. Before joining the University of Cyprus, he was an assistant professor with Vrije Universiteit Amsterdam. He holds a BSc in physics

from the University of Athens and a Ph.D. in computer science from the University of Crete. Elias is a Microsoft Research Ph.D. Scholar. He has interned with Microsoft Research in Cambridge and worked as a research assistant with FORTH in Greece from 2005 to 2011. Elias is also a Marie Curie fellow. Before joining the faculty of Vrije Universiteit Amsterdam, he was a postdoctoral research scientist with Columbia University and a collaborating researcher with FORTH.

Herbert Bos is a professor of systems and network security at Vrije Universiteit Amsterdam, where he heads the VUsec research group. He obtained his Ph.D. from Cambridge University Computer Laboratory (UK). Coming from a systems background, he drifted into security a few years ago and never left.

George Candea heads the Dependable Systems Lab at EPFL, where he conducts research on both the fundamentals and the practice of achieving reliability and security in complex software systems. His main focus is on real-world large-scale systems—millions of lines of code written by hundreds of programmers—because going from a small program to a large system introduces fundamental challenges that cannot be addressed with the techniques that work at small scale. George is also Chairman of Cyberhaven, a cybersecurity company he co-founded with his former students to defend sensitive data against advanced attacks, social engineering, and malicious insiders. In the past, George was CTO and later Chief Scientist of Aster Data Systems (now Teradata Aster). Before that, he held positions at Oracle, Microsoft Research, and IBM Research. George is a recipient of the first Eurosys Jochen Liedtke Young Researcher Award (2014), an ERC StG award (2011), and the MIT TR35 Young Innovators award (2005). He received his Ph.D. (2005) in computer science from Stanford and his B.S. (1997) and M.Eng. (1998) in electrical engineering and computer science from MIT.

Bart Coppens is a postdoctoral researcher at Ghent University in the Computer Systems Lab. He received his Ph.D. in computer science engineering from the Faculty of Engineering and Architecture at Ghent University in 2013. His research focuses on protecting software against different forms of attacks using compiler-based techniques and run-time techniques.

Stephen Crane started seriously diving into security during his undergrad at Cal Poly Pomona, competing in CCDC. From there he worked on research somewhere in the intersection of systems security and compilers at UC Irvine. After transforming into Dr. Crane, Stephen founded Immunant with fellow UCI researchers, where he tries to get exploit mitigation tools into the hands of developers.

Lucas Davi is an assistant professor of computer science at University of Duisburg-Essen, Germany, and associated researcher at the Intel Collaborative Research

Institute for Secure Computing (ICRI-SC) at TU Darmstadt, Germany. He received his Ph.D. in computer science from TU Darmstadt. His research focus includes system security, software security, and trusted computing. His Ph.D. thesis on code-reuse attacks and defenses has been awarded with the ACM SIGSAC Dissertation Award 2016.

Bjorn De Sutter is a professor at Ghent University in the Computer Systems Lab. He obtained his MSc. and Ph.D. degrees in computer science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques and run-time techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and software protection. He has published over 80 peer-reviewed papers on these topics.

Michael Franz is the director of the Secure Systems and Software Laboratory at the University of California, Irvine (UCI). He is a full professor of computer science in UCI's Donald Bren School of Information and Computer Sciences and a full professor of electrical engineering and computer science (by courtesy) in UCI's Henry Samueli School of Engineering. Prof. Franz was an early pioneer in the areas of mobile code and dynamic compilation. He created an early just-in-time compilation system, contributed to the theory and practice of continuous compilation and optimization, and co-invented the trace compilation technology that eventually became the JavaScript engine in Mozilla's Firefox browser. Franz received a Dr. sc. techn. degree in computer science and a Dipl. Informatik-Ing. ETH degree, both from the Swiss Federal Institute of Technology, ETH Zurich.

Enes Göktas is a Ph.D. Student in the systems and network security group at the Vrije Universiteit Amsterdam. His research focus is on evaluating and developing mitigations against memory corruption vulnerabilities. His previous work includes evaluation and proposal of Control-Flow Integrity based mitigations. His interests lie in the area of software security, as well as binary analysis and instrumentation.

Thorsten Holz is a professor in the Faculty of Electrical Engineering and Information Technology at Ruhr-University Bochum, Germany. His research interests include systems-oriented aspects of secure systems, with a specific focus on applied computer security. Currently, his work concentrates on bots/botnets, automated analysis of malicious software, and studying the latest attack vectors. He received the Dipl.-Inform. degree in computer science from RWTH Aachen, Germany (2005), and a Ph.D. degree from University of Mannheim (2009). Prior to joining Ruhr-University Bochum in April 2010, he was a postdoctoral researcher in the Automation Systems Group at the Technical University of Vienna, Austria. In

2011, Thorsten received the Heinz Maier-Leibnitz Prize from the German Research Foundation (DFG).

Andrei Homescu finished his doctoral studies at UC Irvine in 2015, after which he co-founded Immunant with three of the present co-authors. Andrei has published widely in the areas of systems security, language runtimes, and exploit mitigation. He also led the development of selfrando, a production-ready, open-source randomization engine, and is the lead author on several US patents and patent applications.

Yier Jin received his Ph.D. in electrical engineering from Yale University in 2012. He is an assistant professor in the Department of Electrical and Computer Engineering, University of Central Florida, USA. His research interests include hardware security, IoT security, and formal methods. He is a member of IEEE and ACM.

Volodymyr Kuznetsov is a security researcher who focuses on practical applications of program analysis and other formal methods in systems security, with particular interest in protecting sensitive data in applications and systems with security vulnerabilities. Volodymyr's research was released as open source projects that have become widely used in research community and industry (<http://s2e.epfl.ch/>, <http://clang.llvm.org/docs/SafeStack.html>, and others) and were recognized with an open source award. Volodymyr obtained his Ph.D. in Computer Science at EPFL in 2016 and now leads a cyber security company that brings state-of-the-art research into the world of enterprise cyber security.

Ben Niu earned his Ph.D. degree in computer science from Lehigh University, USA, in 2016, advised by professor Gang Tan. He is currently a security software engineer at Microsoft Corporation. His research interests are system security and parallel programming.

Hamed Okhravi is a senior staff member at the Cyber Analytics and Decision Systems Group of MIT Lincoln Laboratory, where he leads programs and conducts research in the area of systems security. His research interests include cyber security, science of security, security evaluation, and operating systems. He is the recipient of the 2014 MIT Lincoln Laboratory Early Career Technical Achievement Award and 2015 Team Award for his work on cyber moving target research. He is also the recipient of an honorable mention (runner-up) at the 2015 NSA's 3rd Annual Best Scientific Cybersecurity Paper Competition. Currently, his research is focused on analyzing and developing system security defenses.

He has served as a program chair for the ACM CCS Moving Target Defense (MTD) workshop and program committee member for a number of academic conferences and workshops including ACM CCS, NDSS, RAID, AsiaCCS, ACNS, and IEEE SecDev.

Dr. Okhravi earned his MS and Ph.D. in electrical and computer engineering from University of Illinois at Urbana-Champaign in 2006 and 2010, respectively.

Mathias Payer is a security researcher and assistant professor in computer science at Purdue University, leading the HexHive group. His research focuses on protecting applications even in the presence of vulnerabilities, with a focus on memory corruption. He is interested in system security, binary exploitation, user-space software-based fault isolation, binary translation/recompilation, and (application) virtualization. All implementation prototypes from his group are open source. In 2014, he founded the b01lers Purdue CTF team. Before joining Purdue in 2014, he spent two years as a postdoc in Dawn Song's BitBlaze group at UC Berkeley. He graduated from ETH Zurich with a Dr. sc. ETH in 2012.

Georgios Portokalidis is an assistant professor in the Department of Computer Science at Stevens Institute of Technology. He obtained his Ph.D. from Vrije Universiteit in Amsterdam in February 2010, and also spent a couple of years as a postdoc at Columbia University in New York. His research interests center mainly around the area of systems and security, including software and network security, authentication, privacy, and software resiliency. His recent work has revolved around code-reuse attacks, efficient information-flow tracking, and security applications using the Internet of Things. During his Ph.D. he worked on the Argos emulator, a platform for hosting high-interaction honeypots that can automatically detect zero-day control-flow hijacking attacks, and Paranoid Android, a record-replay system for the Android OS.

Felix Schuster has been a researcher at the Microsoft Research Cambridge (UK) lab since 2015. Before joining Microsoft Research, he obtained a Ph.D. from Ruhr-Universität Bochum. Felix is broadly interested in applied systems and software security and is part of the lab's Constructive Security Group. In the past, he worked on topics like code-reuse attacks and defenses (e.g., COOP) and automated binary code analysis. Currently, Felix's research focusses on the design of practical solutions for the trusted cloud like the VC3 system or the Coco blockchain framework.

R. Sekar is a Professor of Computer Science and the Director of the Secure Systems Laboratory and the Center for Cyber Security at Stony Brook University. He received his Bachelor's degree in Electrical Engineering from IIT, Madras (India) in 1986, and his Ph.D. in Computer Science from Stony Brook in 1991. He then served as a Research Scientist at Bellcore until 1996, and then as faculty at Iowa State University. Sekar's research interests are focused on software exploit detection and mitigation, malware and untrusted code defense, and security policies and their enforcement.

Dawn Song is a professor in the Department of Electrical Engineering and Computer Science at UC Berkeley. Her research interest lies in deep learning and security. She has studied diverse security and privacy issues in computer systems and networks, including areas ranging from software security, networking security, database security, distributed systems security, and applied cryptography, to the intersection of machine learning and security. She is the recipient of various awards including the MacArthur Fellowship, the Guggenheim Fellowship, the NSF CAREER Award, the Alfred P. Sloan Research Fellowship, the MIT Technology Review TR-35 Award, the George Tallman Ladd Research Award, the Okawa Foundation Research Award, the Li Ka Shing Foundation Women in Science Distinguished Lecture Series Award, the Faculty Research Award from IBM, Google and other major tech companies, and Best Paper Awards from top conferences. She obtained her Ph.D. from UC Berkeley. Prior to joining UC Berkeley as a faculty, she was an Assistant Professor at Carnegie Mellon University from 2002–2007.

Dean Sullivan is pursuing his Ph.D. in computer engineering from the University of Central Florida under the advisement of Dr. Yier Jin. His research interests include system security and computer architecture.

László Szekeres is a software security researcher at Google. He works on developing techniques for protecting against security bugs, primarily in C/C++ code. His research is focused on finding and hardening against vulnerabilities using automated test generation, program analysis, compiler techniques, and machine learning. He obtained his Ph.D. in Computer Science from Stony Brook University in 2017. During his studies he spent a year as a visiting researcher at UC Berkeley. In 2010, he was awarded the Fulbright Foreign Student Scholarship. Before returning to academia for his doctorate degree, he led a security research team at a spin-off company of the Budapest University of Technology and Economics.

Gang Tan received his Ph.D. in computer science from Princeton University in 2005. He is an associate professor in the Department of Computer Science and Engineering, Pennsylvania State University, USA. His research interests include software security, programming languages, and formal methods. He is a member of IEEE and ACM.

Stijn Volckaert received his Ph.D. degree from Ghent University's Faculty of Engineering and Architecture. He is currently a postdoctoral scholar in the Department of Computer Science at the University of California, Irvine. His research interests include security, operating systems, and software protection.