UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Enhancing and Extending Software Diversity

Permalink
https://escholarship.org/uc/item/45w3n70k

Author
Crane, Stephen

Publication Date
2015

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Enhancing and Extending Software Diversity

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Stephen James Crane

Dissertation Committee:
Professor Michael Franz, Chair
Professor Ian Harris
Professor Gene Tsudik

2015

Chapter 2 © 2013 ACM.

Portions of Chapter 3 © 2015 Stephen Crane et al.

Chapter 4 © 2015 Internet Society.

All other materials © 2015 Stephen James Crane.

DEDICATION

I humbly dedicate this work to the almighty God—Father, Son, and Holy Spirit—who sent me upon this journey, supported me, and sustained me and to my wonderful wife, Sharon, who has stood by my side encouraging me the whole way.

Whatever you do, work heartily, as for the Lord and not for men. . .

— Colossians 3:23

TABLE OF CONTENTS

Page

LIST OF FIGURES v

LIST OF TABLES vi

LIST OF ALGORITHMS vii

ACKNOWLEDGMENTS viii

CURRICULUM VITAE x

ABSTRACT OF THE DISSERTATION xii

INTRODUCTION 1

CHAPTER 1: Background 4
- 1.1 Unsafe Languages and Memory Corruption 4
- 1.2 Co-evolution of Exploits and Defenses 5
- 1.3 Side-Channel Attacks 12

CHAPTER 2: Booby Traps 14
- 2.1 Motivation 14
- 2.2 Booby Traps 18
- 2.3 Active Responses 22
- 2.4 Conclusion 29

CHAPTER 3: Readactor++ 31
- 3.1 Motivation 31
- 3.2 Background 34
- 3.3 Adversary Model and Assumptions 40
- 3.4 Design 41
- 3.5 Implementation 50
- 3.6 Evaluation 66
- 3.7 Discussion 79
- 3.8 Conclusion 81

CHAPTER 4: Control-Flow Diversity 83
4.1 Motivation 83
4.2 Side-Channel Background 86
4.3 Dynamic Control-Flow Diversity 92
4.4 Evaluation 98
4.5 Discussion 108
4.6 Conclusion and Outlook 111

CHAPTER 5: Related Work 112
5.1 Code Reuse 112
5.2 Active Defense 126
5.3 Hypervisors 127
5.4 Side Channels 128

CHAPTER 6: Conclusion 132

BIBLIOGRAPHY 134

LIST OF FIGURES

Page

2.1 Combining software diversity with booby trapping. 17
2.2 Weaving booby traps into target binaries. 19
2.3 Cooperative Situational Awareness 23

3.1 Direct vs. indirect memory disclosure. 35
3.2 Readactor++ System Overview 43
3.3 Code-Pointer Hiding Overview 45
3.4 Vtable Splitting and Randomization 46
3.5 PLT Randomization 49
3.6 Relationship between virtual, guest physical, and host physical memory. 51
3.7 Readactor++ Hypervisor Architecture 53
3.8 Switch table code-data separation 56
3.9 Hiding function pointers 57
3.10 Hiding return addresses 58
3.11 Example C++ class hierarchy 63
3.12 Performance overhead of Readactor++ on SPEC CPU2006 73
3.13 Performance overhead of Readactor++ table randomization on SPEC CPU2006 C++ benchmarks 75

4.1 Example of a time-based side channel 85
4.2 Example of cache structure on a modern processor 89
4.3 Side-channel resistance of diversification techniques. 91
4.4 Function call graph before and after replicating the function d(). 93
4.5 Detailed view of randomized trampoline d'() and interaction with the runtime system. 94
4.6 Memory layout of runtime address tables 95
4.7 Average accuracy of AES side-channel attacks with our defense 98
4.8 Control-flow diversity effectiveness as attacker capabilities increas 103
4.9 Performance slowdown factor of control-flow diversity on AES benchmarks . . . 104
4.10 Performance slowdown factor on SPEC CPU2006 with function-level dynamic control-flow diversity 105

LIST OF TABLES

Page

4.1 File size increase for libgcrypt 106

LIST OF ALGORITHMS

Page

3.1 Update vtable references after randomization. 66

4.1 EVICT+TIME attack. 90
4.2 PRIME+PROBE attack. 90

ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Prof. Michael Franz, for his wonderful encouragement, guidance, and support during my studies at UCI. Thank you for giving me such an amazing academic home in your group.

Prof. Franz's Secure Systems and Software lab has been an amazing group of researchers, collaborators, and (most importantly) friends. I am especially grateful to my postdocs, Dr. Stefan Brunthaler and Dr. Per Larsen, for all of their great ideas, critical assistance, and major contributions to my research efforts. I am also immensely grateful to Andrei Homescu for his collaboration and support on many of my research endeavors (and gaming escapades). Finally, I want to thank all of my friends in the lab for their wonderful community. Michael, Gregor, Mason, Todd, Christoph, Eric, Wei, Gülfem, Andrei, Codruţ, Mark, Mohaned, Julian, Bryan, Steven, and Chen — working with you has been awesome, thanks!

I would also like to thank those outside my research group whom I have had the privilege of collaborating with, especially Lucas Davi, Thorsten Holz, Mathias Payer, Ahmad-Reza Sadeghi, Felix Schuster, Christopher Liebchen, and Stijn Volckeart. Thanks for all your hard work. Christopher and Stijn, without your implementation efforts, Readactor++ would never have become a reality. I am also grateful to my first research mentor, Ben Cooper, and my undergraduate advisor, Prof. Craig Rich, for guiding and assisting in my early research career.

I owe a huge thank you to Julian Lettner, Mark Murphy, Mathias Payer, Robert Turner, and all of my labmates for their detailed and constructive feedback on drafts of this and other work.

I am sincerely grateful to Prof. Ian Harris and Prof. Gene Tsudik for sitting on my advancement and doctoral committees. I would also like to thank Prof. Guoqing (Harry) Xu and Prof. David Reinkensmeyer for sitting on my advancement committee. Thank you for your time, consideration, and support.

Finally, I am so immensely and truly grateful to my family and friends who have supported me along my studies. Sharon, thank you for all of your support, encouragement, and help as I have been in school. You have been such a wonderful and supportive wife. Mom and Dad, thank you so much for the world-class education you gave me growing up and all your support throughout my life. I owe much of my success in school to your hard work. My heartfelt gratitude goes out to all of my family; your encouragement has been wonderful. I am especially grateful to my brothers and sisters in Christ from the Graduate Christian Fellowship at UCI who have encouraged, supported, and prayed for me as I have walked this path of graduate school.

Portions of this dissertation have been previously published in conference proceedings. I am extremely grateful to my co-authors for all their assistance in these publications.

Chapter 2 is reprinted, with permission, from Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. *Booby trapping software.* In Proceedings of the 2013 New Security Paradigms Workshop (NSPW '13). 2013. `http://dx.doi.org/10.1145/2535813.2535824`.

Portions of chapter 3 are reprinted, with permission, from Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. *Readactor: Practical Code Randomization Resilient to Memory Disclosure.* In Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P '15). 2015.

Chapter 4 is reprinted, with permission, from Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. *Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity.* In Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS '15). 2015.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014 and by gifts from Oracle and Mozilla.

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 National Science Foundation, or any other agency of the U.S. Government.

CURRICULUM VITAE

Stephen James Crane

EDUCATION

Doctor of Philosophy in Computer Science — **2015**
University of California, Irvine — *Irvine, California*

Master of Science in Computer Science — **2014**
University of California, Irvine — *Irvine, California*

Bachelor of Science in Computer Science — **2011**
California State Polytechnic University, Pomona — *Pomona, California*

RESEARCH EXPERIENCE

Graduate Student Researcher — **2011–2015**
University of California, Irvine — *Irvine, California*

Research Intern — **Summer 2012**
Mozilla — *Mountain View, CA*

Undergraduate Researcher — **Summer 2010**
University of Southern California — *Los Angeles, CA*
NSF CS Research Experience for Undergraduates

Student Researcher — **2003–2005**
Hawthorne Center — *Fontana, CA*

TEACHING EXPERIENCE

Teaching Assistant — **Spring 2014**
University of California, Irvine — *Irvine, California*
CS 141: Concepts in Programming Languages I

Teaching Assistant — **Fall 2012**
University of California, Irvine — *Irvine, California*
ICS 139W: Critical Writing on Information Technology

REFEREED CONFERENCE PUBLICATIONS

Readactor: Practical Code Randomization Resilient to Memory Disclosure. **IEEE S&P 2015**
Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz.
IEEE Symposium on Security and Privacy

Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. **NDSS 2015**
Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz.
Network and Distributed System Security Symposium

REFEREED JOURNAL PUBLICATIONS

Large-scale Automated Software Diversity—Program Evolution Redux. **2015**
Andrei Homescu, Todd Jackson, Stephen Crane, Stefan Brunthaler, Per Larsen, and Michael Franz.
IEEE Transactions on Dependable and Secure Computing

REFEREED WORKSHOP PUBLICATIONS

Booby Trapping Software. **NSPW 2013**
Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz.
Workshop on New Security Paradigms

OTHER PUBLICATIONS

Diversifying the Software Stack using Randomized NOP Insertion. **2012**
Todd Jackson, Andrei Homescu, Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz.
Book Chapter in Moving Target Defense II: Application of Game Theory and Adversarial Modeling, volume 100 of Advances in Information Security

ABSTRACT OF THE DISSERTATION

Enhancing and Extending Software Diversity

By

Stephen James Crane

Doctor of Philosophy in Computer Science

University of California, Irvine, 2015

Professor Michael Franz, Chair

Software immunity through diversity is a promising research direction. Address Space Layout Randomization has been widely deployed to defend against code-reuse attacks and significantly raises the bar for attackers. However, automated software diversity is still exploitable by adroit and adaptable adversaries. Using powerful memory disclosure attacks, offensive researchers have demonstrated weaknesses in conventional randomization techniques. In addition, current defenses are largely passive and allow attackers to continuously brute-force randomized defenses with little impediment.

Building on the foundation of automated software diversity, we propose novel techniques to strengthen the security and broaden the impact of code randomization. We first discuss software booby traps, a new active defense technique enabled by randomized program contents. We then propose, implement, and evaluate a comprehensive randomization-based system, Readactor++, which is resilient to all types of memory disclosure attacks. Readactor++ enforces execute-only memory protections on commodity x86 processors, thus preventing direct disclosure of randomized code. We also identify the indirect disclosure attack, a new class of code leakage via data disclosure, and mitigate this attack as well. By integrating booby traps into our system, we protect against brute-force memory disclosure attempts. In our evaluation we find that

Readactor++ compares favorably to other memory-disclosure resilient code-reuse defenses and that it scales effectively to complex, real-world software.

Finally, we propose a novel extension of code randomization to mitigate side-channel rather than code-reuse attacks. Using control-flow diversity, a novel control-flow transformation, we introduce dynamic behavior into program side effects with fast, static code. As an example, we apply this technique to mitigate an AES cache side-channel attack.

With our techniques, software diversity can now be efficiently secured against advanced attacks, including memory disclosure and function table reuse, and is adaptable to combat new classes of threats, such as side-channel attacks.

Introduction

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say.

— J. R. R. Tolkien, *The Fellowship of the Ring*

Much of today's software is still implemented in insecure, unsafe systems languages such as C or C++. These languages have the benefits of being extremely fast, efficient, and compatible with existing software and systems. However, programming mistakes in unsafe languages can lead to disastrous results; attackers can often exploit memory corruption vulnerabilities to take control of the target and remotely execute arbitrary code. To secure systems against these powerful attacks, we need strong but efficient defenses. Currently deployed defenses leave much to be desired on this account.

Current software defenses are almost entirely passive—they attempt to harden applications against attacks, but do not perform any action in response to attacks. Passive defenses result in an imbalance in the power struggle between attackers and defenders, since attackers are largely free to attack without repercussions. Attackers can throw attacks at the proverbial wall

and see what sticks. We see this problem echoed throughout the information security industry and research communities: attacks are easier than defenses.

Another major factor in software insecurity is the current software ecosystem monoculture—every copy of a given application is identical. Unfortunately attackers benefit greatly from this monoculture, since every instance of a target has exactly the same attack surface and set of vulnerabilities. In contrast, automated software diversity makes every target look different to an attacker by randomizing the attack surface of the application without changing its functionality.

Software diversity is particularly well suited to defend against code-reuse attacks. To successfully reuse code in an attack, the attacker needs to know or disclose the exact machine code of the application, which is easy to automatically randomize. Consequently, defensive researchers have proposed many code-reuse defenses based on code randomization. For example, Address Space Layout Randomization (ASLR), which randomizes the starting address of loaded segments, has been widely adopted in industry.

However, randomization-based defenses have a systemic weakness; they rely on the secrecy of the randomized implementation aspects. Recent research has shown that memory disclosure attacks can violate the assumed secrecy of program code after randomization. Randomization-based defenses fail to protect against attackers who can disclose the code layout at runtime and automatically generate customized attacks based on this disclosure.

In this dissertation we attempt to tackle these shortcomings by building on the foundation of automated software diversity. In particular, we extend diversity with software booby traps and utilize execute-only memory to protect against disclosure of randomized code. We also explore a novel application of automated software diversity to the problem of side-channel attacks. These advances provide new capabilities and directions for future defensive research.

We first introduce the use of software booby traps to disincentivize attackers from repeated or brute-force attacks where they are free to arbitrarily continue until succeeding. We show

an example of how booby traps can be hidden in existing code using conventional software randomization techniques. To improve on the status quo of passive defenses, we explore potential active defenses utilizing booby traps.

We then propose, implement, and evaluate a novel system which prevents information disclosure attacks against diversity with hardware enforcement in current commodity processors. In our system, Readactor++, we combine diversity, execute-only memory, and booby traps into a comprehensive defense resilient to all currently proposed forms of code reuse. In particular, Readactor++ is the first defense to protect against direct code disclosure (reading program code) as well as indirect disclosure (disclosing code locations by reading pointers to code). In our evaluation we show that our techniques are applicable to real-world software, such as complex, modern web browsers.

Finally, we extend the reach of automated software diversity to combat side-channel attacks. Conventional software diversity research has focused on randomizing static aspects of programs, such as machine code layout. Since side-channel attacks observe dynamic runtime side-effects of software, these techniques are not directly applicable to randomize side-channel observations. Thus, we introduce a new technique, control-flow diversity, which allows compile- or load-time randomization to randomize dynamic runtime side-effects with low overhead. We evaluate the effectiveness and performance overhead of control-flow diversity against a representative cache-based side-channel attack.

Chapter 1

Background

> *If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.*
>
> — Sun Tzu, *The Art of War*

To put our work in context, we begin with a short introduction to the arbitrary code execution attacks. We then introduce the general ideas behind side-channel attacks. In section 4.2 we give further background on the technical details of the specific side-channel attacks that we evaluate control-flow diversity against.

1.1 Unsafe Languages and Memory Corruption

A significant portion of currently deployed software is written in low-level, fast, and unsafe systems languages such as C and C++. All major web-browsers (e.g., Chrome, Firefox, Internet Exporer, Safari), operating system kernels (e.g., BSD variants, Linux, OS X, Windows), many

important desktop applications (e.g., Adobe Reader, Microsoft Office), web servers (e.g., Apache, Nginx), and interpreted language runtimes (e.g., Java, PHP, Python) are written in C or C++. While we can attribute some of this popularity to historical accident and tradition, programs written in low-level, compiled languages tend to be fast and memory-efficient. By allowing programmers to manage their own memory allocation, systems languages avoid the overhead of automatic garbage collection usually required by memory-safe languages such as Java or Python. In addition, C and C++ do not enforce runtime checks to ensure safety, since these checks can significantly slow performance-critical programs [107, 108].

Execution of arbitrary machine code by an attacker is possible because these widely deployed, fast languages (e.g., C and C++) are not memory safe. This means that programming bugs can result in corruption of arbitrary memory locations. These languages do not enforce spatial memory safety, and therefore array indices and pointers can go out of bounds. In addition, languages with manual memory management are prone to temporal safety violations such as use-after-free bugs where the program attempts to use memory which has already been freed and reallocated. Szekeres et al. [145] provide a comprehensive classification of memory corruption vulnerabilities and defenses in the academic literature.

1.2 Co-evolution of Exploits and Defenses

Building on the fundamental technique of memory corruption, attackers have found methods to reliably corrupt program control flow and control the execution of the target program. This has resulted in a war of escalating complexity between attackers and defenders, as advanced defenses spawn even more advanced attacks. We give a brief account of the evolution of attacks and defenses here and cover related defenses in greater detail in chapter 5.

By exploiting a memory corruption vulnerability, an attacker can redirect the correct execution flow of the target application in a controlled manner. To then achieve arbitrary code execution, the attacker can either redirect control flow into injected malicious code or carefully guide the redirected execution flow through existing code, which is termed a code-reuse attack. We discuss all known variants of both attacks below.

1.2.1 Code Injection

Arbitrary code execution attacks originally utilized injection of malicious, executable code. In code injection attacks, the attacker injects machine code into the program's memory and diverts control flow into the injected code. The injected code payload is usually called "shellcode," because the traditional goal of code execution is to gain remote access to a shell on the target. The attacker injects shellcode by crafting program input containing malicious code which the program then stores in (executable) data memory. After injecting shellcode into the target program's address space, the attacker must redirect the program instruction pointer into this payload.

Overflowing a buffer on the stack, colloquially known as "smashing the stack," is a conventional way to both inject shellcode on the stack and overwrite a code pointer to hijack control into the injected shellcode. By overflowing a stack buffer, the attacker can arbitrarily rewrite the stack contents with a NOP sled followed by shellcode and overwrite the next return address with the approximate address of the injected buffer. Exploitation of buffer overflows dates back to at least the Morris worm of 1998 [113], and but the technique was first widely disseminated in the classic Phrack article "Smashing the Stack for Fun and Profit" [5].

In response to the threat of memory corruption, and specifically stack smashing, researchers first proposed two major lines of defenses: preventing buffer overflows from corrupting return addresses and disallowing execution of writable memory areas. Both protections are now

deployed, rendering stack smashing attacks ineffective when these protections are properly used.

To mitigate control-flow hijacking via stack buffer overflows, Cowan et al. proposed the use of "stack canaries" to detect corruption of stack memory [49]. A stack canary is a special value inserted on the stack between potentially vulnerable buffers and the stored return address. When returning from a function, the program first checks to ensure that this canary value has not been modified before using the return address to return program execution back to the caller. This protection is now widely deployed in commercial and open-source compilers, and while not perfect [122, 154], forms a first line of defense against stack-based buffer overflows.

Another defensive approach is to prevent the injection of executable code into an application. To prevent execution of writable memory, compiler toolchains can mark pages containing data, such as the stack, as non-executable (NX). While marking pages as NX is not strictly compatible with the uniform view of memory in the Von Neumann architecture, in practice, compilers do not create programs which need to execute writable data memory. To also protect program code from modification, we can mark memory pages containing code as non-writable. With the threat of code injection, disallowing self-modifying code seems a small price to pay to enforce protection against malicious code modification.

Since NX memory was not originally supported on commodity hardware, e.g., the Intel x86 architecture, the PaX team pioneered a Linux kernel patch to enforce a writable XOR executable (W⊕X) memory policy in software. After commodity x86 processors included the capability to disable execution on a per-page basis, all major operating systems adopted NX protection for the stack and other writable memory.

After canaries were deployed to mitigate stack buffer overflows, offensive researchers found many other useful ways to corrupt memory and launch attacks, e.g., heap overflows and use-after-free bugs. Of particular note, vtable hijacking [124] is an effective and useful method

to gain control of the execution flow. In a vtable hijacking attack, the attacker overwrites a vtable pointer in heap object and triggers the execution of a virtual method using that object. The vtable pointer points to a table of virtual methods for an object's class, and by controlling this pointer the attacker can arbitrarily redirect control flow. We discuss this attack further in section 3.2.4.

1.2.2 Code Reuse

With the widespread adoption of W⊕X memory, direct code injection is no longer a generally viable avenue of exploitation (although applications that generate new executable code at runtime may still be vulnerable). However, injecting machine code is not the only way to achieve arbitrary computation. By carefully controlling the control flow of the target program, an attacker can reuse existing, write-protected code for nefarious purposes. This type of attack, known as a code-reuse attack, has proven exceptionally resilient as researchers have repeatedly extended the approach to bypass proposed defenses.

Return-into-lib(c). Code-reuse attacks began with the return-into-lib(c) attack [140]. This attack redirects program control flow to execute legitimate library functions, typically from the C standard library (`libc`). By stringing together functions and controlling the inputs to each, the attacker can perform arbitrary, Turing-complete computation on the target [147].

Attackers can control execution flow between reused functions by corrupting the targets of indirect control-flow transfers (also known as free branches), such as return instructions. The attacker rewrites the stack so that each return address points to a function to reuse, rather than a normal return address. When each function in the attack finishes and returns, it transfers control to the next function to reuse in the attack, and thus the attacker completely controls the execution flow of the attack.

Return-Oriented Programming. In 2007, Shacham introduced a new, Turing-complete class of code-reuse attacks called *return-oriented programming (ROP)* where the attacker reuses small sections of code called *gadgets*, rather than whole functions [135, 125]. Gadgets are short sequences of existing code in an application or library that can be reused as building blocks in an attack. In the basic ROP technique, gadgets always end with a return instruction to allow the attacker to control execution flow by rewriting return addresses, as in the return-into-lib(c) attack. Later research showed that the gadget terminator constraint can be relaxed to any free branch (a return, indirect jump, or indirect call instruction), since these free branches allow the attacker to maintain control of the execution flow [31, 24].

Software Diversity. In traditional code-reuse attacks such as ROP, the attacker needs to know the addresses of all gadgets or functions he wishes to reuse *a priori* in order to construct a successful attack. To deprive the attacker of this knowledge, randomization-based defenses have been proposed to randomize the in-memory layout of code. The use of randomized code in software defenses was first suggested by Cohen in 1993 as a method to increase attacker workload [43]. Since then, researchers and industry practitioners alike have proposed and implemented many schemes to automatically randomize program code at granularities ranging from the whole program down to the level of individual instructions.

The most widely adopted randomization-based defense, *Address Space Layout Randomization (ASLR)*, was originally proposed by the PaX team in 2001 for Linux and has since been implemented for all major operating systems. Its usage is now standard for shared libraries on Linux and all executable code on Windows. ASLR randomizes the starting virtual memory address of each loaded segment during the loading process so that an attacker cannot easily redirect control flow into code segments normally located at a fixed, known address. While ASLR certainly raises the bar for attacks when properly implemented and applied to all executable code, it is still vulnerable to bypasses [136, 61, 82]. ASLR only randomizes the base address of each segment

and does not disturb intra-segment distances. Thus, disclosure of a single code address allows the attacker to shift all gadget addresses in the attack to adjust for the randomized base offset, allowing the attack to bypass ASLR. Attacks in the wild already use this relocation technique to bypass deployed ASLR defenses [35].

Just-in-Time ROP. To ensure resilience against trivial single-pointer disclosure attacks, researchers have proposed many code randomization schemes which operate at finer granularities than ASLR [43, 63, 39, 18, 91, 14, 20, 92, 144, 86, 118, 76, 70, 153, 85, 78, 57, 77]. Larsen et al. systematize these defenses [100] and this dissertation discusses diversity-based defenses in more detail in chapter 5. Most proposed code randomization defenses assume that the full run-time code layout is unavailable to an attacker. Thus, these defenses do not tolerate an attacker capable of disclosing arbitrary portions of the program address space. Snow et al. demonstrated the JIT-ROP attack [139] which reads the randomized code at run time in a browser context. Using a run-time, client-side scripting language such as JavaScript, the JIT-ROP attack exploits an arbitrary buffer over-read primitive [143] to read code pages, disassemble their contents, and recursively follow references to other code pages in order to disclose the entire application contents without crashing. This allows the attacker to read the program code after any randomization has occurred and then construct a ROP attack on-the-fly. In light of this advanced attack model, it is clear that any code-reuse defense for complex client-side applications should tolerate information disclosure. We present a diversity-based defense secure against information disclosure in chapter 3.

Blind ROP. Another technique to bypass diversity-based defenses is brute-force guessing. Brute-force attacks have been shown to effectively bypass weak implementations of ASLR [136]. More recently, Bittau et al. demonstrated Blind ROP, a more powerful brute-force attack targeting defenses with arbitrary randomization granularity [22]. The Blind ROP attack specifically exploits services which spawn processes to handle attacker-controlled requests and which

respawn any crashed processes. Since the entire service will not go down if a response handler crashes, the attacker can arbitrarily probe memory without fear of a segmentation violation or other program crash. In Unix systems, such as Linux, services often respawn identical processes by forking, without a redundant `exec()` call for increased performance. Since `fork()` does not re-invoke the loader, diversity-based defenses such as ASLR do not re-randomize the program, so information about the code layout gathered by the attacker's random probes applies to all spawned processes. The Blind ROP attack model highlights that defenders should carefully handle program faults so that an attacker cannot repeatedly gather information to de-randomize the code layout. In chapter 2 we discuss a novel defensive paradigm which probabilistically detects and prevents brute-force attacks on randomization.

Side-Channel Attacks against Diversity. Side-channel attacks are another potential vector targeting software diversity. If the attacker cannot directly read and disclose the code layout, as the JIT-ROP attack does, a side-channel may allow disclosure of enough information to de-randomize the program. Seibert et al. organized and evaluated potential side-channel attacks against fine-grained code randomization [134]. While this research direction is still relatively unexplored, it is a promising technique to attack diversity defenses when direct and indirect disclosure of the code layout is impossible. We discuss the potential impact and practicality of this new attack paradigm further in section 3.7.1.

Control-Flow Integrity. In parallel to research on software diversity, defensive researchers have introduced and refined *control-flow integrity (CFI)* [2, 3, 1] which aims to prevent the initial control-flow hijacking step in code-reuse attacks. The initial trend in proposed CFI research was to trade off granularity of control-flow checks for increased performance (see chapter 5 for more detailed discussion of CFI defenses). However, recent attacks [29, 56, 71] have demonstrated that course-grained CFI schemes are still vulnerable to control-flow hijacking and code reuse. Thus, a secure CFI scheme must constrain each function call to the minimal set of possible

targets and properly constrain return instructions to allow only legitimate callers. In an ideal CFI system, this means that the defense should maintain a shadow stack, since a function's run-time caller cannot always be statically determined. Additionally, to tolerate attacks which can disclose and alter all program memory (e.g., JIT-ROP [139] and Isomeron [55]), this shadow stack should be hidden and inaccessible. Finally, the recent COOP attack [130] demonstrates that attackers can misuse the C++ virtual dispatch mechanism to execute a whole-function reuse attack using only virtual methods invoked at legitimate virtual call sites. Thus, secure CFI schemes should also be aware of C++ semantics and the full class hierarchy, which is generally only available at compile time. Overall, CFI defenses are promising, and commercial compilers are starting to include optional CFI enforcement. However, CFI implementers must be careful to avoid advanced attack vectors. In this dissertation we instead focus on randomization-based defense which is a complementary line of research to CFI.

1.3 Side-Channel Attacks

Software diversity has been primarily applied to mitigate direct attacks such as code reuse, code injection, and even reverse engineering. In this dissertation we explore extending automated software diversity to a new category of attacks: side-channel attacks (see chapter 4). In a side-channel attack the attacker observes dynamic side-effects of a program to derive secret information. As described previously, this secret may be the layout of randomized code. However, the target of side-channel attacks is more often the secret key in an encryption routine. We specifically focus on side-channel attacks targeting the hardware side effects of encryption, such as the impact of the program on processor cache contents, since these side channels are so prominent and damaging. However, it is important to note that this is only an illustrative example for discussion and evaluation.

Kocher initially described a timing side-channel attack on public-key cryptography [96]. Timing side-channels measure the time a computation takes under varying inputs or external conditions, such as hardware usage. By observing how the time to encrypt a block varies when encrypting different ciphertexts, the attacker can derive information about the key. For example, assume the targeted cryptographic operation is measurably slower when a bit of the key and the plaintext are correlated. By observing execution times for random inputs and splitting these times into two sets based on the bit he is targeting, the attacker can guess the corresponding key bit with high likelihood. Timing attacks are the simplest type of side-channel attack, since they require no special observations besides accurate timing and are often feasible without proximity to the target [17, 27]. Other commonly targeted computational side effects are physical effects such as power usage [95] or acoustic emanations [68], hardware features such as branch prediction [4], cache usage [115], and occurrences of computational faults [21], to name a few.

In this dissertation we specifically focus on defending against attacks which indirectly observe data cache usage. Cache-based attacks on cryptographic routines were first analyzed by Page [115]. Researchers then applied these techniques to practical symmetric ciphers, needing only co-location on the same machine [149, 114]. By controlling the contents of cache sets in the data cache, the attacker can influence the side effects of the target computation (e.g., a symmetric cipher encryption routine). Since the timing of the encryption operation depends on whether needed data values are loaded into the cache already, evicting data from the cache allows the attacker to observe whether this data is reloaded during a given encryption. Conversely, the attacker can observe whether the encryption routine evicts attacker-loaded data from the cache, deriving the cache footprint of the encryption. By analyzing many observations of the cache footprint of the cryptographic operation along with the corresponding inputs, the attacker can derive the secret key (see section 4.2.2 and Tromer et al. [148] for details of these attacks).

Chapter 2

Booby Traps

Hold out baits to entice the enemy. Feign disorder, and crush him.
— Sun Tzu, *The Art of War*

2.1 Motivation

The status quo in cyber security puts defenders at a disadvantage over attackers: the attacker only has to find one way in, while the defender needs to guard multiple points of exposure, some of which may be unknown. Even a defender with vastly superior resources is still at risk from unknown bugs [9]. The current context in which we use software greatly amplifies this asymmetry. Two key factors multiply the attacker's advantage:

1. *Software Monoculture*: Hundreds of millions of computers run identical versions of popular software, such as Microsoft Windows, Acrobat Reader, Adobe Flash, and modern web browsers. This monoculture favors attackers disproportionately, since the exact same attack vector will be effective on large numbers of targets. Moreover, since the attacker

can easily replicate the environment which he or she will eventually execute the attack against, the attacker can develop, debug, and test the attack before releasing it into the wild.

2. *Passive Defense Strategies and Tactics*: Aside from network-based defenses, the defenses deployed on the computers that are the actual targets of attacks are mainly *passive*. We define *passive* defenses as anything which makes an attack harder to accomplish but does not react automatically during an attack. This includes traditional defenses such as firewalls, hardened secure software, and scans for resident malware after infection.

Automated software diversity offers a solution to the first of these problems. By randomizing attack surface of an application, software diversity forces attackers to expend the additional effort of adapting their attack to a specific target. With sufficiently robust defenses, this adaptation may not even be possible for some types of attacks (see chapter 3). However, due to their passive nature and reliance on confidentiality, randomization-based defenses are generally susceptible to brute-forcing attacks such as the bypass of weak 32-bit ASLR implementations [136] and the Blind ROP attack [22].

The second factor favoring the attacker, the passive nature of current defense mechanisms on target computers, has received much less attention so far. In contrast to *passive* defenses, we define *active* defenses as countermeasures triggered as a direct reaction to an attack. Intrusion prevention systems (IPS) which together with a firewall actively block network connections when the IPS detects an ongoing attack is a good example of such *active* defenses. Part of the reason for the current lack of active defenses is that active monitoring solutions consume resources. Frequently, users are unwilling to deploy heavyweight schemes. Additionally, active defenses may have high false positive rates and detrimental effects for legitimate users, making system administrators even more reluctant to deploy them.

This chapter describes a new defense paradigm that address both factors. Rather than just trying to build stronger defenses, we suggest also introducing active responses to discourage the attacker. Our technique protects software without human intervention using either compile-time or load-time rewriting to add active, diversity-based defenses with no run-time overhead. In particular, we focus on extending the compiler or loader to weave dormant booby traps into generated binaries. Among other capabilities, these active defenses can discourage brute-force attacks by costing the attacker time or other resources. Since the defenses that our system weaves into the target code are not executed during normal operation, the protected programs run at full speed. We use the term *arming* a program to refer to this process of automatically adding these active defenses to strengthen a program.

We have identified one such active defense which we call the *booby trap*. We define booby traps as code providing active defense that is only triggered by an attack. These booby traps do not implement program functionality and do not influence its operation—in fact, the program does not know about its own booby traps and under normal operation cannot trigger them. We propose to automatically insert booby traps into the original program code during compilation or program loading. Whenever an attack triggers one of the booby traps within the program, the trap instantly knows that the software is under attack, and is in a position to adequately react to the threat (see Figure figure 2.1). For example, a booby trap might perform advanced forensics to identify an attack in real-time to facilitate a deceptive response. In section 2.3 we describe further research directions and new opportunities to engage attackers.

The use and effect of cybersecurity booby traps bears striking resemblance to their real-world equivalents. First, a booby trap signals the position of the enemy to the defender. Second, it is in the interest of the defender to place booby traps in locations with a high likelihood of being triggered by the enemy. Third, booby traps affect the mindset of the enemy: they demoralize attackers, keep them stressed, make them cautious instead of aggressive, and slow down the enemy's movement. We anticipate that all of these effects hold true in the digital domain, too.

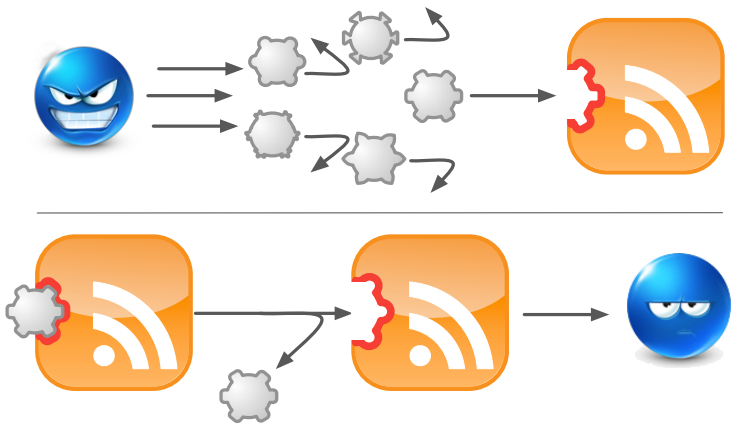

Figure 2.1: Software diversity forces the attacker to bombard targets to find a working exploit. Even if the attacker manages to find an exploit that sticks, active defenses now trigger a reaction in the target.

By arming a program, we fundamentally alter the terrain of cyber warfare, since active, direct response to intrusion gives defenders an equalizing advantage. Defenders need not constrain themselves to preparing for attacks or cleaning up after them, instead they can automatically and reliably respond during an attack. These responses are not triggered by heuristics and observation, as the few current active approaches are, but rather as a direct side-effect of the attack. This means that defenders can pursue more aggressive reactions, with the assurance that there really was an attempted attack. This ability to respond tips the scales of cyber war back towards the defender.

Summing up, our contributions are:

- We propose a new active cyber defense tool, *booby traps*, which will help to correct the current asymmetry in cyber warfare initiative. In section 2.2 we discuss a concrete example of how defenders could insert booby traps in the context of code-reuse attacks.
- We provide examples that demonstrate how cyber booby traps can actively defend software by enabling responses such as providing additional information to defenders, preventing

and recovering from attacks, and disincentivizing attackers. In section 2.3 we enumerate and discuss useful responses particularly suited to booby traps.

2.2 Booby Traps

As discussed previously, a booby trap provides an active defense triggered directly by an attack. Booby traps could be useful to defend at many different levels of the software stack. As an example in the web domain, booby traps could be inserted in web frameworks to protect against attacks which attempt to use functions or SQL statements which are not normally used by the framework. We could also insert booby traps into the operating system to trap syscalls or OS interaction which we guarantee the system will never use during normal operation.

However, we will focus on low-level booby traps as a running example in this chapter. By extending the idea of artificial software diversity, we can create booby traps for code-reuse attacks. Attack code which inadvertently lands in a booby trap will trigger active defenses, which we describe in section 2.3. The following sub-sections detail research questions we need to address before cyber booby traps enter the arsenal of defenders. Specifically, we describe what code-reuse booby traps are, and then determine when and where booby traps should be inserted for the greatest probability of catching an attack.

2.2.1 What to add?

As we discuss in section 1.2.2, researchers have presented many approaches to randomize program layouts. We assume that one of the proposed fine-grained diversity solutions is in place, and that consequently a realistic ROP attack will crash at some point when it tries to call a gadget which is broken or no longer in the expected position. However, by inserting booby trap hooks into the binary while diversifying, we propose to catch the attack and react

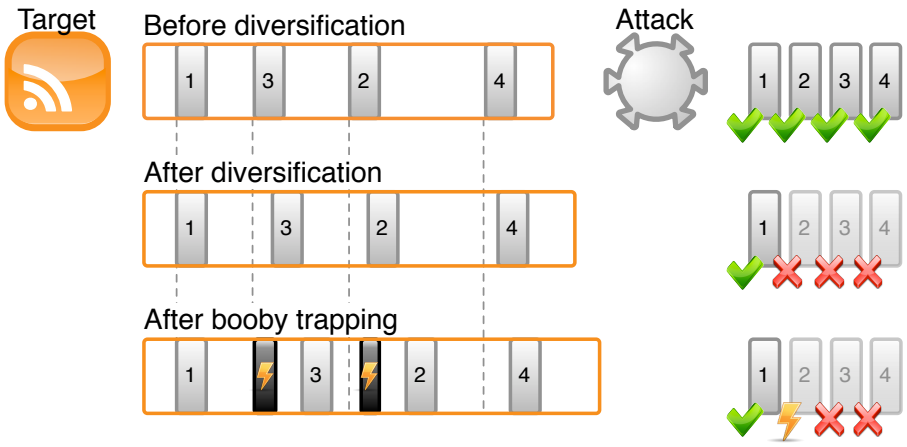

Figure 2.2: After diversification, the attack is disarmed since gadgets have moved from their expected locations. After booby traps are weaved into the binary, the attack is detected and the system reacts to the anomaly.

to it directly. When the attack redirects execution to what the attacker thought was a useful gadget, but is now a booby trap, the booby trap takes over execution and responds to the attack. By inserting booby traps where the attacker expects a gadget, or simply sprinkle traps throughout the binary so that any attack has some probability of triggering a trap instead of executing a gadget, we can reliably catch attacks while they occur. These booby traps will then pass execution to a handler which can actively respond to an attack in real-time and provide additional security to disincentivize the attacker from blindly attempting additional code-reuse attacks (see section 2.3 for discussion of possible responses).

Additionally, we must decide what these handlers actually do when triggered. What makes a suitable booby trap, and who supplies the booby trap code? Obviously, an attacker should not be able to use a booby trap to mount an attack, which means that we have to either find a restricted set of eligible instructions to be used for booby traps, or supply hardened booby trap code ourselves. Either way, we have to make sure that it does not compromise security. An interesting direction to explore here is to enable programmers to supply domain specific booby trap code, which is then automatically hardened and woven into the application code.

2.2.2 When to add?

After we determine how booby traps function, we must decide when and how to best insert them. Essentially, we think that there are two options to arm a program:

1. inserting code at compile-time,
2. inserting code at load-time, or

As is usually the case, there are several trade-offs involved in deciding which technique to use. Since each has its advantages and disadvantages, we shall now explore both techniques to offer a complete and applicable solution.

Compile-time Insertion. Arming at compile-time has the benefit that it happens ahead of time, i.e., we have greater flexibility to diversify and choose suitable booby traps to insert into a program. The compiler has full control of all code emitted and can place booby trap code at arbitrary locations. This process is far simpler and more practical than trying to rewrite existing binary code, which may not even be automatically disassemblable, but requires the availability of source code.

Compiler diversity can create a nearly infinite number of versions of software, so we can give each end-user a custom diversified and armed binary. However, this affects the release and distribution of programs, as publishers cannot realistically ship software on a physical medium anymore. Fortunately with the growing popularity of digital software distribution, we do not need to worry about this difficulty. A previous work by Franz [65] describes this paradigm shift in more detail.

Load-time Insertion. If we diversify and arm a program at load-time, we could still ship a single binary to the users. Inserting code at load-time could provide even higher security guarantees, as not even an informed insider would be able determine what the program layout is without access after it is loaded. Further, it would extend our techniques to programs whose source code is not readily available. In this case, however, we would also have to properly handle all indirect-branches and make sure that they get properly "redirected" to the intended branch target. One way to optimize this detour away would be to add a table listing all indirect branch targets when compiling the program. Research in the area of load-time diversity by Wartell et al [153] may be a starting point which booby trap arming at load-time could be based upon.

So, we see that there is no clear best approach, but rather varying points on the same spectrum. Furthermore, diversifying/arming at both times, i.e., at compile-time and load-time, results in a system offering higher security in face of potentially compromised parties: compile-time arming protects against a compromised or disabled load-time diversifier, and load-time arming protects against a compromised code producer or use of an insecure transmission medium.

2.2.3 Where to add?

If we choose to insert booby traps either at compile-time or load-time, we need to address the question of *where* we actually insert our trap code. Guiding this decision is the goal of inserting in locations an attacker will execute, but where normal execution will never trigger the trap.

The primary factor in placing booby traps is the type and target of the randomization approach they are coupled with. We need to ensure that we place booby traps where the given diversity scheme will make their location unpredictable. In most proposed fine-grained diversity systems, this means that we should insert booby traps directly into code, as the machine code layout and contents are randomized. When randomizing tables of code pointers, as in chapter 3, we can

insert booby trap entries into these tables. Other diversification systems will require analogous adjustment of booby trap implementation and placement.

We expect that finer grained insertion will result in stronger security, although at the cost of slightly slower performance. For example, inserting a `NOP` slide ending in a booby trap within an existing basic-block is going to increase its size, decreasing instruction cache utilization and thus increase instruction cache misses. Whenever we insert a booby trap into a basic block, we need to make sure that the program's control flow does not reach it. This can be done by breaking the basic block in sub-blocks and jumping past the booby trap. Using this technique, we expect to reduce performance penalties. We expect that this fine-grained level of insertion does positively affect security, as we can insert booby traps in all places that are attractive to attackers.

To insert booby traps in positions which an attacker will likely execute, we can find all possible ROP gadget locations and insert traps at these locations, displacing the existing gadgets. However, this assumes a known attack model, which is generally not the case. To properly combat new and evolving threats, we recommend inserting traps in random locations throughout the binary so that any code-reuse attack will have some probability of triggering a trap.

2.3 Active Responses

Since attacks directly trigger booby traps, which then run inside the program space under attack, booby trap handlers are in a unique position to react to attacks. A booby trap is the fastest possible reaction to an attack, since it runs during the attack itself. Booby traps also have full access to the program and attempted exploit, which allows them to modify the execution of both the program and attack. Finally, booby traps are an integral part of the application code itself, and as such cannot be easily disabled by an attacker, especially if the operating system

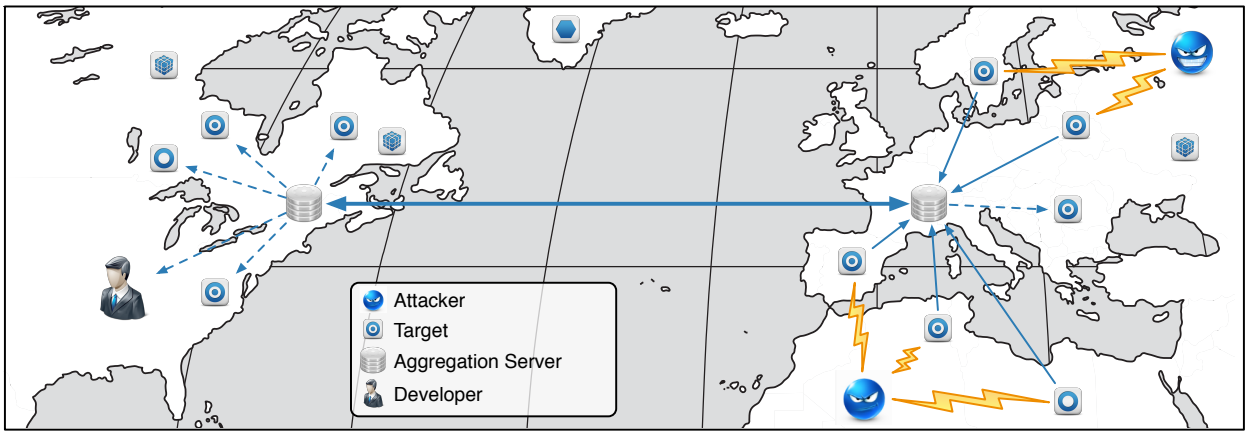

Figure 2.3: Cooperative Situational Awareness — When attacked, armed binaries report the attack to aggregation servers that monitor the attacks in real-time. Defenders can use the early warning capabilities to alert developers and cause potentially vulnerable systems to take preventive action ahead of the attack.

verifies that binaries are properly signed. Based on this unique position, we think that there are many interesting ways to use booby trapping to increase the cost of attacks.

When considering responses, we must first determine how much information we will allow to the attacker about our response. Stealthy responses allow defenders to maintain an advantage by responding as the attacker expects and therefore not leaking knowledge that the attack occurred. We can engineer many responses to either be stealthy or not, although a few that we suggest below will by nature betray their existence.

Recovery

Since we have access to the running program in the booby trap, we can attempt to recover from the attack, rather than simply crash. For certain high-availability services, it is more desirable that the program stay running, even if the system cannot guarantee program correctness. A recovery handler could use standard software fault tolerance techniques such as checkpointing, or more domain specific techniques such as inferring what data was corrupted in the attack and

synthesizing replacement data. Additionally, we can use the booby trap to re-activate programs, memory protection flags, and other security mechanisms that the attack disabled by reverse engineering the attack payload.

Recovery will generally leak information back to the attacker, although we are unsure how attackers might interpret this information. Depending on when the attack triggers the trap, we may recover before the attacker actually knows that the attack has partially succeeded. However, this recovery should not alter correct execution of the software, lest the attacker be able to observe unexpected side-effects.

Version Flux

Take a snapshot of the current program state and re-start another program instance with this state. This is particularly effective if the new program is a program containing a different set of diversifications and/or booby traps. There are several compelling applications of mixing program diversification with execution, for example, we could supply a custom program loader which chooses one program out of many diversified/armed ones at random. In such a situation, even insider knowledge might not be sufficient to sabotage the computing infrastructure, because the attacker cannot deduce which of the variations is currently running, or will be running on the next launch.

Version flux naturally leaks information to the attacker, since a new version of the software ideally behaves differently in response to the same attack. However, we can introduce noise into this leak by switching software versions randomly without detecting an attack.

Honeypots

Honeypots are network machines specifically configured to be vulnerable to attack in order to gather information about new exploits and tactics. Booby traps are ideal for this purpose, since they can provide detailed information about attacks, especially the initial exploitation vector. By using advanced forensics capabilities (see section 2.3) armed binaries can report exactly what the attack is doing in real-time without requiring further execution in a virtual machine. We believe that armed binaries will be a very useful new tool to existing honeypot research such as the Honeynet project [142].

Honeypots are ideal for implementing stealthy booby traps. We expect that hiding most responses from the attacker will require the use of some sort of a honeypot to simulate the attack completing successfully.

Closely related to the concept of honeypot environments, decoys such as honeywords (decoy passwords) [90] and honeyfiles (decoy documents) [25] are additional examples of defenses designed to trick attackers. These decoys are effectively booby traps at a higher abstraction, and therefore our booby trap approach fits well into the framework of decoy objects as a lower level defense.

Feedback Directed Instrumentation

Until now, we have only talked about the technicalities of inserting active defenses, i.e., how, where, and what. Doing this, however, at all places that could be a potential attack is a conservative approach. If we assume a software monoculture, as we have today, where all binaries of a single version of an application are identical, we can insert booby traps where known gadgets would be located in the "expected" binary. While this increases security if applied for only a minority of users, attackers will not have an "expected" binary if these techniques are widely

utilized. In this situation, we can insert booby traps of various sizes randomly into the code, and probabilistically catch an attack targeting some particular binary (assuming we protect the binary itself from exfiltration or diversify at load-time).

Whenever we discover classes of new attacks, we can add a dedicated analysis phase and make sure that targeted locations in the "expected" binary are prepared to trigger booby traps in our version. Arming programs in this way allows us to make better guesses as to where to actually add booby traps and where a simple `NOP` pad suffices. It also allows us to guarantee that no gadgets survive (remain usable) between any two programs in a population of diversified binaries. Finally, by also using feedback obtained from a performance profiler, we can also avoid instrumenting the most frequently executed code as demonstrated by Homescu et al. [78].

Enhanced Forensics

Complimentary to improved accuracy, which is possible by using adaptive techniques as the previous paragraph describes, we can use booby traps to trigger a host of forensic analyses. The purpose of such analysis is to determine the origin and signature of the attack. Since it is difficult to determine if a program crash is caused by an attack or simply a coding error, triggering forensics in the early stages of an attack improves our opportunities to distinguish the two cases.

Since attacks trigger booby traps while they are still running we can deeply inspect the memory of a running attack. For example, return-oriented programming uses the return-stack of the native machine to manage the control of the attack, similar to an instruction pointer. When the ROP-attack triggers a booby trap, we can walk the stack of the program in reverse order and use a *gadget look-up table* to find which words hold data, and which (most likely) point to gadgets. Whenever we have looked at enough data points, it is likely that we have "recovered" the original stack that the attacker prepared. At this point, we can actually run analysis on the

attack, and use well-known techniques, such as signatures, to identify how the actual attack works.

While some of this is possible with conventional forensics techniques, triggering these techniques from a booby trap lets us catch an attack in progress, and thus allows us to inspect the initial attack vector. Using booby traps we can do this without costly deep packet inspection and storage to replay attacks after-the-fact. Since booby traps trigger in real-time, we get immediate feedback on techniques and intentions of the initial attack vector. We can use these features to our advantage in the following way: First, we could use forensics analysis results, possibly based on the signatures to find out the entry point and possible goals of the attack. Next, we can potentially deceive the attacker with incorrect knowledge in real-time, or move the whole attack to a safe environment, e.g., to a virtual machine, while it is still running. Finally, we can drive or validate the feedback directed insertion process using forensics information to prevent future similar attacks.

We think that deception based on advanced forensics is a promising application for booby traps, since this allows defenders to respond stealthily to the attack. Cohen has already done significant work in the field of deception, and has shown that deception is an extremely powerful technique [42]. Using booby traps to trigger deceptive campaigns can especially provide low-level deception that the attack succeeded when in fact it did not. Real-time forensics provides the details of the attack, allowing defenders to possibly simulate the attack in a secure environment. Defenders then have full control over the attack and can construct a simulated environment to deceive the attacker.

Counterattack

Although counterattacking (colloquially termed “hacking back”) is rife with dangers, pitfalls, and questions of legality, we believe that booby traps might be a good point to launch counterattacks

from. Previous literature has discussed the legality and morality of counterattacks [89, 59], and we defer this discussion to more qualified experts on the subject. Instead we simply address the role that booby traps might play in facilitating counterattacks.

Booby traps could serve as an automatic start to the counterattack process. As an automated response, booby traps could quickly engage the attacker before he or she even realizes that the attack has failed. This might allow the counterattack to catch the attacker unaware and before he or she has a chance to retreat and abandon the source of the attack. However, since counterattacking is risky, any automated response would need to use the forensics techniques we touched upon earlier to verify the trigger was a real attack rather than a software bug. In addition, counterattack, or even counter-intelligence, betrays that the booby trap detected the attack, which is a large disadvantage. We imagine that booby traps might serve as only a starting point in a counterattack campaign, perhaps providing initial port scan or automated vulnerability scan results to a human operator who would then continue the campaign. However, automated attack toolkits such as Metasploit may be sufficient for an automated counterattack to gain access to the offending system. While counterattacking is still under debate, there may be contexts, such as internal use in an organization, where it is a valid tactic, and booby traps could provide the perfect starting point to trigger such counterattacks.

During the workshop we also discussed the possibility of internal cooperation for counter-intelligence. If both the victim and attacking machines are in the same administrative domain, booby traps could trigger a mechanism installed on all internal machines, giving direct access to the attacking machine for forensics. This would allow the booby trap at the very least to instantly follow the attack trail until it left the organization.

Cooperative Situational Awareness

In the battlefield, access to relevant information regarding the situational awareness plays a crucial role. Consequently, military leaders depend on networked and interconnected systems to use these information to their tactical advantage. We think that duplicating this approach in the digital domain results in a similar competitive edge. Binaries could be armed so that triggering a booby trap causes the target binary to broadcast information about the ongoing attack to other interested parties. For instance, cloud computing resources could aggregate attack reports and thus let defenders track and locate attackers in real time (see figure 2.3.) Probabilistic and machine learning techniques let us identify causal events and detect denial of service and decoy attacks in such a cooperative situational awareness system.

We think that defenders will benefit in multiple ways. First, this system can further automate the task of tracking and locking down malicious IP traffic and identifying the ISPs hosting the attacker-controlled servers. We can notify local law enforcement much earlier relative to current practice and simultaneously collect more evidence, too. Second, the triggering of booby traps in a single armed binary could provide an early warning to all other armed binaries created from the same input program. This means, for example that we can start *preventive re-diversification* of the binary started even *before* the attack reaches the hosting computer. Finally, the alerted hosts can take steps to confuse the attackers by sending them decoy information to increase the likelihood of their apprehension.

2.4 Conclusion

Active defenses are a critical part of a strong defense strategy, and we believe that booby traps are a great tool to provide active response. As attackers become more powerful and persistent, it is critical that we find new ways to handle these attacks. Sitting idly by or monitoring

for compromises while attackers bombard critical systems is no longer sufficient to fight off adversaries. However, with booby traps, defenders can respond immediately to attacks with intimate knowledge of the exploit itself and the vulnerable software.

By triggering from the attack itself, booby traps give defenders a variety of unique responses. Booby traps have the potential to allow software to recover from attacks, rather than crashing immediately. They also provide a platform for automatic and accurate forensics inside an attack. Since booby traps trigger in real-time, defenders could even automatically launch attribution or deception responses, while the attack is in progress and an open connection is still available. We are just beginning to scratch the surface of what might be possible with real-time, active responses triggered by attacks directly, but we believe that booby traps are a powerful weapon to add to the defense arsenal. We look forward to the development of new types of booby traps, especially to combat higher-level vulnerabilities, such as are prevalent in web applications.

Automatically arming software by inserting cyber booby traps will have a lasting effect on the field of cybersecurity. Booby traps are the sentries guarding existing defensive walls such as diversity. With these sentries, software armed with booby traps actively protects itself by handling attacks as they occur, allowing defenders to raise the cost and risk of attempted attacks. Using active response to raise the bar for attacks tips the balance of power in cyber war towards the side of the defender. By booby trapping software defenders can now take the initiative and fight back against incoming attacks.

Chapter 3

Readactor++

The ultimate in disposing one's troops is to be without ascertainable shape. Then the most penetrating spies cannot pry in nor can the wise lay plans against you.

— Sun Tzu, *The Art of War*

3.1 Motivation

Design and implementation of practical and resilient defenses against code-reuse attacks is challenging, and many defenses have been proposed over the last few years. So far, these defense mechanisms can roughly be classified into two primary categories: control-flow integrity (CFI) and code randomization. CFI, when properly implemented [1], prevents attackers from executing control-flow edges outside a static control-flow graph (CFG) [3]. However, fine-grained CFI solutions suffer from performance problems and the precision of the CFI policy is only as good as that of the underlying CFG. Obtaining a completely precise CFG is generally not possible, even with source code. Recent work on control-flow integrity has therefore focused on coarse-grained solutions that trade security for performance [119, 38, 66, 158, 157].

Unfortunately, all of these solutions have been successfully bypassed due to their imprecise CFI policies [71, 56, 72, 131, 29].

Code randomization (see [100] for an overview), on the other hand, has suffered a blow from information disclosure, which breaks the fundamental memory secrecy assumption of randomization, namely, that the code layout of a running program is unknown to attackers [143]. We distinguish between two types of memory disclosure: *direct* and *indirect*. In a direct memory disclosure attack, the adversary reads code pages directly and mounts a return-oriented programming (ROP) attack based on the leakage of code pointers embedded in instructions residing on code pages, as shown in the just-in-time code-reuse (JIT-ROP) attack [139]. In an indirect memory disclosure attack, the adversary reads multiple code pointers that are located on data pages (e.g., stack and heap) to infer the memory layout of an application (as we show in an experiment in section 3.2.1).

Since randomization is known to be efficient [78, 57], recently proposed defenses [12, 11] focus on reducing or eliminating memory disclosure. For instance, Oxymoron [12] aims at hiding direct code and data references in instructions, whereas Execute-no-Read (XnR) marks all memory pages (except a sliding window) as non-accessible to prevent memory pages from being dynamically read and disassembled [11]. However, information disclosure is surprisingly hard to prevent. As we explain in section 3.2.1, none of these techniques provide sufficient protection against memory disclosure and can be bypassed. They are also not sufficiently practical to protect complex applications such as web browsers that contain just-in-time compilers. Finally, we note that Szekeres et al. [145] propose a different approach called Code-Pointer Integrity (CPI) which separates code pointers from non-control data. Kuznetsov et al. [99] implement CPI by placing all code pointers in a secure region which (in 64-bit mode) is hidden by randomizing its offset in the virtual address space. However, Evans et al. [62] successfully bypass this CPI implementation using side-channel attacks enabled by the large size of the secure region.

Both insufficiently fine-grained CFI approaches and existing diversity-based defenses are vulnerable to whole-function reuse attacks, such as the traditional return-into-lib(c) attack [140]. More recently, Schuster et al. [130] described a new type of whole-function reuse called *counterfeit object-oriented programming* (COOP), which reuses legitimate functions from legitimate call sites by abusing the C++ virtual dispatch mechanism. Targets of C++ virtual function calls and calls to dynamically linked functions are looked up at runtime. More precisely, these are *dynamically-bound* functions. *Function-reuse attacks* exploiting dynamically-bound functions are unaffected by current randomization-based defenses because prior defenses do not introduce randomness into the dynamic dispatch mechanisms found in system software. Additionally, the COOP attack bypasses all published binary-only CFI approaches, since these defenses cannot derive a sufficiently precise CFG for dynamically-bound virtual functions.

Goals and contributions. In this chapter, we focus on improving the security properties of code randomization. Our goal is to tackle the shortcomings of existing defenses by closing memory disclosure channels and defending against whole-function reuse while allowing a reasonable granularity of code randomization. We classify information disclosure attacks into direct and indirect memory leakage. We then present the design and implementation of Readactor++, the first practical fine-grained code randomization defense that resists both classes of memory disclosure attacks. Our defense combines novel compiler transformations with a hardware-based enforcement mechanism that prevents adversaries from reading any code. Specifically, we use virtualization support in current, commodity Intel processors to enforce execute-only pages [83]. This support allows us to avoid two important shortcomings of prior work [11, 69]: either requiring a sliding window of readable code or legacy hardware, respectively. Finally, we build upon the foundation of execute-only code disclosure protection to implement a defense against function reuse attacks such as COOP and RILC.

In summary, our primary contributions are:

- **Comprehensive ROP resilience.** Readactor++ defends against all existing code-reuse attacks: return-into-lib(c), conventional ROP [135], ROP without returns [31], dynamic ROP [139, 22], and COOP [130]. We designed Readactor++ under the challenging and realistic assumption that an adversary can read and write arbitrary memory. Thus, Readactor++ improves the state of the art in JIT-ROP defenses by preventing indirect memory disclosure through code-pointer hiding.

- **Novel techniques.** We introduce compiler transformations that extend execute-only memory to protect against the new class of indirect information disclosure. We also present a new implementation of execute-only memory that leverages hardware-accelerated memory protections. Additionally, we demonstrate a new approach to randomize and defend dynamically-bound function tables.

- **Realistic and extensive evaluation.** We provide a full-fledged prototype implementation of Readactor++ that protects applications from code-reuse attacks, and present the results of a detailed performance and security evaluation. We report an average overhead of 6.0% on compute-intensive benchmarks. Moreover, our solution scales beyond benchmarks to programs as complex as Google's popular Chromium web browser.

3.2 Background

Readactor++ prevents memory disclosure attacks, which we discuss in detail in this section. In addition, we cover the technical details involved in dynamic function dispatch and related attacks to give context to the dynamic-binding table randomization component of Readactor++.

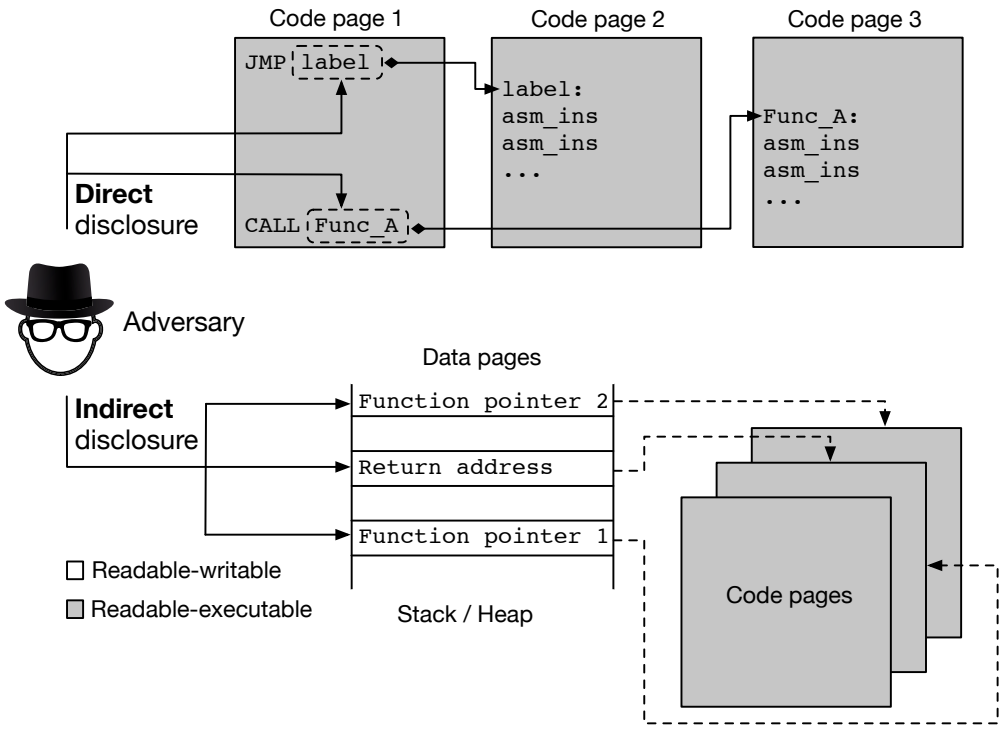

Figure 3.1: Direct vs. indirect memory disclosure.

3.2.1 The Threat of Memory Disclosure

As we briefly discuss in section 1.2.2, memory disclosure vulnerabilities allow an attacker to learn the memory layout and randomized locations of machine code in an application. Using this information, the adversary can reliably infer the runtime addresses of instruction sequences and bypass the underlying code randomization. We divide the space of disclosure attacks into direct and indirect memory disclosure attacks; figure 3.1 illustrates both classes of disclosure.

In a direct memory disclosure attack, the adversary is able to directly read code pointers from code pages. Such pointers are typically embedded in direct branch instructions such as direct jumps and calls. The top of figure 3.1 shows how the adversary can access a single code page (code page 1), dynamically disassemble it, and identify other code pages (pages 2 and 3) via direct call and jump instructions. By performing this recursive disassembly process on-the-fly,

the adversary can directly disclose all gadgets needed to relocate a ROP attack to match the diversified code [139].

Two protection methods have been proposed to prevent direct memory disclosure: rewriting inter-page references and redirecting attempts to read code pages. In the first approach, direct code references in calls and jumps between code pages are replaced by indirect branches to prevent the adversary from following these code pointers [12]. A conceptually simpler alternative is to prevent read access to code pages that are not currently executing [11], e.g., code page 2 and 3 in figure 3.1.

Unfortunately, obfuscating code pointers between pages does not prevent *indirect* memory disclosure attacks, where the adversary only harvests code pointers stored on the data pages of the application which are necessarily readable (e.g., the stack and heap). Examples of such pointers are return addresses and function pointers on the stack, and code pointers in C++ virtual method tables (vtables). We conducted experiments that indicate that the adversary can bypass countermeasures that only hide code pointers in direct calls and jumps. We collected code addresses from virtual table pointers on the heap, and disassembled code pages to identify useful gadgets, similar to the original JIT-ROP attack. We found 144 virtual function pointers pointing to 74 code pages in IE 8 and showed that it is possible to construct a JIT-ROP attack from those 74 code pages [55]. We call this updated attack *indirect JIT-ROP* to distinguish it from the original JIT-ROP attack that directly reads the code layout.

We must also consider whether preventing read access to code pages suffices to protect against indirect memory disclosure vulnerabilities. Since code pages are not readable, the adversary cannot disassemble code to construct an attack. The adversary still gains information from leaked code pointers, however, as our experiment on indirect JIT-ROP demonstrates. By leaking pointers to known code locations, the adversary is able to infer the contents of code surrounding the pointer targets. The severity of this threat depends on the type of code randomization that is deployed in conjunction with non-readable code pages. For example, if function permutation is

used, each leaked code pointer allows the adversary to correctly infer the location and the entire content of the function surrounding the leaked code address, since there is no randomization used within functions. Thus, the security of making code pages non-readable depends on the granularity of the code randomization.

3.2.2 Virtual Function Calls

Object-oriented languages such as C++ and Objective-C support polymorphism; the target of a function call depends on the type of the object that is used to make the call. C++ supports object orientation based on classes, which are simply user-defined data types with fields and functions (also called *methods*). Classes can be extended through inheritance of methods and fields. Whereas many modern languages limit inheritance to a single base class, C++ supports multiple inheritance.

The C++ compiler generates different call mechanisms for invoking an receiving object's method depending on whether the callee is a virtual function. A subclass can override functions marked as virtual in its base class(es). Non-virtual calls are simply compiled to a static, direct call to the corresponding function. For virtual function calls however, the exact callee depends on the type of the object that receives the call at runtime. Therefore, at virtual function call sites, the program first retrieves a pointer to a virtual function table (*vtable*) from the receiver object and then indexes into this table to retrieve a function pointer to the correct callee. As explained in section 3.2.4, adversaries abuse indirect calls to virtual function by corrupting objects to point to a vtable of their choice, thus controlling the call destination.

3.2.3 Dynamic Linking

Dynamic linking allows programmers to group functions and classes with related functionality into a single module. Programs can share dynamically linked modules, which simplifies maintenance and reduces code duplication both on disk and in memory.

Symbols are the basic building blocks of dynamic linking. Each module defines a set of symbols which it exposes to other modules through a symbol table. These symbols typically consist of exported functions and variables. A module can refer indirectly to symbols which are not defined within the module itself. Symbol addresses in the table are resolved at run time by the dynamic linker. Many binary file formats support *lazy binding*. With lazy binding, the addresses of external symbols are resolved when they are first used. Lazy binding can improve the startup time of some applications, which is why some binary formats enable it by default.

Unfortunately, the tables that support dynamic linking are an ideal target for attackers. Computed symbol addresses for instance are usually kept in tables in writable (and readable) memory. Furthermore, to support lazy binding, the meta-data necessary to resolve a specific symbol's address is also kept in readable memory at all times. These sources of information can be of use in an information leakage attack.

In ELF binaries, several specific dynamic linking tables are prone to information leakage. Attackers can collect function addresses from the Global Offset Table (GOT). The GOT stores the addresses of global and static elements of the program, including the addresses of all imported functions for use in the Procedure Linkage Table (PLT). The PLT on the other hand contains a set of trampolines that each correspond to a code pointer in the GOT. An attacker can infer the layout of both the GOT and the PLT tables from the relocation, symbol and string tables in the binary.

3.2.4 Attacking Dynamic Function Dispatch

Conventional ROP attacks are stack-oriented because they require the return instruction to read the next gadget address from the corrupted stack [125]. Since stack-based vulnerabilities are rare, attackers shifted to heap-oriented ROP. However, these attacks require a heap-based vulnerability along with a "stack pivot" sequence which sets the stack pointer to the start of the injected return-oriented payload on the heap. The former is typically achieved by vtable hijacking attacks [124]. As described above, a vtable holds pointers to virtual functions. While the function pointers are stored in read-only memory, the vtable pointer itself is stored in writable memory. Hence, an adversary can inject her own malicious vtable, overwrite the original vtable pointer, write the return-oriented payload, and wait until the next indirect call dereferences the vtable pointer to invoke a virtual function. The correct virtual function is not called, due to the vtable pointer overwrite, but instead the code referenced by the previously injected malicious vtable is executed. In many exploits, the first invoked sequence is the aforementioned stack pivot, which sets the stack pointer to the start of the injected return-oriented payload.

The more recent COOP code-reuse technique described by Schuster et al. [130] starts by hijacking the control flow, as in other code-reuse attacks. Instead of directing the control flow to a chain of ROP gadgets, COOP attacks invoke a sequence of virtual function calls on a set of counterfeit C++ objects. This carefully crafted exploit payload is injected in bulk into a memory region that the attacker can access. Notice that constructing this payload requires knowledge of the exact layout of the relevant objects and vtables.

Whereas ROP relies on return instructions to transfer control from one gadget to another, COOP uses a so-called *main loop* gadget (or *ML-G*), which is defined as follows: "A virtual function that iterates over a container [...] of pointers to C++ objects and invokes a virtual function on each of these objects" [130]. In addition, there are certain platform-specific requirements that a virtual function must meet in order to be usable as a ML-G in a COOP attack. For example,

on x86-64, registers that are, per calling convention, used for passing explicit arguments to C++ functions should not be overwritten within the ML-G's loop. The ML-G is the first virtual function that is executed in a COOP attack and its role is to dispatch to all other virtual functions (*vfgadgets* in COOP parlance) that make up the COOP attack. Similar to ROP gadgets, vfgadgets fall into different categories such as those that perform arithmetic and logic operations, read and write to memory, manipulate the stack and registers, etc. We refer to Schuster et al. for a full treatment of COOP [130].

3.3 Adversary Model and Assumptions

Throughout this chapter we will assume a powerful adversary model consistent with the most advanced attacks presented in the current literature [139, 55, 105, 130]. We build our defense assuming the following adversary capabilities and existing, complementary mitigations:

Adversarial Capabilities

- **System Configuration:** The adversary is aware of the applied defenses and has access to the source and non-randomized binary of the target application.
- **Vulnerability:** The target application suffers from a memory corruption vulnerability that allows the adversary to corrupt memory objects. We assume that the attacker can exploit this flaw to read from and write to any arbitrary memory addresses.
- **Scripting Environment:** The attacker may utilize a client-side scripting environment to process memory disclosure information at run time, adjust the attack payload, and subsequently launch a code-reuse attack.

Defensive Requirements

- **Writable ⊕ Executable Memory:** The target system ensures that memory can be either writable or executable, but not both. This prevents an attacker from injecting new code or modifying existing executable code.
- **JIT Protection:** We assume mitigations are in place to prevent code injection into the JIT code cache and prevent reuse of JIT compiled code [77, 51, 141]. These protections are orthogonal to Readactor++.
- **Brute-forcing Mitigation:** We require that the protected software does not automatically restart after hitting a booby trap which terminates execution. In the browser context, this may be accomplished by displaying a warning message to the user and closing the offending process.

We cannot rule out the existence of timing, cache, and fault side channels that can leak information about the code layout to attackers. Although information disclosure through side-channels is outside the scope of this particular work, we note that Readactor++ mitigates recent remote side-channel attacks against diversified code since they also involve direct memory disclosure [22, 134].

3.4 Design

3.4.1 Overview

Readactor++ protects against both direct and indirect disclosure (see section 3.2.1). To handle attacks based on direct disclosure, it leverages the virtualization capabilities in commodity x86

processors to map code pages with execute-only permissions at *all* times. Hence, in contrast to previous related work [11], the adversary cannot read or disassemble a code page at *any* time during program execution. To prevent indirect disclosure attacks, Readactor++ hides the targets of all function pointers and return addresses. We hide code pointers by converting these into direct branches stored in a dedicated *trampoline* code area with execute-only permissions. Code-pointer hiding allows the use of practical and efficient fine-grained code randomization, while maintaining security against indirect memory disclosure.

We also defend against reuse of dynamically bound functions (e.g., through vtables or the PLT). Unlike ROP attacks, which reuse short instruction sequences, COOP and RILC reuse dynamically-bound functions called through code pointers normally stored in read-only memory. To construct a COOP payload, the adversary must know (or disclose) the exact layout of needed vtables. Similarly, in return-into-PLT attacks, the adversary requires knowledge of the layout of the PLT. Our key insight is that we can permute and hide the contents of these tables using execute-only memory, even from an adversary who can disclose arbitrary readable memory.

Figure 3.2 shows the overall architecture of Readactor++. Since our approach benefits from precise control-flow information, which binary analysis cannot easily provide, we opt for a compiler-based solution. Our choice to use a compiler also improves the efficiency and practicality of our solution. In particular, our technique scales to complex, real-world software: the Chromium web browser and its V8 JavaScript engine (see section 3.6.2). To demonstrate that our techniques can also be used to randomize at load time, we implement vtable and PLT randomization using a staged randomization approach [18, 105]; we instrument binaries during compilation so that they randomize dynamic-binding tables when the program loads. Note that although our prototype uses a compiler, our approach is compatible with binary rewriting as long as the vtable hierarchy and all virtual and PLT call sites can be recovered, as in VTint [156].

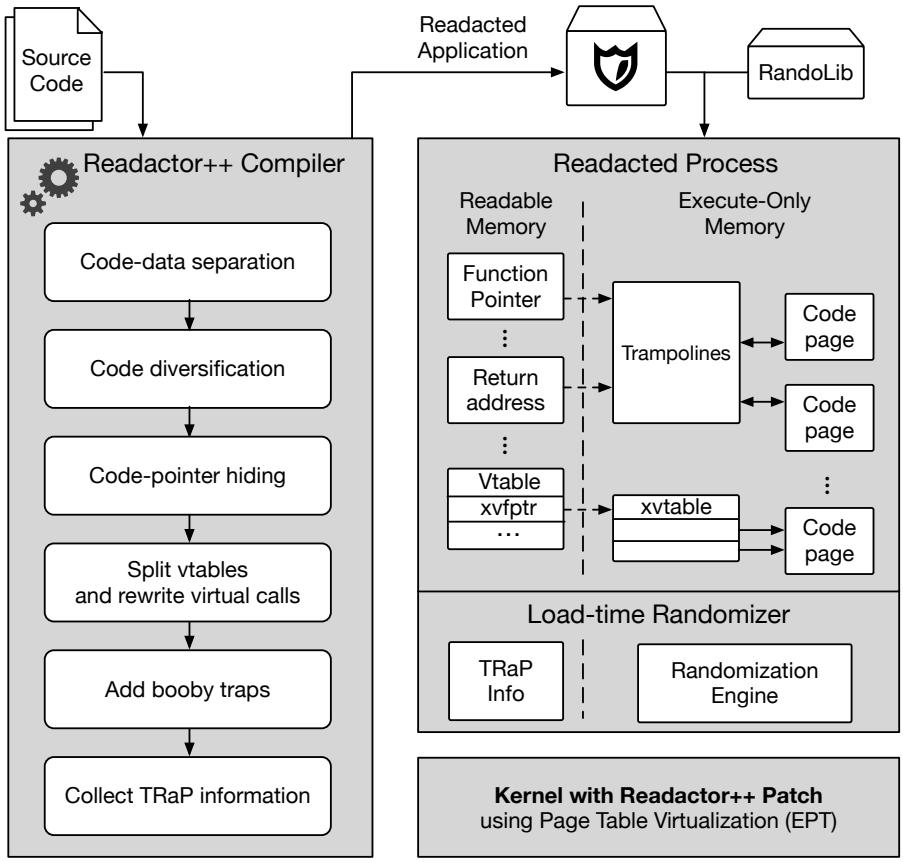

Figure 3.2: System overview. Our compiler generates diversified code that can be mapped with execute-only permissions, inserts trampolines to hide code pointers, and rewrites code pointer tables for randomization at runtime. We modify the kernel to use EPT permissions enabling execute-only pages.

As shown on the left of figure 3.2, our compiler converts unmodified source code into a *readacted* application. It does so by (i) separating code and data to eliminate benign read accesses to code pages, (ii) randomizing the code layout, and (iii) emitting trampoline code to hide code pointers from the adversary. To protect against dynamically-bound function reuse, the compiler additionally (iv) rewrites vtables and their respective references to hide virtual function addresses in executable trampoline tables (called xvtables), (v) inserts booby trap entries into dynamic-dispatch tables, and (vi) records "TRanslation and Protection" (TRaP) meta-data to prepare the application for load-time randomization. The right side of figure 3.2 illustrates a Readacted process at runtime. The load-time randomization component uses

the TRaP information stored in the application to randomize the ordering of the PLT and all xvtables, and rewrite all references to these tables. Our patched kernel maps all executable code pages with execute-only permissions at runtime. We do not alter the permissions of data areas, including the stack and heap. Hence, these are still readable and writable.

3.4.2 Code-Pointer Hiding

In an ideal fine-grained code randomization scheme, the content and location of every single instruction is random. Execute-only pages offer sufficient protection against all forms of memory disclosure at this granularity, since indirect disclosure of a code address gives the adversary no information about the location of any other instruction. However, ideal fine-grained randomization is inefficient and does not allow code sharing between processes. Hence, practical protection schemes randomize code at a coarser granularity to reduce the performance overhead [100]. Efficient use of modern processor instruction caches requires that frequently executed instructions are adjacent, e.g., in sequential basic blocks. Furthermore, randomization schemes such as Oxymoron [12] that allow code pages to be shared between processes lead to significantly lower memory usage but randomize at an even coarser granularity (i.e., page-level randomization).

To relax the requirement of ideal fine-grained code randomization, we observe that indirect JIT-ROP relies on disclosing code pointers in readable memory. The sources of code pointers in data pages are (i) C++ virtual tables, (ii) function pointers stored on the stack and heap, (iii) return addresses, (iv) dynamic linker structures (i.e., the global offset table on Linux), and (v) C++ exception handling. Our prototype system currently handles sources (i)-(iv); protecting code pointers related to C++ exceptions is an ongoing effort requiring additional compiler modifications which we discuss in section 3.5.2.

figure 3.3 illustrates our high-level technique to hide code pointers from readable memory pages. Whenever the program takes the address of a code location to store in readable memory,

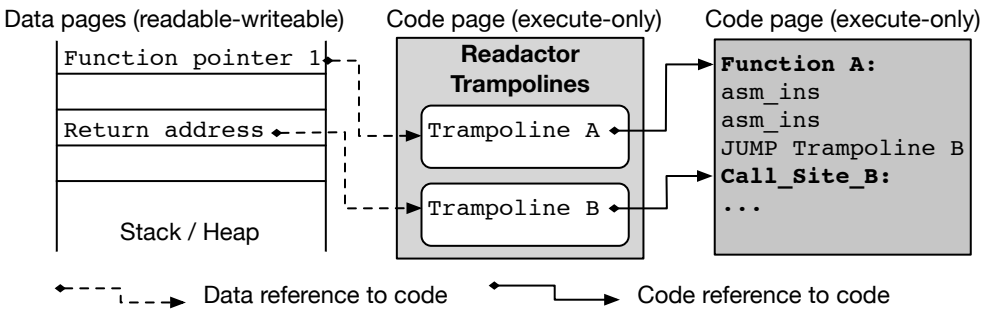

Figure 3.3: Readacted applications replace code pointers in readable memory with trampoline pointers. The trampoline layout is not correlated with the function layout. Therefore, trampoline addresses do not leak information about the code to which they point.

we instead store a pointer to a corresponding trampoline. Function pointers, for example, now point to trampolines rather than functions. When a call is made via `Function pointer 1` in figure 3.3, the execution is redirected to a Readactor++ trampoline (`Trampoline A`), which then branches directly to `Function A`.

Because trampolines are located in execute-only memory and because the trampoline layout is not correlated with the layout of functions, trampoline addresses do not leak information about non-trampoline code. Hence, trampolines protect the original code pages from indirect memory disclosure (see section 3.5.2 for details). This combination allows us to use a more practical fine-grained randomization scheme, e.g., function permutation and register randomization, which adds negligible performance overhead and aligns with current cache models.

In the following sections, we describe each component of Readactor++ in detail. First, we describe how we enable hardware-assisted execute-only permission on code pages (section 3.5.1). We then present our augmented compiler that implements fine-grained code randomization and code-pointer hiding (section 3.5.2).

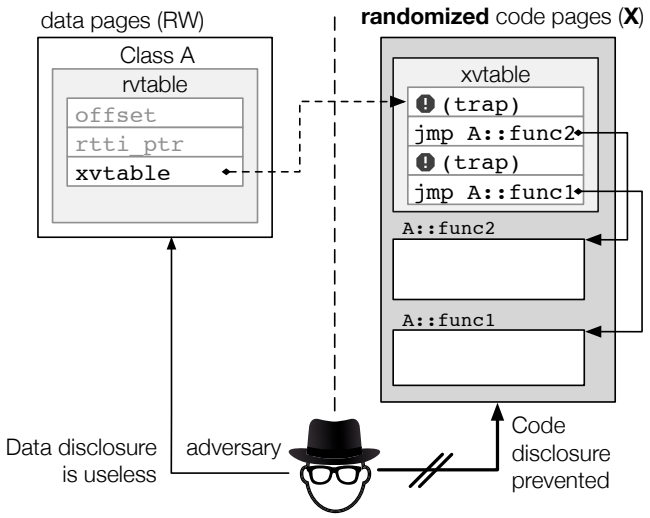

Figure 3.4: We split C++ vtables into a readable rvtable pointing to an executable xvtable. The readable part of the vtable (rvtable) contain no code pointers and the executable and randomized part (xvtable) is not readable.

3.4.3 Vtable Randomization

C++ objects contain a hidden member, `vptr`, that points to a vtable. We could randomize the layout of objects to make the `vptr` harder to locate. However, C++ objects are stored on the heap and stack which are necessarily readable, so an adversary can use a memory disclosure vulnerability to parse the heap and discover how objects of a given class are randomized. We therefore chose to randomize the layout of vtables instead.

If we only randomize the ordering of vtable entries, an adversary can still follow `vptr`'s in the heap or stack to disclose the vtable layout. Legitimate vtables are most commonly stored on pages with read permissions. To prevent disclosure of the vtable layout after randomization, we want to prevent adversaries from reading the part of vtables that contains code pointers. Our solution is to transform read-only code pointers into execute-only code. We encode each code pointer p as a direct jump using the value of p as the immediate operand. In addition to code pointers, vtables contain other data such as Run-Time Type Information (RTTI). We therefore split each vtable into two parts: a read-only part called the *rvtable* and an execute-only part

called the *xvtable*. We can either place the rvtable and xvtable on successive memory pages or we can add a pointer from the rvtable to the xvtable. If we choose the former approach, we need to pad rvtables to the nearest page boundary which wastes memory. We instead add a new field to the rvtable, `xpointer`, referencing the corresponding xvtable. Figure 3.4 shows how a traditional vtable (left) is split into an rvtable and an xvtable (right).

After splitting the vtable and inserting booby trap entries, we can securely randomize the ordering of each class's virtual functions. Since we store the xvtable in execute-only memory, an attacker cannot de-randomize the ordering of functions in the xvtable. We randomly permute the ordering of each class's virtual functions and rewrite all vtables in a semantics preserving way; section 3.5.3 describes how we do this. After randomizing the function ordering in the vtables, we must rewrite all virtual call sites in the program to use the new, permuted ordering. These virtual calls use a static offset from the start of a vtable to dispatch to the correct function. We rewrite this offset to correspond to the new, randomized index of the callee. As Readactor++ extends Readactor, all virtual call sites are also augmented with trampolines that hide actual return addresses at run time.

3.4.4 Procedure Linkage Table Randomization

A call to a dynamically linked function goes though the PLT of the calling module. Each entry in the PLT is a code sequence that (i) loads a function pointer from the GOT into a register and (ii) transfers control to the target function through an indirect jump. To enable lazy binding, each function's default address in the GOT initially points to a function that resolves the external function address. The left side of figure 3.5 shows the lazy binding process.

RILC attacks need to know or discover the layout of the PLT to succeed. For ASLR'ed Linux applications, PLT entries are laid out contiguously and only the base address is randomized. One way to improve security is to eliminate the PLT altogether. However, this approach is

problematic because the PLT or similar tables on Windows are essential to support dynamically linked functions.

As shown on the right side of figure 3.5, we chose to randomize the order of PLT entries, insert booby traps, and store the result in execute-only memory. If we had only done this, an adversary could still read GOT or relocation information to disclose the ordering of PLT entries indirectly. We therefore switch from lazy to eager binding resolution of PLT entries. This allows us to discard the GOT. Our specialized compiler determines which calls will be routed through the PLT and stores the location of each such call site in the TRaP section of the output ELF file. At load time, our runtime randomization engine, `RandoLib`, converts all pointers in the readable GOT into execute-only trampolines in the PLT. To do so, we precompute all of the function addresses in the GOT. Then, we rewrite all the trampolines in the PLT so they jump to the target function directly rather than reading the function address from the GOT and indirectly jumping to it. This allows us to remove the code pointers from the GOT altogether. By stripping the GOT from all code pointers, we can also remove its associated relocation information from the memory. We then shuffle the trampolines in the PLT. This transformation prevents attackers from inferring the layout of the PLT or leaking the code pointers it contains. Finally, we use the TRaP information stored by our compiler to rewrite all call sites that point into the PLT with the corresponding randomized address.

3.4.5 Countering Guessing Attacks

Since we prevent attackers from directly reading valuable tables such as the PLT and vtables, we expect that attackers may try to execute xvtable entries and other addresses in execute-only memory to guess their contents. Researchers have shown that brute-force attacks can bypass diversity, especially in services that automatically restart without re-randomization after crashing [136, 22, 134, 62]. We use software booby traps, discussed in chapter 2, to counter this

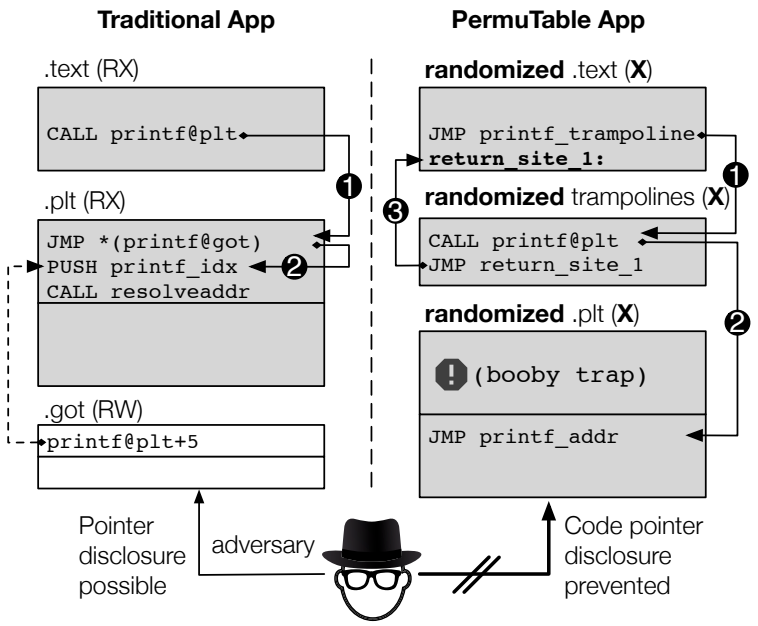

Figure 3.5: In traditional apps, functions call PLT entries directly (left). In Readactor++ apps, functions first jump to a trampoline which performs the actual call, so the return address pushed on the stack does not reveal the function layout. Moreover, we eagerly resolve the targets of PLT entries which eliminates so we can remove the GOT and add booby traps to deter probing.

threat. The idea is that booby traps lie dormant during normal program operation but are likely triggered by adversarial probing. Booby traps therefore terminate the program immediately and notify an administrator or security software of the issue. Terminating the program causes it to re-randomize as it starts up again, invalidating whatever knowledge the adversary has collected so far. When randomizing xvtables and procedure linkage tables, we insert booby traps. These are simply direct jumps to a booby trap handling routine. This probabilistically bounds the number of undetected guesses.

We take special care to ensure that there are many possible permutations for each table. Specifically, we ensure that each permuted table contains at least $n_{min} = 16$ entries. As explained section 3.5.3, portions of each xvtable, which we call *sub-xvtables*, are permuted together. However we cannot alter the relative ordering of sub-xvtables. Thus, we ensure that each *sub*-xvtable has at least $n_{min} = 16$ entries. We also require that at least $k = \frac{1}{4}$ of the entries of each table are booby traps, so guessing the table entries by executing them will quickly lead to

detection. Given a class with n virtual functions, we add $\max(\lceil k \cdot n \rceil, n_{min} - n)$ booby traps to its xvtable to meet these requirements. Both n_{min} and k are fully tunable for increased security. See section 3.6.1 for a discussion of how these parameters affect the security of our system.

3.5 Implementation

3.5.1 Execute-Only Memory

Enforcing execute-only memory for all executable code is one of the key components of our system. Below we discuss the challenges of implementing hardware enforced execute-only memory on the x86 architecture.

Extended Page Tables

The x86 architecture provides two hardware mechanisms to enforce memory protection: segmentation and paging. Segmentation is a legacy feature and is fully supported only in 32-bit mode. In contrast, paging is used by modern operating systems to enforce memory protection. While modern x86 CPUs include a permission to mark memory as non-executable [83, 7], it used to be impossible to mark memory as executable and non-readable at the same time. This changed in late 2008 when Intel introduced a new virtualization feature called *Extended Page Tables* (EPTs) [83]. Modern AMD processors contain a similar feature called *Rapid Virtualization Indexing*.

Readactor++ uses EPTs to enforce execute-only page permissions in hardware. EPTs add an additional abstraction layer during the memory translation. Just as standard paging translates virtual memory addresses to physical addresses, EPTs translate the physical addresses of a virtual machine (VM)—the so-called *guest physical memory*—to real physical addresses or

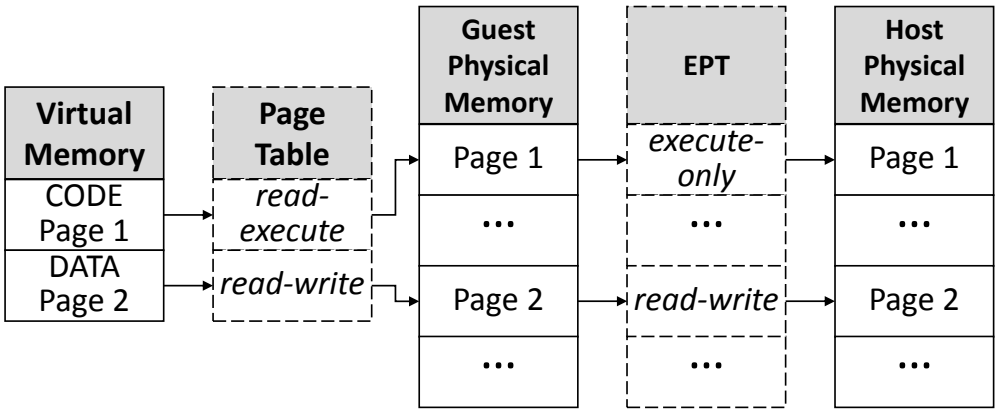

Figure 3.6: Relationship between virtual, guest physical, and host physical memory. Page tables and the EPT contain the access permissions that are enforced during the address translation.

host physical memory. The access permissions of each page are enforced during the respective translations. Hence, the final permission is determined by the intersection of the permissions of both translations. EPTs conveniently allow us to enforce (non-)readable, (non-)writable, and (non-)executable memory permissions independently, thereby enabling efficient enforcement of execute-only code pages.

figure 3.6 shows the role of the page table and the EPT during the translation from a virtual page to a host physical page. In this example, the loaded application consists of two pages: a code page, marked execute-only, and a data page marked as readable and writable. These page permissions are set by the compiler and linker. If a code page is labeled with only execute permission, the operating system sets the page to point to a page marked execute-only in the EPT. Note that access control is enforced for each translation step. Hence, a read operation on the code page is allowed during the translation of the virtual to the guest physical page. But, when the guest physical page is translated to the host physical page, an access fault is generated, because the EPT permission is set to execute-only. Similar to the access permissions of a standard x86 page table, the EPT permissions cannot be bypassed by software. However, EPTs are only available when the operating system is executing as a virtualized guest. The next section describes how we addressed this challenge.

Hypervisor

Our approach can be used in two different scenarios: software already operating inside a virtualized environment, and software executing directly on physical hardware. For the former case, common in cloud computing environments, the execute-only interface can be implemented as an extension to an existing hypervisor. We chose to focus on the second, non-virtualized scenario for two reasons: First, while standard virtualization is common for cloud computing, we want a more general approach that does not require the use of a conventional hypervisor (and its associated overhead). Many of the attacks we defend against (including our indirect JIT-ROP attack in section 3.2.1) require some form of scripting capability [139, 36, 37, 56] and therefore target software like browsers and document viewers running on non-virtualized end-user systems. Second, implementing a thin hypervisor allows us to measure the overhead of our technique with greater precision.

Our hypervisor is designed to transparently transfer the currently running operating system into a virtual environment on-the-fly. Our thin hypervisor design is inspired by hypervisor rootkits that transparently switch an operating system from executing on physical hardware to executing inside a virtual environment that hides the rootkit [93, 128]. Unlike rootkits, however, our hypervisor interfaces with the operating system it hosts by providing an interface to manage EPT permissions and to forward EPT access violations to the OS. Our hypervisor also has the capability to revert the virtualized operating system back to direct hardware execution without rebooting if needed for testing or error handling. For performance and security reasons, we keep our hypervisor as small as possible; it uses less than 500 lines of C code.

figure 3.7 shows how we enable execute-only page permissions by creating two mappings of the host physical memory: a *normal* and a *readacted* mapping. The EPT permissions for the normal mapping allow the conventional page table to fully control the effective page permissions. As previously mentioned, the final permission for a page is the intersection of the page table

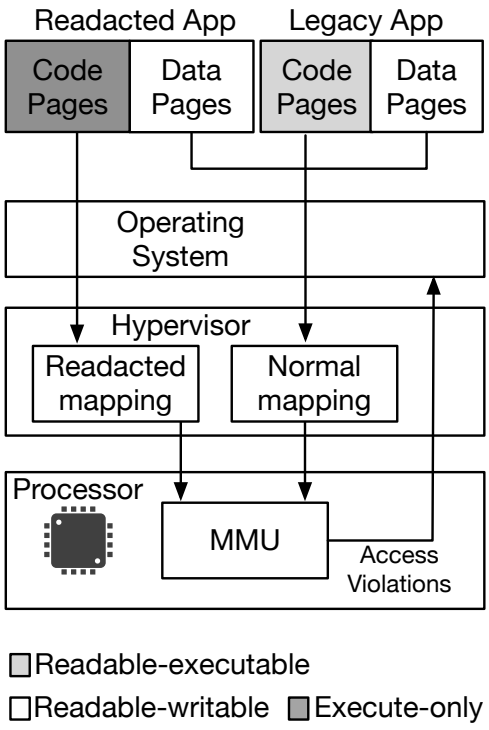

Figure 3.7: Readactor++ uses a thin hypervisor to enable the extended page tables feature of modern x86 processors. Virtual memory addresses of protected applications (top left) are translated to physical memory using a *readacted* mapping to allow execute-only permissions whereas legacy applications (top right) use a *normal* mapping to preserve compatibility. The hypervisor informs the operating systems of access violations.

permission and the EPT permission. Hence, setting the EPT permissions to RWX for the normal mapping means that only the permissions of the regular page table are enforced. We set the EPT permissions for the readacted mapping to execute-only so that any read or write access to an address using this mapping results in an access fault. The operating system can map virtual memory to physical memory using either of these mappings. When a block of memory is mapped through the readacted mapping, execute-only permissions are enforced. When the normal mapping is used, executable memory is also readable.

Our use of extended page tables is fully compatible with legacy applications. Legacy applications can execute without any modification when Readactor++ is active, because the normal mapping is used by default. Readactor++ also supports code sharing between legacy and readacted applications. Legacy applications accessing readacted libraries will receive an execute-only

mapping of the library, thus securing the library from disclosure. Readacted applications that require a legacy, un-readacted library can load it normally, but the legacy library will still be vulnerable to information disclosure.

Operating System

To simplify implementation and testing, our prototype uses the Linux kernel. However, our fundamental approach is operating system agnostic and can be ported to other operating systems. We keep our patches to the Linux kernel as small as possible (the patch contains 82 lines of code and simply supports the mapping of execute-only pages). Our patch changes how the Linux kernel writes page table entries. When a readacted application requests execute-only memory, we set the guest physical address to point to the readacted mapping rather than the normal mapping.

3.5.2 Compiler Instrumentation

To support the Readactor++ protections, we modified the LLVM compiler infrastructure [101] to (i) generate diversified code, (ii) prevent benign code from reading data residing in code pages, and (iii) prevent the adversary from exploiting code pointers to perform indirect disclosure attacks.

Fine-grained Code Diversification

Our compiler supports several fine-grained code randomization techniques: function permutation, basic-block insertion, NOP (no-operation) insertion, instruction schedule randomization, equivalent instruction substitution, register allocation randomization, and callee-saved register save slot reordering. The last technique randomizes the stack locations that a function uses to

save and restore register values that it must preserve during its execution. In our prototype implementation of Readactor++, we use function permutation [92], register allocation randomization, and callee-saved register save slot reordering [118]. We selected these transformations because they permute the code layout effectively, have low performance impact, and make the dataflow between gadgets unpredictable.

Our prototype implementation performs fine-grained code randomization at compile-time. With additional implementation effort, we can make the compiler emit binaries that randomize themselves at load-time [20, 153, 105]. Self-randomizing binaries eliminate the need to generate and distribute multiple distinct binaries, which improves the practicality of diversity. However, the security properties of compile-time and load-time solutions are largely similar. Hence, we focus on how to randomize programs and how to protect the code from disclosure irrespective of when randomization happens.

Code and Data Separation

To increase efficiency, compilers sometimes intersperse code and data. Since Readactor++ enforces execute-only permissions for code pages, we must prevent the compiler from embedding data in the code it generates. That is, we must generate Harvard-architecture compatible code. If we do not strictly separate code and data, we run the risk of raising false alarms as a result of benign attempts to read data from code pages.

We found that the LLVM compiler only emits data in the executable `.text` section of x86 binaries when optimizing a switch-case statement. LLVM emits the basic block address corresponding to each switch-case in a table after the current function. As shown in the left part of figure 3.8, the switch statement is then implemented as a load from this table and an indirect branch to the loaded address.

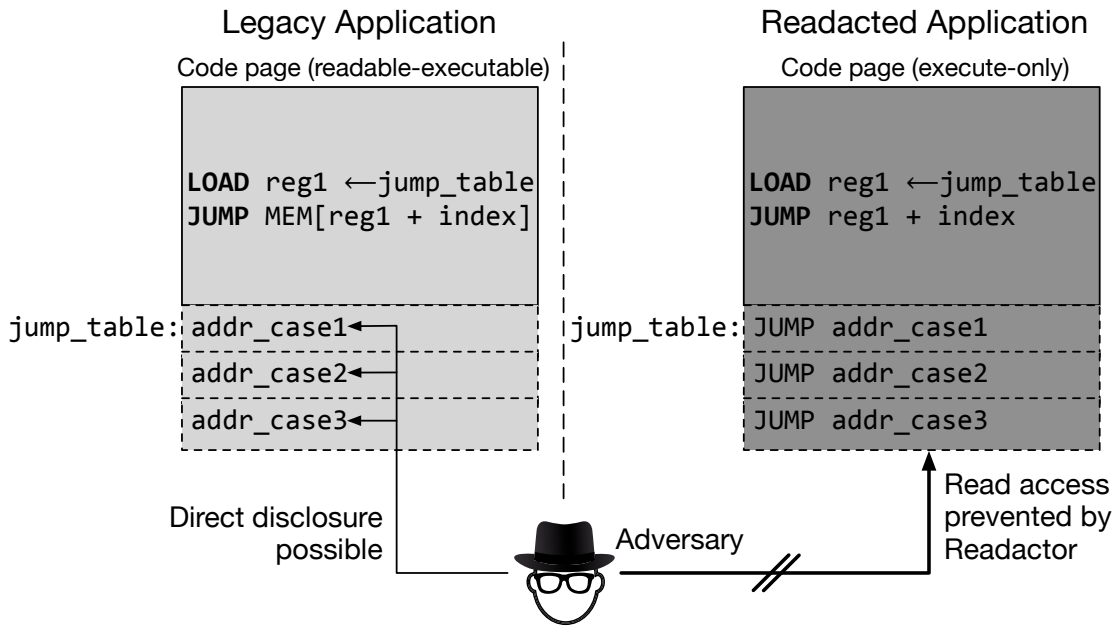

Figure 3.8: We rewrite switch-case tables to be executable instructions, rather than data embedded in executable code.

Our compiler translates switch statements to a table of direct branches rather than a list of code pointers that an attacker can read. Each direct branch targets the first basic block corresponding to a switch-case. The switch statement is then generated as an indirect branch into the sequence of direct branches rather than an indirect load and branch sequence. This entirely avoids emitting the switch-case pointers as data, thereby making LLVM generate x86 code compatible with execute-only permissions. figure 3.8 shows how code pointers (`addr_case1...addr_case3`) are converted to direct jumps in an example switch-case statement. We quantify the impact of this compiler transformation in section 3.6.2.

While examining x86 binaries on Linux, we noticed that contemporary linkers include both the readable ELF header data and executable code on the first page of the mapped ELF file. Hence, we created a patch for both the BFD and Gold linkers to start the executable code on a separate page from the readable ELF headers and to adjust the page permissions appropriately. This separation allows the ELF loader to map the headers as readable while mapping the first code page as execute-only.

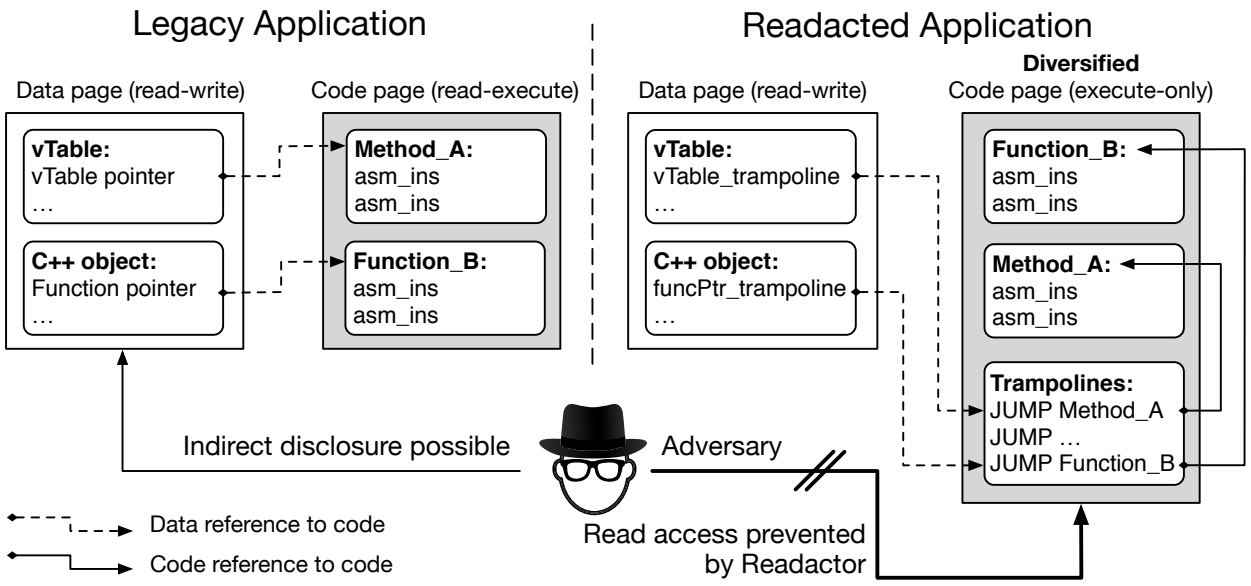

Figure 3.9: Hiding code pointers stored in the heap and in C++ vtables. Without Readactor++, pointers to functions and methods may leak (left). With Readactor++, only pointers to jump trampolines may leak and the layouts of functions and jump trampolines are randomized (right).

Code-Pointer Hiding

Making code non-readable prevents the original JIT-ROP attack but not indirect JIT-ROP. In the latter attack, an attacker combines pointer harvesting with partial a priori knowledge of the code layout, e.g., the layout of individual code pages or functions (see section 3.2.1). To thwart indirect JIT-ROP, we hide code pointers so they are no longer stored in readable memory pages.

We protect against the sources of indirect code disclosure identified in section 3.4.2 by adding a level of indirection to code pointers. The two steps in code-pointer hiding are (i) creating trampolines for each instruction reachable through an indirect branch and (ii) replacing all code pointers in readable memory with trampoline pointers. We use two kinds of trampolines: *jump trampolines* and *call trampolines*, to protect function addresses and call sites respectively.

We generate a *jump trampoline* for each function that has its address taken. figure 3.9 shows how we replace a vtable and function pointer with pointers to jump trampolines. For example,

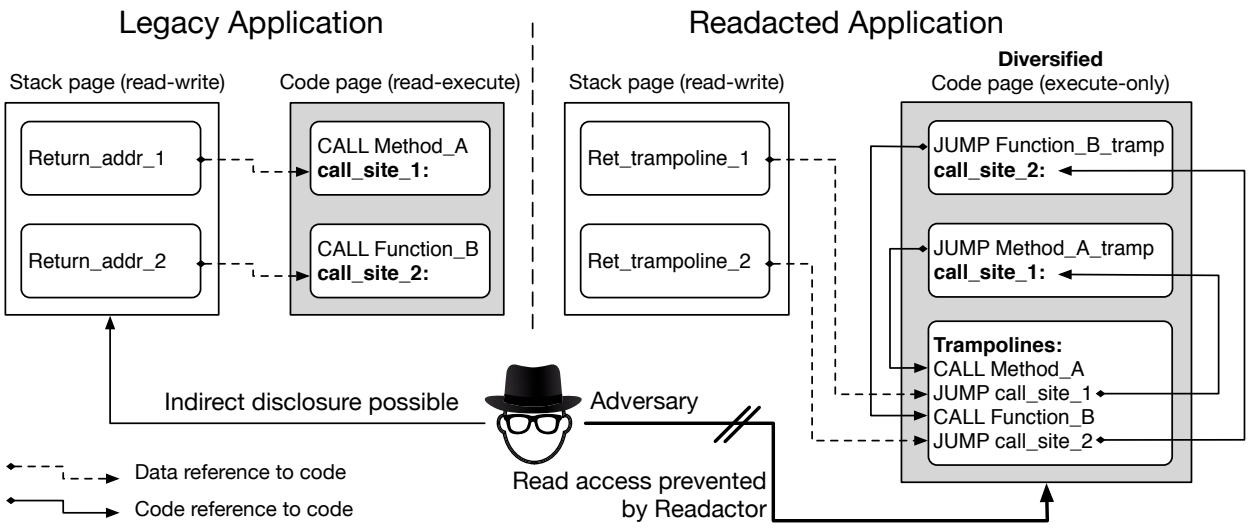

Figure 3.10: Hiding return addresses stored on the machine stack. Without Readactor++, each activation frame on the stack leaks the location of a function (left). With Readactor++, calls go through call trampolines so the return addresses pushed on the stack can only leak trampoline locations – not return sites (right).

when a call is made through funcPtr_trampoline, execution is redirected to the original target of the call: Function_B.

The call trampolines that hide return addresses on the stack are shown in figure 3.10. Normally, a call to Method_A will push the address of the following instruction (call_site_1) onto the stack. The Readactor++ compiler moves the call into a call trampoline such that the return address that is pushed onto the stack points to the call trampoline rather than the calling function. When the callee returns to the call trampoline, a direct branch transfers execution back to the original caller. Like jump trampolines, call trampolines cannot be read by attackers and therefore do not leak information about the function layout.

A final source of indirect code leakage is related to C++ exception handling. When an exception occurs, the C++ runtime library must unwind the stack, which is the process of walking back up the call chain and destroying locally allocated objects until an appropriate exception handler is found. Modern C++ compilers implement efficient exception handling by generating an exception handling (EH) table that informs the unwinding routine of the stack contents. These

data tables are stored in readable memory during execution and contain the range of code addresses for each function and the information to unwind each stack frame. During stack unwinding, the C++ runtime library locates the exception handling entry for each return address encountered on the stack. Since our call trampolines push the address of a trampoline onto the stack rather than the real return address, the runtime will try to locate the address of the call trampoline in the exception handling tables. Hence, we need to replace the real function bounds in the EH table with the bounds of the trampolines for that function.

Our prototype compiler implementation does not rewrite the EH table to refer to trampolines; however, doing so is merely a matter of engineering effort. No aspect of our approach prevents us from correctly supporting C++ exception handling. We found that disabling C++ exception handling was not a critical limitation in practice, since many C++ projects, including Chromium, choose not to use C++ exceptions for performance or compatibility reasons.

Handwritten assembly routines are occasionally used to optimize performance critical program sections where standard C/C++ code is insufficient. To prevent this assembly code from leaking code pointers to the stack, we can rewrite it to use trampolines at all call sites. Additionally, we can guarantee that no code pointers are stored into readable memory from assembly code. Our current implementation does not perform such analysis and rewriting of handwritten assembly code but again, doing so would involve no additional research.

While code pointers are hidden from adversaries, trampoline pointers are stored in readable memory as shown on the right-hand sides of figure 3.9 and figure 3.10. Therefore, we must carefully consider whether leaked trampoline pointers are useful to adversaries. If the layout of trampolines is correlated with the function layout, knowing the trampoline pointers informs adversaries of the code layout. To ensure there is no correlation between the layout of trampolines and functions, we permute the list of trampolines. We also insert *dummy* entries into the list of trampolines that consist of privileged instructions. Hence, any attempts to guess the contents of the trampoline list by executing them in a row will eventually trigger an exception [50].

Each trampoline entry contains a direct jump that consists of an opcode and an immediate operand encoding the destination of the jump. Because we permute the order of functions, the value of the immediate operand is randomized as a side effect. This makes the contents of trampolines unpredictable to attackers and prevents use of any unintended gadgets contained in the immediate operands of direct jumps.

An attacker can use trampoline pointers just as they are used by legitimate code: to execute the target of the trampoline pointer without knowledge of the target address. Because we only create trampolines for functions that have their address taken and for all return sites, our code mechanism restricts the attacker to reuse only function-entry gadgets and call-preceded gadgets. Note that CFI-based defenses constrain attackers similarly and some CFI implementations use trampoline mechanisms similar to ours [157, 146, 158]. Coarse-grained CFI defenses are vulnerable to gadget stitching attacks where adversaries construct ROP chains out of the types of gadgets that are reachable via trampoline pointers [71, 56]. Although gadget stitching attacks against Readactor will be hard to construct because the required trampoline pointers may be non-trivial to leak, we still included protection against such attacks. We observe that gadget chains (including those that bypass CFI) pass information from one gadget to another via stack slots and registers. Because we perform register allocation randomization and callee-saved register save slot reordering, the attacker cannot predict how data flows through the gadgets reachable via trampolines.

Modern processors have deep pipelines and fetch instructions long before they may be needed. On the one hand, pipelining can hide the cost of the additional indirection introduced by trampolines. On the other hand, we must ensure that our use of trampolines to hide code pointers does not increase the number of costly branch predictor misses that stall the processor pipeline until the correct instructions are fetched. Thanks to the use of a dedicated return-address stack in modern branch predictor units, targets of function returns are rarely mispredicted as long as function calls and returns are paired. By preserving this pairing, we ensure that our

trampolines do not introduce additional branch predictor misses. We evaluate the performance impact of jump and call trampolines in section 3.6.2.

Splitting Vtables

As we discuss in section 3.4.3, we must split vtables into a read-only rvtable and an execute-only xvtable in order to protect the randomized table contents from disclosure. We modify the Clang C++ front end to split vtables accordingly.

We must also modify all virtual call sites to handle the new, split layout. Rewritten call sites must first dereference the vtable pointer from the object, as usual, to get the address of the correct rvtable. The call must then dereference the xvtable pointer found in this rvtable. After the xvtable address is obtained, the virtual call indexes into this table to find the correct trampoline address for the virtual function, which can then be called. Altogether, we add one additional memory reference to each virtual method call.

In C++, one can also store the address of a class method into a method pointer and later dereference this pointer to call the method. We also handle this slightly more complex case by storing an index in the method pointer struct, as normal. We then handle the xvtable pointer dereference whenever a method pointer is invoked.

Collecting TRaP Information

We use several types of information, available at compile time, to properly randomize the PLT and vtable at load time. We embed this meta-data in a special section of the output object files.

To randomize the xvtables, we need the location of each class's vtable, as well as the number of virtual functions in the class, which is not present in the binary. We modify both the C++ front end and the code emission back end of LLVM to add additional metadata (TRaP information)

which marks the location of and number of functions in each vtable for use at load time. We also mark the class inheritance hierarchy for each component of the vtable, since this information is difficult to derive from the vtable alone.

Additionally, we need the location of each reference to the PLT and each virtual call in the program code, in order to rewrite these uses after randomization. We modify the Clang C++ front end to find all PLT references and virtual function call sites and modify the compiler back end to mark these locations in the binary. Specifically, we mark instructions which hold the index of a virtual function so that these indices can be rewritten.

3.5.3 Runtime Randomization Engine

Our runtime randomization component, `RandoLib`, consumes the TRaP information emitted by the compiler, permutes the PLT and all vtables, and finally updates all references to these tables to refer to a new, randomized index. We built `RandoLib` as a dynamically linked module that randomizes the other loaded modules as the program starts. Our modified compiler automatically emits an initialization routine into each module that registers the module with `RandoLib`. `RandoLib` then randomizes all registered modules together after all modules have been loaded but before the dynamic linker passes control to the program entry point.

`RandoLib` can then be safely unloaded after it has randomized the registered modules. When it unloads, it frees all of the memory it had allocated and overwrites all of the code pointers in its private data with zeros. It therefore does not increase the attack surface.

Identifying tables

The first step of the randomization process is to collect the locations and sizes of the PLT and xvtables from the TRaP section. After identifying all xvtables, we must identify and distinguish

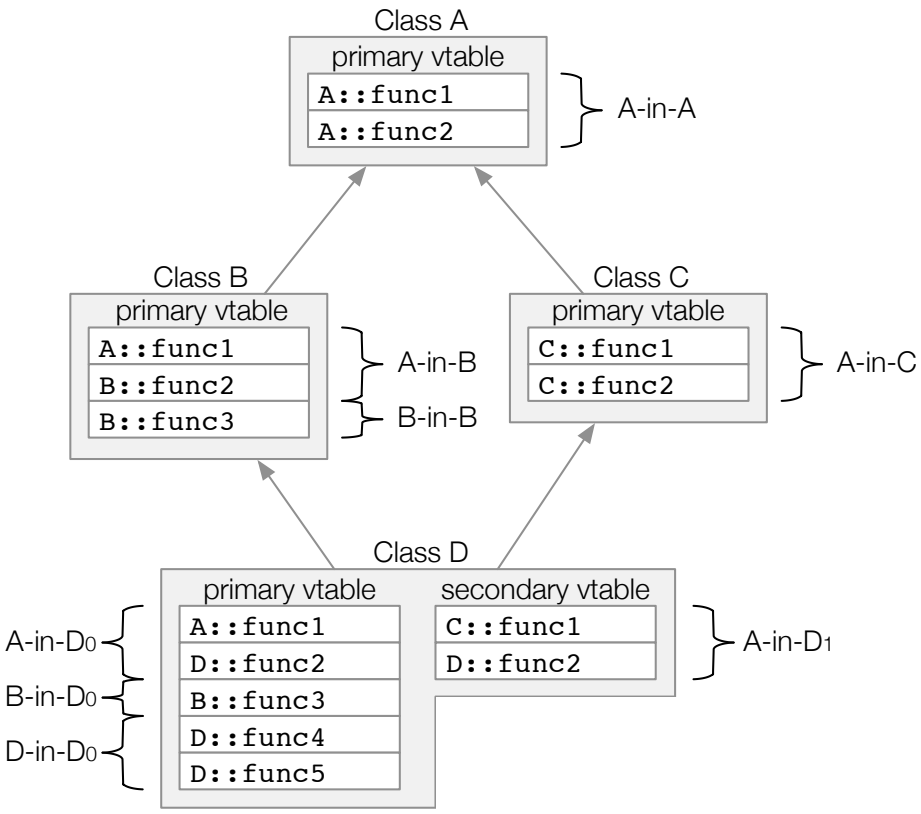

Figure 3.11: Example class hierarchy rooted in class A. The direct subclasses of A use single inheritance and provide their own implementation of `func2`. Class D uses multiple inheritance and therefore has two virtual function tables.

all copies of each class's xvtable. Vtables contain sub-vtables corresponding to the vtables of their base classes, and each of these corresponding sub-vtables must be randomized in the same order.

To see why this process is necessary, we give an example of a simple inheritance hierarchy and the corresponding vtable layouts. figure 3.11 shows the "diamond" inheritance pattern in which a class D has two base classes, B and C that both inherit from a single class A. If a call is made to `func1` on an instance of D, the target of the call depends on the call site. If the call site expects an instance of C, `C::func1` is invoked, and `A::func1` is invoked in all other cases. This behavior is implemented using a primary and secondary vtable for instances of class D (see bottom third of figure 3.11). The primary vtable is used from call sites that expect objects

of type A, B, or D and the second vtable is used when a call expects an instance of class C. For this reason, all instances of class D contain two vtable pointers.

To explain why xvtables must be split into sub-xvtables for randomization, let us consider randomizing the class hierarchy shown in figure 3.11. Three classes derive directly or indirectly from class A. Each of these subclasses therefore has a set of vtable entries that correspond to the virtual methods defined by class A; we call such sets sub-vtables. The layout of a sub-vtable must be the same for all classes related by inheritance. In figure 3.11, the first sub-vtables shared among the class A and its descendants share the A-in- prefix in their names.

A virtual method call that operates on an object of type A can effectively use any entry in the A-in- sub-vtables because any object of one of A's sub-types can be assigned to a pointer of type A. This is why we must ensure that the layout of each of these sub-vtables are mutually compatible.

Since the TRaP section contains the locations and sizes of whole xvtables, `RandoLib` must first split the xvtables into sub-xvtables and then group sub-xvtables together by their type. `RandoLib` relies on the vtable inheritance relationships to identify sub-xvtables. Consider again the previous example. If we know that B's xvtable is based on A's xvtable, then the compiler will guarantee that the lower part of B's xvtable will be the A-in-B sub-xvtable.

Information about the inheritance relationships between xvtables is not readily available at run time. This information can in principle be derived from the Run-Time Type Information (RTTI) structures that are embedded in C++ programs by default [41]. During the course of this research, however, we have found several ambiguous cases where unused vtables are not emitted and therefore we could not always correctly disambiguate complex vtable layouts. Additionally, we found that compilers implementing the Itanium ABI do not always agree on sub-vtable layouts, although this is presumably due to compiler bugs. We have therefore chosen to embed the names of the bases for each sub-vtable in the TRaP information as well. For

example, we mark classes A, B, D as the bases of the primary vtable of Class D in figure 3.11. This information, combined with the RTTI structures, is sufficient to recover the whole xvtable hierarchy in the program. If it is undesirable to embed C++ RTTI information in the binary, we can easily extend the TRaP information to store all necessary meta-data.

Identifying references

The second step in the randomization process is to resolve the target of all references to the tables we have identified in the first step. These references must be updated to reflect the new table layouts after permutation.

The TRaP section contains the locations of all index operations into the PLT and vtables. To save space we do not embed the indices used in these the references in the TRaP section, since they can be read by disassembling the marked instructions. We can disassemble references quickly and accurately because they are valid, single instructions.

The TRaP section also contains the name of the intended target class for each xvtable reference. `RandoLib` can then resolve the target to the set of sub-xvtables corresponding to receiving class, which are all randomized using the same permutation as described previously.

Randomization

The final step in the randomization process is to randomly permute all tables we identify in the first step and update the references we identify in the second step.

We use algorithm 3.1 to update xvtable references. In this algorithm, G is a directed graph that represents the class hierarchy. We calculate G based on the RTTI information. Each node in G represents a class that has an array of sub-xvtable pointers ($VTables$). For each sub-xvtable, we store a link to the sub-xvtable that preceded it in the original (whole) xvtable ($Preceding$)

and the offset of the sub-xvtable within that original xvtable ($Offset$). $XRefsVector$ is an array of ($Location$, $ClassName$, $Index$) tuples. Each tuple represents one xvtable reference. The $ClassName$, which represents the name of the intended object type for the virtual method call, as well as the $Location$ are read from the TRaP section. We determine the $Index$ by disassembling the reference.

For each xvtable reference, we start by finding the primary xvtable for the call's intended target type (line 4). We then resolve the reference to an exact sub-xvtable group based on the sub-xvtable offsets (line 5-7). We then calculate the index of the intended target of the reference within that sub-xvtable and update the reference (line 8-10).

```
Input  : Directed graph G, Vector XRefsVector

foreach XRef in XRefsVector do
    Class ← G.Find(XRef.ClassName)
    SubVtbl ← Class.VTables[0]
    while XRef.Index < SubVtbl.Offset do
        SubVtbl ← SubVtbl.Preceding
    end
    Idx ← XRef.Index − SubVtbl.Offset
    Tmp ← SubVtbl.Permutation[Idx]
    XRef.Index ← SubVtbl.Offset + Tmp
end
```

Algorithm 3.1: Update vtable references after randomization.

3.6 Evaluation

3.6.1 Security

We designed Readactor++ to prevent all forms of code-reuse attacks, including those utilizing direct or indirect disclosure vulnerabilities, as well as attacks exploiting dynamic function binding. Thus, we have analyzed and tested its effectiveness based on five different variants of

code-reuse attacks, namely (i) static ROP attacks using direct and indirect disclosure, (ii) just-in-time ROP attacks using direct disclosure, (iii) just-in-time ROP attacks using indirect disclosure, (iv) return-into-lib(c) attacks, and (v) COOP attacks. We present a detailed discussion on each type of code-reuse attack and then evaluate the effectiveness of Readactor++ using a sophisticated proof-of-concept JIT-ROP exploit.

Static ROP

To launch a traditional ROP attack [135, 31], the adversary must know the runtime memory layout of an application and identify ROP gadgets based on an offline analysis phase. To defeat regular ASLR the adversary needs to leak a single runtime address through either direct or indirect disclosure. Afterwards, the addresses of all target gadgets can be reliably determined.

Since Readactor++ performs fine-grained randomization using function permutation, the static adversary can only guess the addresses of the target gadgets. In other words, the underlying fine-grained randomization ensures that an adversary can no longer statically determine the addresses of all gadgets as offsets from the runtime address of a single leaked function pointer. In addition, we randomize register allocation and the ordering of stack locations where registers are saved to ensure that the adversary cannot predict the runtime effects of gadgets. Using these fine-grained diversifications, Readactor++ fully prevents static ROP attacks.

JIT-ROP with direct disclosure

JIT-ROP attacks bypass fine-grained code randomization schemes by disassembling code pages and identifying ROP gadgets dynamically at runtime. One way to identify a set of useful gadgets for a ROP attack is to exploit direct references in call and jump instructions [139]. Readactor++ prevents this attack by marking all code pages as non-readable, i.e., execute-only. This differs

from a recent proposal, XnR [11], that always leaves a window of one or more pages readable to the adversary. Readactor++ prevents all reading and disassembly of code pages by design.

JIT-ROP with indirect disclosure

Preventing JIT-ROP attacks that rely on direct disclosure is insufficient, since advanced attacks can exploit indirect disclosure, i.e., harvesting code pointers from the program's heap and stack (see section 3.2.1). Readactor++ defends against these attacks with a combination of fine-grained code randomization and code-pointer hiding. Recall that pointer hiding ensures that the adversary can access only trampoline addresses but cannot disclose actual runtime addresses of functions and call sites (see section 3.4.2). Hence, even if trampoline addresses are leaked and known to the adversary, it is not possible to use arbitrary gadgets inside a function because the original function addresses are hidden in execute-only trampoline pages. As discussed in section 3.5.2, code-pointer hiding effectively provides at least the same protection as coarse-grained CFI, since only valid address-taken function entries and call-sites can be reused by an attacker. However, our scheme is strictly more secure, since the adversary is forced to first disclose the address of each trampoline from the stack or heap before he can reuse the function or call-site. In addition, we strengthen our protection by employing fine-grained diversifications to randomize the dataflow of this limited set of control-flow targets.

Specifically, when exploiting an indirect call (i.e., using knowledge of a trampoline address corresponding to a function pointer), the adversary can only redirect execution to the trampoline but not to other gadgets located inside the corresponding function. In other words, we restrict the adversary who has disclosed a function pointer to whole-function reuse.

On the other hand, disclosing a call trampoline allows the adversary to redirect execution to a valid call site (e.g., call-preceded instruction). However, this still does not allow the adversary to mount the same ROP attacks that have been recently launched against coarse-grained CFI

schemes [71, 56, 29, 131], because the adversary only knows the trampoline address and not the actual runtime address of the call site. Hence, leaking one return address does not help to determine the runtime addresses of other useful call sites inside the address space of the application. Furthermore, the adversary is restricted to only those return trampoline addresses that are leaked from the program's stack. Not every return trampoline address will be present on the stack, only those that are actually used and executed by the program are potentially available. This reduces the number of valid call sites that the adversary can target, in contrast to the recent CFI attacks, where the adversary can redirect execution to *every* call site in the address space of the application without needing any disclosure.

Finally, to further protect call-site gadgets from reuse through call trampolines, we use two fine-grained diversifications proposed by Pappas et al. [118] to randomize the dataflow between gadgets: register allocation and stack slot randomization. Randomizing register allocation causes gadgets to have varying sets of input and output registers, thus disrupting how data can flow between gadgets. We also randomly reorder the stack slots used to preserve registers across calls. The program's application binary interface (ABI) specifies a set of callee-saved registers that functions must save and restore before returning to their caller. In the function epilogue, the program restores register values from the stack into the appropriate registers. By randomizing the storage order of these registers, we randomize the dataflow of attacker-controlled values from the stack into registers in function epilogues.

Whole-Function Reuse

Most of the papers dealing with code-reuse attacks do not provide a security analysis of whole-function reuse, such classic return-into-lib(c) attacks [109], i.e., attacks that only invoke entire functions rather than short ROP code sequences. In general, it is very hard to prevent whole-function reuse attacks, since they target legitimate addresses, such as exported library functions and virtual functions. In Readactor++, we limit the attack surface for such attacks.

When the target of a return-into-lib(c) attack is protected by diversity, the adversary must first identify the addresses of functions of interest, e.g., `system` or `mprotect` on Linux. Typically the attacker discloses the function address from a known position within either code or data sections. We prevent disclosure from code because Readactor++ maps code pages as execute-only. On the other hand, code pointers in data sections, e.g., pointers in the import address table (IAT) in Windows or the global offset table (GOT) in Linux which are used for shared library calls, can be exploited in an indirect disclosure attack. Since code-pointer hiding indirects all code pointers through execute-only trampolines, the adversary cannot read IAT/GOT entries to determine the actual address of a targeted function in a readacted library. However, if the function of interest is imported by the program then the adversary can directly invoke the corresponding PLT entry to directly invoke the function, without knowing its address.

To handle this *return-into-plt* attack, we randomize order of the entries in the PLT at runtime and remove all function pointers from the GOT (see section 3.4.4). Since this table is already composed of execute-only trampolines, the attacker cannot read the entries to determine their targets. We also add booby trap PLT entries to prevent an attacker from randomly invoking entries to find useful functions. Thus, an attacker cannot reliably perform a return-into-lib(c) style attack using the PLT or GOT.

The COOP attack [130] also reuses whole functions, but does so using virtual method tables rather than the PLT to refer to targeted functions. We apply the same randomization protection to vtables as we do with the PLT, and thus an attacker does not know which vtable indices correspond to which virtual functions. As with the PLT, booby traps offer additional protection against brute-forcing attacks. Since an attacker cannot determine the location of targeted virtual function trampolines inside vtables, we prevent all COOP-style attacks.

Guessing Table Layouts. The security of our function table randomization depends on the difficulty of de-randomizing the permuted function ordering. Critical to this defense is a mechanism

to prevent brute-force guessing and execution of table entries. Our adversary model (section 3.3) assumes a *brute-forcing mitigation* that permanently terminates the application when it executes a booby trap. The program will never execute a booby trap during correct execution. Even benign programming errors have extremely low likelihood of accidently triggering a booby trap, since we place booby traps in tables the programmer should never access directly.

Since hitting a booby trap will terminate the attack, a successful adversary needs to make an uninterrupted sequence of *good guesses*. What exactly constitutes a good guess depends on the concrete attack scenario. In the best case, the adversary always needs to guess a particular entry in a particular xvtable or the PLT; in the worst case, a good guess for the adversary may be any entry that is not a booby trap. Considering the nature of existing COOP and RILC attacks [130, 147], we believe that the former case is the most realistic. Further, assuming in favor of the adversary that he will only attempt to guess entries in tables with exactly 16 entries (the minimum), we get can roughly approximate the probability for Readactor++ to prevent an attack that reuses n functions with $P \approx 1-(\frac{1}{16})^n$. Our experiments in the following indicate that an attacker needs at least two or three hand-picked functions (from likely distinct tables) to mount a successful RILC or COOP attack respectively. Thus, the probability of preventing these attacks is lower bounded by $P_{RILC,min} \approx 1-(\frac{1}{16})^2 = 0.9960$ and $P_{COOP,min} \approx 1-(\frac{1}{16})^3 = 0.9997$.

Proof-of-concept exploit

To demonstrate the effectiveness of our protection, we introduced an artificial vulnerability into V8 that allows an attacker to read and write arbitrary memory. This vulnerability is similar to a vulnerability in V8[1] that was used during the 2014 Pwnium contest to get arbitrary code execution in the Chrome browser. In an unprotected version of V8, the exploitation of the introduced vulnerability is straightforward. From JavaScript code, we first disclose the address of a function that resides in the JIT-compiled code memory. Next, we use our capability to write

[1] `CVE-2014-1705`

arbitrary memory to overwrite the function with our shellcode. This is possible because the JIT-compiled code memory is mapped as RWX in the unprotected version of V8. Finally, we call the overwritten function, which executes our shellcode instead of the original function. This attack fails under Readactor++, because the attacker can no longer write shellcode to the JIT-compiled code memory, since we set all JIT-compiled code pages execute-only. Further, we prevent any JIT-ROP like attack that first discloses the content of JIT-compiled code memory, because that memory is not readable. We tested this by using a modified version of the attack that reads and discloses the contents of a code object. Readactor++ successfully prevented this disclosure by terminating execution of the JavaScript program when it attempted to read the code.

3.6.2 Performance

We rigorously evaluated the performance impact of Readactor++ on both the SPEC CPU2006 benchmark suite and a large real-world application, the Chromium browser.

SPEC CPU2006

The SPEC CPU2006 benchmark suite contains CPU-intensive programs which are ideal to test the worst-case overhead of our compiler transformations and hypervisor. To fully understand the impact of each of the components that make up the Readactor++ system, we measured and report their performance impact independently.

We performed all evaluations using Ubuntu 14.04 with Linux kernel version 3.13.0. Unless otherwise noted, we performed our performance evaluation using SPEC on an Intel Core i5-2400 desktop CPU running at 3.1 GHz with dynamic voltage and frequency scaling (Turbo Boost) enabled (*System I*). We also independently verified these evaluations using an Intel Xeon E5-

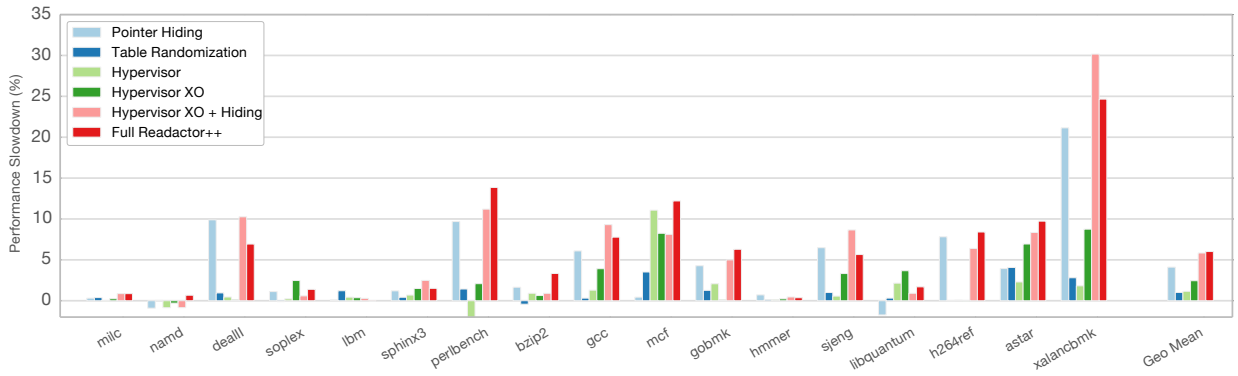

Figure 3.12: Performance overhead for SPEC CPU2006 with Readactor++ enabled relative to an unprotected baseline build.

2660 server CPU running at 2.20 GHz with Turbo Boost disabled, and observed identical trends and nearly identical performance (within one percent on all averages) (*System II*). We performed all measurements which include our table randomization techniques exclusively on *System II* for reasons of convenience. We summarize our SPEC measurements in figure 3.12. Overall, we found that Readactor++, with all protections enabled, incurs an average performance overhead of just 6.0% for SPEC CPU2006.

Code-Data Separation

First we evaluated the performance overhead of separating code from data by rewriting how the compiler emits switch tables in code (see section 3.5.2). We found the impact of transforming switch table data into executable code to be less than half of a percent on average, with a maximum overhead of 1.1%. This overhead is minimal because we maintain good cache locality by keeping the switch table close to its use. In addition, modern processors can prefetch instructions past direct jumps, which means these jumps have a low performance impact. We omit this experiment from figure 3.12 for clarity, since it showed such minimal overheads.

Code-Pointer Hiding

We then evaluated full code pointer protection, with hiding of both function pointers and return addresses enabled. We found that code-pointer hiding resulted in a performance slowdown of 4.1% on average over all benchmarks (*Pointer Hiding* in figure 3.12). This protection introduces two extra direct jumps for each method call and one direct jump when de-referencing function pointers. On closer inspection using hardware performance counters, we observed that these jumps back and forth from the regular code section to the trampolines slightly increased the instruction-cache pressure, resulting in more instruction-cache misses.

We hypothesized that the bulk of code-pointer hiding overhead was due to call trampolines, which are far more common than function pointers. To verify this, we disabled return address hiding while keeping function pointer protection enabled. We found that function pointer protection by itself incurred an average overhead of only 0.2% on SPEC, with no benchmark exceeding 2%. Thus, most of the overhead for code-pointer hiding is caused by the frequent use of call trampolines. This effect is amplified in benchmarks which make many function calls, such as *xalancbmk*.

Table Randomization

Next we evaluated the performance overhead of table randomization and booby trapping without any other transformations such as code-pointer hiding. We randomized the ordering of the PLT and inserted enough booby trap entries to make up 25% of the table entries. For the C++ benchmarks we also split and randomized vtables, adding booby traps so that all sub-vtables contained at least 16 entries. To properly evaluate the impact of our vtable transformations on the entire program as well as all relevant libraries, we applied our protections to libc++ and used this library as the standard C++ library. The baseline uses an unprotected version of libc++ to avoid any differences due to variations between C++ standard library implementations. PLT and

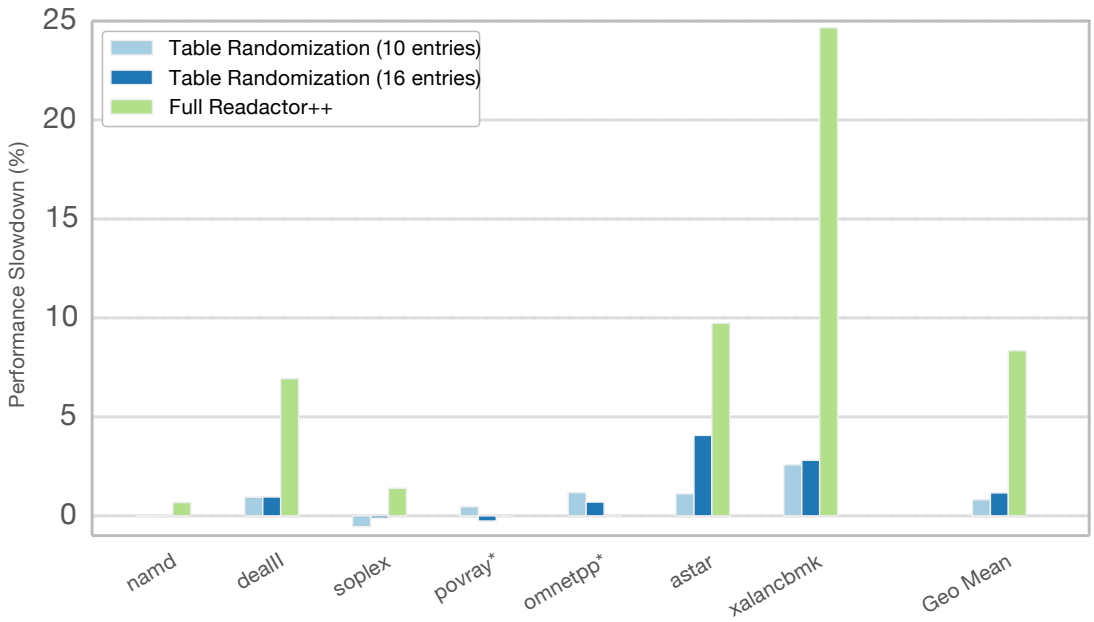

Figure 3.13: Overhead of table randomization on SPEC CPU2006 C++ benchmarks. Starred benchmarks are incompatible with the Readactor protection since they require C++ exception handling.

vtable randomization resulted in a total performance overhead of 1.0% (Table Randomization in figure 3.12).

Since this defense specifically focuses on defending C++ vtables and therefore introduces some overhead into C++ virtual method dispatch, we also report our evaluation on only the C++ benchmarks in SPEC CPU2006. The first two columns of figure 3.13 show the results of our evaluation of table randomization on the C++ benchmarks, without any other components of Readactor++. For a smaller minimum xvtable size of ten entries ($n_{min} = 10$), we observed a geometric mean slowdown of 0.8%. We observed a 1.2% geometric mean performance slowdown with a minimum xvtable size of $n_{min} = 16$. Since the additional performance slowdown is so minimal, we recommend a minimum xvtable size of at least 16 entries.

Hypervisor

To understand the performance impact of the hypervisor layer, including the additional EPT translation overhead, we ran SPEC under our hypervisor without any execute-only page protections or code-pointer hiding enabled (*Hypervisor* in figure 3.12). We observed that the virtualization overhead was 1.1% on average. Since we allow the virtual processor full control of all hardware and registers, this overhead is mainly caused by the extra memory translation overhead from translating guest physical addresses to host physical addresses through the EPT. Even though we use an identity mapping from guest physical to host physical addresses, the processor must still walk through the whole EPT whenever translating a new memory address. The larger overhead observed for the *mcf* benchmark supports this theory, as it has a higher and more stable memory footprint (845Mib) than the other benchmarks [75]. This results in more swapping in and out of the cache, which in turn triggers more EPT address translations.

After compiling SPEC with separation of code and data and marking code pages as execute-only during linking, we ran the benchmarks under the hypervisor, enforcing execute-only page permissions (*Hypervisor XO* in figure 3.12). This configuration incurred a performance slowdown of 2.5%, somewhat higher than the overhead of the hypervisor itself. Much of this overhead difference is due to the separation of code and data, which de-optimizes execution slightly. We attribute the rest of this difference to measurement variance, since the hypervisor itself should not add any significant overhead when enforcing execute-only pages versus legacy readable and executable pages. In either case the processor must still translate all addresses through the EPT when the hypervisor is enabled.

Full Readactor++

Enabling code-pointer hiding along with page protections provided by the hypervisor resulted in a slowdown of 5.8% (*Hypervisor XO + Hiding* in figure 3.12). This overhead is approximately

the sum of the overheads of both components of the system, the execute-only hypervisor enforcement and pointer hiding. This confirms our hypothesis that each component of the Readactor++ system is orthogonal with respect to performance.

With the addition of our fine-grained diversity scheme (function, register, and callee-saved register slot permutation) and table randomization we observed a total geometric mean performance overhead of 6.0% (*Full Readactor++* in figure 3.12) for the full defense on all C and C++ benchmarks in SPEC CPU2006. Looking at only the C++ benchmarks, we found that Readactor++ introduces an 8.4% performance overhead (Full Readactor++ in figure 3.13). We believe this additional overhead is reasonable since these benchmarks use more indirect dispatches, resulting in more code-pointer hiding overhead.

Our system compares favorably to the performance of recent CFI implementations, after accounting for the need to protect return addresses with a shadow stack. VTV [146], a C++ aware forward-CFI implementation, which can thus defend against COOP, incurs an average geometric mean overhead of 4.0% on the SPEC CPU2006 C++ benchmarks using comparable optimization techniques. Dang et al. [53] report that a protected traditional shadow stack, necessary to defend against an attacker with arbitrary memory read/write capabilities, incurs an average overhead of 9.7% on all of CSPEC CPU2006 (excluding perl and gcc). Thus, the comparable overhead to fully protect against both traditional ROP attacks and COOP attacks using state-of-the-art CFI is 13.7%, in contrast to our lower overhead of 8.4% on the directly comparable set of C++ benchmarks or 6.0% on all of SPEC.

Chromium Browser

To test the performance impact of our protections on complex, real-world software, we compiled and tested the Chromium browser, which is the open-source variant of Google's Chrome browser. Chromium is a highly complex application, consisting of over 16 million lines of code [23]. We

were able to easily apply all our protections to Chromium with the few minor changes described below. Overall, we found that the perceived performance impact on web browsing with the protected Chromium, as measured by Chromium's internal UI smoothness benchmark, was 4.0%, which is in line with the average slowdown we observed for SPEC.

Since our protection system interferes with conventional stack walking, we had to disable debugging components of Chromium that use stack walking. We found that the optimized memory allocator used in Chromium, TCMalloc, uses stack tracing to provide detailed memory profiling information to developers. We disabled this functionality, which is not needed for normal execution. We also observed that Chromium gathers stack traces at tracing points during execution, again for debugging. Conveniently, we could disable this stack tracing with a single-line source code change. With these minor modifications we could compile and test the current development version[2] of Chromium with our LLVM-based Readactor++ compiler.

To understand the perceived performance impact during normal web browsing we benchmarked page scrolling smoothness with Chromium's internal performance testing framework. We ran the scrolling smoothness benchmark from the Chromium source tree on the Top 25 sites selected by Google as representatives of popular websites. This list includes 13 of the Alexa USA Top 25 sites including Google properties such as Google search, GMail and Youtube, Facebook, and news websites such as CNN and Yahoo. The Chromium scrolling benchmark quantifies perceived smoothness by computing the mean time to render each frame while automatically scrolling down the page. We report the average slowdown as time per frame averaged over 3 runs of the benchmark suite to account for random variance.

Overall, we found that the slowdown in rendering speed for our full Readactor++ system was about 4.0%, averaged over 3 different diversified builds of Chromium. This overhead is slightly lower than what we found for SPEC, which is natural considering that browser rendering is not

[2]Chromium sources checked out on 2014-11-04.

as CPU-intensive as the SPEC benchmarks. However, browser smoothness and responsiveness are critical factors for daily web browsing, rather than raw computing performance.

We also evaluated the performance impact of our techniques on Chromium using the extensive Dromaeo benchmark suite to give a worst-case estimate for browser performance. This suite, composed of 55 individual benchmarks, includes standard JavaScript benchmarks such as the Sunspider and V8 JavaScript benchmarks, as well as benchmarks that exercise DOM and CSS processing. Dromaeo is comprehensive, and hence, ideal to evaluate the overall impact of our protections on performance-critical browser components.

We found that execute-only code protection alone, without code-pointer hiding, introduced a 2.8% overall performance slowdown on Dromaeo. Combining the hypervisor execute-only code pages along with code-pointer hiding resulted in a 12% performance slowdown. We attribute this higher overhead to increased instruction cache pressure caused by our call pointer protection. However, Dromaeo represents a worst-case performance test, and rendering smoothness on real websites is a far more important factor in browsing.

3.7 Discussion

3.7.1 Limitations

We support all C++ programs that comply with the language specification and do not rely on compiler-specific vtable implementation details. Rare C++ programs which parse or modify their own vtables would need minor modification in order to handle our new split vtable layout. In practice we have not seen this issue, since vtables are not a part of the C++ standard at all and vary between compilers, e.g., Itanium-style vtables on Linux and MSVC vftables on Windows. Thus, compiler-agnostic programs should not rely on vtable structure.

Due to these application binary interface (ABI) incompatibilities, programs which import C++ library interfaces need to be compiled with the same C++ ABI version as the external library. Since we modify the vtable portion of the ABI to split vtables, we must also recompile any C++ dependencies with the same ABI. In practice, the only external dependency we found for Chromium or SPEC was the C++ standard library. We rebuilt the libc++ standard library with our modified compiler without any difficulty.

As mentioned in section 3.5.2, our implementation does not cover handwritten assembly or C++ exception handling. An industrial implementation of these techniques should apply code-pointer hiding to handwritten assembly and rewrite any necessary C++ exception handling tables to refer to call trampolines, which are located at the return address found on the stack.

Side-channel attacks may be a potential attack vector to de-randomize code protected by Readactor++. Seibert et al. [134] propose and discuss several potential side-channel attacks targeting diversity. However, the majority of these proposed attacks focus on disclosing code layout without a direct memory disclosure vulnerability, but do not consider execute-only memory preventing legitimate program functionality from its own code. Thus, the only class of attacks proposed by Seibert et al. that Readactor++ may still be vulnerable are those that blindly execute protected code and observe the side-effects of this execution (include potentially crashing the program). Therefore, for optimal security we require that a brute-forcing mitigation is in place to prevent the attacker from arbitrarily crashing and restarting the program.

3.7.2 Extensions

Our code pointer table protection protects the two main targets of function reuse attacks: dynamic linking tables and C++ vtables. However, similar tables sometimes exist in other contexts where dynamic dispatch is required. For example, C programs which emulate a variant of object orientation sometimes keep tables of function pointers to perform virtual dispatch,

depending on the type in question. Previous work has explored randomizing the layout of data structures [103], and these techniques could be extended to randomize structures or arrays of function pointers. In combination with code-pointer hiding, data structure randomization could protect these pointers from disclosure and reuse.

Dynamic loading of libraries via `dlopen` on Linux and analogous methods on Windows is a special case of dynamic linking. The `dlopen` function is used to load a new library at run time, after the program has started. The program can then use the `dlsym` function to retrieve the address of an exported function from the newly imported library. We can extend Readactor++ to randomize any C++ libraries imported during execution and update all relevant unmodified call sites referring to classes from the imported library.

While we only cover protection of ahead-of-time generated code in this dissertation, Crane et al. present a version of Readactor++ which also integrates disclosure protection for just-in-time (JIT) compiled code [51]. Code-pointer hiding and table randomization should also be integrated into JIT compilers in a similar manner. Homescu et al. propose an automatic, generic JIT-rewriting library [77] which we believe could be extended to integrate Readactor++ protections into JIT compilers without manual modification.

3.8 Conclusion

Numerous papers demonstrate that code randomization is a practical and efficient mitigation against code-reuse attacks. However, memory leakage poses a threat to all these probabilistic defenses. Without resistance to such leaks, code randomization loses much of its appeal. This motivates our efforts to construct a code randomization defense that is not only practical but also resilient to all recent bypasses.

We built a fully-fledged prototype system, Readactor++, to prevent attackers from disclosing the code layout directly by reading code pages and indirectly by harvesting code pointers from the data areas of a program. We prevent direct disclosure by implementing hardware-enforced execute-only memory and prevent indirect disclosure through code-pointer hiding.

Our careful and detailed evaluation verifies the security properties of our approach and shows that it scales beyond simple benchmarks to complex, real-world software such as Google's Chromium web browser and its V8 JavaScript engine. Compared to prior JIT-ROP mitigations, Readactor++ provides comprehensive and efficient protection against direct disclosure, is the first defense to address indirect disclosure, and is also the first technique to provide uniform protection for both statically and dynamically compiled code.

Chapter 4

Control-Flow Diversity

By altering his arrangements and changing his plans, he keeps the enemy without definite knowledge. By shifting his camp and taking circuitous routes, he prevents the enemy from anticipating his purpose.

— Sun Tzu, *The Art of War*

4.1 Motivation

Artificial software diversity, like its biological counterpart, is a highly flexible and efficient defense mechanism. Code injection, code reuse, and reverse engineering attacks are all significantly harder against diversified software ([43, 63, 118, 76, 153, 70, 47, 45]). We propose to extend software diversity to protect against side-channel attacks, in particular cache side channels.

Essentially, artificial software diversity denies attackers precise knowledge of their target by randomizing implementation features of a program. Because code reuse and other related attacks rely on *static properties* of a program, previous work on software diversity predominantly focuses on randomizing the program *representation*, e.g., the in-memory addresses of code and

data. Side-channel attacks, on the other hand, rely on *dynamic properties* of programs, e.g., execution time, memory latencies, or power consumption. Consequently, diversification against side channels must randomize a program's execution rather than its representation.

Most existing diversification approaches randomize programs before execution, e.g., during compilation, installation, or loading. Ahead-of-time randomization is desirable because re-diversification during runtime impacts performance (similar to just-in-time compilation). Some approaches interleave program randomization and program execution ([111, 70, 137, 77]). However, the granularity of randomization in these approaches is quite coarse, potentially allowing an attacker to observe the program uninterrupted for long enough to carry out a successful side-channel attack. We avoid this problem by extending techniques used to prevent reverse engineering such as code replication and control-flow randomization ([8, 47]). Unlike these approaches, however, we replicate code at a finer grained level and produce a nearly unlimited number of runtime paths by randomly switching between these replicas. Rather than making control flow difficult to reverse engineer, our technique randomly switches execution between different copies of program fragments, which we refer to as replicas, to randomize executed code and thus side-channel observations. We call this new capability *dynamic control-flow diversity*.

To vary the side-channel characteristics of replicas, we employ diversifying transformations. Diversification preserves the original program semantics while ensuring that each replica differs at the level of machine instructions. To protect against cache side-channel attacks we use diversifications that vary observable execution characteristics. Like other cache side-channel mitigations, such as reloading the cache on context switches and rewriting encryption routines to avoid optimized lookup tables, introducing diversity has some performance impact which we rigorously quantify in this paper.

In combination, dynamic control-flow diversity and diversifying transformations create binaries with randomized program traces, without requiring hardware or developer assistance. In this

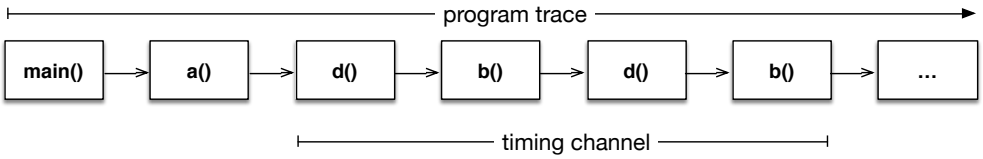

Figure 4.1: Time based side channel exploitable through a sequence of function calls in a program trace.

paper we explore the use of dynamic control-flow diversity against cache-based side-channel attacks on cryptographic algorithms. Our main contributions are the following:

- We apply the new capability of dynamic control-flow diversity to the problem of side channels. To the best of our knowledge, this is the first use of automated software diversity to mitigate cache side channels.
- We show how to generate machine code for efficient randomized control-flow transfers and combine this with a diversifying transformation to counter cache-based side-channel attacks.
- We present a careful and detailed evaluation of applying diversity to protect cache side channels and report the following:
  - **Security**: Our techniques successfully mitigate two realistic cache side-channel attacks against AES on modern hardware.
  - **Performance**: Applying dynamic control-flow diversity with effective security settings to an AES micro-benchmark of the libgcrypt library results in performance impacts of 1.75x and protecting a real-world application using AES results in a slowdown of 1.5x.

4.2 Side-Channel Background

The execution of a program is described by its control flow. The sequence of all control-flow transitions a program takes during execution is usually referred to as an execution path, or a *program trace*. A program trace describes the dynamic behavior of a program. figure 4.1 illustrates a program trace at the granularity of function calls.

Executing programs on real hardware results in dynamic properties that leak information, such as timing or power variation. For example, figure 4.1 shows a side channel based on time spent in executing the function sequence d(), b(), d(), b(). By observing dynamic properties of a program trace through a side channel, attackers can derive information about the actual program execution, such as inferring secret inputs to the program.

4.2.1 Threat Model

Since side-channel attacks often target secret keys of a process performing encryption, in this paper we assume that an attacker is targeting such a secret key. To demonstrate the applicability of our techniques, we assume an advantageous scenario for this attacker and reason that our defense remains effective under weaker assumptions.

Tromer et al. [148] classified side-channel attacks into synchronous and asynchronous attacks depending on whether or not the attacker can trigger processing of known inputs (usually plain- or cipher-texts). Synchronous encryption attacks, where the attacker can trigger and observe encryption of known messages, are generally easier to perform, and thus harder to defend against, since the attack does not need to determine the start and end of each encryption. We assume as strong a position for the attacker as possible and therefore will consider the scenario where an attacker can request and observe encryption of arbitrary chosen plaintexts.

To minimize external noise, we assume that the attacker is co-resident on the same machine as the target process. We also assume that the attacker can execute arbitrary user-mode code on a processor core shared with the target process but does not have access to the address space of the target process.

In the interest of allowing a strong attacker model, we advise but do not require that the protected binary be kept secret. Since we randomly generate diverse but semantically equivalent binaries, preventing the attacker from reconstructing the target environment is an advisable defense-in-depth against off-line attacks, such as the cross-VM attack described by Zhang et al. [159]. Deploying protected programs with differing layouts is also an effective defense against code-reuse attacks [78] and we can defend in the same manner by deploying randomized binaries which include dynamic control-flow diversity.

If we allow access to the binary, we must be careful that the attacker is not able to accurately determine which replica of each program unit was executed in an observed program trace. An attacker who observes a complete trace of control-flow transfers could filter out the effects of the replicas' diversifying transformations, regardless of what those effects are. In practice, a user-level process cannot observe all control-flow transfers of another process, especially at the granularity of basic blocks[1].

4.2.2 Example Attacks

To demonstrate an example of our dynamic control-flow diversity defense, we chose two synchronous, known input cache attacks on AES described by Tromer et al. [148]: EVICT+TIME and PRIME+PROBE. Although these representative cache attacks have limited scope, an attacker could use this type of attack to compromise a system-wide filesystem encryption key or target

[1] Gullasch et al. [74] describe a DoS attack against the OS scheduler which could result in such fine-grained information, but the OS scheduler can be hardened to prevent such attacks.

a proxy server where an attacker can trigger encryption of known plaintexts. In addition, these attacks are representative of cache-based side channels and are the basis of several more complex side-channel attacks [159, 155, 123]. While we demonstrate the effectiveness of our technique against cache-based side channels in particular, we expect that the same general defense paradigm can be applied to other categories of side channels using different diversifying transformations than the ones we discuss in section 4.3.1.

Caches exploit temporal and spatial locality to speed up access to recently used data. This helps to compensate for the speed gap between processors and main memories. As a side effect, caches increase the correlation between program inputs and its execution characteristics.

Modern processors access the cache in units called "cache lines," which are typically 64 bytes long. Each cache level is partitioned into n "cache sets," and each memory line can be placed into exactly one of these n sets. Each set stores at most m lines simultaneously, in which case the cache is called "m-way set associative". In practice, caches are 4-, 8-, 12- and 16-way associative. figure 4.2 shows the structure of a 3MB 12-way set associative cache found in our test system.

For efficiency, the processor shares these caches between running processes but prevents processes from accessing data belonging to other processes via the virtual memory abstraction. However, since data from multiple processes is concurrently stored in the cache, adversaries can indirectly deduce information about which cache locations a target process accesses by observing side-effects of cache accesses. Since the data cache access patterns of many programs are input-dependent and predictable, attackers can use knowledge of some inputs and the target's data access patterns to derive the secret input.

To exploit cache access patterns, all cache timing attacks rely on the same fundamental principle of cache behavior: accessing data stored in the cache is measurably faster than accessing the data from main memory. As a result, attacks can exploit this principle as a side channel and observe different cache behavior for certain segments of a program trace. In the EVICT+TIME

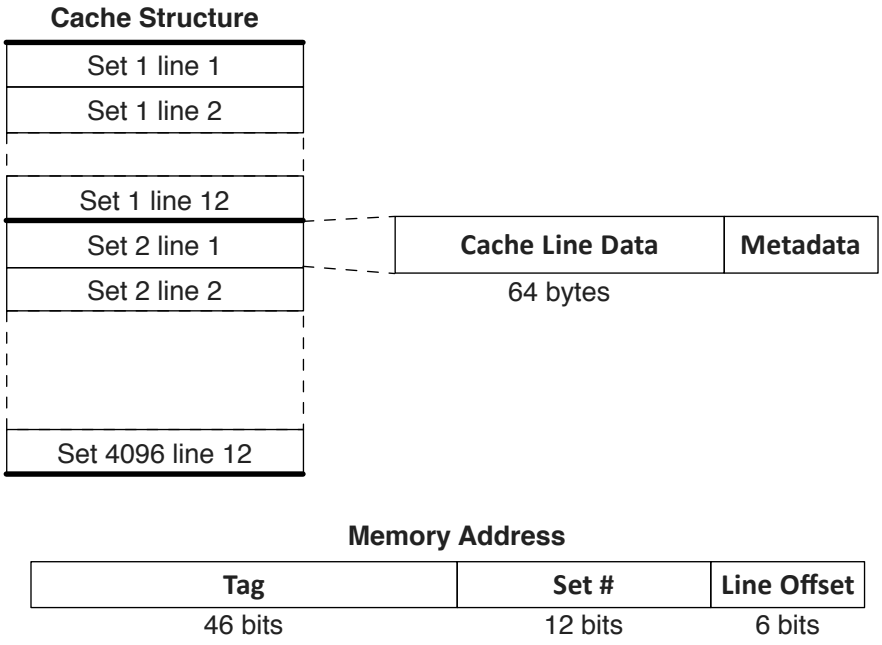

Figure 4.2: Example of cache structure on a modern processor. Cache shown is 3MB in size, with 4096 (2^{12}) sets, 12-way associativity and 64-byte cache lines. Memory addresses are broken into a 46-bit (or less) tag, a 12-bit set number and a 6-bit line offset.

attack, we observe the effect of evicting an entire cache set and forcing the encryption program to fetch values from main memory, while in the PRIME+PROBE attack we fill a cache set and check which cache lines the encryption evicts by observing the time to reload our data.

For convenience we summarize both AES attacks here but refer interested readers to Tromer et al. [148] for further details. Optimized AES implementations use four in-memory tables (T_0 through T_3, each containing 256 four-byte values) during encryption, and the access pattern of these tables varies according to the key and plaintext inputs. Specifically, during the first of ten encryption rounds for plaintext $\mathbf{p}$ and key k, the encryption process will access table T_l at index $p_i \oplus k_i$ for all $i = 0, \ldots, 15$ where $l = i \bmod 4$. Since we assume the attacker knows the plaintext $\mathbf{p}$, the attacker is able to derive information about the key from information about which table elements are loaded from memory.

Input : Cache set c to probe, plaintext p, key k.
Output : Time needed to encrypt the plaintext after probing c.

`Encrypt(`k, p`)`
Evict cache set c
$t_0 \leftarrow$`Time()`
`Encrypt(`k, p`)`
$t_1 \leftarrow$`Time()`
return $t_1 - t_0$

Algorithm 4.1: EVICT+TIME attack.

Input : Array C of cache sets to probe, plaintext p, key k.
Output : Array T of times needed to probe each set in C.

foreach $c \in C$ **do**
    Read w values into cache set c from memory
end
`Encrypt(`k, p`)`
foreach $c \in C$ **do**
    $t_0 \leftarrow$`Time()`
    Read w values from cache set c
    $t_1 \leftarrow$`Time()`
    $T[c] \leftarrow t_1 - t_0$
end
return T

Algorithm 4.2: PRIME+PROBE attack.

algorithm 4.1 shows the EVICT+TIME attack. We derive the table access patterns by observing the total execution time of the encryption routine. By first running the encryption on a chosen, random plaintext, we prime the cache with the table entries required during the encryption of this plaintext. We then completely evict a cache set by loading a set of memory locations that all map into the chosen cache set. By timing another encryption of the same plaintext, we can then, by averaging over many runs, determine whether the encryption used a table value from that cache set, since the encryption routine will take longer when accessing an evicted table entry due to the cache miss.

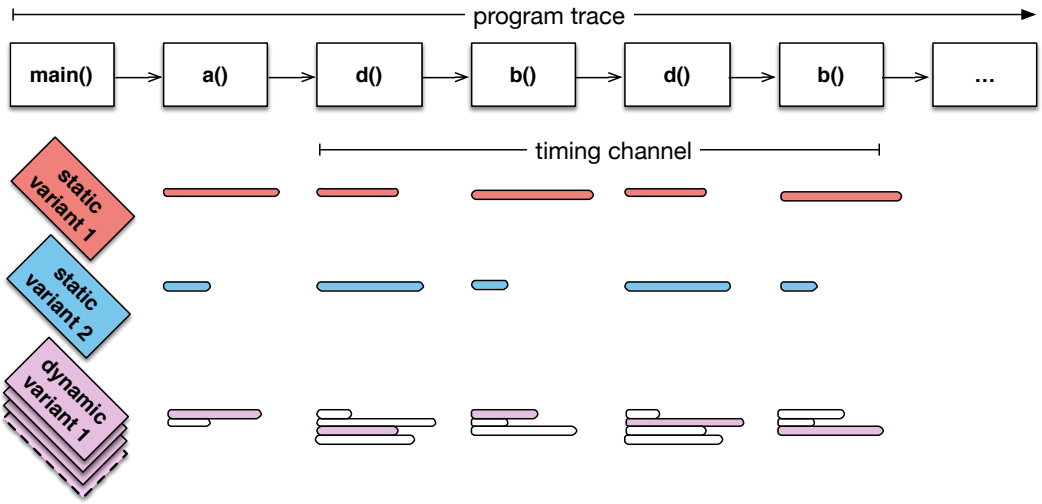

Figure 4.3: Side-channel resistance of diversification techniques.

The PRIME+PROBE attack (shown in algorithm 4.2) is very similar to the EVICT+TIME attack, but with the timing and eviction roles flipped. In this attack we first create a known starting cache state by loading a set of memory locations into each relevant cache set. We then trigger encryption of a chosen plaintext, which will modify this cache state by caching accessed table entries. Finally, we determine which cache sets were modified by timing a load of each cache set again. The cache sets corresponding to table entries that the encryption accessed will take longer to load than those not used, since the encryption table entry will have displaced one of the original entries loaded by the attacker and thus incur at least one cache miss.

By analyzing a large set of these cache observations for randomly chosen plaintexts, we can determine the key bits that correspond to table indices in the first round of encryption. For each guess of a key byte $\hat{k}_i$, we average all observed timings for the cache set evictions corresponding to table entry $T_{i \bmod 4}[\hat{k}_i \oplus p_i]$. In both attacks, the highest observed average time should correspond to the correctly guessed key byte. However, with 64 byte cache lines, four table entries fit into each cache line, and we can only observe accesses at the granularity of cache lines, which means that we can only determine the high nibble of each key byte with this analysis. Therefore, to determine the lower four bits of each key byte, we must analyze the second round of encryption

as described by Tromer et al. This analysis, while more involved, is conceptually analogous to the first round analysis and we refer interested readers to the description in the original paper.

4.3 Dynamic Control-Flow Diversity

Most diversification techniques prevent attackers from constructing reliable attacks by randomizing the layout of a program's data and code. Since modern exploits such as code reuse attacks depend on detailed knowledge of the program layout and internals, automatically modifying these aspects of the program implementation hinders development of reliable exploits using techniques such as return-oriented programming. However, software diversity affects not only program layout but also alters program side-effects, such as run time, power usage and cache usage. Even simply re-ordering functions can have a large effect on cache usage and performance since code will be aligned differently in the instruction cache.

Since software diversity affects performance and cache usage, by extension we observed that it could be useful to disrupt or add noise to side channels. However static compile-time or load-time diversity is insufficient, since side-channel attacks are online dynamic attacks and attackers can simply profile the static target binary to learn its runtime characteristics. Re-diversifying and switching to a new variant during execution is also insufficient since side-channel attacks are fast enough to complete between reasonably spaced re-diversification cycles. Figure 4.3 illustrates the effect of diversification techniques on side channels. While the program trace of the original program leaves a specific footprint on the executing hardware, diversified program variants (labeled as static variant 1 and static variant 2) each have a different footprint. This diversity is likely to thwart offline profiling attacks, but online side-channel attacks that deduce information by monitoring the running program are not affected by these diversification techniques.

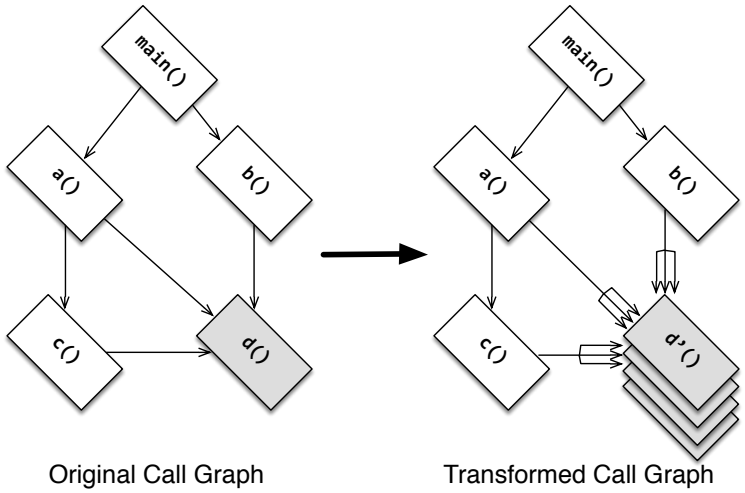

Figure 4.4: Function call graph before and after replicating the function d().

We extend previous, mostly static software diversification approaches by dynamically randomizing the control flow of the program while it is running. Rather than statically executing a single variant each time a program unit is executed, we create a program consisting of replicated code fragments with randomized control flow to switch between alternative code replicas at runtime.

In figure 4.3, we see the effect of dynamic control-flow diversity in the bottom row, labeled dynamic variant 1. For the trace segment the attacker is interested in, the program can now take numerous different paths, effectively preventing the attacker from constructing a reliable model to infer program execution information from side-channel characteristics, such as timing.

We build our control-flow diversity on a conventional compiler-based diversification system that creates randomized variants of a program fragment, such as a function or a basic block, by applying diversifying transformations. A diversifying transformation preserves program semantics but transforms implementation details. Examples of previously proposed diversifying transformations include insertion of NOP instructions, permutation of function or basic block layout, and randomization of register assignments. In section 4.3.1 we discuss a diversifying transformation to illustrate the effects of control-flow diversity on cache side channels, but other transformations could be used to protect against other instances and varieties of side-channel attacks.

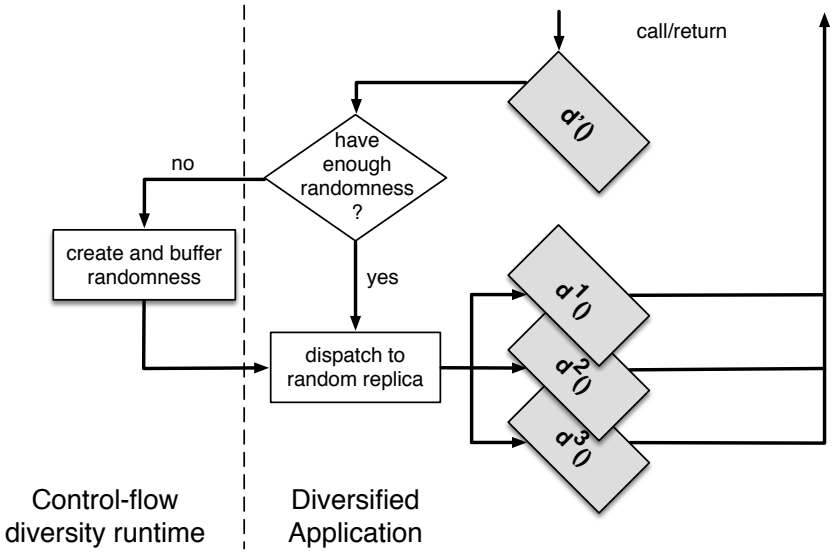

Figure 4.5: Detailed view of randomized trampoline d'() and interaction with the runtime system.

To create control-flow diversity, we begin by choosing a set of program fragments (either functions or basic blocks) to transform. If a developer knows that some sections of the program, such as encryption routines, are particularly interesting targets for side-channel attacks, the developer can manually specify this set of program fragments to diversify. In addition, for blanket coverage we can randomly select candidate program fragments. Since randomized control-flow transfers add performance overhead, the software distributor should adjust the percentage of duplicated fragments to balance security and performance.

After choosing a set of functions and/or basic blocks, we clone each chosen program fragment a configurable number of times. We then use different diversifying transformations for each clone to create functionally-equivalent replicas that differ in runtime characteristics. The set of transformations applied to each program fragment may include completely different transformations, applications of the same transformation with different parameters, or some combination of both. figure 4.4 shows an example of this process applied to a function.

We then integrate these randomized replicas into a program that dynamically chooses control-flow paths at runtime. For each replica, we replace all references to the original fragment with a

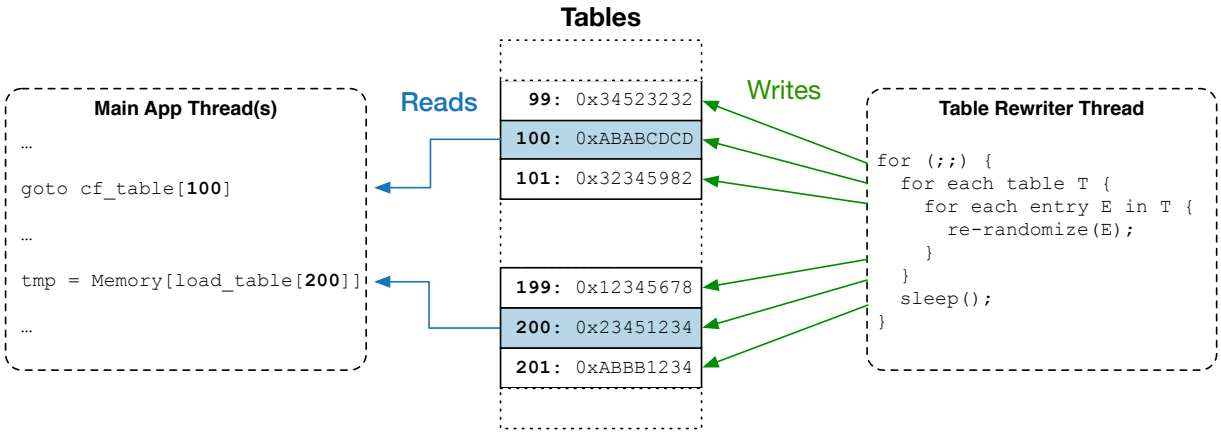

Figure 4.6: Memory layout of runtime address tables, along with pseudocode of the randomization algorithm. The randomization algorithm runs periodically in an infinite loop for the entire duration of the program.

randomized trampoline. As illustrated in figure 4.5, whenever the program executes a trampoline it randomly chooses a replica to transfer control to.

We use the SIMD-Oriented Fast Mersenne Twister pseudorandom number generator (PRNG) [129], since the runtime needs to quickly generate random numbers. Although our chosen PRNG is not cryptographically secure, it is sufficient for our purposes, since we assume the attacker cannot extract every control-flow transfer through the noisy side channel. If defending a side channel through which extracting the dynamic control flow and predicting the PRNG stream is easier than extracting the targeted secret information, this PRNG could easily be replaced by a cryptographically secure PRNG. Processor-integrated random number generators would be ideal to fill this role, and, as processors with this capability become widespread, we expect that the processor can fill a randomness buffer instead of using a software PRNG.

4.3.1 Cache Noise Transformation

In order to produce structurally different but semantically identical variants, we randomly apply diversifying transformations to the program code. These transformations change how a

program looks to an observer (who might either read the binary itself, or observe it through side channels) without affecting program semantics. We investigated one specific transformation, inserting cache noise, to disrupt cache side-channel observations. However, this technique is only one example of possible side-channel disrupting transformations. When protecting other side channels, one may need different transformations, e.g., disrupting power observations might require randomly weaving in another unrelated program to ensure that the inserted code is indistinguishable from the original program code.

We initially investigated disrupting the EVICT+TIME attack by randomly inserting NOP instructions into the code. However, after optimizing our randomness generation, we found that NOP instructions do not add enough time fluctuation to disrupt the attack. In addition, NOP instructions have no effect on cache usage, and thus do nothing to affect the PRIME+PROBE attack. We therefore turned our attention to inserting random memory loads, which disrupt both timing and cache snooping side channels.

To ensure that inserted loads have a high likelihood of actually impacting the performance of the targeted program, we want to create loads that evict a specific set of cache lines, specifically those that the target uses. In addition, attempting to read from invalid addresses (such as unallocated regions in the process address space) can potentially crash the target program, stopping the attack. For these reasons, we restrict the loads to a linear region, selected at program load-time. In the case of our AES experiments, this region covers only the AES S-box tables but in general is adjustable for other applications.

Our compiler randomly picks the locations to insert loads at compile time, and the target program itself picks the base and size of the region at load-time during program initialization. We leave the size of each load (in bytes) up to the implementation, and use single-byte loads in our evaluation. While implementing this cache diversification technique, we identified two ways of computing the address accessed by each load instruction: (i) static address and (ii) dynamic address computation.

In the first technique, static address computation, the compiler randomly picks an address (inside the range), then hard-codes it inside the program so the load is the same for every execution:

```
offset = 0x123 // Random constant < region_size
addr = region_base + offset
tmp = Memory[addr] // Volatile load
```

The second technique, dynamic address computation, loads addresses chosen dynamically while the program is running. We extend the same cached random tables used for control-flow diversity described below, and constantly rerandomize this table to contain valid addresses. This results in the following code for inserted load `i`:

```
addr = Memory[random_table[i]]
tmp = Memory[addr] // Volatile load
```

Static address computation requires the region size to be defined at compile time and a global variable `region_base` to be initialized at run time. The background thread for dynamic address computation randomly picks addresses for each table slot using global variables `region_base` and `region_size` that are initialized at run time.

4.3.2 Table Randomization Optimization

One of our main design goals was to make the randomized trampolines and memory loads be fast enough for practical usage. A naive initial implementation that called a random number generator for every control-flow transfer or memory operation proved to have unacceptably large overhead, even when buffering randomness. We instead chose to store branch targets and memory load addresses in tables and periodically re-randomize this table asynchronously in a background thread. At program startup we create a background thread that repeatedly iterates over all tables and randomizes each entry. Trampolines are then just a single indirect

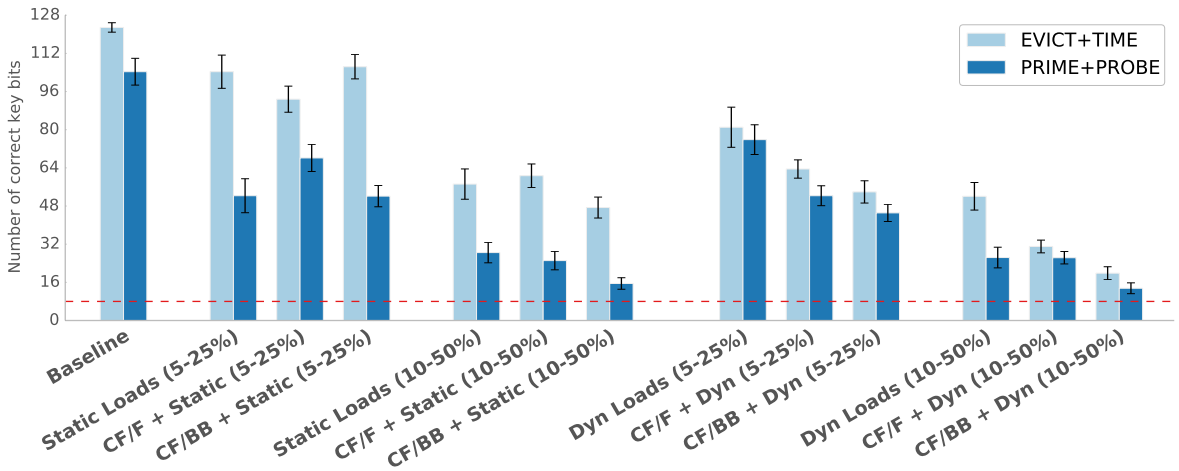

Figure 4.7: Average accuracy of AES side-channel attacks with our defense. The dashed line shows the expected number of correct bits for randomly chosen keys (8 bits). Error bars represent two standard errors from the mean.

branch through a control-flow cache table, while the memory loads require an extra load from the table. figure 4.6 shows the memory layout of the tables and the pseudocode of the table randomization algorithm which runs in the background thread.

Our dynamic control-flow transfer implementation could be further optimized to use inline caching and rewrite static branch instructions rather than an external table in data memory. However, branch targets are rerandomized frequently, so changing code page permissions from executable to writable and back may trump the performance improvement from inline caching. Alternatively, code pages could be left writable and executable, although this increases the risk of a code injection attack and may still be slow if instruction cache flushes are required.

4.4 Evaluation

To analyze the security and performance characteristics of our techniques in a real-world setting, we evaluated dynamic control-flow diversity as a defense for the two side-channel attacks proposed by Tromer et al. [148] and discussed in section 4.2.2. We implemented these attacks

targeting the AES-128 encryption routine in libgcrypt 1.6.1, which is the current version of the cryptographic library underlying GnuPG. Since our implementation does not currently support diversification of inline assembly, we disabled the assembly implementation of AES to force libgcrypt to use its standard C implementation. It is worth noting that this is an implementation limitation, and an industrial-strength implementation of our transformations could easily support rewriting of inline assembly as well.

To simplify our attack implementation, we made a slight change to the libgcrypt source code. We added an annotation to each of the targeted tables to force the compiler to align table entries such that no entries crossed a 64 byte cache line boundary. While both attacks could work around this alignment issue with further engineering effort, this change allowed us to more accurately measure the results of our protections.

We performed all security evaluations on an Intel Core 2 Quad Q9300 running Ubuntu 12.04 with Linux kernel 3.5.0. We targeted our attacks at the L2 cache of the processor; the Q9300 contains a 6MB L2 cache split into two halves, with each 3MB half being shared by two of the cores. The cache is 12-way set associative with 64-byte lines and 4096 sets. To minimize system interference, we stopped all unnecessary system daemons and pinned the attack to two cores, where the second core accommodated the background rewriting thread. In addition, to create the most advantageous situation possible for an attacker, our example attacks call the libgcrypt encryption function as a black box in the same process, rather than spawning or communicating with a separate process. Attacks in a more realistic setting would require even more observations to reliably extract the key, and our transformations would create additional uncertainty when coupled with the extra intra-process system noise.

Modern processors implement a cache prefetching algorithm that assumes spatial locality of cache accesses and speculatively loads additional cache lines that the prefetching unit expects might be accessed soon. Prefetching improves performance, especially for algorithms that access long linear regions of memory. However, prefetching negatively impacts cache-based

side-channel attacks by introducing the difficulty of determining which lines were loaded by the encryption algorithm and which by the prefetcher. For this reason, we disabled the prefetcher completely on our test machine by setting several configuration bits in machine status registers. While this slightly reduces the overall system performance, it makes attacks much more consistent.

We implemented all transformations and insertion of dynamic control-flow diversity as passes in version 3.3 of the Clang/LLVM compiler framework [101]. These new passes operate at the LLVM intermediate representation (IR) level, and are thus platform-independent.

4.4.1 Security Evaluation

After testing our example attacks, we empirically found that 5 million iterations of the EVICT+TIME attack and 75 thousand iterations of the PRIME+PROBE attack were sufficient to derive 96% and 82% of the random key bits on average, respectively. Although our attack implementation does not derive the full key in all cases due to random system noise and complex processor variations, this is an implementation concern, and an attacker would likely tune these attacks for increased accuracy.

To ensure that our baseline was accurate, we averaged 50 runs without any diversification, using a new random secret key for each iteration. Since our transformations rely on random choices during compilation, we tested each instance with ten different random seeds and each seed with five random keys (resulting in 50 runs total for each configuration) and report the average accuracy over all seeds and keys for each configuration.

To ensure that functions or basic blocks relevant to the AES encryption implementation were replicated, we manually inspected the libgcrypt implementation and selected nine functions that the program executes for every AES encryption. To collect comparable data for each experiment,

we configured our compiler to select all nine functions (or all basic blocks in the selected functions) for replication.

We fixed the number of generated replicas for each program fragment to ten in all cases. We found that further increasing this parameter had little effect on the attack success with the number of iterations we tested. However, increasing the replica count also had no measurable effect on runtime performance, and only a moderate effect on file size. Therefore adding additional variants may be a viable option to combat increasing attacker capabilities.

Security results for both the EVICT+TIME and PRIME+PROBE attacks are found in figure 4.7. We label the all static cache load variants with Static and the dynamic variants with Dyn. Control-flow diversity with function and basic-block replicas is labeled respectively with CF/F and CF/BB. We report key recovery in number of bits for clarity, however, it is important to note that both attacks derive the key in nibble-sized increments.

Static Loads

Static cache noise at a 5–25% insertion rate had little effect on the EVICT+TIME attack, resulting in 104–108 of 128 key bits recovered. Adding dynamic control-flow diversity to static noise also had little effect, since there is little timing variance between replicas when using static loads. Increasing this percentage to 10–50% had a more pronounced effect. More loads naturally imply that execution will be slower and thus more sensitive to cache collisions.

Dynamic control-flow diversity did have a significant effect on the PRIME+PROBE attack when combined with static cache loads. Function-level dynamic control-flow diversity reduced the correctly key recovered key bits from 52 with static loads to 41, and basic-block level replication furthered reduced this to 31 bits. With 10–50% cache noise insertion, we saw further reduction to 16 key bits correctly recovered using basic-block dynamic control-flow diversity.

Dynamic Loads

Dynamic loads had a larger effect on the EVICT+TIME attack. Dynamic cache noise alone at a 5–25% rate reduced the average correctly recovered key bits to 81. Adding dynamic control-flow diversity on top of this further reduced the recovered key bits to 64 and 54 for function-level and basic block-level diversity respectively. At the 10–50% insertion rate we observed similar trends, with CF/BB and dynamic loads reducing the EVICT+TIME key recovery to 20 bits. Dynamic cache loads naturally have a higher performance variation, since they require an extra indirect load to implement runtime dynamic randomness. This results in a more pronounced impact on the EVICT+TIME attack.

We observed similar trends for the PRIME+PROBE attack. While dynamic loads have some effect on the attack by themselves, they are most effective when combined with function or basic-block dynamic control-flow diversity. In the best case (CF/BB + Dyn) we observed an average correct key recovery of only 14 bits. This result is near the theoretical limit of 8 bits where an attacker gains no information from the side channel. Recovering 8 bits of the key is equivalent to an adversary randomly guessing the key by nibbles without side-channel information, since such an adversary has a 1 in 16 chance to guess each nibble correctly and each key nibble is independent for uniform random keys. This expected number of correctly guessed key bits with no knowledge is a lower bound on the accuracy of any side-channel attack, and we show this bound as a dashed line in figure 4.7.

Increasing samples

To investigate whether the attacks could feasibly overcome our defense by gathering more side-channel observations, we increased the iteration count for both attacks. We show these results in figure 4.8. We found that while the attack accuracy increased marginally with 4x and 8x the number of original attack measurements, a realistic attack is still infeasible. With

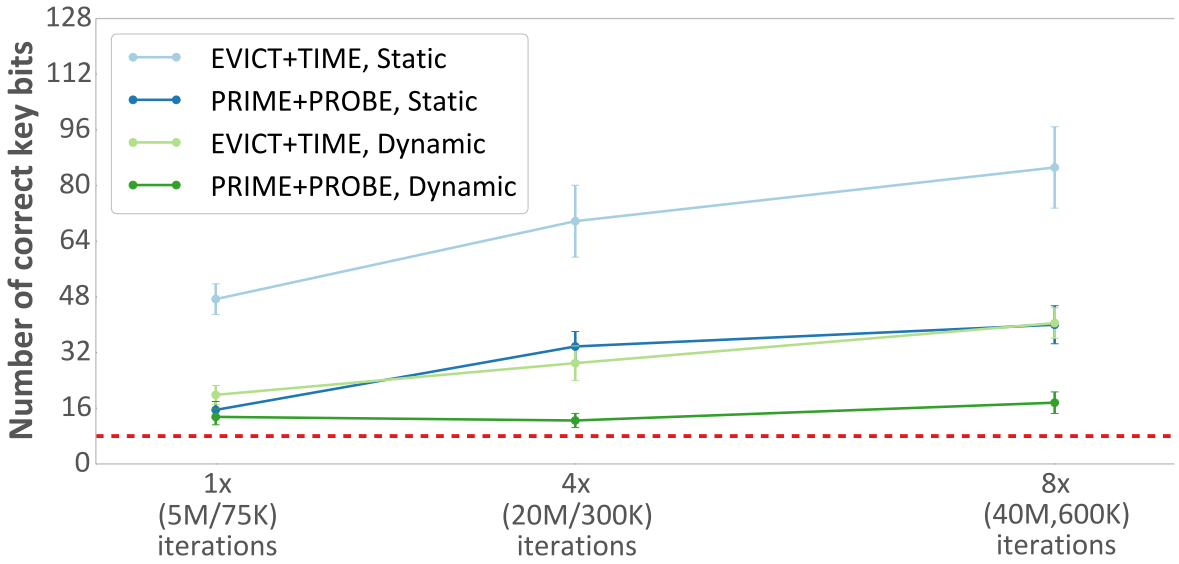

Figure 4.8: Average accuracy of AES side-channel attacks with our defense as the number of attack iterations increases. Error bars represent two standard errors from the mean.

the CF/BB + Static (10–50%) setting, 4x iterations resulted in average correctness of 70 bits for the EVICT+TIME attack and 34 bits for the PRIME+PROBE attack. 8x iterations resulted in 42 correct key bits on average for the PRIME+PROBE attack. We observed that dynamic cache-noise diversity was significantly more effective as attacker capabilities increased. With 8x iterations and dynamic diversity, the EVICT+TIME attack correctly identified only 40 key bits on average and the PRIME+PROBE attack was limited to only about 18 key bits, barely more effective than the lower bound of 16 bits. These results indicate that dynamic control-flow diversity is still effective in the presence of better resourced attackers, although it may require a different diversifying transformation to be more effective against the EVICT+TIME attack.

Collecting eight times more samples than in our baseline attack required about five minutes of attack time, resulted in a 1.5GiB data file, and analysis took about an hour on a high end, quad-core c3.xlarge Amazon EC2 instance. In a more realistic situation collecting many more samples than this is likely prohibitive. It is important to remember that our attack is simply encrypting a single block, with no inter-process communication or application overhead. Our tests represent a best-case scenario for an attacker. A realistic attack would target a service

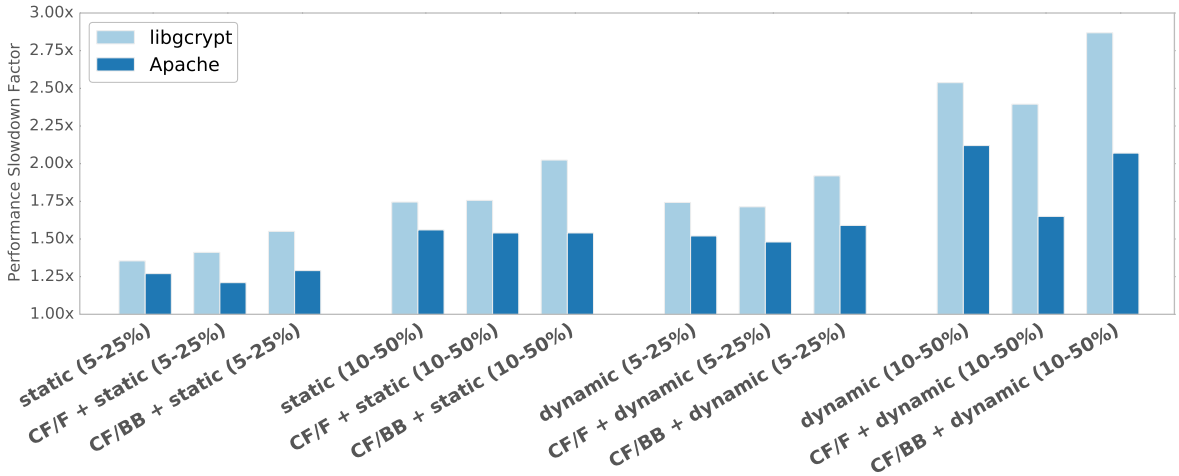

Figure 4.9: Performance slowdown factor. libgcrypt: AES micro-benchmark encryption performance slowdown, relative to a non-diversified baseline. Apache: slowdown when serving a 4MB file over HTTPS with AES block cipher, relative to a non-diversified baseline.

which is doing more work than our test attacks, and thus data collection would be far slower and noisier in practice.

4.4.2 Performance Evaluation

Most existing defenses against cache side-channel attacks, e.g., reloading sensitive tables into cache after every context switch or rewriting encryption algorithms to not use cached tables at all, introduce moderate overheads. Our transformations also marginally increase the cost of AES encryption. However we believe this overhead to be quite reasonable for an automated and general side-channel defense. To properly quantify this impact, we studied an AES micro-benchmark, a full-fledged service — Apache serving files over HTTPS using AES — and the SPEC CPU2006 benchmark suite.

From this performance analysis, in conjunction with attack success, we found that the optimal trade-off between security and performance is the CF/F + Static Loads setting. The CF/BB + Static Loads setting was slightly more effective, with only a small marginal decrease in performance, and is thus also an ideal candidate setting. Using dynamic loads, while slightly

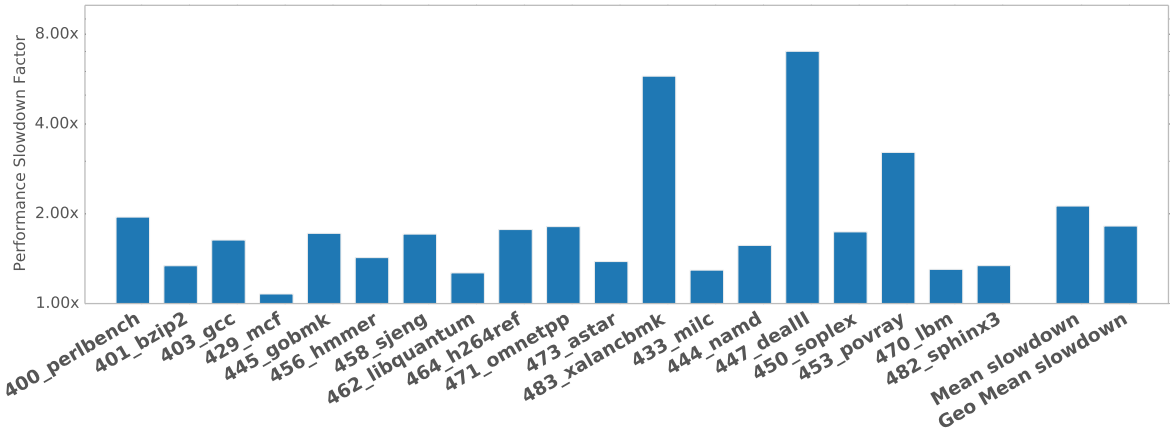

Figure 4.10: Performance slowdown factor for SPEC CPU2006 with function-level dynamic control-flow diversity on 25% of functions and 10–50% static cache noise inserted in all functions. Y-axis is on a log scale.

more effective, has a significantly larger performance impact for comparably little marginal security benefit.

AES Micro-benchmark

We first measured the increase in time introduced by each transformation with an AES micro-benchmark. We generated ten random different versions of libgcrypt for each set of parameters, ran each version of the AES encryption function five million times on random plaintexts for each of ten different random keys and measured the number of cycles for each encryption. The first column of each group in figure 4.9 shows the slowdown for the libgcrypt micro-benchmark.

We found that using function or basic-block level dynamic control-flow diversity along with static cache noise results in a performance slowdown of 1.76x–2.02x compared to the baseline AES encryption when using 10–50% cache noise insertion. Dynamic cache noise at a 5–25% rate results in similar performance, but 10–50% insertion of dynamic loads has significantly more impact on performance (2.39–2.87x slowdown).

Table 4.1: File size increase for libgcrypt, relative to a non-diversified baseline.

Transformation	File Size (KiB)	Increase Factor
Baseline	657	1.00
Static Loads (5-25%)	657	1.00
CF/F + Static (5-25%)	702	1.07
CF/BB + Static (5-25%)	716	1.09
Dyn Loads (5-25%)	658	1.00
CF/F + Dyn Loads (5-25%)	755	1.15
CF/BB + Dyn Loads (5-25%)	727	1.11
Static Loads (10-50%)	657	1.00
CF/F + Static (10-50%)	766	1.17
CF/F + Static (10-50%)	941	1.43
CF/BB + Static (10-50%)	737	1.12
CF/BB + Static (25@10-50%)	837	1.27
Dyn Loads (10-50%)	660	1.00
CF/BB + Dyn Loads (10-50%)	759	1.15
CF/F + Dyn Loads (10-50%)	784	1.19

In addition to measuring encryption time, we investigated the impact of our transformations on the size of the encryption library. While desktop disk space is currently plentiful, this is not the case for embedded or mobile systems. Many programs are also distributed over the Internet through communication links that have either bandwidth or data limits. section 4.4.2 shows the impact of our transformations on the size of the libgcrypt shared object.

Application Benchmark

In the previous section we measured the performance impact on AES encryption alone, encrypting a single block. However, to get a more realistic picture of the performance impact of our techniques, we also evaluated the performance overhead of dynamic control-flow diversity and our transformations on Apache 2.4.10 serving AES encrypted data. We used the standard apachebench (ab) tool to evaluate performance, connecting over https to an Apache instance using a diversified version of the OpenSSL 1.0.1 library[2].

[2]While we have not tested the effectiveness of the side-channel attack on this library, we believe it would take minimal effort to port the attack to OpenSSL or other table-based AES implementations.

As seen in the second column of each group in figure 4.9, the overall slowdown of our techniques varies from 1.25x for static cache noise to 2.1x for dynamic. The static noise CF/F and CF/BB settings in fact have identical overheads in this test, and we therefore recommend the CF/BB setting for practical applications which consist of more than just block cipher encryption. The overall performance impact is naturally lower than the simple micro-benchmark, since Apache does other processing in addition to encryption. However, this workload is more representative of a real-world application of cryptography and AES in particular.

SPEC CPU2006

To illustrate the effects of our techniques on CPU intensive workloads, we tested with the C and C++ portions of the SPEC CPU2006 benchmark suite. We selected one parameter setting: function-level dynamic control-flow diversity with static noise. However, since SPEC does not have any particular targets for cache side-channel attacks, we applied dynamic control-flow diversity universally over all functions with a 25% probability. We also applied static cache noise over all functions with a probability to insert noise for each instruction chosen randomly for each basic block from the range 10–50%. These parameters represent a worst-case for the CF/F + Dyn setting. To account for random choices, we built and ran SPEC with four different random seeds

As we show in figure 4.10, our transformations introduce a 1.82x geometric mean overhead across all benchmarks. The xalancbmk and dealII benchmarks stand out in this test. These particular benchmarks are large, complex C++ programs with many function calls. Since we applied function dynamic control-flow diversity across the entire program in this case, we naturally incur a higher overhead when the program calls many small functions. In practice users of dynamic control-flow diversity should target transformations in only the sections of code which might be vulnerable to a side-channel attack, instead.

4.5 Discussion

4.5.1 Parameter Settings

In our experiments we determined that a 5–25% insertion percentage range for cache noise instructions is too narrow. Dynamic control-flow diversity works best when replicas have very different runtime behavior, since it relies on switching between replicas with varying side-channel effects. In addition, libgcrypt is mostly straight line code and thus has a relatively low number of functions and basic blocks used for AES encryption. We expect that more complex cryptographic algorithms such as RSA will have more control flow, and thus more opportunity to insert dynamic control-flow diversity and switch between variants.

Cache noise, especially the dynamic variant, has an impact on execution time and thus the EVICT+TIME attack. However, this transformation is designed specifically to disrupt the PRIME+PROBE attack by polluting the cache and masking real AES table cache accesses. A transformation targeted at varying the running time of each replica would be more suited to disrupting this attack. We could adapt proposed hardware junk code insertion techniques [84, 6] to work with dynamic control-flow diversity by inserting differing code with varying runtimes into each replica.

In the best case, CF/BB + Dyn (10–50%), our EVICT+TIME attack can derive only 4.96 key nibbles, or about 20 key bits. Even with a more performance conscious alternative, CF/BB + Static (10–50%), we still prevent the attacker from finding 80 of 128 key bits. In the PRIME+PROBE attack our experiments show an average of 3.32 correctly recovered key nibbles, or 13.28 key bits, for the CF/BB + Static (10–50%) setting. The remaining approximately unknown key bits are too much to brute-force search, since this would require checking 2^n key guesses, where n is the number of unknown key bits. With this low correctness an attacker is unlikely to even

be able to determine which key nibbles are correct, and thus would gain no useful information from the attack. Thus, we conclude that our techniques effectively mitigate the PRIME+PROBE attack, given a realistic attack scenario.

We chose example parameters of ten replicas for each program unit along with 5–25 and 10–50 percent probability of inserting cache noise operations at each instruction as a starting point after initial experimentation. These parameters are representative of a narrow and wider range of insertion. However, these parameters may not represent an ideal trade-off between security and performance. In fact, these parameter settings are not mutually exclusive, e.g., some functions may be diversified with static noise while others get dynamic noise. Some combination of function and basic block replicas may also be useful for some applications. For future work, we propose to develop heuristics for automatic parameter selection through application and attack profiling.

4.5.2 Disabled Cache

Disabling caching of critical memory is an often suggested naive approach to preventing cache side-channel attacks [148]. This approach is attractive since existing commodity processors support selectively disabling page caching, but unfortunately it is prohibitively slow. To verify that this mitigation is impractical, we carefully measured the performance of the AES routine in libgcrypt with caching disabled for the AES lookup tables. This required writing a custom Linux kernel module to map and mark a page of memory as uncacheable using the Page Attribute Table (PAT) available on x86 CPUs. The user mode application, in this case libgcrypt, can then map this page into its address space and store the lookup table into it. This interface, while technically possible, is complex and not available in the standard Linux kernel.

We modified libgcrypt to utilize this approach and tested the same AES micro-benchmark described above. We found that disabling caching on only the single AES lookup page caused

the encryption routine to be 75 times slower than normal. Therefore disabling caching, even for a single page, is impractical on modern hardware. We discuss other related hardware based cache protections in subsection 5.4.2, however, these approaches are not available in commodity processors.

4.5.3 Implementation Limitations

For our initial investigation of applying control-flow diversity to side channels, we manually inspected the libgcrypt AES implementation to select nine functions relevant to the encryption algorithm. This simple step required no modification to the original sources, and could be easily automated by supplying only an encryption entry point. We forced our system to replicate these functions and their basic blocks to demonstrate the effectiveness of our techniques in a controlled environment, without the additional complication of having the system automatically select program units for diversification at random. However, this small manual effort was done to arrive at a controlled experiment and is not required to use control-flow diversity. By randomly selecting program units for replication with some configurable probability, our system can probabilistically protect an entire application from side-channel attacks with no manual effort.

Instead of random or manual program unit selection, we believe that side-channel analysis tools such as CacheAudit [60] can guide the selection of the critical program fragments and parameters for diversification. This should eliminate all manual effort while preserving a high level of security.

4.5.4 Related Attacks

Diversifying transformations, such as inserting cache noise instructions, can also be used to perform fine grained code layout randomization. This provides probabilistic protection against return-oriented programming and its variants which makes it realistic to expect that our defense technique can simultaneously defend against two or more fundamentally different classes of attacks. We will pursue this research direction in follow up work as well.

4.6 Conclusion and Outlook

We provide the first evaluation of software diversity as a side-channel mitigation. To that end, we developed dynamic control-flow diversity which performs fine-grained program trace randomization. Our technique does not require source code modification or specialized hardware so it can be automatically applied to existing software. We have implemented a prototype diversifier atop LLVM and rigorously evaluated the performance of our techniques using modern, realistic cache side-channel attacks in a setting that favors attackers. Our experimental evaluation shows that our technique mitigates cryptographic side channels with high efficacy and moderate overhead of 1.5–2x in practice, making it viable for deployment.

Beyond the cryptographic side-channel problem addressed in this paper, we expect that control-flow diversity is simultaneously effective against other implementation-dependent attacks, including code reuse and reverse engineering.

Chapter 5

Related Work

Nescire autem quid ante quam natus sis acciderit, id est semper esse puerum.

— Marcus Tullius Cicero, *Orator ad M. Brutum*

5.1 Code Reuse

5.1.1 Code-Reuse Attacks

Return-oriented programming. ROP, initially demonstrated by Shacham [135] in 2007 for the x86 architecture, is a generalization of the return-to-lib(c) attack described by Alexander Peslyak (Solar Designer) [140] in 2001 and an evolution of the "borrowed code chunks" technique described by Krahmer in 2005 [98]. Buchanan et al. [28] extended this in 2008 to a generalized version of ROP, targeting fixed-width instruction set architectures such as SPARC. Researchers have additionally extended ROP to many other processor architectures [64, 32, 97, 54]. Checkoway et al. [31] showed how any *Update-Load-Branch* instruction sequence can effectively

be used as a gadget terminator. Bletsch et al. [24] also further generalized ROP to gadgets terminating in any free branch, rather than only return instructions.

The original ROP proposal used a hand-picked set of gadgets found in the C standard library. To automate ROP attack creation, automated tools [81, 133, 79] scan a given binary, produce a set of useful gadgets, and optionally a payload that uses those gadgets. These tools build a database of gadgets that is, in most cases, Turing-complete. In reality, however, attacks require just enough gadgets and computational power to disable existing protections, such as W⊕X, and mount a more conventional shellcode injection attack.

Memory disclosure attacks. A major research direction in code-reuse attacks focuses on bypassing code randomization through memory disclosure. Such disclosure attacks first focused on disclosing the randomized base offset added by ASLR. Shacham et al. [136] demonstrated a brute-force attack exploiting the low entropy available in the 32-bit address space.

Linux binaries are in practice often not built as position-independent executables (PIE). Without PIE the executable itself contains gadgets at un-randomized locations, and Roglia et al. [126] demonstrated an attack to de-randomize ASLR protected libraries using these unprotected gadgets by reading the program linking structures at runtime.

Strackx et al. [143] proposed an even broader attack, buffer overreads, to completely break the assumption of code layout secrecy that randomization-based defenses depend on. By coercing the program into reading past the end of a buffer, the attacker can potentially read any valid memory location, thus bypassing all randomization of readable memory locations. Building on this attack approach, Snow et al. [139] improved upon the traditional static ROP attack by disclosing randomized code contents and creating a *JIT-ROP* payload at runtime, customized to the current code layout. Bittau et al. [22] demonstrated a brute-force attack to generically de-randomize memory contents of Unix services which respawn crashed threads.

5.1.2 Software Diversity Defenses

Manual software diversity has a long history in the fault-tolerant systems community. In 1977, Avizienis and Chen introduced *n-version programming*, [33] where multiple independent teams implemented the same program, potentially using different operating systems, languages, and algorithms. However, n-version programming is too costly for widespread adoption, and later work by Knight and Leveson in 1986 [94] identified that correlations may exist in the errors between multiple implementations.

Instead of manually introduced software diversity, researchers turned to automated software diversity to mitigate vulnerabilities. Cohen's seminal paper "Operating system protection through program evolution" [43] anticipates much of the development in what we now call artificial or automated software diversity. Consequently, it is safe to say that this work has motivated subsequent research in using automated software diversity and program randomization for security. Cohen describes several automatic program evolution techniques, e.g., adding garbage computation to an application using a source-to-source compiler. Forrest et al. [63] also advocated diversity to combat the currently insecure software monoculture.

Address Space Layout Randomization

The first randomization-based code-reuse defense to be widely adopted, ASLR, was pioneered in 2001 by the PaX project[1] for Linux. Since then it has been implemented in all major operating systems, and its usage is now standard practice. ASLR randomizes the starting virtual memory address of loaded segments during the loading process so that an attacker cannot easily redirect control flow into code segments normally located at a fixed, known address. However, ASLR has suffered from practical implementation problems and is insufficient to entirely prevent code

[1] `http://pax.grsecurity.net`

reuse. Shacham et al. [136] found that implementations of ASLR do not have enough entropy on 32-bit systems to properly prevent brute-force attacks which guess where code has been relocated. Additionally, ASLR requires that applications or libraries be compiled as position-independent so that they can be arbitrarily relocated in memory at load time. Many legacy applications which the user cannot recompile are therefore incompatible with ASLR, although most currently deployed dynamic libraries are now compatible. Additionally, unless all code in an application is position independent and protected with ASLR, attackers will simply target code-reuse attacks at any un-randomized portions of the code. In practice, while shared libraries are all compiled as position independent, the main binary of even a modern application on Linux is often compiled as position-*dependent* code and is therefore not relocatable. On 32-bit x86 systems, position-independent executables suffer a performance slowdown of 10% compared to standard, fixed address code, due to the scarcity of registers [120]. Thus, attackers can often find sufficient un-randomized gadgets in the main binary of non-trivial applications [133, 132].

In practice, even with full randomization of all executable memory, ASLR was shown to be bypassable [61, 136, 106, 35] due to its extremely course, segment-level granularity. Subsequent research on address space randomization has focused on randomizing program code at a finer granularity to raise the bar by forcing adversaries to disclose more than one code address.

Fine-Grained Diversity

Binary-rewriting diversification. Many proposed fine-grained diversity approaches use binary rewriting to randomize the application code at load-time. Load-time randomization allows protection of legacy and commercial applications and produces a freshly randomized variant each time the application is loaded. However, safe binary rewriting introduces additional runtime, since perfect disassembly and rewriting is not possible in general [80].

To defend against return-into-lib(c) attacks and malicious syscalls, Chew and Song [39] first suggested randomizing syscall numbering and library function locations. In 2003 Bhatkar et al. [18] proposed Address Obfuscation, a binary rewriting approach to randomize the location of not only program code but also data objects. Address Obfuscation randomizes the starting address of loaded segments (as in ASLR), permutes the order of variables and functions, and adds random padding between memory objects. Bhatkar et al. [20] later extended this approach to be more comprehensive. Kil et al. also proposed address space layout permutation (ASLP) [92], another fine-grained randomization approach which uses a binary rewriter and a patched kernel to randomize the program address space at a finer granularity (down to individual functions) with lower overhead. Binary Stirring [153] is another binary rewriting approach which permutes both functions and basic blocks within functions. Instruction Layout Randomization (ILR) [76] takes this progression to its extreme by randomizing the code layout at the level of individual instructions. Pappas et al. [118] also described a technique for introducing diversity at load-time, but their approach constrains diversification to techniques which do not reposition code.

Compile-time diversification. Producing randomized program variants at compile-time has also been a prominent research approach, since it offers the flexibility and safety to easily create randomized machine code without the need for disassembly and rewriting. In 2008 Jacob et al. [87] introduced the idea of a "superdiversifier," a compiler that performs superoptimization [104] for the purposes of increasing computer security. In 2010 Franz [65] suggested compiling a unique version of an application for every user and distributing these variants transparently via existing online distribution channels, e.g., mobile application stores. Giuffrida et al. [70] proposed a diversifying compilation scheme to protect operating systems from kernel level exploits. Their approach collects meta-data during compilation to allow re-randomization of kernel components while the system is running. In 2013 Homescu et al. [78] presented compilation techniques to introduce diversification using profiling feedback to reduce the performance impact of introducing additional `NOP` instructions.

Disclosure-resilient diversity

The JIT-ROP attack [139] capitalizes on the expansive threat surface of modern, complex applications by using a scripting language to control disclosure of randomized code, disassemble that code, and build a customized ROP payload on the fly. In the face of this attack, we require software diversity which is resilient to disclosure and disassembly of running code.

Hiding code addresses. In response to JIT-ROP, Backes and Nürnberger proposed Oxymoron [12] which randomizes the code layout at a granularity of 4KB memory pages. This has the additional benefit of being able to save memory by sharing code pages between multiple protected applications running in different virtual memory spaces on a single system. Oxymoron seeks to make code references in direct calls and jumps that span code page boundaries opaque to attackers. Internally, Oxymoron uses segmentation and redirects inter-page calls and jumps through a dedicated table randomly placed and hidden behind a segment register. This prevents direct memory disclosure, i.e., it prevents the recursive-disassembly step in the original JIT-ROP attack. Unfortunately, Oxymoron can be bypassed via indirect memory disclosure attacks as we have described in section 3.2.1.

Execute-only memory. Another defense against JIT-ROP, Execute-no-Read (XnR) by Backes et al. [11], is conceptually similar to Readactor++ as it is also based on execute-only pages. However, it emulates execute-only pages in software by keeping a sliding window of n pages as both readable and executable, while all other pages are marked as *non-present*. XnR does not fully protect against direct memory disclosure because a window of currently executing pages are still readable. Hence, the adversary can disclose function addresses (see section 3.2.1), and force XnR to map a target page as readable by calling the target function through an embedded script. This can be repeated until the disclosed code base is large enough to perform a JIT-ROP attack. In Readactor++, such an attack is not possible by design, because code pages are always

mapped execute-only and these permissions are constantly enforced in hardware. Further, it remains unclear how well XnR can protect against indirect memory disclosure. First, it only assumes a code randomization scheme without implementing and evaluating one. Second, as we have discussed in detail in section 3.4.2, defending against indirect code disclosure with randomization alone (i.e., without code-pointer hiding) requires XnR to use a very fine-grained and unpractical code randomization scheme.

HideM by Gionta et al. [69] also implements execute-only pages. Rather than supporting execute-only pages by unmapping code pages when they are not actively executing, HideM uses split translation lookaside buffers (TLBs) to direct instruction fetches and data reads to different physical memory locations. This allows HideM to effectively enforce execute-only page permissions in hardware. HideM allows instruction fetches of code but prevents data accesses except whitelisted reads of embedded constants. This is the same technique PaX used to implement W⊕X memory before processors supported RX memory natively. However, the split TLB technique does not work on recent x86 hardware because most processors released after 2008 contain unified second-level TLBs.

Other approaches. Davi et al. [55] also proposed a defense mechanism, Isomeron, that tolerates full disclosure of the code layout. To do so, Isomeron keeps two isomers (clones) of all functions in memory; one isomer retains the original program layout while the other is diversified. On each function call, Isomeron randomly determines whether the return instruction should switch execution to the other isomer or keep executing code in the current isomer. Upon each function return, the result of the random trial is retrieved, and if a decision to switch was made, an offset (the distance between the calling function f and its isomer f') is added to the return address. Since the attacker does not know which return addresses will have an offset added and which will not, return addresses injected during a ROP attack will no longer be used as is and instead,

the ROP attack becomes unreliable due to the possible addition of offsets to the injected gadget addresses.

Mohan et al. [105] presented Opaque CFI (O-CFI), a binary instrumentation-based solution which combines coarse-grained CFI with code randomization. Similar to Isomeron, the code layout is no longer a secret. O-CFI works by identifying a unique set of possible target addresses for each indirect branch instruction. Afterwards, it leverages the per-indirect branch set to restrict the target address of the indirect branch to only its minimal and maximal members. To further reduce the set of possible addresses, it arranges basic blocks belonging to an indirect branch set into clusters (so that they are located nearby to each other), and also randomizes their location.

Other Applications of Diversity

Automated diversity is also useful to mitigate attacks utilizing predictable data locations and representations. Researchers have proposed many approaches to randomize the locations or relative distances of stack frames, heap objects, and static data. Lightweight encryption of data values has also been proposed to randomize their stored representation in memory.

Data layout randomization. DieHard and DieHarder [16, 111] are randomizing memory allocators which secure the heap against memory management errors. All new memory allocations are placed randomly in memory and stay in place until their deallocation. DieHard recommends replicating execution with differently randomized heaps and comparing results. In DieHarder, freed pages are overwritten with random data to mitigate use-after-free attacks.

Lin et al. [103] suggest an approach called Data Structure Layout Randomization (DSLR) and implement it as part of GCC. In their DSLR system, the layout of structures, classes, and local stack variables are randomized and dummy fields are added.

Data representation randomization. Randomizing the representation of data in memory can prevent attackers from leaking or modifying objects. Cowan et al. [48] suggested encrypting all pointers with a simple XOR mask and decrypting pointers into registers before use. Bhatkar and Sekar [19] extended this approach by masking all data objects which might overflow, rather than pointers.

A related technique, constant blinding, protects against JIT spraying attacks [127]. JIT compilers often preserve constants in high-level scripting language statements as-is in generated machine code. By carefully constructing a sequence of operations with embedded constants, the attacker can predict and control the generated machine code. Constant blinding masks embedded constants by XORing the embedded, masked value with a random constant key before use.

Obfuscation. Research in obfuscation has strong ties to automated software diversity, and our work in control-flow diversity (see chapter 4) builds on approaches intended for obfuscation. Collberg et al. [44] first presented obfuscating transformations against reverse engineering attacks and introduced the notion of opaque predicates [46]. While opaque predicates usually refer to predicates that have a known outcome at obfuscation time but are expensive to decide afterward via static analysis, Collberg et al. also mention variable opaque predicates that flip-flop between true and false at runtime. These ideas were evaluated in depth by Anckaert et al. [8] as a defense against reverse engineering, by Collberg et al. [45] in context of client-side tampering of networked systems, and by Coppens et al. [47] to prevent reverse engineering of patches. Our work differs in its use of control flow randomization: we use it to switch among implementation variants (replicas) with fine-granularity—not as a randomizing transformation in itself. Furthermore, we aim to thwart side-channel attacks rather than reverse engineering.

5.1.3 Integrity Checking Defenses

Rather than tackling code reuse by randomizing program contents so they cannot be reused, integrity-checking defenses attempt to prevent malicious diversion of program execution.

Stack Integrity

StackGuard, by Cowan et al. [49] is a simple compile-time transformation to stack frames that catches traditional stack buffer overwrites attempting to rewrite the return address. StackGuard places a value called a *canary* on the stack between buffers on the stack and the return address. This canary must survive an integrity check when the function terminates before the function returns to its caller. Stackshield [150] is another approach that protects against return address overwrites by saving the return address to a location safe from overwrites and restoring it before the return instruction.

Control-flow integrity

In 2005 Abadi et al. [2, 3] proposed control-flow integrity (CFI), another prominent type of integrity checking defense which has spawned extensive research. Abadi et al. [1] later formalized the notion of CFI and showed that it can indeed constrain program control flow to a given control-flow graph. Ideally, CFI constrains each indirect branch such that it can only reach valid, statically identified targets from the program's correct control-flow graph. In practice, each possible control-flow destination is associated with a label, and each control-flow check ensures that the label associated with the runtime destination is an allowed destination for the given branch. However, for efficiency and practicality reasons, multiple destinations are often coalesced under a single label. It is often difficult to construct a complete, correct control-flow graph, especially in a binary rewriter.

We can classify CFI techniques based on which classes of indirect branches are checked. Many proposed CFI approaches concentrate on protecting *forward* branches, i.e., indirect jumps and calls, and rely on an auxiliary technique, such as a shadow stack, to protect *backwards* branches (returns).

Several CFI approaches (including the original CFI proposal) insert CFI checks using binary rewriting and use a single label for all possible backwards branch targets. CFI for COTS binaries [158] relies on static binary rewriting to identify all potential targets for indirect branches (including returns) and instruments all branches to go through a validation routine. CFI for COTS binaries enforces the policy that branch targets are either call-preceded or target an address-taken basic block.

Compact control-flow integrity and randomization (CCFIR) [157] is another coarse-grained CFI approach based on static binary rewriting. CCFIR collects all indirect branch targets into a *springboard* section and ensures that all indirect branches target a springboard entry. Although the layout of springboard entries is randomized to prevent traditional ROP attacks, CCFIR does not include countermeasures against JIT-ROP, and is therefore vulnerable to disclosure of the springboard. Similar to CFI for COTS binaries, CCFIR uses a set of three labels for CFI checks, adding an additional label for return sites from "sensitive" functions.

Recent research [71, 72, 29] has shown that these efficient but course-grained CFI approaches are insecure. Since CFI does not randomize the code layout, attackers can inspect the code layout ahead of time and carefully choose only valid call-proceeded gadgets and address-taken functions allowed by coarse-grained CFI policies.

SafeDispatch [88] and forward-edge CFI [146] are two compiler-based implementations of fine-grained forward CFI. The former prevents virtual table hijacking by instrumenting all virtual method call sites to check that the target is a valid method for the calling object. The latter work is a set of similar techniques which protect both virtual method calls and calls through

function pointers. It maintains tables to store trusted code pointers and halts program execution whenever the target of an indirect branch is not found in one of these tables. Even though both techniques add only minimal performance overhead, these approaches require additional protection to constrain function returns.

A number of recent CFI approaches specifically focus on analyzing and protecting vtables in binaries created from C++ code [67, 156, 121]. Although these approaches do not require source code access, their CFI policy is not as fine-grained as their compiler-based counterparts. A novel attack technique for C++ applications, *counterfeit object-oriented programming* (COOP) [130], undermines the protection of these binary instrumentation based defenses by invoking a chain of virtual methods through legitimate call sites to induce malicious program behavior.

Code-Pointer Integrity (CPI) was first identified as an alternative to CFI by Szekeres et al. [145] and was first implemented by Kuznetsov et al. [99]. CPI prevents the first step of control-flow hijacking during which the adversary overwrites a code pointer. In contrast, CFI verifies code pointers *after* they may have been overwritten. CPI protects control data such as code pointers, pointers to code pointers, etc. by separating them from non-sensitive data. The results of a static analysis are used to partition the memory space into a normal area and a safe region. Code pointers and other sensitive control data are stored in the safe region. The safe region also includes meta-data such as the sizes of buffers. Loads and stores that may affect sensitive control data are instrumented and checked using meta-data stored in the safe region. By ensuring that accesses to potentially dangerous objects, e.g., buffers, cannot overwrite control data, CPI provides spatial safety for code pointers. In 32-bit mode, CPI uses x86 segmentation to restrict access to the safe region. However segmentation is not enforced in 64-bit mode so the safe region is merely hidden from attackers whenever segmentation is not fully supported. Evans et al. [62] demonstrated that (in 64-bit mode) the size of the safe region is (in some implementations) so large that an attacker can use a corrupted data pointer to locate it and

thereby bypass the CPI enforcement mechanism using an extension of the side-channel attack by Siebert et al. [134].

Niu and Tan [110] proposed a CFI solution that also applies CFI to JIT-compiled code. Their RockJIT framework enforces fine-grained CFI policies for static code and coarse-grained CFI policies for JIT-compiled code. RockJIT maps the physical pages containing JIT-compiled code twice in virtual memory: once as readable and executable, and once as a readable and writable for the JIT compiler to modify—a *shadow code heap* accessible only to the JIT. This protects JIT-compiled code from code injection and tampering attacks, in addition to the protections provided by CFI enforcement. However, since only coarse-grained CFI is applied, the adversary can still leverage memory disclosure attacks to identify valid gadgets in JIT-compiled code and redirect execution to critical call sites in static code (i.e., calls that legitimately invoke a dangerous API function or a system call) to induce malicious program behavior.

Song et al. [141] demonstrated a multi-threaded attack vector on web browser JIT compilers that rewrites code while it is both executable and writable. To prevent modification of JIT-compiled code, they also proposed a split-process defense where the JIT compiler code cache is mapped as executable in the main browser process, and a specialized JIT compiler process handles all generation of code by writing to a shared, writable (but not executable) mapping of the code cache.

In 2010, Onarlioglu et al. presented G-Free, a set of techniques that "de-generalize the [ROP] threat to a traditional return-to-lib(c) attack" [112] by aligning all indirect control-flow transfers and checking them before dispatching. Their technique allows for comprehensive protection against (jump-) and return-oriented programming attacks at the expense of adding run-time checks to the secured programs. G-Free protects return addresses by XORing the return address with a random key at the beginning of each function and unmasking the address at the end of the function. Indirect jumps and calls are also protected by verifying a stack cookie before dispatching so that an indirect branch cannot be reused without computing the cookie at the

beginning of the function containing the branch. However, G-Free is not resilient to memory disclosure. If an attacker discloses these cookie values, he can forge valid cookies and reuse free branches.

Finally, Li et al. [102] presented a technique to remove all return instructions from a binary by rewriting returns as a table lookup of the return site corresponding to an index passed to the function. While this technique prevents traditional ROP using returns, it does not prevent other forms of code reuse.

5.1.4 Monitoring Defenses

Researchers have also proposed defensive approaches which monitor for characteristics of ROP to prevent these attacks. A number of these approaches have near-zero overheads because they use existing hardware features to constrain the control-flow before potentially dangerous system calls. In particular, x86 processors contain a last branch record (LBR) register which kBouncer [119] and ROPecker [38] use to inspect a small window of recently executed indirect branches for ROP-like characteristics. Similar policies are enforced by Microsoft's security tool EMET, which builds upon the research of ROPGuard [66]. However, recent attack research has shown that all proposed monitoring policies can be bypassed with carefully constructed attacks [71, 72, 29]. We expect that all policy-based monitoring defenses will either have false negatives, i.e., disallow some legitimate programs or false positives, i.e., be vulnerable to some attacks.

Frequent return detection by Chen et al. [34], aims to disrupt ROP by monitoring for frequent use of return instructions. However, Checkoway et al. [31] demonstrated a return-less code-reuse attack that thwarts Li et al.'s [102] and Chen et al.'s [34] defenses. In contrast to the conventional ROP attack which depends on RET instructions, their attack makes use of certain

instructions that behave like a return instruction. Similarly, *jump-oriented programming* [24] does not use the stack in any way and does not require that gadgets end in the `RET` instruction.

5.2 Active Defense

Much of the closest work in defenses which we term *active* has been in the realm of intrusion prevention and response systems, which are intrusion detection systems extended with techniques to actively mitigate or repair detected intrusions. Common commercial intrusion prevention systems often include functionality to automatically block network connections detected as malicious [10]. Recently, research intrusion response systems have included capabilities of attempting to trace intrusions to their sources [151]. There has also been considerable work in automatically choosing proper responses to intrusions [30, 13].

In 2011, Cui et al. [52] presented their symbiotes approach to secure embedded networked devices. Their approach injects symbiotes into Cisco routers which monitor the firmware while it is running. To create a symbiote, it is necessary to obtain a program binary and scan points where "code interception" is possible. A symbiote inserts itself at a randomly selected subset of these points and monitors the firmware operation of the router. Symbiotes serve a similar purpose to booby traps, but require execution overhead to monitor the process, rather than staying dormant until triggered by invalid program execution flow.

Most of the previous work in intrusion prevention and response systems is compatible with our techniques. Booby traps can directly serve as a replacement or enhancement to the intrusion detection portions of these systems. However, booby traps have the advantage of being a reliable source of attack information, since they can only be triggered by a program bug which attackers then exploit. We expect that existing IPS and IRS responses will be a good starting point to develop responses tailored to the advantages of booby traps specifically.

As specific examples of possible active responses, we highlight recovery and self-healing. There has been research into both developing patches to immunize against later attempts at the same exploit (such as [138]), as well as recovery to keep the compromised process running (such as [58]). Booby traps could directly trigger and use these responses.

Finally, there has been some work in creating active decoy documents to catch information exfiltration [25]. This is a similar to our booby traps, since this system attempts to actively track exfiltration when triggered. However, the document decoys attempt to prevent information loss through existing known channels, while we focus on protecting software against exploitation.

5.3 Hypervisors

Readactor++ uses an ultra-thin hypervisor to enable use of the Intel EPT feature. Previous researchers have utilized lightweight hypervisors in various security applications, however, to the best of our knowledge our use of a hypervisor layer for code-reuse protection is unique.

Security researchers have created several proof-of-concept hypervisor rootkits which switch a running operating system into a virtualized environment to control and hide from the operating system [93, 128]. Readactor++ takes the same approach to virtualization but cooperates with the existing OS to provide execute-only page protections.

Our hypervisor implementation and usage is also similar to that of Dune [15], which exposes hardware features to sandboxed user-level processes executing under a virtual CPU. Dune intercepts and forwards all necessary system calls back to the regular Linux operating system running outside a virtualized environment. Rather than exposing supervisor mode hardware access to user mode applications, we expose hypervisor-level hardware features to the operating system. However, we execute the entire operating system inside the virtualized environment to avoid incompatibility and overhead.

5.4 Side Channels

5.4.1 Side-Channel Attacks

Since Kocher described an initial timing side-channel attack on public-key cryptosystems [96], researchers have proposed a multitude of side-channel attacks against cryptographic algorithms. While researchers have proposed many different side-channel vectors ranging from power analysis [95] to acoustic analysis [68], we focus on applying our techniques against timing and cache-based attacks not requiring physical access. Cache-based attacks were first theoretically described by Page [115] in 2002. In 2003, Tsunoo et al. [149] demonstrated cache-based attacks against DES in practice. Bernstein [17] then presented a simple timing attack on AES, along with potential causes of this timing variability, including variable cache behavior and latency. Shortly after, Osvik, Shamir, and Tromer [114, 148] presented their attacks on AES, including the two example attacks used in this paper. In addition to the two synchronous attacks we evaluated our techniques against, Osvik et al. also described an asynchronous attack relying only on passively observing encryptions of plaintexts from a known but non-uniform distribution.

Recently, Hund et al. [82] used a cache-based timing side-channel attack to de-randomize kernel space ASLR in order to accurately perform code-reuse attacks in the kernel address space. Since we build our control-flow diversity on techniques proven to be effective against code-reuse attacks, our dynamic control-flow diversity with NOP insertion is a perfect fit to defend in depth against this attack.

As side-channel attacks have matured, researchers have proposed numerous defenses using both hardware and software. We will now briefly describe a few of the relevant defenses.

5.4.2 Hardware Defenses

Several different methods of preventing side channels at the hardware level have been proposed, with varying degrees of practicality. In the context of differential power analysis attacks, Irwin et al [84] proposed a new stage in the processor execution pipeline which randomly mutates the instruction stream with the assistance of a compiler-generated register liveness map. Among other peephole transformations, this mutation unit adds instructions that do not affect the correct functioning of the program, which are a super-set of our compiler-based NOP insertion transformation. Since our transformations in software are similar to the techniques Irwin et al. applied to differential power analysis, we expect that our technique will apply directly to power analysis attacks as well. Finally, Irwin et al. proposed a new probabilistic branch instruction, *maybe*, that would allow us to efficiently randomize control flow without the use of a random buffer. Ambrose et al. [6] also proposed inserting random instructions but with the added requirement that inserted instructions modify processor state, e.g., registers, so the new instructions are indistinguishable from legitimate program code.

To specifically target cache-based attacks, Page [117] proposed partitioning the cache into disjoint configurable sets so that a sensitive program cannot share cache resources with an attacker. However this would require a radical change to current cache designs. Bernstein [17] suggested the addition of a new CPU instruction to load an entire table into L1 cache and perform a lookup. This approach provides consistent cache access behavior regardless of input, and as such would eliminate cache side channels through table lookups. Wang and Lee [152] also proposed two new hardware cache designs to mitigate cache side channels: PLcache and RPcache. PLcache has the new capability of locking a sensitive cache partition into cache, while RPcache randomizes the mapping from memory locations to cache sets. While these techniques are powerful mitigations against cache side-channel attacks, they all require additional hardware

features which major processor vendors are unlikely to implement. In contrast, our techniques require no special hardware support and can be used immediately.

Intel has recently implemented a new hardware instruction to perform encryption and decryption for AES [73]. Since this instruction is data independent, using it instead of a software routine should protect against side-channel attacks on AES. However, this hardware only implements AES, and thus we still need defensive measures to protect other cryptographic algorithms.

5.4.3 Software Defenses

The ideal defense against side-channel attacks is to modify the sensitive program so that it has no input-dependent side-effects, however this is an extremely labor-intensive solution and is often infeasible. Developers generally take this approach to removing timing side channels by creating algorithms that run in constant-time regardless of inputs. Bernstein [17] strongly recommends this approach, while cautioning that software which the programmer expected to run in constant time may not do so due to hardware complexity.

Page [116] suggested manually adding noise to encryption to make cache side-channel attacks more difficult in a manner conceptually similar to our automatic randomizing transformations. For instance, Page manually inserted garbage instructions and random loads into the encryption routine to combat timing and trace based attacks respectively. Page's work is a form of obfuscation rather than diversification since all users run the same binaries with the same runtime control-flow. Our combination of control flow randomization and garbage code insertion *simultaneously* defends against code reuse attacks and side channels whereas garbage code in itself does not protect against side channels and Page's transformations do not protect against code reuse.

Brickell et al. [26] proposed the use of compressed and randomized tables for AES that would alleviate cache-based attacks. However, this implementation process requires manually rewriting the AES implementation and is specific to the operation of AES.

Cleemput et al. [40] proposed defenses that do not require manual program modification. In particular, they described the use of compiler transformations to reduce timing variability. Our approach, while also compiler-based, seeks to mask variability rather than remove it entirely, since opportunities to automatically eliminate variable-time routines are limited.

In their recent paper addressing side-channel attacks in the context of virtualized cloud computing, Zhang and Reiter [160] proposed periodically scrubbing shared caches used by sensitive processes. This scheme potentially breaks cache snooping by a time-shared process on the same core, but will not necessarily combat cache attacks in a Simultaneous Multithreading (SMT) context or continuous power analysis attacks. Since our random decision points are more fine grained than the scrubbing interval, our techniques have greater potential against these fine-grained attacks, although this would require more investigation. In addition, control-flow diversity does not depend on any resources outside the program and is thus applicable in situations without hypervisors, such as embedded software.

Finally, Tromer et al. [148] mention adding noise to memory accesses with spurious accesses to decrease the signal available to the attacker. Effectively, our technique accomplishes this goal in a general way that could be extended to other side channels, and we provide a concrete evaluation showing its effectiveness in practice. Since adding replicas exponentially increases the number of possible execution traces, we can ratchet our defense up sufficiently so that an attacker cannot feasibly collect and analyze enough samples.

Chapter 6

Conclusion

So long and thanks for all the fish.
— Douglas Adams, *Hitchhiker's Guide to the Galaxy*

Extensive existing research in software diversity has shown that it has the potential to be a valuable paradigm in hardening software against widespread attacks. However, recent attacks have cast doubt on the resilience of diversity against targeted, determined attacks. We have shown that not only can code randomization be hardened against such attacks, but it can actually be cheaper than comparable protections against memory-disclosure code-reuse attacks while also being more widely applicable to other attacks.

Our software booby traps give defenders an additional tool for active defenses. Booby traps effectively mitigate many brute-forcing attacks, but also serve as a reliable trigger for active defensive mechanisms, which we propose and discuss.

Building on the foundation of software diversity and booby traps, we present Readactor++ which fully protects randomized code from run-time disclosure attacke. Readactor++ enforces execute-only protection in hardware on current commodity processors. We also describe two

novel transformations, code-pointer hiding and table randomization, which protect against indirect code disclosure and vtable reuse respectively. Our complete protection system incurs reasonable overhead (6.0%) and scales to complex, real-world software.

Finally, we propose a new application of software diversity techniques to mitigate side-channel attacks. Using dynamic control-flow diversity, we can create dynamic run-time variation of program side effects while using static code.

These new techniques strengthen and extend the reach of automated software diversity. We hope that researchers continue to investigate diversity as a generic software defense paradigm and look into additional applications of this paradigm.

Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*, pages 340–353, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM '05)*, 2005.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4:1–4:40, Nov. 2009.

[4] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In M. Abe, editor, *Topics in Cryptology – CT-RSA 2007*, number 4377 in Lecture Notes in Computer Science, pages 225–242. Springer Berlin Heidelberg, Jan. 2006.

[5] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, Issue 49, 1996.

[6] J. A. Ambrose, R. G. Ragel, and S. Parameswaran. RIJID: random code injection to mask power analysis based side channel attacks. In *Proceedings of the 44th Design Automation Conference (DAC '07)*, pages 489–492. ACM, 2007.

[7] AMD. Intel 64 and IA-32 architectures software developer's manual - Chapter 15 Secure Virtual Machine nested paging. `http://developer.amd.com/resources/documentation-articles/developer-guides-manuals`, 2012.

[8] B. Anckaert, M. Jakubowski, R. Venkatesan, and K. D. Bosschere. Run-time randomization to mitigate tampering. In *Proceedings of the 2nd International Workshop on Security (IWSEC '07)*, pages 153–168, 2007.

[9] R. Anderson. Why information security is hard - an economic perspective. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC '01)*, pages 358–365, 2001.

[10] N. B. Anuar, M. Papadaki, S. Furnell, and N. Clarke. An investigation and survey of response options for intrusion response systems (IRSs). In *Proceedings of the 10th Annual Information Security for South Africa Conference (ISSA '10)*, 2010.

[11] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS '14)*, 2014.

[12] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[13] I. Balepin, S. Maltsev, J. Rowe, and K. Levitt. Using specification-based intrusion detection for automated response. In *Proceedings of the 6th Interntional Symposium on Recent Advances in Intrusion Detection (RAID '03)*, number 2820 in Lecture Notes in Computer Science, pages 136–154, 2003.

[14] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, 2005.

[15] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazieres, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 335–348, 2012.

[16] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, pages 158–168, 2006.

[17] D. J. Bernstein. Cache-timing attacks on AES. Preprint, 2005.

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

[19] S. Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA '08)*, 2008.

[20] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, 2005.

[21] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In B. S. K. Jr, editor, *Advances in Cryptology — CRYPTO '97*, number 1294 in Lecture Notes in Computer Science, pages 513–525. Springer Berlin Heidelberg, 1997.

[22] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P '14)*, 2014.

[23] Black Duck Software, Inc. Chromium Project on Open Hub. `https://www.openhub.net/p/chrome`, 2014.

[24] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*, pages 30–40, 2011.

[25] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. In Y. Chen, T. D. Dimitriou, and J. Zhou, editors, *Security and Privacy in Communication Networks*, number 19 in Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 51–70. Springer Berlin Heidelberg, 2009.

[26] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, 2006.

[27] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, 2005.

[28] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*, pages 27–38. ACM, 2008.

[29] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[30] C. Carver, J. M. Hill, J. R. Surdu, and U. W. Pooch. A methodology for using intelligent agents to provide automated intrusion response. In *Proceedings of the IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop, West Point, NY*, pages 110–116, 2000.

[31] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 559–72, 2010.

[32] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. In *Proceedings of the 4th Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, 2009.

[33] L. Chen and A. Avizienis. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS '95)*, 1995.

[34] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie. Efficient Detection of the Return-oriented Programming Malicious Code. In *Proceedings of the 6th International Conference on Information Systems Security (ICISS '10)*, pages 140–155, 2010.

[35] X. Chen. ASLR bypass apocalypse in recent zero-day exploits. FireEye Threat Research Blog, Oct. 2013.

[36] X. Chen and D. Caselden. CVE-2013-3346/5065 technical analysis. FireEye Threat Research Blog, Dec. 2013.

[37] X. Chen, D. Caselden, and M. Scott. The dual use exploit: CVE-2013-3906 used in both targeted attacks and crimeware campaigns. FireEye Threat Research Blog, Nov. 2013.

[38] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[39] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, 2002.

[40] J. V. Cleemput, B. Coppens, and B. De Sutter. Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):23:1–23:20, Jan. 2012.

[41] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ application binary interface (ABI). Technical report, 2001.

[42] F. Cohen. A note on the role of deception in information protection. *Computers & Security*, 17(6):483–506, 1998.

[43] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, Oct. 1993.

[44] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, 1997.

[45] C. S. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*, pages 319–328, 2012.

[46] C. S. Collberg, C. D. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL '98)*, pages 184–196, 1998.

[47] B. Coppens, B. De Sutter, and J. Maebe. Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):24:1–24:26, Jan. 2013.

[48] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard™: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[49] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[50] S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Booby trapping software. In *Proceedings of the 2013 Workshop on New Security Paradigms (NSPW '13)*, 2013.

[51] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P '15)*, 2015.

[52] A. Cui, J. Kataria, and S. J. Stolfo. From prey to hunter: transforming legacy embedded devices into exploitation sensor grids. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*, pages 393–402, 2011.

[53] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS '15)*, pages 555–566, 2015.

[54] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-oriented programming without returns on ARM. *System Security Lab-Ruhr University Bochum, Tech. Rep*, 2010.

[55] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[56] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[57] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS '13)*, 2013.

[58] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 78–95, 2003.

[59] D. Dittrich and K. E. Himma. Active response to computer intrusions. In *The Handbook of Information Security*, volume III. 2005.

[60] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22nd USENIX Security Symposium*, pages 431–446, 2013.

[61] T. Durden. Bypassing PaX ASLR protection. *Phrack*, 11(59), July 2002.

[62] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point: On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P '15)*, 2015.

[63] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HOTOS '97)*, pages 67–72, 1997.

[64] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*, pages 15–26, 2008.

[65] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms (NSPW '10)*, pages 7–16, 2010.

[66] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. Technical report, 2012.

[67] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*, 2014.

[68] D. Genkin, A. Shamir, and E. Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology (CRYPTO '14)*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2014.

[69] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY '15)*, 2015.

[70] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Security Symposium*, pages 475–490, 2012.

[71] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P '14)*, 2014.

[72] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium*, 2014.

[73] S. Gueron. Intel advanced encryption standard (AES) instructions set. Technical report, Intel, 2010.

[74] D. Gullasch, E. Bangerter, and S. Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P '11)*, pages 490–505, 2011.

[75] J. L. Henning. SPEC CPU2006 memory footprint. *SIGARCH Computer Architecture News*, 35, 2007.

[76] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12)*, pages 571–585, 2012.

[77] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. librando: Transparent code randomization for just-in-time compilers. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS '13)*, pages 993–1004, 2013.

[78] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automatic software diversity. In *Proceedings of the 11th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13)*, pages 1–11, 2013.

[79] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT '12)*, pages 64–76, 2012.

[80] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, Aug. 1980.

[81] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, pages 383–398, 2009.

[82] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P '13)*, 2013.

[83] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Chapter 28 VMX Support for address translation*.

[84] J. Irwin, D. Page, and N. Smart. Instruction stream mutation for non-deterministic processors. In *Proceedings of the 13th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '02)*, pages 286–295, 2002.

[85] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Diversifying the software stack using randomized NOP insertion. In S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense II*, volume 100 of *Advances in Information Security*, pages 151–173. Springer New York, 2013.

[86] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. In S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense*, number 54 in Advances in Information Security, pages 77–98. Springer New York, 2011.

[87] M. Jacob, M. Jakubowski, P. Naldurg, C. Saw, and R. Venkatesan. The superdiversifier: Peephole individualization for software protection. In *Proceedings of the 3rd International Workshop on Security*, volume 5312 of *Lecture Notes in Computer Science*, pages 100–120. Springer Berlin Heidelberg, 2008.

[88] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[89] V. Jayaswal, W. Yurcik, and D. Doss. Internet hack back: counter attacks as self-defense or vigilantism? In *2002 International Symposium on Technology and Society, (ISTAS'02)*, pages 380–386, 2002.

[90] A. Juels and R. L. Rivest. Honeywords: Making password-cracking detectable. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS '13)*, pages 145–160, 2013.

[91] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*, pages 272–280, 2003.

[92] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*, 2006.

[93] S. T. King, P. M. Chen, Y. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P '06)*, 2006.

[94] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12:96–109, 1986.

[95] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology (CRYPTO '99)*, number 1666 in Lecture Notes in Computer Science, pages 388–397. Springer Berlin Heidelberg, Jan. 1999.

[96] P. C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology (CRYPTO '96)*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer Berlin Heidelberg, Jan. 1996.

[97] T. Kornau. *Return oriented programming for the ARM architecture*. Master's thesis, Ruhr-Universität Bochum, 2010.

[98] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation techniques. Technical report, Sept. 2005.

[99] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, 2014.

[100] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P '14)*, 2014.

[101] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO '04)*, pages 75–87, 2004.

[102] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 195–208. ACM, 2010.

[103] Z. Lin, R. D. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *Proceedings of the 6th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA~'09)*, pages 107–126, 2009.

[104] H. Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the 2nd International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126, 1987.

[105] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[106] T. Müller. ASLR Smack and Laugh Reference. Technical report, RWTH Aachen, Germany, Feb. 2008.

[107] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pages 245–258, 2009.

[108] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*, pages 31–40, 2010.

[109] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 58, 2001.

[110] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS '14)*, 2014.

[111] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*, pages 573–584, 2010.

[112] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*, pages 49–58, 2010.

[113] H. Orman. The morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, (5):35–43, 2003.

[114] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval, editor, *Topics in Cryptology (CT-RSA '06)*, number 3860 in Lecture Notes in Computer Science, pages 1–20. Springer Berlin Heidelberg, Jan. 2006.

[115] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169, 2002.

[116] D. Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, 2003.

[117] D. Page. Partitioned cache architecture as a side-channel defence mechanism. Cryptology ePrint Archive, Report 2005/280, 2005.

[118] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12)*, pages 601–615, 2012.

[119] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[120] M. Payer. Too much PIE is bad for performance. Technical report, ETH Zurich, Department of Computer Science, 2012.

[121] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[122] G. Richarte and u. others. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[123] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pages 199–212, 2009.

[124] rix. Smashing C++ VPTRS. *Phrack*, 56, 2000.

[125] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security (TISSEC)*, 15(1):2:1–2:34, Mar. 2012.

[126] G. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09)*, pages 60–69, Dec. 2009.

[127] C. Rohlf and Y. Ivnitskiy. Attacking clientside JIT compilers. In *Black Hat USA Briefings*, 2011.

[128] J. Rutkowska and A. Tereshkin. IsGameOver() anyone? In *Black Hat USA Briefings*, 2007.

[129] M. Saito and M. Matsumoto. SIMD-Oriented fast mersenne twister: a 128-bit pseudo-random number generator. In A. Keller, S. Heinrich, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer Berlin Heidelberg, Jan. 2008.

[130] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P '15)*, 2015.

[131] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '14)*, 2014.

[132] E. Schwartz. The danger of unrandomized code. *USENIX; login*, 36(6):97, 2011.

[133] E. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 19th USENIX Security Symposium*, 2011.

[134] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS '14)*, 2014.

[135] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pages 552–561, 2007.

[136] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, pages 298–307, 2004.

[137] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu. Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*, pages 309–318, 2012.

[138] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. *Proceedings of the 2005 USENIX annual technical conference*, pages 149–161, 2005.

[139] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P '13)*, 2013.

[140] Solar Designer. Getting around non-executable stack (and fix). Email to the Bugtraq mailing list, Aug. 1997.

[141] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[142] L. Spitzner. The honeynet project: trapping the hackers. *IEEE Security Privacy*, 1(2):15–23, 2003.

[143] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the 2nd European Workshop on System Security (EUROSEC '09)*, pages 1–8, 2009.

[144] B. D. Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. D. Bosschere. Instruction set limitation in support of software diversity. In *Proceedings of the 11th Internation Conference on Information Security and Cryptology (ICISC 08)*, number 5461 in Lecture Notes in Computer Science, pages 152–165. 2009.

[145] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P '13)*, pages 48–62, May 2013.

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

[147] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th Interntional Symposium on Recent Advances in Intrusion Detection (RAID '11)*, number 6961 in Lecture Notes in Computer Science, pages 121–141. Jan. 2011.

[148] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, Jan. 2010.

[149] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of the 2003 Workshop on Cryptographic Hardware and Embedded Systems (CHES '03)*, number 2779 in Lecture Notes in Computer Science, pages 62–76. Jan. 2003.

[150] Vendicator. StackShield: A "stack smashing" Technique Protection Tool for Linux. `http://www.angelfire.com/sk/stackshield/`, 2000. Last accessed July, 2015.

[151] X. Wang, D. S. Reeves, S. F. Wu, and J. Yuill. Sleepy watermark tracing: An active network-based intrusion response framework. In *Proceedings of the 16th International Conference on Information Security*, pages 369–384, 2001.

[152] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA '07)*, pages 494–505, 2007.

[153] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12)*, pages 157–168, 2012.

[154] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS '03)*, 2003.

[155] Y. Yarom and K. Falkner. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. Cryptology ePrint Archive, Report 2013/448, 2013.

[156] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[157] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P '13)*, 2013.

[158] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[159] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12)*, pages 305–316, 2012.

[160] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS '13)*, pages 827–838, 2013.