UNIVERSITY OF CALIFORNIA,
IRVINE

Securing Statically and Dynamically Compiled Programs using Software Diversity

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Andrei Homescu

Dissertation Committee:
Professor Michael Franz, Chair
Professor Ian Harris
Professor Guoqing Xu

2015

Chapter 3 © 2013 IEEE.
Reprinted, with permission, from **Profile-guided Automated Software Diversity**, Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, Michael Franz, in *Proceedings of the 2013 International Symposium on Code Generation and Optimization*, **CGO 2013**.

Portions of Chapters 2 and 6 © 2014 IEEE.
Reprinted, with permission, from **SoK: Automated Software Diversity**, Per Larsen, Andrei Homescu, Stefan Brunthaler, Michael Franz, in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, **Oakland S&P 2014**.

Portions of Chapter 5 © 2015 IEEE.
Reprinted, with permission, from **Readactor: Practical Code Randomization Resilient to Memory Disclosure**, Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, Michael Franz, in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, **Oakland S&P 2015**.

All other materials © 2015 Andrei Homescu

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

ACKNOWLEDGMENTS **vi**

CURRICULUM VITAE **vii**

ABSTRACT OF THE DISSERTATION **ix**

1 Introduction **1**
- 1.1 Contributions 3

2 Background **6**
- 2.1 Arbitrary Code Execution Attacks 6
  - 2.1.1 Code Injection 6
  - 2.1.2 Code Reuse Attacks 7
- 2.2 Software Diversity 10
- 2.3 Just-in-Time Compilers 13
  - 2.3.1 Attacks on JIT Compilers 14
- 2.4 Information Disclosure 16

3 Profile-guided Automated Software Diversity **19**
- 3.1 Our Approach 20
  - 3.1.1 Profile-guided Diversification 23
- 3.2 Implementation 26
- 3.3 Evaluation 28
  - 3.3.1 Performance Evaluation 28
  - 3.3.2 Security Impact 30

4 Librando: Transparent Code Randomization for Just-in-Time Compilers **35**
- 4.1 Design 36
  - 4.1.1 Code Relocation 43
  - 4.1.2 Diversification 46
  - 4.1.3 Optimizations 48

4.2 Evaluation . . . 52

5 Readactor: Execute-only Pages for JIT Compilers **57**
5.1 Readactor Design . . . 58
5.2 V8 JIT Compiler Protection . . . 61
5.2.1 Separation of Code and Data . . . 64
5.2.2 Switching Between Execute-Only and RW Permissions . . . 65
5.3 Evaluation . . . 68
5.3.1 Performance . . . 68
5.3.2 Security . . . 70

6 Related Work **71**
6.1 Software Diversity . . . 71
6.2 Integrity-based Defenses . . . 77
6.3 Code Hiding . . . 81
6.4 Other Defenses . . . 82
6.5 JIT Protection . . . 83
6.6 Profile-Guided Optimization . . . 85
6.7 Binary Rewriting . . . 86

7 Conclusions and Future Work **89**
7.1 Diversifying Transformations . . . 91
7.2 Future Work . . . 92

Bibliography **93**

LIST OF FIGURES

Page

2.1 Return-oriented programming attack example. 8
2.2 High-level structure of a JIT compiler. 14
2.3 JIT spraying example. 15
2.4 JIT code reuse example. 16
2.5 Example of loop with memory-controlled counter. 18

3.1 Effect of NOP insertion on program code. 21
3.2 Life of a program from source code to execution. 26
3.3 SPEC CPU 2006 performance overhead of NOP insertion. 29

4.1 A JIT compiler with librando attached. 37
4.2 Black and white box diversification architecture. 38
4.3 Block contents and diversification example. 39
4.4 Rewriting the `CALL` instruction. 43
4.5 Per-block finite state machine that handles block changes. 45
4.6 Blocks spanning several memory pages and crossing page boundaries. 45
4.7 Blinding the immediate operand of an instruction. 47
4.8 Rewriting the `RET` instruction to use the *Return Address Map*. 49
4.9 V8 benchmark results for librando. 54
4.10 HotSpot benchmark results for librando. 55

5.1 Readactor system overview. 61
5.2 Readactor hypervisor overview. 62
5.3 Relation between virtual, guest physical, and host physical memory. 63
5.4 Timeline of the execution of a JIT-compiled program. 64
5.5 Transforming V8 `Code` objects to separate code and data. 64
5.6 Performance of modified V8 running under Readactor. 69

LIST OF TABLES

Page

3.1 NOP insertion candidate instructions. 22
3.2 Most executed basic block for each SPEC CPU 2006 benchmark. 25
3.3 Average surviving gadgets on SPEC CPU 2006 binaries. 33
3.4 Total surviving gadgets on SPEC CPU 2006 binaries on 25 different binaries. . . . 34

4.1 x86 instructions with 32-bit immediate operands. 48
4.2 Application Programming Interface provided by librando. 51

5.1 V8 benchmark performance impact of **Method A**. 69

ACKNOWLEDGMENTS

First and foremost, I will be forever grateful to my advisor Prof. Michael Franz for his mentorship and support during my years in graduate school. He brought me into a great research group where I was surrounded by amazing people and could work on interesting research projects, none of which would have been possible without his efforts. In addition, he has spared no effort to ensure the academic success and well-being of his students, myself included.

I would also like to thank Prof. Ian Harris and Prof. Harry Xu for accepting to serve on my committee.

I am deeply grateful to Dr. Per Larsen and Dr. Stefan Brunthaler for closely working with me on this research, guiding me and helping me solve any problems, as well as helping me greatly improve my English writing skills. In addition, I would like to thank Dr. Christian Wimmer for his guidance and insights in the early stages of this research.

I have been fortunate to be part of a great research group, surrounded by great colleagues. I would like to thank everyone in the group for making graduate school enjoyable and fun, but also educational and productive, especially: Stephen Crane, Julian Lettner, Mark Murphy, Gülfem Savrun Yeniçeri, Wei Zhang, Codruţ Stancu, Mohaned Qunaibit, Brian Belleville, Eric Hennigan, Christoph Kerschbaumer, Todd Jackson, Mason Chang, Michael Bebenita, Gregor Wagner.

I would also like to thank Prof. Ahmad-Reza Sadeghi and his students Christopher Liebchen and Lucas Davi for the collaboration on the Readactor project.

This research was supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Google. 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.

Portions of this dissertation have appeared in publications published by IEEE, who owns the respective copyrights.

CURRICULUM VITAE

Andrei Homescu

EDUCATION

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

Engineering Diploma in Computer Engineering **2009**
Politehnica University of Bucharest *Bucharest, Romania*

RESEARCH EXPERIENCE

Graduate Research Assistant **2012–2015**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

141 - Concepts in Programming Languages I **Spring 2012**
University of California, Irvine *Irvine, California*

142A - Compilers and Interpreters **Winter 2012**
University of California, Irvine *Irvine, California*

BOOK CHAPTERS

Diversifying the Software Stack Using Randomized NOP Insertion
Todd Jackson, Andrei Homescu, Stephen Crane, Per Larsen, Stefan Brunthaler, Michael Franz.
In **Moving Target Defense II: Application of Game Theory and Adversarial Modeling, Springer Advances in Information Security**, S. Jajodia, A. K. Ghosh, V. S. Subrahmanian, V. Swarup, C. Wang, X. S. Wang (Eds.), Springer, ISBN 978-1-4614-5415-1, 2013.

Compiler-Generated Software Diversity
Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, Michael Franz.
In **Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats**, S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, X. S. Wang (Eds.), Springer, ISBN 978-1-4614-0976-2, 2011.

REFEREED PUBLICATIONS

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

Readactor: Practical Code Randomization Resilient to Memory Disclosure **Oakland S&P 2015**
Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, Michael Franz.
In *Proceedings of the 36th 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, Michael Franz.
In *Proceedings of the 2015 Network and Distributed System Security Symposium*.

SoK: Automated Software Diversity **Oakland S&P 2014**
Per Larsen, Andrei Homescu, Stefan Brunthaler, Michael Franz.
In *Proceedings of the 35th IEEE Symposium on Security and Privacy*.

librando: Transparent Code Randomization for Just-in-Time Compilers **CCS 2013**
Andrei Homescu, Stefan Brunthaler, Per Larsen, Michael Franz.
In *Proceedings of the 20th ACM Conference on Computer and Communications Security*.

Profile-guided Automated Software Diversity **CGO 2013**
Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, Michael Franz.
In *Proceedings of the 2013 International Symposium on Code Generation and Optimization*.

Microgadgets: Size Does Matter in Turing-complete Return-oriented Programming **WOOT 2012**
Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, Michael Franz.
In *Proceedings of the 6th USENIX Workshop on Offensive Technologies*.

HappyJIT: A Tracing JIT Compiler for PHP **DLS 2011**
Andrei Homescu, Alex Şuhan.
In *Proceedings of the 2011 Symposium on Dynamic Languages*.

ABSTRACT OF THE DISSERTATION

Securing Statically and Dynamically Compiled Programs using Software Diversity

By

Andrei Homescu

Doctor of Philosophy in Computer Science

University of California, Irvine, 2015

Professor Michael Franz, Chair

Code-reuse attacks are notoriously hard to defeat, and many current solutions to the problem focus on automated software diversity. This is a promising area of research, as diversity attacks one cause of code reuse attacks—the software monoculture. Software diversity raises the costs of an attack by providing users with different variations of the same program. However, modern software diversity implementations are still vulnerable to certain threats: code disclosure attacks and attacks targeted at JIT (just-in-time) compilers for dynamically compiled languages.

In this dissertation, we address the pressing problem of building secure systems out of programs written in unsafe languages. Specifically, we use software diversity to present attackers with an unpredictable attack surface. This dissertation contributes new techniques that improve the security, efficiency, and coverage of software diversity. We discuss three practical aspects of software diversity deployment: (i) performance optimization using profile-guided code randomization, (ii) transparent code randomization for JIT compilers, and (iii) code hiding support for JIT compilers. We make the following contributions: we show a generic technique to reduce the runtime cost of software diversity, describe the first technique that diversifies the output of JIT compilers and requires no source code changes to the JIT engine, and contribute new techniques to prevent disclosure of diversified code. Specifically, we demonstrate how to switch between execute-only and read-write page permissions to efficiently and comprehensively prevent JIT-oriented exploits.

Our in-depth performance and security evaluation shows that software diversity can be efficiently implemented with low overhead (as low as 1% for profile-guided NOP insertion and 7.8% for JIT code hiding) and is an effective defense against a large class of code reuse and code disclosure attacks.

Chapter 1

Introduction

Borrowing a term from biology, we refer to the prevalent practice of shipping identical binaries to all customers as the *software monoculture*. Dating back to 1993, Cohen [28] identified this as a practice with detrimental effects on computer security. Software diversity is a biology-inspired response to this problem. Whereas users would run identical copies of a program under a monoculture, each user runs a different version in a diversified environment. Previously, the attacker could study a program and build a single instance of an attack that would run on all instances of the program, but diversification forces the attacker to customize the attack for every target. Consequently, software diversity increases the costs for attackers, ultimately rendering them too costly. Due to a particularly malign class of attacks—known as *code reuse attacks*—community interest recently surged around protections using automated software diversity.

Code randomization is one concrete embodiment of the idea of software diversity. There are many different code randomization techniques. The main one we discuss is NOP insertion [60] (termed "garbage code insertion" by Cohen). This transformation inserts NOP instructions between regular native machine instructions in the program. While naive insertion has a positive impact on security,

it can adversely affect performance. This is not surprising, because truly random NOP insertion is indifferent towards frequently executed code.

Software profiling changes this: it allows us to diversify liberally over most of the program, and also reduce diversification overhead in code where the performance impact is most severe. Feedback directed or profile-guided approaches have long been a major line of research in compilation, in particular for generating and optimizing native machine code, starting with the work of Knuth [67, 68]. A profiling run separates frequently executed—or *hot*—parts of code from infrequently executed—or *cold*—parts. Subsequently, a second compilation run uses this information to optimize the generated code, e.g., by co-locating frequently executed basic blocks [95]. Similar use of software profiling has led to successful research in the areas of code compression [39] and hardware error detection [64]. Whereas profiling is most commonly used to enhance code optimizations to increase their performance benefits, it can also be used to reduce the performance cost of program transformations that slow down the program. In this dissertation, we investigate the use of profiling to reduce the cost of NOP insertion. In Chapter 3, we present the design and implementation of profile-guided NOP insertion using the LLVM 3.1 compiler and its profiling infrastructure.

Unfortunately, existing approaches to artificial software diversity do not protect dynamically emitted code from a just-in-time compiler. In Chapter 4, we address this challenge by describing the first fully automatic technique to diversify existing JIT compilers in a black-box fashion. Similar to the successful frontend-backend separation in traditional compilers, our proposed black-box approach has the advantage over a white-box solution—where developers would manually add diversification directly to JIT compiler source code—of not requiring duplicated work for every existing JIT compiler. Besides the obvious savings in implementation time, the black-box approach allows for patches to be released earlier, without having to rely on vendors to supply patches to known vulnerabilities. Another benefit is the added security for legacy JIT compilers available only in binary form, where extra defenses cannot be added by changing the source code.

As a reaction to the increased popularity to software diversity, attackers are increasingly relying on information disclosure attacks to counter randomization. One defense is to hide randomized code from the attacker, so they are unable read the code. XnR [6], HideM [49] and Readactor [34] use different technical approaches to solve the same problem: preventing the attacker from reading code. Of all these solutions, Readactor is the only one that extends protection to dynamically generated code, such as code generated by a JIT compiler. In Chapter 5, we describe the challenges encountered in modifying Readactor and a popular JavaScript JIT compiler—V8—to support JIT code hiding.

1.1 Contributions

This dissertation makes the following contributions, organized by chapter:

Chapter 3

- We introduce a technique that uses profiling information to optimize away overhead introduced by automated software diversity.
- We describe a heuristic formula controlling randomization based on basic block execution frequencies.
- We present the results of a thorough evaluation of our implementation. Our results indicate:

  Performance We are able to reduce overhead of probabilistic NOP insertion for SPEC CPU 2006 down to a negligible 1% performance overhead. This is an important result allowing us to use stronger diversifying transformations without sacrificing performance, which is a key barrier to software diversity adoption.

Security We briefly describe a way to objectively measure the success of diversifying a binary. Our profiling-driven optimization preserves the security properties of code layout randomization. In addition, we show that our transformation thwarts concrete attacks.

Chapter 4

- We present librando, the first automated software diversity solution for hardening existing JIT compilers in a black box fashion. Our solution implements two popular diversifying transformations: NOP insertion and constant blinding.
- We describe two optimizations (the *Return Address Map* and optional white box diversification—taking advantage of compiler cooperation) to improve the performance of librando.
- We demonstrate applicability of black box diversification on two pervasive industrial-strength JIT compilers: Oracle's HotSpot (used in the Java Virtual Machine) and Google's V8 (used in the Chrome web browser). We then report the results of our analysis of librando performance. We show that we successfully protect:

  HotSpot (a JIT compiler for Java—a statically typed language) with an overhead of 15%.

  V8 (a JIT compiler for JavaScript—a dynamically typed language) with a slowdown factor of $3.5\times$.

Chapter 5

- We discuss the challenges encountered when adding support for code hiding to a popular industry-strength JIT compiler—V8. Our work consists of changes to both V8 (to add support for non-readable code cache pages), and to Readactor (to efficiently support frequent changes to page permissions).

- We discuss and evaluate two different virtual-to-physical page mapping strategies for JIT code cache pages on a system running Readactor.

- We evaluate the performance and security of our changes to V8. We show that, with our changes and running under Readactor, V8 is resilient to code disclosure attacks and the average performance overhead of our approach is 7.8%.

Chapter 2

Background

2.1 Arbitrary Code Execution Attacks

For performance and practical reasons, a large part of the modern software stack is written in low-level systems programming languages. Since the programmer is assumed not to make any mistakes, little error checking happens at run-time. Consequently, even simple programming mistakes can lead to security vulnerabilities for attackers to exploit. Exploiting these vulnerabilities generally allows the attacker to take control of the target machine or program, i.e., execute arbitrary code under the attacker's control on the target. While the attacker can also have other goals, such as leaking secret information from the target or performing denial-of-service attacks, arbitrary code execution on the target machine is still highly valuable.

2.1.1 Code Injection

Originally, attackers would perform arbitrary code execution by injecting a binary payload (for example, x86 code) into a vulnerable application running on the target, then use some other vul-

nerability in the application to force it to execute the payload. For example, the attacker would overwrite the return address on the program stack using a stack buffer overflow [3], causing the program to execute the payload when returning from the current program function. This approach requires that the application be able to write data to a buffer, then execute code from that same buffer. In other words, the processor has to be able to (or be allowed to) execute instructions from an area of memory that the application can directly write to. However, modern processors allow for a page to be marked as non-writable or non-executable.

Modern operating systems use this feature to prevent code injection attacks. A page cannot be both executable and writable at the same time, unless specifically requested by the application. This restriction is known under several names: W$\oplus$X (implemented by the PaX [92] kernel patch on Linux), DEP (Data Execution Prevention), NX (No-eXecute), and XN (eXecute Never). In practice, W$\oplus$X renders most code injection attacks ineffective. Since the payload is stored in a writable but non-executable buffer, the processor cannot execute its contents. In addition, some operating systems disallow execution of unsigned code.

2.1.2 Code Reuse Attacks

To bypass W$\oplus$X, a new class of attacks against applications surfaced and gained popularity: code reuse attacks. These attacks circumvent anti-code injection restrictions by reusing code from the application itself to perform the attack. Shacham [106] introduced Return-Oriented Programming (ROP), a code reuse attack that uses small code snippets (called *gadgets* in the ROP paper) from the executable section of the program as the attack payload itself. A gadget is any valid sequence of binary code that the attacker can execute successfully (the gadget decodes correctly and does not contain invalid instructions) and that ends in a control flow transfer instruction. The original version of ROP uses only gadgets that end in a `RET` instruction (encoded by the `C3` byte on x86); the attacker places addresses of gadgets on the stack on consecutive stack slots, so that each gadget

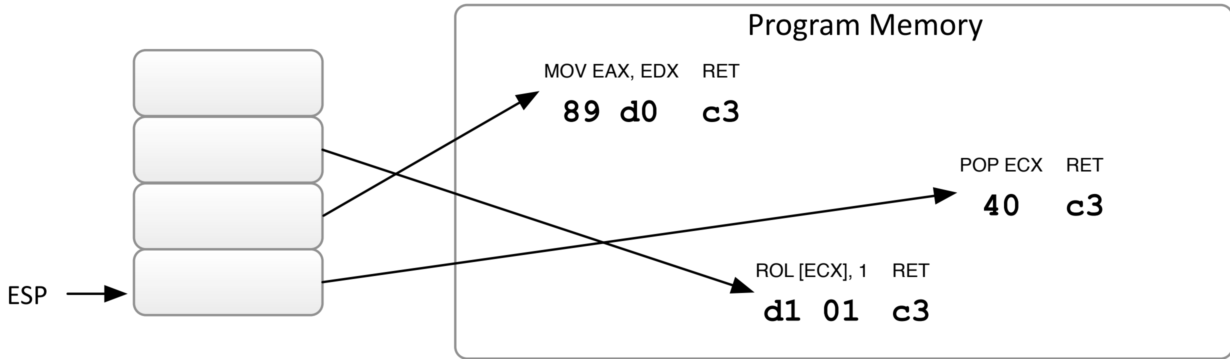

Figure 2.1: Return-oriented programming attack example.

The attack proceeds in several steps: (i) the attacker locates gadgets in the program code; (ii) the attacker puts the gadget addresses on the stack, overwriting the return address; (iii) when the attacked function returns, it executes the first gadget; (iv) each gadget executes the following one by returning to it.

proceeds to the next one using a return. A gadget can start anywhere inside the generated code (including in the middle of a proper instruction) and spans one or more of the original instructions emitted by the compiler. Figure 2.1 shows a high-level example of a return-oriented programming attack. Later variations of ROP, such as JOP [15] (jump-oriented programming), ROP-without-returns [21], and `sigreturn`-oriented programming [16], lift the requirement that gadgets end in a return and extend the approach to other control flow instructions.

ROP uses gadgets to implement the instructions of a virtual machine (VM). The attacker writes their payload as a program written in this virtual machine, then converts each VM operation into the address of the equivalent gadget. The ROP VM usually offers a significant number of useful operations, like simple arithmetic, memory loads and stores, and conditional jumps. In the original ROP design, the operations formed a Turing-complete set, providing an attacker with a simple and effective way to perform arbitrary computations inside the target. However, it is possible to successfully launch an attack with even a restricted subset of operations, without providing Turing-completeness. Often, the attacker only needs to call some system function (like `mmap`), store a payload into a memory area and then redirect control flow to that memory region. The *borrowed*

code chunks attack by Krahmer [70] (which preceded ROP by several years) and *microgadgets* [56] are two simplified code reuse attacks that call the `system` system call to obtain a system shell.

Early variations of code reuse attacks mainly targeted the x86 instruction set, as the leading desktop processor architecture. With the increasing popularity of other architectures (such as ARM for phones), researchers also demonstrated code reuse attacks against ARM [69, 38], SPARC [99], Zilog Z80 [22] and Atmel [44]. As a practical example, we note that ROP was successfully used to jailbreak an earlier version of the iPhone [124].

While ROP initially used short and simple gadgets ending in returns, code reuse attacks are feasible even with longer and less restricted sequences. Return-into-lib(c) [83] attacks (introduced several years before ROP) redirect the program's execution to a function (usually `system`) in a system library (such as libc), after manipulating the stack so the attacker controls the function parameters. Later work [114] implements a Turing machine using libc functions, proving that return-into-lib(c) is Turing-complete and therefore able to perform arbitrary computations. Another whole-function code reuse attack—COOP [103]—uses C++ class methods chained using counterfeit object instances to perform arbitrary computations.

Control-flow integrity (CFI) defenses have been proposed as a viable mitigation strategy against code reuse attacks. The main idea of CFI is to restrict the control flow graph of the program exclusively to intended transitions, i.e., instrument indirect branches and returns to validate the target of the branch, as attackers often rely on replacing code pointers with gadget addresses to hijack program execution. In return-oriented programming, for example, attackers replace return addresses on the stack with gadget addresses. CFI defends against ROP by checking that each return address on the stack is the address of a valid return site (the instruction immediately succeeding a function call). This requires that the CFI implementation builds a conservative list of valid targets for every indirect branch in the program (including returns), either at compile-time or through analysis of the program binary. Due to the difficulty of building this list, as well as the high performance overhead of enforcing strict control flow integrity, a less strict version of CFI

has been proposed: coarse-grained CFI. In a coarse-grained implementation, any return instruction can return to any return site (but only a return-site), and indirect function calls can call all valid function entry points. However, new variations of ROP [51, 35, 104, 20] have been presented that work even under these restrictions. Restricting the set of gadgets to entry-point (EP) gadgets (contiguous code sequences starting from a valid function entry point and continuing up to the first indirect call/return) and call-site (CS) gadgets (gadgets starting immediately after a function call) still allows the attacker to perform a successful attack against a program secured by a coarse-grained CFI solution. This leaves ROP attacks (and code reuse attacks in general) as a potent class of attacks, requiring stronger defenses than coarse-grained CFI.

2.2 Software Diversity

The idea of software diversity was originally explored as a way to obtain fault-tolerance in mission critical software. Approaches to software fault-tolerance are broadly classified as single-version or multi-version techniques. Early examples of the latter kind include Recovery Blocks [98] and N-Version programming [5] that are based on *design diversity*. The conjecture of design diversity is that components designed and implemented differently, e.g., using separate teams, different programming languages and algorithms, have a very low probability of containing similar errors. When combined with a voting mechanism that selects among the outputs of each component, it is possible to construct a robust system out of faulty components.

Obfuscation to prevent reverse engineering attacks [29, 31] is closely related to diversity and relies on many of the same code transformations. Diversity requires that program implementations are kept private and that implementations differ among systems; this is not required for obfuscation. Pucella and Schneider perform a comparative analysis of obfuscation, diversification, and type systems within a single semantic framework [97].

Software diversity is a powerful tool in the defense against code reuse attacks. All code reuse attacks have one trait in common: the attacker obtains and uses knowledge of the program code itself. In some cases, the attacker does not have such knowledge a priori, but they can gain obtain it dynamically (we discuss this case in greater detail in Section 2.4). Any interaction between the program and attacker may help the attacker, and they frequently take advantage of this to obtain information on program internals (such as contents of data structures or program instructions). One practical restriction is that an attacker can only use code from the executable sections of the program, due to W⊕X. Therefore, one way to defend against code reuse attacks is to prevent the attacker from gaining any useful information about the program itself, such as the addresses of known gadgets inside the program.

Cohen [28] proposes the following strategy to defend against attacks:

> *The ultimate defense is to drive the complexity of the ultimate attack up so high that the cost of attack is too high to be worth performing.*

Cohen proposes program evolution as a defense strategy, where programs "evolve" into different, but semantically equivalent versions, of the original program. He then demonstrates several different evolution techniques, such as equivalent instruction replacement, instruction reordering and garbage insertion. As these techniques all change the program code in different ways, by either adding new instruction or shifting the existing ones around, they have a potentially large impact on both the locations and order of instructions in the program. Since code reuse attacks require the attacker to have knowledge of the contents of the binary, performing such attacks becomes much more difficult.

Many arbitrary code execution attacks require the attacker to redirect the execution of the program from its regular path to some malicious behavior under the attacker's control. Often, the attacker does this by overwriting a memory location controlling an indirect branch. In return-oriented programming, for example, the attacker overwrites the return address of the current function and all

following stack locations. Often, attackers use a buffer overflow vulnerability to overwrite these locations, by writing past the end of a buffer stored in a function frame. Stack frame randomization [43] is an implementation of software diversity targeted at stack-based attacks, using variable reordering and stack frame padding to diversify the binaries.

With the growing popularity of code reuse attacks, the need arises for mitigations against this class of attacks. One such mitigation is software diversity focused on the program's binary code. By preventing the attacker from having a priori information about the code layout of a program, we significantly raise the cost of a code reuse attack. In practice, one way we achieve is code layout randomization. This can be done in several places: in the compiler (by randomizing the code inside the compiler [63, 50]), in the operating system loader (disassembling the program and applying diversifying transformations to it; also, it can be done by replacing the loader) [54] or in between, as a separate step [90]. These implementations use techniques similar to the ones described by Cohen and show that code layout randomization is very effective at preventing code reuse attacks, preventing many ROP attacks currently at large.

Modern operating systems also implement a form of code layout randomization called Address Space Layout Randomization (ASLR). This works by randomizing the base addresses of objects in the program, like the stack, heap, program code, and dynamic libraries. However, as this randomization changes the location of most program objects, the operating system loader must now perform a significant number of relocations. An alternative implementation, currently used in Linux, changes the program so that it does not require fixed locations for any of its objects; instead, the program determines its own randomized base address at run time, then accesses all its objects by offsets from the base address. This approach has a performance penalty on 32-bit x86 systems [93], so it is currently disabled for most applications. Consequently, the code section of a program is always loaded at the same address (`0x8048000` on Linux), and only libraries are loaded at random addresses.

Another weakness of ASLR is that it only diversifies the base addresses of program objects, such as code and data sections. There is no diversity inside the sections. If an attacker were to gain this base address by some information leak, they would have all the code layout information for a code reuse attack (assuming attackers also possess a copy of the binary). On 32-bit systems, ASLR also suffers from lack of entropy; recent work [107, 100] shows that ASLR can be defeated in a matter of minutes. However, ASLR has been successful at raising the bar for attackers, illustrating the effectiveness of randomization against attacks. On the other hand, it has also shown the effect low entropy can have on randomization, emphasizing the importance of high-entropy defenses.

2.3 Just-in-Time Compilers

Our computing infrastructure depends on high performance delivered by just-in-time (JIT) compilers to a large degree. Efficiently executing JavaScript is a prerequisite for complex Web applications. Similarly, Java's success rests on performance delivered by efficient dynamic code generation. To achieve this, a JIT compiler transforms a program written in a high-level language (HLL), such as Java or JavaScript, generating native code as needed during program execution. The compiler emits native code into a code cache, after parsing and optimizing HLL source code. The compiler itself is most often written in another programming language, which we call the host language (which is C or C++ in most cases). JIT compilers also contain a language runtime, which is a library of functions that are written in the host language and provide or manage access to system resources. Some examples of such resources are files, networks, operating system threads and complex data structures such as maps and trees. When compiling a HLL program, the JIT compiler emits native calls into the runtime whenever the program uses these resources. Figure 2.2 shows a high-level structure of a JIT compiler and its interactions with the generated code. After emitting all or part of the native code (usually enough code to start execution of the program), the compiler branches to the entry point of the HLL program. The generated code continues execution,

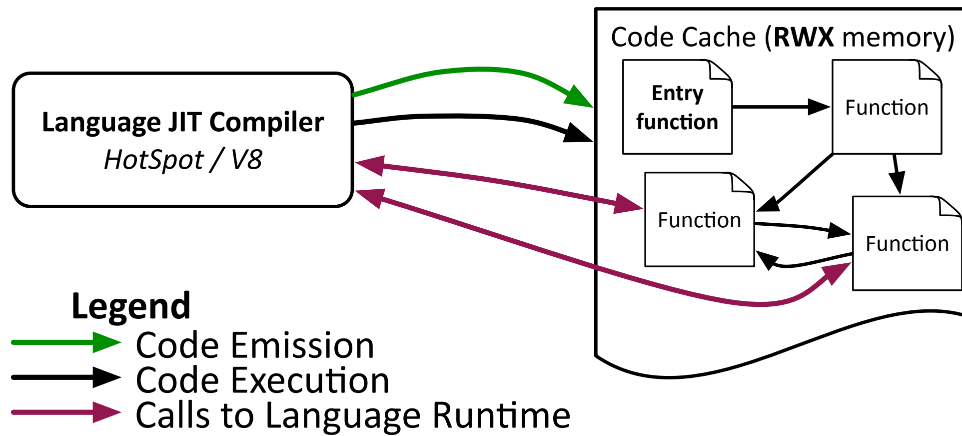

Figure 2.2: High-level structure of a JIT compiler.

calling into other generated functions or the language runtime. The HLL program continues until termination, making repeated calls into the runtime whenever needed.

2.3.1 Attacks on JIT Compilers

Security-wise, JIT compilers have one characteristic that is important in our context: predictability. Current JIT compilers are generally deterministic. When presented with the same HLL code many times repeatedly, a compiler will emit the same native code; attackers can use this characteristic to their advantage. This is not a problem specific to JIT compilers, but compilers in general; however, predictability of JIT compilers has not been fully explored.

JIT spraying [13] is one recent attack that relies on predictability of JIT compilers. This attack is a form of code injection targeted at dynamically generated code. In its original form, it relies on one unintended behavior of many JIT compilers: HLL program constants reach native code unmodified, therefore becoming part of the executable code. The attacker injects short sequences (32-bit constants in the original paper), and later jumps to the injected sequence through a separate attack vector. Figure 2.3 shows an example of code injected using constants. For the attack to work, the attacker must also predict the remaining bytes inserted between the controlled sequences, and

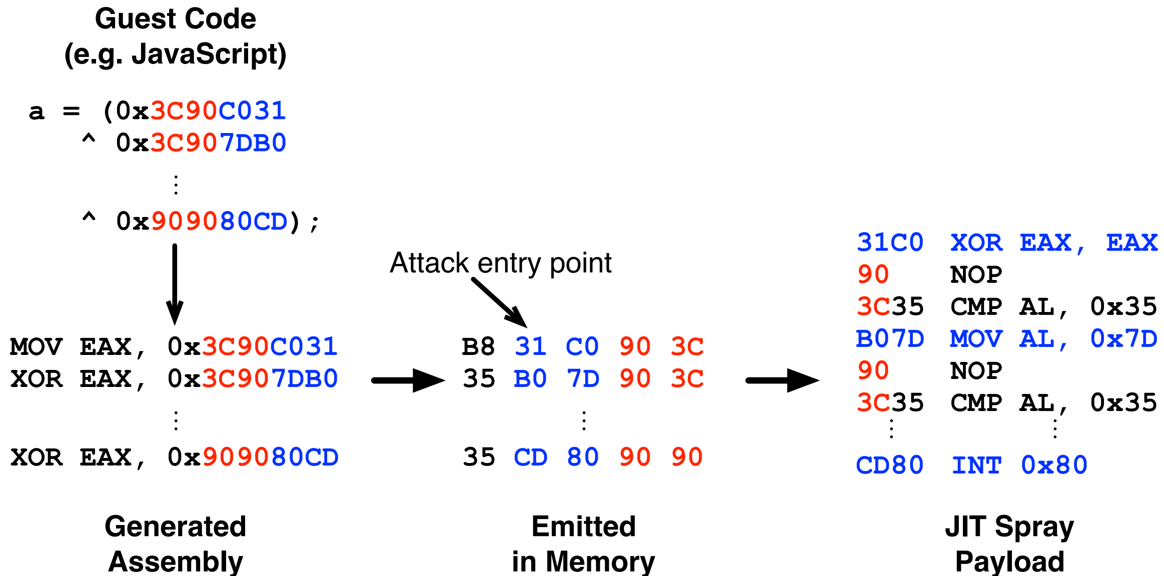

Figure 2.3: JIT spraying example.

The 32-bit constants from HLL code (shown in red and blue) appear inside native code, in little-endian form.

use those bytes as part of the payload; this is often possible in practice, due to the predictability of the compiler. This allows the attacker to execute arbitrary native code, even when running on a compiler that runs the generated code in a sandbox (with restricted access to memory, for example). The original JIT spraying attack targeted x86 systems, but Lian et al. [75] later extended this approach to JITs targeting ARM processors.

As an additional threat, code reuse attacks are even more potent in the presence of a JIT compiler, as an attacker that controls HLL code can emit an arbitrary amount of native code containing gadgets (by emitting as much HLL code as needed to generate all the gadgets for the attack); Figure 2.4 illustrates how HLL code sequences become gadgets. For example, this can be a problem for web browsers that include a JavaScript compiler, as many web pages include JavaScript code from unreliable (or hostile) sources. Another problem is that current anti-ROP defenses [74, 88, 90, 54, 60, 55] target ahead-of-time compilers or binary rewriters, but do not offer protection to dynamically generated code.

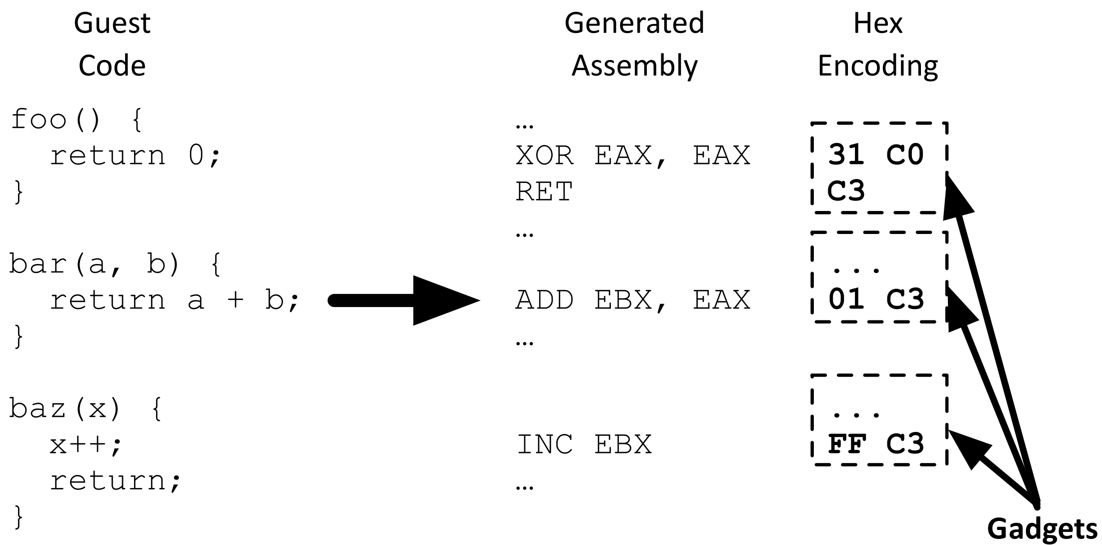

Figure 2.4: JIT code reuse example.

The compiler transforms HLL code into native code containing ROP gadgets. The `C3` byte encodes the `RET` instruction at the end of a gadget.

2.4 Information Disclosure

All code reuse attacks have one trait in common: the attacker must have knowledge of the program code itself. The original threat model of software diversity-based defenses made one significant assumption: that once the code is diversified, the attacker cannot gain access to it. Later research proved this assumption to be false under certain circumstances. In some cases, the attacker can gain gain information about code layout and contents dynamically. JIT-ROP [110] is an advanced attack that counters code randomization by reading code pages at run-time. This attack has two main requirements: (i) that the target program offers the capability to run sandboxed Turing-complete code (written in a HLL) under the control of the attacker, e.g., run JavaScript in a browser or Java code in the JVM, and (ii) that the target contains some memory read vulnerability that allows a HLL script provided by the attacker to read (and possibly write) arbitrary memory locations inside the target (even those outside the sandbox). Using an arbitrary memory read coupled with a control flow hijack, the attacker locates all required gadgets and builds a ROP attack at run-time. With the

growing popularity of the JavaScript language for client-side web development, browsers have become a significant target for ROP attacks.

In other cases, the attacker does not need to leak the contents of code pages. Instead, they can just de-randomize the location of known gadgets through brute force or side channels. Blind ROP [12] starts from the assumption that any Linux program always contains certain gadgets and libc functions, and finds their addresses by sequentially trying all addresses in a contiguous range. This attack works against a specific set of programs: networked servers that restart to a known fixed state after a crash (this is a common property of Linux server programs that use the `fork` system call for multi-processing, such as Apache and nginx). When the attacker puts a bad address on the stack, the server crashes and closes the network connection, but also restarts in an identical state and accepts new connections. The attacker simply reconnects and tries the next address. Using this approach, Blind ROP successfully finds enough gadget addresses to perform a system call, leading to further arbitrary code execution, even if the program has been diversified using code randomization or ASLR, unless the program is re-diversified upon restart.

Another viable attack against diversified programs are memory access timing attacks. The main requirement of their attack is that the target program contains a loop whose iteration count is read from memory. In other words, the running time of the loop is proportional to some value read from a memory address. The attacker can point this address to a code location, and deduce the contents of that location from the duration of the loop (assuming the duration of the loop is directly proportional with the memory value). Figure 2.5 shows an example of such a loop. Siebert et al. [109] implement such an attack and evaluate it against several forms of diversity. Their attack requires a modest amount of time to defeat ASLR (30.58 seconds), but fine-grained code randomization is more resilient (it takes them 2.2 hours to attack a program randomized using NOP insertion).

Information disclosure attacks significantly raise the bar for defenders, but do not completely defeat software diversity defenses. Instead of assuming that the attacker cannot read code, defenders

```
struct S {
  ...
  int n;
  ...
};

void foo(struct S *p) {
  for (int i = 0; i < p->n; i++) {
    ...
  }
}
```

Figure 2.5: Example of loop with memory-controlled counter.

The duration of the loop is directly proportional with the value of `p->n`, which the attacker can change by redirecting the pointer `p`.

must now take measures to hide the code. XnR [6], HideM [49] and Readactor [34] all successfully make program code unreadable, preventing attacks such as JIT-ROP. In Chapter 5, we discuss code hiding for the code cache of a JIT compiler. Increasing entropy and re-randomizing code frequently are viable defenses against brute force and timing attacks.

Chapter 3

Profile-guided Automated Software Diversity

While modern diversification techniques successfully protect against code-reuse attacks, they suffer from performance impacts ranging from negligible (1-5%) to significant [72] (as high as 30-50% for realistic approaches and an extreme of 287x for the slowest solution). Interestingly, this spectrum blends well with the security guarantees of the approaches, i.e., the slower ones have the strongest security properties. Taking a closer look at the implementations of automated software diversity, we notice that many of them use a "one-off" design: they take an input program and diversify it in one pass. This resembles the state-of-the-art in compiler construction *before* the advent of profile-guided optimizations triggered by Pettis and Hansen's profile guided code positioning of 1990 [95].

Feedback-directed, or profile-guided, approaches have been a major line of research in compilation, in particular for generating and optimizing native machine code. A profiling run separates frequently executed—or *hot*—parts of code from infrequently executed—or *cold*—parts. Subse-

quently, a second compilation run uses these information to optimize the generated code, e.g., by co-locating frequently executed basic blocks.

Previous research focuses almost exclusively on optimizing the hot code parts. Our idea is to combine profiling information with automated software diversity. By doing so, we substantially reduce the costs of even expensive diversifying transformations. One such transformation inserts NOP instructions in between intentionally emitted native machine instructions, a technique known as NOP insertion. To add software diversity to an input program, we have to insert NOPs probabilistically, i.e., depending on some random information, we decide whether to insert a NOP instruction or not. While blind insertion has a positive impact on security, it also affects performance. This is not surprising, because the random NOP insertion trial has no information about the whereabouts of frequently executed code. Profiling information gives us these clues: it tells us that we can diversify as much as we want to in cold code, and reduce diversification overhead in hot code. A similar insight led to successful research in the areas of code compression [39] and hardware error detection [64]. In this chapter, we discuss the use of profiling information to drive decisions taken by a randomized NOP insertion algorithm.

3.1 Our Approach

In this dissertation, we mainly use randomized NOP insertion to randomize the code layout of a program. Cohen [28] discusses this approach under the name "garbage code insertion"; later, Jackson et al. [60] discuss the practical details of implementing NOP insertion, and implement and evaluate its performance overhead [59]. We build upon their ideas and extend their work with an optimization that significantly improves overhead.

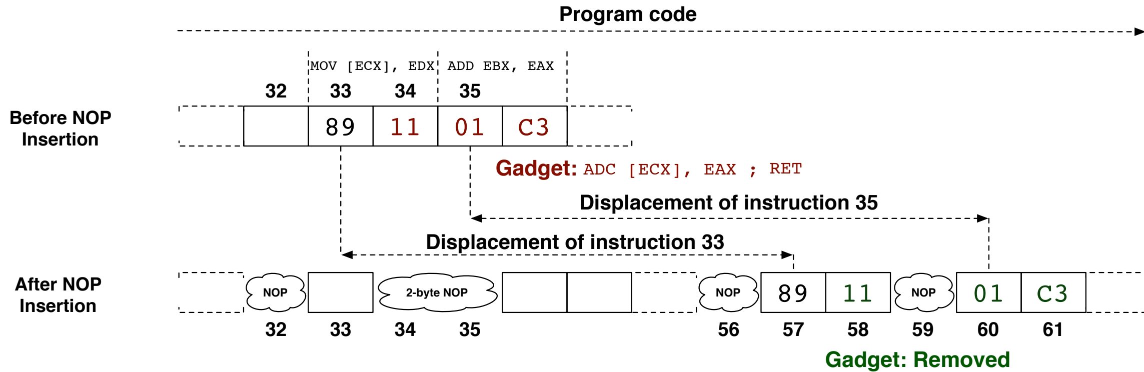

Figure 3.1: Effect of NOP insertion on program code.

A NOP [1] is an instruction that the processor fetches and executes without any effect on the processor register or machine memory. Compilers insert NOPs in programs for various purposes: (i) to enforce alignment of basic blocks, functions or other code blocks (for performance and security); (ii) to add timing delays to code fragments (for contention mitigation) [113]; (iii) to compensate for microarchitectural limitations, such as those in the branch predictor [57].

At each program instruction, we insert a randomly chosen number of NOPs (zero or one in our current implementation), so that all following instructions are displaced by a random number of bytes. As the algorithm inserts NOPs through the program, the displacements accumulate, so that later instructions are pushed forward by increasingly larger amounts (Figure 3.1).

For every assembly instruction in the program, we decide whether to prepend a NOP before the instruction, with probability p_{NOP} of success. In case a NOP is inserted, we then pick one of the NOP candidates at random. Consequently, there are two sources of randomness in this transformation: whether to insert and what to insert. Algorithm 1 shows the algorithm in pseudocode.

Table 3.1 shows a list of eligible NOP instructions. We picked only instruction candidates that preserve the processor state at all times (as opposed to a weaker version of NOPs which change some minor part of processor state, e.g., an operation that adds zero to a register, not affecting registers or memory but changing the CPU flags). Second, we selected candidates with return-

[1] NOP is the x86 architecture mnemonic for a “no-operation” instruction.

Data: The list $IList$ of instructions. The probability of insertion p_{NOP}. $NOPTable$, the list of candidate NOPs.
Result: The list $IList$ with NOPs inserted.
begin
  $numNOPs \leftarrow |NOPTable|$
  for $i \in IList$ **do**
    $roll \leftarrow random\,(0.0, 1.0)$
    if $roll < p_{\text{NOP}}$ **then**
      $nopIndex \leftarrow random\,(0, numNOPs)$
      $insert(i, NOPTable\,[nopIndex])$
    end
  end
end

Algorithm 1: NOP insertion algorithm.

Instruction	**Encoding**	**Second Byte Decoding**
`NOP`	`90`	–
`MOV ESP, ESP`	`89 E4`	`IN`
`MOV EBP, EBP`	`89 ED`	`IN`
`LEA ESI, [ESI]`	`8D 36`	`SS:`
`LEA EDI, [EDI]`	`8D 3F`	`AAS`
`XCHG ESP, ESP`	`87 E4`	`IN`
`XCHG EBP, EBP`	`87 ED`	`IN`

Table 3.1: NOP insertion candidate instructions.

oriented programming in mind, carefully picking those that minimize the likelihood of creating new gadgets. In the case of the two-byte instructions, the second byte decodes to an operand or opcode that the attacker cannot use for nefarious purposes. For example, the `IN` instruction causes the processor to read from an input/output port. However, `IN` requires the processor to be in privileged mode to work correctly, causing unprivileged software to fault.

Table 3.1 shows seven NOP instructions that can be inserted as padding, but our implementation only uses five of them. The two `XCHG`-based NOPs, while perfectly suited to our goals, have a larger performance impact than the others. This is because, on current implementations of the x86 architecture, the `XCHG` instruction locks the memory bus [58]. None of the other NOPs require

this, so we use them in our insertion pass. The two extra NOPs provide some extra diversity in the generated code, so they can be enabled at compile-time.

While our goal is to displace the original program instructions by random amounts, these NOPs also have another useful side effect. The x86 architecture is highly irregular, with instructions as short as 1 byte and as long as 20 bytes. Therefore, inserting one or two extra bytes inside an x86 instruction can change its decoding significantly, in many cases even changing the instruction's length. This effect is even more pronounced on ROP gadgets, where changing the decoding of one instruction can cause the next one to start at a different location. This offset propagates to the return instruction (encoded as the `C3` byte) so that the gadget no longer ends in this instruction. This effectively removes that gadget from the binary. Figure 3.1 illustrates this. NOP insertion not only meets our original design goal of displacing instructions by a random offset, but also has the added benefit of removing some gadgets entirely.

3.1.1 Profile-guided Diversification

The approach to NOP insertion described in Section 3.1 uses the same probability for all instructions. The technique inserts the same expected number of NOPs inside loops and other frequently executed parts of code as in the rest of the program. In practice, most of a program's execution time is spent in a very small part of the code, usually a loop. Therefore, it makes much sense to alter the NOP insertion strategy for these regions, to minimize the performance impact of the extra instructions. To change the probability of NOP insertion according to the execution frequency of the current code region, we need a source of information for that frequency.

Run-time profiling is one source of execution counts. When optimizing using this approach, the compiler generates a special, instrumented version of the input program. The developer then runs this instrumented version on a training set of inputs. The purpose of this run is to collect execution statistics from the training run. These statistics include values such as execution counts for

control flow edges and histograms for variable values. The compiler later uses this information for optimizations during a second compilation.

Most modern compilers support some form of run-time profiling and profile-guided optimization. In every case, the profile contains per-basic-block execution counts, or similar information. LLVM, for example, only inserts counters for the minimal required subset of edges on the control flow graph [86]. The compiler derives all basic block execution counts from that minimal set of per-edge counters.

This approach provides accurate information on which parts of the code are executed most frequently, assuming that the training set is a proper sample of real-world usage of the input program. Note that per-basic-block execution counts are sufficient for our purpose; all instructions in a basic block are executed the same number of times. Therefore, we propagate basic-block execution counts to all instructions, and use the same probability of inserting a NOP for all instructions inside a basic block. More frequently executed blocks will correspond to lower NOP insertion probabilities.

To achieve this, we replace the singular NOP insertion probability with a range of probabilities. The hottest basic blocks get the lowest value in this range, while the coldest blocks get the highest probability. Blocks between these extremes need some probability linked to their execution frequency. The simplest way to model this is a linear function:

$$p_{NOP}(x) = p_{max} - (p_{max} - p_{min}) \frac{x}{x_{max}}$$

where x is the execution count of the current basic block, x_{max} is the maximum execution count in the program and $[p_{min}, p_{max}]$ is the probability range. The compiler uses this function to compute the NOP insertion percentage of each basic block.

Benchmark	Maximum Block Execution Count	Benchmark	Maximum Block Execution Count
403.gcc	14,294,732	447.dealII	1,031,411,560
444.namd	98,877,833	483.xalancbmk	1,210,195,955
433.milc	101,025,792	482.sphinx3	1,268,512,791
453.povray	132,687,016	445.gobmk	1,772,254,272
450.soplex	215,314,492	473.astar	2,072,966,645
470.lbm	390,000,000	464.h264ref	2,769,429,824
429.mcf	437,284,625	458.sjeng	3,629,257,134
400.perlbench	651,734,032	401.bzip2	3,703,065,891
462.libquantum	681,771,008	456.hmmer	4,024,737,000
471.omnetpp	768,762,446		

Table 3.2: Most executed basic block for each SPEC CPU 2006 benchmark.

We collected this data from one SPEC CPU run in `train` mode.

This is a simple and effective heuristic that improves our NOP insertion strategy. However, in practice, execution counts of basic blocks are not distributed linearly. Not only do loops often have large number of iterations, but inner loops are occasionally contained in one or more outer loops. In the latter case, the total number of iterations is the product of the iterations of the inner and outer loops. This means that the counts grow exponentially in the number of loops, while the base itself can be a large number. A linear function would simply polarize the probabilities toward either the maximum or the minimum, since the maximum execution count has a very large value. For example, our measurements on a profiled run of SPEC CPU 2006 show that the maximum execution count ranges from 14 million (for 403.gcc) to 4 billion (for 456.hmmer), as Table 3.2 illustrates. Also, for many applications, the execution counts are spread out between the minimum and maximum count. The 473.astar benchmark is one example of this: the median of all basic block execution counts is $117,635$, well under the maximum of 2 billion.

This means that a linear function is not an appropriate fit; a logarithm-based function serves our purpose much better. The updated function for our profile-guided NOP insertion is:

$$p_{NOP}(x) = p_{max} - (p_{max} - p_{min}) \frac{\log(1+x)}{\log(1+x_{max})}$$

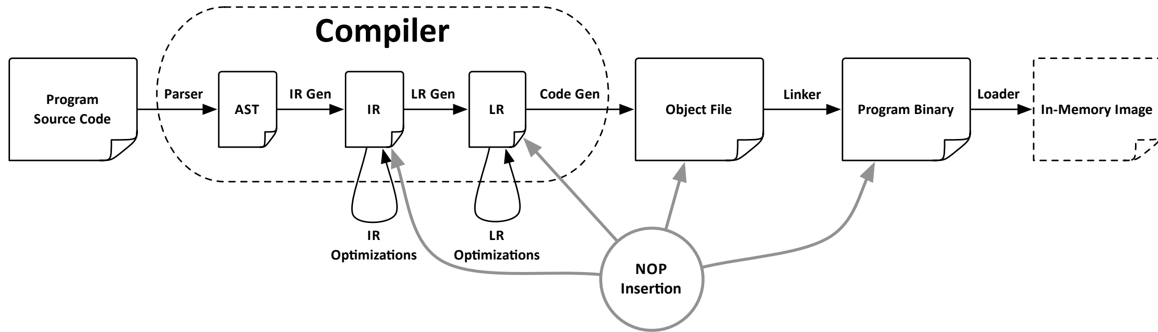

Figure 3.2: Life of a program from source code to execution.

We can insert NOPs at several stages of this process.

Using this function, the numerator and denominator of the fraction become much smaller. For an x_{max} of 4 billion and a logarithm base of 10, the denominator is approximately 10. Therefore, intermediate values get much smaller logarithms as well (in our case, logarithms range between 0 and 10). This also guarantees that counts that are smaller than the maximum by several orders of magnitude are still placed well inside the probability interval, not toward the minimum. The median from 473.astar, for a probability range of $[10, 50]$, now gets an approximate probability of $p_{NOP} \approx 50 - 40 * \frac{5}{10} = 30\%$ instead of $p_{NOP} \approx 50 - 40 * \frac{10^5}{10^{10}} \approx 50\%$. Therefore, using logarithms offers us a much better distribution of probabilities inside the interval, given the particular distribution of the execution counts gathered from profiling.

3.2 Implementation

One major design choice in the implementation of NOP insertion is the stage of the compilation process at which to insert the extra instructions. Figure 3.2 shows the stages a program goes through, from source code to final image in process memory. Most modern compilers take a program as source code, parse it into an abstract syntax tree, then flatten that tree into an intermediate representation (IR). One such IR is the LLVM IR used by the LLVM compilation framework [73]. The compiler performs a suite of optimizations on this IR, then lowers the IR into a lower-level

representation (LR), such as the Register Transfer Language from GCC [46]. After performing even more optimizations (such as register allocation), the compiler generates native code directly from the LR, into an object file. A linker later links one or more object files together into the final program binary. The operating system loader loads the program into memory and executes it.

It is theoretically possible to perform NOP insertion during any of these steps. However, each choice comes with its particular set of disadvantages. Inserting NOPs inside native code, either in object files or in the final program binary, requires complete information about the locations of all program instructions. As modern processors allow for read-only data to be mixed with code in the same stream, accurate disassembly has been proven impossible in the general case [28]. Another option is to use debug information, but that requires the program to be compiled with such information present. Inserting NOPs too early also presents a similar challenge: some IR operations might be lowered to more than one native instruction, while some other operations might disappear completely. Fortunately, most LR operations in a compiler have a one-to-one correspondence to the native code instructions in the object files. Therefore, our strategy is to insert NOPs into the lower-level representation, after the compiler performs all optimizations and just before it emits native code.

Our profiling-based randomization uses execution counts for basic blocks to tune the probability that a NOP is inserted. We derive the per-basic-block information from edge execution counts generated through instrumentation. Fortunately, LLVM currently implements a profiling framework which provides instrumentation for profiling [86]. At present, LLVM does not perform any optimizations that use the profiling information, but such optimizations can be easily added. The profiler adds a counter to each control-flow graph edge, and the counter is incremented at runtime each time that path is taken during program execution. Using this profiling framework, we implemented NOP insertion as a new backend pass in LLVM 3.1.

3.3 Evaluation

We performed all our tests on a 2.66 GHz Intel Xeon 5150 with 4GB of memory. We used Ubuntu 12.04 as the host operating system, running Linux kernel version 3.2.0.

3.3.1 Performance Evaluation

To evaluate the performance impact of our technique, we built and ran the SPEC CPU 2006 benchmark suite with our diversifying compiler. We used the test system described above to run the benchmarks. We compiled all tests using the `-O2` optimization level. For the profile-guided diversification tests, we collected profiling information by running SPEC on the train input set.These inputs were designed specifically to provide an accurate profile for each program.

As the NOP insertion process uses a random number generator, there is some potential noise in the measurement due to the performance differences between different randomized versions. To account for this, we evaluated five different versions of each individual benchmark. We ran each version three times and averaged the results.

Figure 3.3 shows the performance impact of NOP insertion, with and without using profiling information and for a few combinations of probability parameters. First, we evaluated $p_{\text{NOP}} = 50\%$, as this is the parameter that offers the maximum diversity; the number of possible versions of a program is maximized at this value. However, smaller values of p_{NOP} can also provide sufficient diversity, while also lessening the performance impact. For this reason, we also ran our tests with $p_{\text{NOP}} = 30\%$. To measure the impact of profiling, we evaluated three sets of probability ranges for the profile-guided version of NOP-insertion: $p_{\text{NOP}} = 25 - 50\%$, $p_{\text{NOP}} = 10 - 50\%$ and $p_{\text{NOP}} = 0 - 30\%$. In each range, the first parameter is the minimum probability that a NOP is inserted, while the second parameter is the maximum probability. We used the logarithm-based function to derive the per-basic block probability from the execution count.

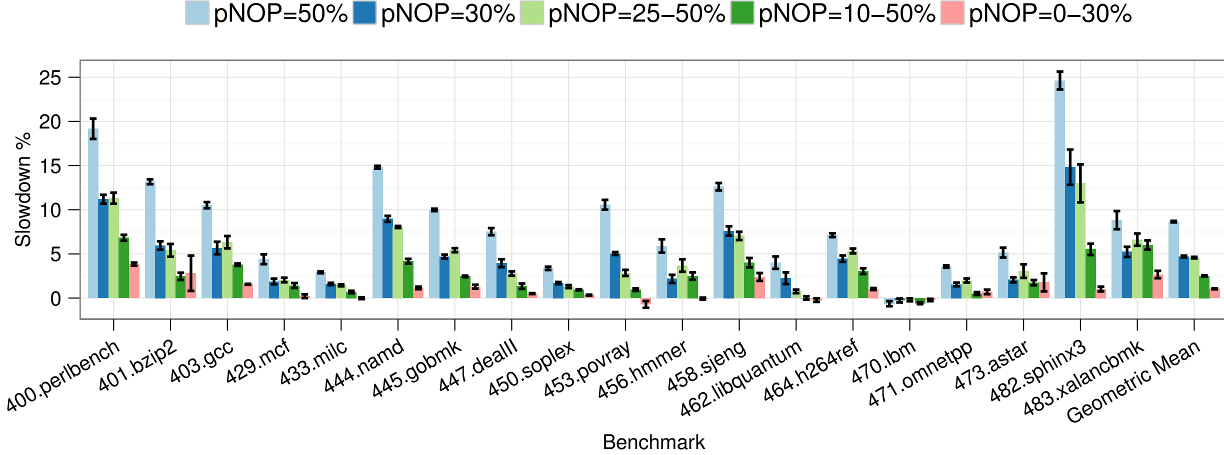

Figure 3.3: SPEC CPU 2006 performance overhead of NOP insertion.

Our results confirm that profiling has a significant impact on the performance overhead of NOP insertion. The performance overhead is around 8% for $p_{\text{NOP}} = 50\%$ and a little less than 5% for $p_{\text{NOP}} = 30\%$, but profile-guided randomization reduces that to 2.5% for $p_{\text{NOP}} = 10 - 50\%$ and 1% for $p_{\text{NOP}} = 0 - 30\%$ (a reduction factor of 5x compared to naive NOP insertion). For the benchmarks where the NOP insertion overhead is highest (400.perlbench and 482.sphinx3), profiling also has the highest impact. For the latter, the impact is reduced from 25% to 5% (for $p_{\text{NOP}} = 10-50\%$) or as low as 1% (for $p_{\text{NOP}} = 0-30\%$). The 470.lbm benchmark has the smallest overhead from NOP insertion; our measurements actually showed a very small performance gain from NOP insertion (under 0.5%), which we attribute to measurement noise. As LLVM does not currently perform any profile-guided optimizations, the performance gains come solely from inserting fewer NOPs in frequently executed code.

Our results also show that both ends of the probability range have an equally significant impact on the performance overhead of NOP insertion. A side-by-side comparison of $p_{\text{NOP}} = 10 - 50\%$ and $p_{\text{NOP}} = 25 - 50\%$ shows that reducing the minimum probability from 25% to 10% decreased the average overhead by half, as Figure 3.3 shows.

3.3.2 Security Impact

We evaluated the security impact of our diversifying compiler by examining how it affects gadgets in diversified executables. We measured this by counting how many functionally equivalent gadgets remain at the same location in the binary after diversification. Our comparison algorithm (called *Survivor*) extracts the `.text` sections from executables after diversification and compares them to the `.text` sections in unmodified original executables. *Survivor* scans through the sections, looking for common instruction sequences—*candidate* matches—ending in free branches such as returns, indirect calls, or jumps. A candidate match is a pair of instruction sequences with identical offsets; one from the original binary and one from the diversified one. For each candidate, we ensure that both sequences decompile to valid x86 code having no control-flow instructions except a free branch at the end. The algorithm then compensates for the effects of our diversifying transformations by removing all potentially inserted NOP instructions from both instruction sequences. Since this step potentially makes the two instruction sequences more similar, our algorithm conservatively *overestimates* the number of gadgets surviving diversification. If the normalized instruction sequences are equivalent, then the algorithm has identified the candidate as a surviving gadget.

Using this strategy, we determined how many functionally equivalent gadgets exist at the same location in a pair of executables. These two properties are a requirement for a code reuse attack like ROP to work on multiple executables without modification. Because we used `.text` section offsets and not absolute addresses, we were able to perform our analysis in an environment where protections such as address space layout randomization (ASLR) [92] do not interfere with results.

We ran our surviving gadgets algorithm on 25 different versions of each SPEC CPU benchmark. Table 3.3 shows the results of our scans. The benchmarks are sorted by total number of gadgets in the original binary (the **Gadgets Baseline** column). Our scans show that, as the binary gets larger and contains more gadgets, the effectiveness of randomization also increases. On the largest

benchmark, 483.xalancbmk, which contains over half a million gadgets, only 0.05% of the original gadgets survive (a reduction in gadgets of around $2000\times$). The impact of profiling on surviving gadgets is small, with the percentage of extra gadgets between 1% and 30% (473.astar is an outlier, as $p_{\text{NOP}} = 0 - 30\%$ gains a massive 254% extra gadgets; this is due to the large difference between $p_{\text{NOP}} = 50\%$ and $p_{\text{NOP}} = 30\%$, not from the profiling optimization itself). However, the increase in surviving gadgets is 30% out of 0.05% of the gadgets available in the original undiversified binary. Finally, we note that 407.lbm is very small; the undiversified C library assembly code is comparatively large to the program itself, increasing the relative percentage of surviving gadgets. This could be easily fixed in practice by also diversifying the C library code. Overall, the absolute impact of profiling on the number of surviving gadgets is negligible.

In practice, it is possible that the attacker is satisfied with successfully attacking some subset of targets. To that end, they will try to find the largest subset of gadgets common to as many binaries as possible, ignoring the undiversified program. To determine how much diversity there is in the binaries, we measured how many gadgets survive at the same location in at least 2 (10% of the population), 5 (20%) and 12 (50%) of the 25 versions. Table 3.4 shows the results of our scans.

The data provide several interesting insights. First, there are more gadgets in total in at least two binaries than there are in the original, undiversified binary. This is because a gadget at offset O in the original binary can be displaced to offset O_1 in some subset of binaries and to offset O_2 in another subset. Therefore, the same baseline gadget is counted several times in the diversified population, once for each offset. Second, adjusting p_{NOP}has a significant impact on gadgets surviving in at least two binaries (for example, for 400.perlbench, going from $p_{\text{NOP}} = 50\%$ to $p_{\text{NOP}} = 0 - 30\%$ increases the number of gadgets from $6,827$ to $11,117$, an increase of 62%). However, this is less significant when looking at a larger subset of the population; although 471.omnetpp shows an increase from 113 to 390 gadgets (a large percentual increase) surviving in at least five versions, this is a very small increase compared to the initial number of gadgets in the binary. Third, we observe that the number of gadgets surviving in at least half the binaries is essentially constant,

regardless of the program size or diversification parameters. The remaining gadgets (around 40 in total) come from the small C library object files that the linker adds to the binary (which we can also diversify, given their source code).

To verify that profile-guided NOP insertion is effective against a concrete attack, we tested our diversification on a vulnerable application. We picked a popular network-facing application (PHP version 5.3.16) and ran two separate ROP gadget scanners to build ROP attacks against the undiversified PHP binary. We used two publicly documented ROP attack frameworks: ROPgadget [102] and microgadgets [56].

As a preliminary step, we verified that the undiversified PHP binary is indeed vulnerable to both these attacks; with both scanners, the program contained enough gadgets to allow the attacker to execute arbitrary code. Next, we built 25 diversified versions of the PHP interpreter, using the highest-performance, lowest-security setting: $p_{\mathrm{NOP}} = 0 - 30\%$. We ran *Survivor* on each diversified version, then re-ran both ROPgadget and the microgadgets scanner on the surviving gadgets to verify the feasibility of an attack. On all diversified versions of PHP, a ROP-based attack was no longer possible, as the remaining gadgets did not provide the required operations for the attack.

Since there exists no standard profiler-friendly training input for PHP, we profiled several different PHP programs, then built 25 diversified versions for each profile. We used several benchmarks from the Computer Language Benchmarks Game [47]: binarytrees, fannkuchredux, mandelbrot, nbody, pidigits, spectralnorm and fasta. Each benchmark stresses different parts of the PHP interpreter (function calls, arrays, loop operations). None of the profiles produced any binary that we could successfully attack, using either ROPgadget or microgadgets.

Benchmark	Gadgets Baseline	p_{NOP} 50%	25 − 50%	10 − 50%	30%	0 − 30%	Gadgets Extra%	Surviving%
470.lbm	344	61.60	61.92	61.80	62.88	62.92	**2%**	**18.29%**
429.mcf	579	55.60	56.92	56.84	55.88	56.68	**1%**	**9.79%**
462.libquantum	709	52.32	52.28	52.28	52.28	52.92	**1%**	**7.46%**
401.bzip2	1191	16.00	16.24	16.24	16.36	17.40	**8%**	**1.46%**
473.astar	1362	16.64	18.56	22.24	46.20	59.04	**254%**	**4.33%**
433.milc	3149	17.16	16.92	17.28	17.16	17.44	**1%**	**0.55%**
458.sjeng	3317	15.08	16.00	16.04	17.24	17.44	**15%**	**0.53%**
456.hmmer	4535	15.56	15.60	15.84	15.84	16.20	**4%**	**0.36%**
444.namd	5322	38.48	39.12	39.60	42.72	43.24	**12%**	**0.81%**
482.sphinx3	6599	15.76	16.28	16.64	16.32	16.56	**5%**	**0.25%**
464.h264ref	16233	16.32	16.44	15.68	16.76	18.76	**14%**	**0.12%**
450.soplex	23885	23.32	23.88	24.84	25.96	24.48	**4%**	**0.10%**
447.dealII	24654	21.20	22.52	22.80	24.92	26.28	**23%**	**0.11%**
453.povray	41954	21.56	22.68	22.36	24.40	26.08	**20%**	**0.06%**
400.perlbench	43065	24.68	25.32	24.20	24.08	25.68	**4%**	**0.06%**
445.gobmk	56032	37.60	39.92	39.52	42.84	43.72	**16%**	**0.08%**
471.omnetpp	75246	45.28	47.20	48.08	49.56	59.16	**30%**	**0.08%**
403.gcc	105039	38.08	37.24	40.40	42.24	48.28	**26%**	**0.05%**
483.xalancbmk	566342	246.80	254.36	253.68	271.24	274.16	**11%**	**0.05%**

Table 3.3: Average surviving gadgets on SPEC CPU 2006 binaries.

We show the average number of surviving gadgets for each benchmark and NOP insertion strategy. **Gadgets Baseline** is the number of gadgets in the undiversified binary, **Extra%** is the extra number of gadgets in $p_{\text{NOP}} = 0 - 30\%$ compared to $p_{\text{NOP}} = 50\%$ (best-to-worst comparison) and **Surviving%** is the ratio of gadgets surviving diversification in $p_{\text{NOP}} = 0 - 30\%$.

	$p_{\mathrm{NOP}}\%$														
	At least 2 versions					**At least 5 versions**					**At least 12 versions**				
Benchmark	50	25 – 50	10 – 50	30	0 – 30	50	25 – 50	10 – 50	30	0 – 30	50	25 – 50	10 – 50	30	0 – 30
470.lbm	586	608	614	602	723	140	173	56	175	180	50	50	46	50	50
429.mcf	1614	1722	1563	1663	1850	79	166	155	212	196	45	45	45	45	45
462.libquantum	871	819	849	1082	1229	137	47	50	152	62	41	41	41	43	41
401.bzip2	913	1085	1195	1145	1910	46	49	43	53	150	44	42	42	42	44
473.astar	1335	1373	1551	1580	2165	48	54	62	56	100	45	44	44	41	48
433.milc	2022	2014	2358	2496	3443	44	51	48	54	65	44	42	42	44	41
458.sjeng	1502	2110	2008	2927	3593	47	48	45	55	78	41	44	44	42	42
456.hmmer	1721	1829	1898	2427	2779	44	44	45	49	50	44	44	44	44	44
444.namd	2189	2449	2524	3509	4225	77	76	71	81	87	54	64	63	64	67
482.sphinx3	2315	2521	2426	5277	4589	42	45	45	51	55	42	44	44	44	44
464.h264ref	3639	4343	5163	7138	7216	46	42	47	46	56	44	41	42	43	49
450.soplex	6652	7300	6952	10314	11013	110	138	104	142	157	44	48	42	50	49
447.dealII	5764	7647	7723	8759	10550	48	53	55	65	66	44	44	44	44	47
453.povray	9878	9658	11002	12702	13425	70	66	67	63	77	44	44	44	41	49
400.perlbench	6827	10380	7935	8361	11117	47	55	46	52	68	44	48	44	42	40
445.gobmk	10896	11974	11898	14574	17739	58	53	63	81	108	42	44	43	44	42
471.omnetpp	17156	17523	17914	60388	29870	113	106	154	419	390	48	47	47	44	48
403.gcc	16825	16572	20443	19243	25343	90	161	160	113	104	16	16	43	16	16
483.xalancbmk	76765	79688	82053	102370	109543	77	108	103	181	186	42	42	16	16	44

Table 3.4: Total surviving gadgets on SPEC CPU 2006 binaries on 25 different binaries.

The columns show the number of gadgets surviving in at least 2, 5 and 12 out of the 25 versions for each combination of benchmark and randomization parameter.

Chapter 4

Librando: Transparent Code Randomization for Just-in-Time Compilers

From their early beginnings, JIT compilers focused on producing code quickly. Usually, they achieve this by optimizing the common case and forgoing time-intensive optimizations altogether. As a result, this leads to highly predictable code generation, which attackers exploit for malicious purposes. This is evidenced by the rising threats of *JIT spraying* [13] and similar attacks on sandboxes in JITs. The former is particularly interesting: JIT spraying relies on JIT compilers emitting constants present in input source code directly into binary code. Due to variable length encoding, attackers can encode and subsequently divert control flow to arbitrary malicious code arranged this way.

This attack vector is innate and specific to JIT compilers. From a security perspective, the state-of-the-art in the field is to address JIT spraying by encrypting and decrypting constants. This addresses the code injection part of JIT spraying, but attackers can fall back on code-reuse techniques. Specifically, return-oriented programming [106] for JIT compiled code is problematic. Instead of finding gadgets in statically generated code (as they would do an a generic return-oriented

programming attack), an attacker uses the JIT compiler to create new binary code containing the necessary gadgets by supplying specially crafted source code. The ubiquity of JIT compilers amplifies this security risk to such a degree that JITs become a liability.

Software diversity addresses code-reuse attacks by attacking its foundation: the software monoculture. By diversifying the binary code, attackers cannot construct reliable attack code, because the binary code layout differs for each end-user. Consequently, diversity increases the costs for attackers, ultimately rendering them too costly. Unfortunately, existing approaches to artificial software diversity do not protect code emitted dynamically by a just-in-time compiler. We address this challenge by describing the first fully automatic technique to diversify existing JIT compilers in a black-box fashion: librando. Similar to the successful frontend-backend separation in traditional compilers, this black-box approach has the advantage over a white-box solution—where developers would manually add diversification directly to JIT compiler source code—of not requiring duplicated work for every existing JIT compiler. Besides the obvious savings in implementation time, the black-box approach allows for faster time-to-market for patches, without having to rely on vendors to supply patches to known vulnerabilities. Another benefit is the added security for legacy JIT compilers available only in binary form, where extra defenses cannot be added by changing the source code.

4.1 Design

The librando library diversifies code generated by a JIT compiler. It reads all code emitted by the compiler, disassembles the code, randomizes it, and then writes the randomized output to memory. Figure 4.1 shows the structure of dynamically generated code, as a function call graph.

The library diversifies dynamically generated code under one of the following models (illustrated in Figure 4.2):

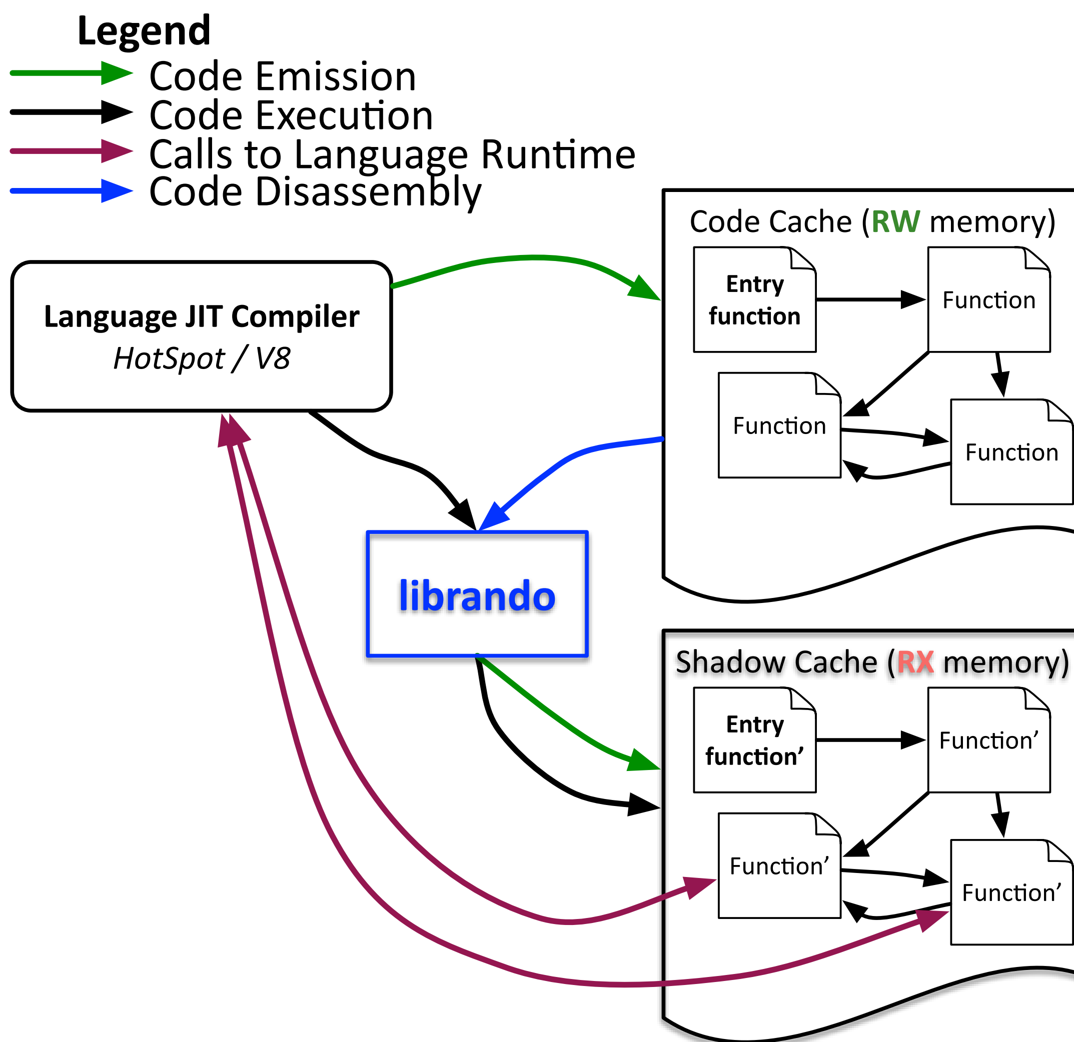

Figure 4.1: A JIT compiler with librando attached.

The control flow graph of the dynamically generated program is also shown. Greyed out edges represent branches that are redirected or never taken after diversification.

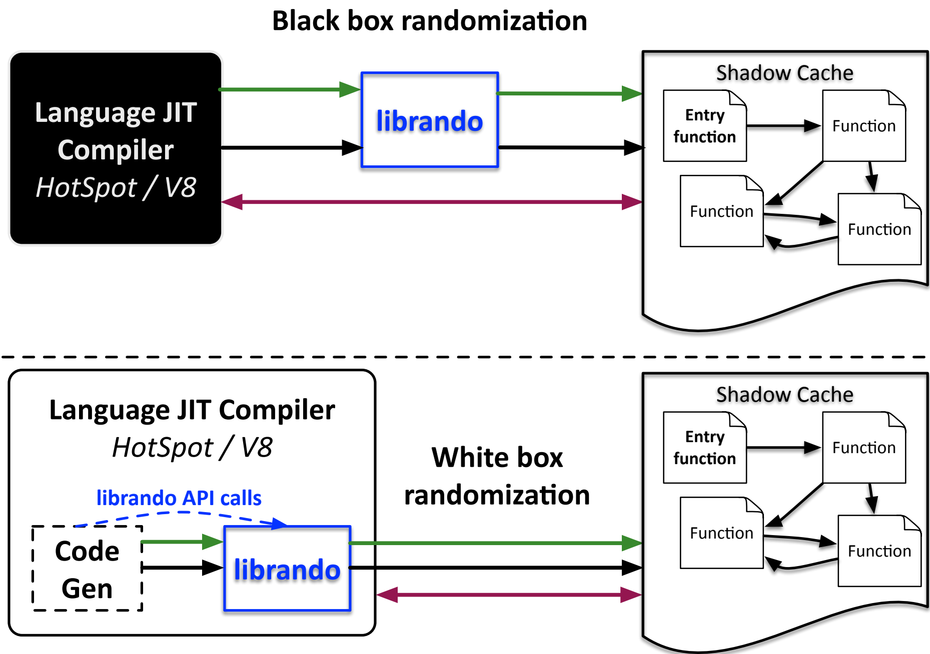

Figure 4.2: Black and white box diversification architecture.

Black box diversification with no assistance from the compiler (the compiler is a black box and the library has no knowledge of compiler internals). The library attaches to the compiler and intercepts all branches into and out of dynamically generated code, without requiring any changes to compiler internals.

White box diversification with some assistance from the compiler (the library has some knowledge of compiler internals). The code emitter notifies librando through an API when it starts running undiversified code. The library provides the diversified code addresses to the compiler, and the compiler executes diversified code directly. We change all compiler branches into emitted code to use the addresses returned by librando. This approach is intended as a middle ground between the previous model and a manual implementation of randomization for each compiler, and it requires that compiler source code is available.

As the first security measure, librando prevents execution of all code generated dynamically by the compiler. Instead, the library disassembles the code into a control flow graph, diversifies every

Undiversified

```
@0x100   CMP RAX, QWORD PTR [R13 - 40]
@0x104   JNE 0x116
@0x10A   MOV RAX, 0x200
@0x114   JMP 0x120
```

Diversified

```
@0x30A   CMP RAX, QWORD PTR [R13 - 40]
         NOP [90]
@0x30F   JNE 0x323
@0x315   MOV RAX, 0x200
         NOP [66 90]
@0x321   JMP 0x330
```

Figure 4.3: Block contents and diversification example.

basic block in this graph, and then writes the diversified blocks to a separate executable area. All branches (including function calls and returns) to the original undiversified code are redirected to the diversified code (Figure 4.3 shows an example of a diversified block). While the undiversified code remains available and writable, the compiler or HLL program cannot execute it anymore. The library intercepts all memory allocation functions (the `mmap` function family on Linux) that return executable memory, then removes the executable flag on all intercepted allocation requests. The library also keeps an internal list of all memory allocated by these requests, and uses this list to distinguish between accesses to undiversified code and all other memory accesses.

In the black box diversification model, the library transparently intercepts all branches to protected blocks. To do this, we protect all undiversified code pages against execution, then catch all attempts to execute these protected pages. The library installs a handler for the segmentation fault signal (`SIGSEGV` in Linux), which is raised whenever a processor instruction attempts to access memory it does not have permission for. Whenever the processor attempts to execute a non-executable

page, it triggers a page fault in the MMU. The operating system handles this fault and calls our `SIGSEGV` signal handler. The library then redirects execution to diversified code.

While static disassembly of binary code accurately is impossible in the general case [28] (due to the need to distinguish code from data), dynamic disassembly is much simpler for one reason: we only disassemble bytes that we are certain represent code. Our use of signals guarantees this: the signal handler is always called when the JIT compiler executes an undiversified instruction at some address, so the bytes at that address (and the entire block starting at that address) are guaranteed to be code. The same is true for all blocks in the control flow graph containing that instruction, since we follow direct branches to find more code.

We make a few simplifying assumptions about the emitted code that reduced our implementation effort significantly:

No stack pointer reuse On the x86 architecture, the stack pointer register (`RSP`) is available as a general-purpose register. The x86 64-bit ABI reserves this register for its intended purpose, as stack register. However, compilers only need to follow the ABI when calling into external libraries and system code; they can ignore its guidelines inside emitted code. Code emitted by a JIT compiler might use another register to keep track of the HLL stack, and re-use `RSP` for another purpose. Generally, JIT compilers do not do this in practice, so we assume that this register always points to the top of a valid native stack. This allows both code emitted by the compiler and by librando to use several stack manipulation instructions, such as `PUSH`, `CALL` and `RET`.

No self-modifying code While it is possible for a JIT compiler to emit code that modifies itself, this is usually not the case. We assume that only the compiler can modify code. Once emitted code starts executing, it remains unchanged. This assumption simplifies our implementation, as we only have to detect code changes originating in the compiler itself; therefore, we can

safely discard old versions of modified code, without the risk of having to continue execution inside discarded code.

Calls paired with returns Compilers often use the `CALL` x86 instruction for calls and `RET` for the corresponding returns, but not always. The former pushes the return address on the native stack and branches to a function, while the latter pops the return address and branches to it. There are equivalent instruction sequences with the same behavior which a compiler can use instead, perhaps to push/pop the return address to another stack. However, there is a performance penalty from using these sequences, as modern processors use an internal return address stack to improve branch prediction for `CALL`/`RET` pairs, and only for those pairs. A compiler optimized for performance always uses this combination for function calls, simplifying our implementation as well. Consequently, librando only needs to analyze and rewrite these instructions when redirecting function calls.

Our technique diversifies code by relocating emitted code blocks, rewriting the code, and inserting instructions. To preserve HLL program semantics, a diversification library must be transparent to both the compiler and the compiled program. We identified several pieces of program state which librando must preserve when rewriting code (related work [17, 76, 105] identifies a superset of these for general-purpose dynamic binary rewriters):

Processor register contents including the flags register. The library inserts instructions that hold intermediate values in registers. Also, some inserted instructions (like `ADD` and `XOR`) modify the processor flags register. The library attempts to only add instructions that do not change any registers or flags; where this is not possible, it temporarily saves the register(s) to the stack, performs the computation, then restores the register(s).

Native stack contents Some JIT compilers use the native stack for HLL code, while others switch to a separate stack when branching to HLL code. To support the former, the library must preserve all contents of the native stack inside diversified code. This must be true not only

during execution of dynamically generated code, but also during calls into the runtime, as the runtime itself may read or write to the native stack. For example, the V8 runtime walks the native stack during garbage collection to find all pointers to data and code. Changing any such pointer or any other data on the stack leads to program errors and crashes. Therefore, native stack contents must be identical when running with and without librando. This includes return addresses; when a diversified function calls another diversified function, the former must push the undiversified address on the stack. Figure 4.4 illustrates this transformation. The original call pushes the address of the next instruction on the stack, then jumps to the called function. We replace it with a `PUSH`/`JMP` pair that pushes the undiversified return address. There is one exception to this restriction: we consider any memory past the top of the stack (below `RSP`) to be unused, so we use it to save registers.

Undiversified code instructions JIT compilers frequently emit code with temporary placeholders, then later replace them with other instructions. The compiler usually uses fixed instruction sequences for these placeholders, so it first checks their contents before patching. In other cases, certain instruction sequences are used to flag properties of emitted code. The library cannot modify any of the original, undiversified code in-place, as it cannot distinguish between regular code and these placeholders. Even if the compiler never reads back the former, it may read back and validate the contents of the latter, then crash or execute incorrectly. The library is required to preserve the undiversified code at all times, as originally emitted by the compiler.

POSIX signals Some JIT compilers (such as HotSpot) intercept POSIX signals and install their own handlers. We intercept the `SIGSEGV` signal to catch execution and write attempts on undiversified code, but pass all other signals (including `SIGSEGV` we do not handle) back to the JIT compiler.

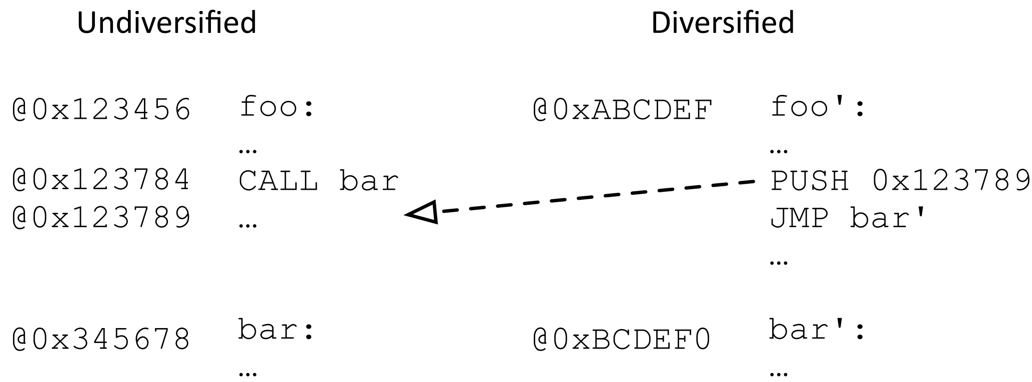

Figure 4.4: Rewriting the `CALL` instruction.

4.1.1 Code Relocation

In our control flow graph, a block is a maximal continuous run of instructions, so that the block ends in an unconditional branch instruction, but contains no such instruction inside. We also break blocks at function calls (the `CALL` instruction) and returns, but not at conditional branches; a basic block can have one or more conditional branches inside. However, we use one heuristic to split blocks: whenever a block contains a number of consecutive zeroes over a given threshold, we break the block after a small number of those zeroes. This is needed because compilers sometimes emit code only partially (or lazily), initializing the remainder with zeroes. The compiler emits the rest of the code at a later moment, after some event triggers generation of the missing code. We implemented this heuristic to support the V8 compiler, which uses lazy code generation.

Many JIT compilers perform garbage collection on generated code, discarding unused code and reusing that space for new code. The compiler will write and later execute this new code in place of the old version. This happens frequently in the modern JIT compilers we investigated. We detect such changes, discard the diversified versions of old blocks, then read the new blocks and integrate them into the existing control flow graph. To detect all changes to a block, librando marks

it read-only after diversification using the `mprotect` function on Linux. If the compiler writes to the block later, the library allows the write to succeed, but marks the block as dirty.

The `mprotect` function has one significant restriction: it can only change access rights on zones aligned to hardware pages; on x86, a page is 4096 bytes by default. There are several issues to consider when using `mprotect` to protect a block. First, a single page may contain more than one block. Second, a block may be spread across several consecutive pages. Third, for each page, one or two blocks might cross its boundaries, one at each end. Figure 4.6 shows examples of all three cases. Instead of marking a block as read-only, librando actually marks all pages containing that block as read-only. However, those pages may contain other blocks that the compiler might never modify.

After librando marks a page as read-only, the operating system starts notifying the library of all writes to read-only pages. We also use POSIX signals to receive these notifications. The operating system calls the `SIGSEGV` signal handler each time a processor instruction tries to write to a protected page, as long as the page is protected. However, this occurs before the actual instruction writes the data; we have to either emulate the instruction itself (while keeping the page read-only), or make the page writable and allow the original instruction to execute. As emulating x86 instructions correctly requires significant development effort, we choose the latter solution. However, once we allow writes to a page, there is no way to catch each write to that page separately; the processor will allow all writes to succeed without generating a page fault. At this point, the only information available to librando is that at least one byte in that page may have been modified, but nothing more. Therefore, we mark all blocks contained in a writable page as dirty. The next time librando intercepts a branch to undiversified code, it checks all dirty blocks to verify that the VM has actually modified their contents. The library only discards and re-diversifies changed blocks, keeping unchanged blocks as they are. Figure 4.5 shows this process per block, in the form of a finite state machine.

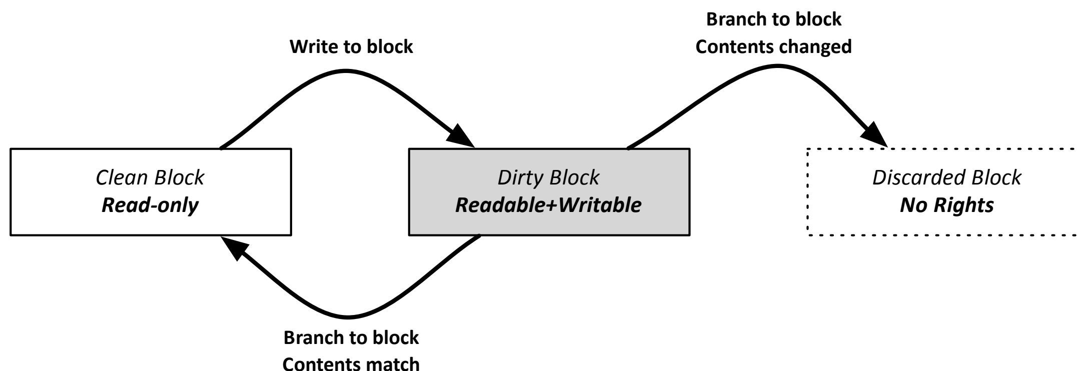

Figure 4.5: Per-block finite state machine that handles block changes.

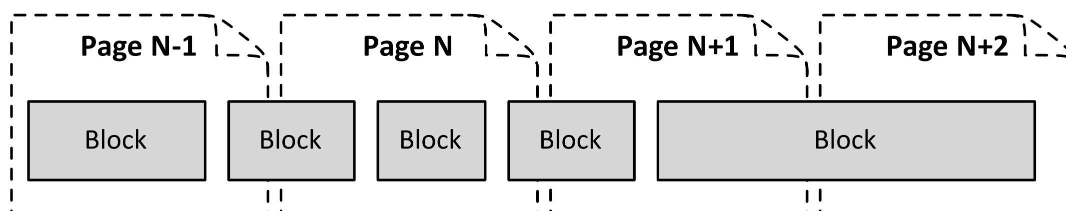

Figure 4.6: Blocks spanning several memory pages and crossing page boundaries.

To check whether a block has been modified, the library compares the new contents of the block against its original contents. However, this requires that two copies of each block be stored: the original and current code. To save memory space, we do not store both copies in memory; instead, each block stores a hash code of its original contents. Every time librando checks if a block has been modified, it hashes the current contents of the block and compares the hash code to the one stored in the block. If the hash codes match, librando assumes that the block has not been changed; otherwise, it discards the block. While there is a very small possibility of collision (where the contents of a block change, but the hash remains the same), we performed all our experiments and benchmarks successfully with this optimization. A librando user has the option of disabling hashing and verifying the entire contents of blocks, which guarantees correctness.

The user attaches librando to a JIT compiler in one of several ways: either by linking it (statically or dynamically) to the compiler, or through the `LD_PRELOAD` environment variable on Linux. The

latter mechanism preloads a library into a process at program start-up time, allowing the library to override some of the program's symbols. We used the latter approach in our evaluation, but the former is preferred in an actual deployment for increased security.

4.1.2 Diversification

The main security benefits from librando come from rewriting the generated code. We propose two rewriting techniques that harden the generated code against attacks.

NOP Insertion For our first randomization technique, we once again used Algorithm 1 from Chapter 3. In contrast with 3, we used another set of instructions: the smallest three NOPs from the set of canonical NOPs recommended by the Intel architecture manual [58]: `90`, `66 90`, and `0F 1F 00`. This changes makes the distribution of the NOP-induced displacement (how far from its original location an instruction is) more uniform, at a minor cost in security (the attacker can now jump into the middle of the 2- and 3-byte NOPs).

Constant Blinding We previously described JIT spraying as an example of a code injection attack against JIT compilers. In general, these attacks rely on the compiler emitting native code that contains some binary sequence from the source-program as-is. In the particular case of JIT spraying, this sequence is a set of 32-bit constants emitted as immediate operands to x86 instructions. Recently, JavaScript compilers have started to implement their own defensive measures particular to this attack [101]. There are two ways that a HLL program value winds up in the executable code region: the compiler stores the value either close to the code (without executing it as code), or as an immediate instruction operand. In the former case, NOP insertion shifts the location of the value by a random offset, making its location hard to guess. For the latter case, there is one simple solution based on obfuscation: emit each immediate operand in an encrypted form, then

Undiversified

```
MOV EAX, 0x12345678
```

Diversified

Randomized value

Randomization cookie

```
PUSH R8
MOV EAX, 0xB1AAB59C
MOV R8D, 0x6089A0DC
LEA R8D, [R8D + EAX]
MOV EAX, R8D
POP R8
```

Figure 4.7: Blinding the immediate operand of an instruction.

`12345678h` is replaced with `B1AAB59Ch`+`6089A0DCh` (modulo 2^{32}).

decrypt its value at run-time using a few extra instructions. We pick a random value (a *cookie*) for every operand, blind the operand using the cookie, emit the original instruction with the blinded immediate, then emit the decryption code. While other implementations use an XOR operation for encryption, we do not use it since it alters the arithmetic flags, so librando would have to save them. Fortunately, the processor provides an addition instruction that leaves the flags intact: `LEA`. Therefore, we blind the original value by subtracting the cookie from it, then add an immediate-operand `LEA` to add the cookie back. Figure 4.7 shows an example of this transformation.

The x86 architecture supports 8-, 16-, 32- and 64-bit immediate values for many instructions. We encountered only the latter two types in most code we analyzed, so we implemented only these sizes; the others can be trivially added. Table 4.1 shows all x86 instructions that accept a 32-bit immediate. There is only a single instruction that accepts a 64-bit immediate (the `REX.W + B8` encoding of `MOV`). Every time librando encounters one of these instructions, it transforms the instruction to the equivalent encrypted sequence. Each instruction from Table 4.1 was sufficiently

Operation	Form	Opcode
MOV	MOV EAX, imm32	B8
	MOV ECX, imm32	B9
	...	
	MOV EDI, imm32	BF
	MOV reg, imm32	C7
PUSH	PUSH imm32	68
IMUL	IMUL reg, reg, imm32	69
TEST	TEST EAX, imm32	A9
	TEST reg, imm32	F7
Arithmetic	op reg, imm32	81
	ADD EAX, imm32	05
	OR EAX, imm32	0D
	ADC EAX, imm32	15
	...	
	CMP EAX, imm32	3D

Table 4.1: x86 instructions with 32-bit immediate operands.

The arithmetic class contains `ADD`, `ADC`, `SUB`, `SBB`, `AND`, `OR`, `XOR` and `CMP`.

different from the others, so we implemented different blinding code manually for each type of instruction.

4.1.3 Optimizations

Intercepting branches to all generated code and rewriting their contents has a cost in program performance, as our evaluation will show. We propose several optimizations to our approach to reduce this cost as much as possible.

The Return Address Map One of the restrictions of our design was transparency of the native stack. Even when executing diversified code, the native stack must have the same contents as it would have under undiversified code. We meet this requirement through one change: we rewrite call instructions to push undiversified return addresses on the stack, as shown in Figure 4.4. However, this has one significant drawback: the later `RET` matching the replaced call pops the

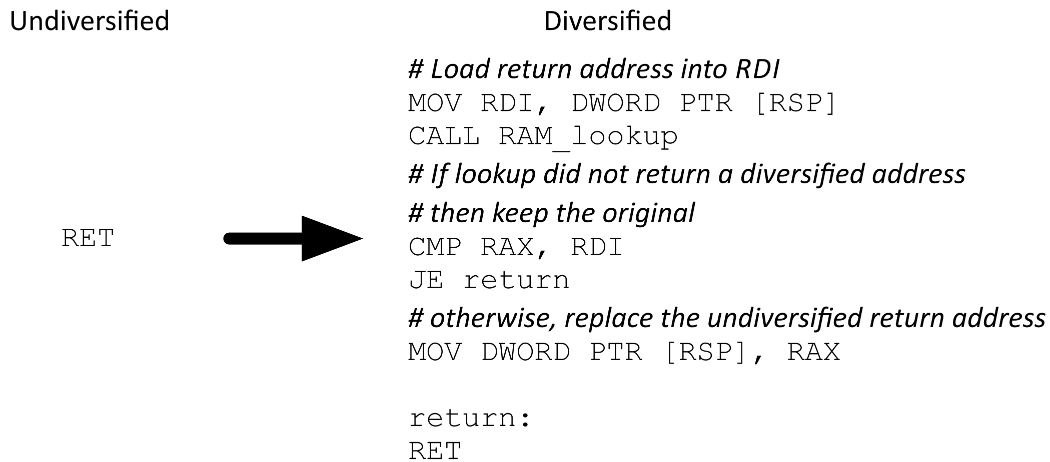

Figure 4.8: Rewriting the RET instruction to use the *Return Address Map*.

undiversified address and branches to it. Since this address is now non-executable, this generates a page fault and a call to our signal handler. As the SIGSEGV handler is called after the processor interrupts the execution of another instruction, the operating system saves a lot of processor state before calling the handler; this makes signal handlers very slow. For this reason, RET instructions in diversified code are also expensive.

To improve performance, we prevent the SIGSEGV signal from being triggered. We rewrite return instructions, adding code to handle the case when the return address is in undiversified code that also has a corresponding diversified address. In as few instructions as possible, the diversified RET instruction now looks for the return address in a data structure that maps undiversified to diversified addresses (we call this data structure the *Return Address Map*). If an entry is found, execution continues in diversified code; otherwise, the original address is used as-is. Figure 4.8 shows how this optimization rewrites the RET instruction.

We store the address mapping in a hash map. We require a data structure that supports three operations: lookup, insertion and removal. The library adds the addresses of a block to the map whenever it diversifies the block, then removes the addresses when it discards the block. One significant factor in the choice of data structure is that lookups are far more frequent than the other

operations (every `RET` performs a lookup), so the data structure implementation must perform the former as efficiently as possible. We use a cuckoo hash map for this goal [89], as it provides both fast constant-time lookups (we implemented a lookup function in 28 lines of assembly code) and good memory utilization.

As returns are essentially just indirect branches (a pop followed by a branch to the popped value), we also use this data structure to optimize all other indirect branches. We extend the map to all basic block addresses (not just return addresses), then prepend hash map lookups to all branches with unknown targets (where the target is not a direct address, but one loaded from a register or memory).

White Box Diversification With the *Return Address Map* handling all diversified-to-diversified-code indirect branches, and the rewriting of all direct branches to target diversified code, the signal handler only intercepts branches entering or exiting the diversified code (mainly the compiler and the language runtime). Some HLL programs make many calls to the runtime (either explicitly as function calls, or implicitly through language features or inline caches), so the overhead from the signal handler remains significant. This overhead is difficult to reduce without making changes to the compiler itself, so the next step in optimization is **white box diversification**. Under this model, librando provides an interface to the compiler, which the latter uses to notify the former of branches to generated code, with the goal of avoiding the signal handler. JIT compilers frequently have one or a few centralized places in their source code that all jumps to generated code pass through; by manually inserting calls to librando in these few places, we can significantly reduce the overhead of compiler-to-undiversified-code jumps. For example, we identified a single function in V8 (called `FUNCTION_CAST`) that returns the memory address of a JavaScript function; by inserting a single line that calls librando from `FUNCTION_CAST`, we completely intercepted all indirect jumps and function calls from V8 to generated code.

Function	Description
`rando_redirect(addr)`	Diversifies (if needed) the code starting at `addr` and returns the diversified address
`rando_hash_return()`	Checks the return address of the current function and replaces it with the diversified equivalent

Table 4.2: Application Programming Interface provided by librando.

In many cases, functions in the language runtime are implemented in a host language such as C; generated code calls these functions directly (using the `CALL` instruction), and control flow returns from the runtime to generated code not through explicit branches, but through function returns. A host language compiler (gcc, for example) generates these returns automatically, so we cannot manually insert calls to librando for all of them. Also, programs written in a higher-level language usually do not have direct access to the native stack, so they cannot change the return address through host language code. For this reason, we implemented a compiler-level extension for LLVM that adds a new function attribute–`rando_hash_return`. The compiler adds calls to librando at the return sites of all functions marked with this attribute; whenever such a function returns, librando checks whether the function returns to undiversified code or not. As there is no automatic analysis that can determine whether a runtime function is called from the runtime or not, the librando user has to find all functions that can be called from generated code and manually add this flag to all of them[1].

Table 4.2 shows the two operations that librando provides to the compiler. Using only this small API, most (if not all) invocations of the signal handler disappear.

[1]Most such functions in V8 are defined using a macro called `RUNTIME_FUNCTION`, so we easily added the flag by searching for all uses of this macro.

4.2 Evaluation

We implemented librando as a dynamic library for Linux and loaded it into compilers using the LD_PRELOAD mechanism. We evaluated the performance impact of librando on two JIT compilers: V8 (the JavaScript compiler included in the Google Chrome browser) and HotSpot (the Java client compiler from Oracle). We ran all benchmarks on a 2.2GHz Intel Xeon E5-2660 system with 32 GiB of memory, running Ubuntu Linux with kernel version 3.2.0.

First, we benchmarked V8 using the benchmark suite included with the compiler. These benchmarks stress different parts of the compiler (integer operation optimizations, floating points optimizations, inline caches, regular expression implementation) differently, and the overhead of librando varies accordingly. We sought to determine the performance impact of control flow rewriting (without diversification), our proposed optimizations, as well as the diversification techniques. We first tested librando under the black box model without optimizations or randomizations, and gradually added them one by one in the following order: the `Return Address Map` (RAM), NOP insertion, then constant blinding. We then implemented white box randomization by manually changing V8 to interact with librando (we changed 70 lines of code in total), then ran all the benchmarks again to measure the effects of this interaction.

Figure 4.9 shows the results of our tests. The benchmark suite reported both a per-test score, as well as the geometric mean of all benchmarks (the `Total` column). The overall impact of librando on the V8 benchmark suite is around 3.5x, with all optimizations and randomizations enabled; without randomizations, this overhead is approximately 2.7x. This overhead is not uniform across all benchmarks; `NavierStokes`, for example, shows an overhead between 1.2× and 1.4×. The fastest benchmarks (as well as the second best-performing one, `Crypto`) have a similar structure where the impact of our approach is small: nested loops of primitive numeric operations, where the types of variables are mostly static (either integers or numbers). Most of the time spent in these benchmarks is in a highly-optimized loop that does not branch or call outside the loop.

The V8 compiler emits very well optimized code once for such structures, then executes it for a substantial amount of time. Primitive operations are implemented directly as native instructions, so they contain very few calls to the runtime. Compared to Chapter 3, 50% NOP insertion on V8 has a larger overhead of about 30%. This is more consistent with the SPEC CPU results for the 400.perlbench benchmark, which is also an implementation of a dynamically typed language. This leads us to believe that profile-guided NOP insertion could also have a significant impact on V8.

Second, we benchmarked the HotSpot client compiler for Java, using the Computer Language Shootout Game benchmarks [47]. Figure 4.10 shows the per-benchmark slowdown factor for each benchmark, as well as the geometric mean over all four tests. Note that the overall slowdown is smaller for HotSpot; the main cause for this is that Java is a statically typed object-oriented language, where the data types of values are known ahead-of-time. While Java object instances can have different types at runtime (based on the program class hierarchy), primitive values have fixed types and primitive operations can be emitted very efficiently on the first attempt. HotSpot needs to collect substantially less type information than V8, as so much is already available, so it requires a lot less time to fully optimize a program. In our Java benchmarks, we see the same behavior we saw in the `NavierStokes` test for V8. We see an average slowdown of about 1.08× (a percentual slowdown of 8%) for rewriting and around 1.15× (15%) with full diversification (a total NOP insertion overhead under 7% is comparable to our results from Chapter 3). However, as the impact of rewriting is now much smaller, we start to notice a significant impact from NOP insertion. Since so much of the benchmarks' execution time is now spent in a very small loop (a few instructions in length), every extra instruction starts to count. On `fannkuchredux`, NOP insertion at $p_{\text{NOP}} = 0.5$ almost triples the overhead (from 15% to almost 40%).

In addition to these tests, we also ran the same benchmarks on V8 and HotSpot without the *Return Address Map* optimization. V8 showed an average slowdown factor of 50× (with a maximum of 232× for `EarleyBoyer`), much higher than the 2.7× slowdown for the optimized version.

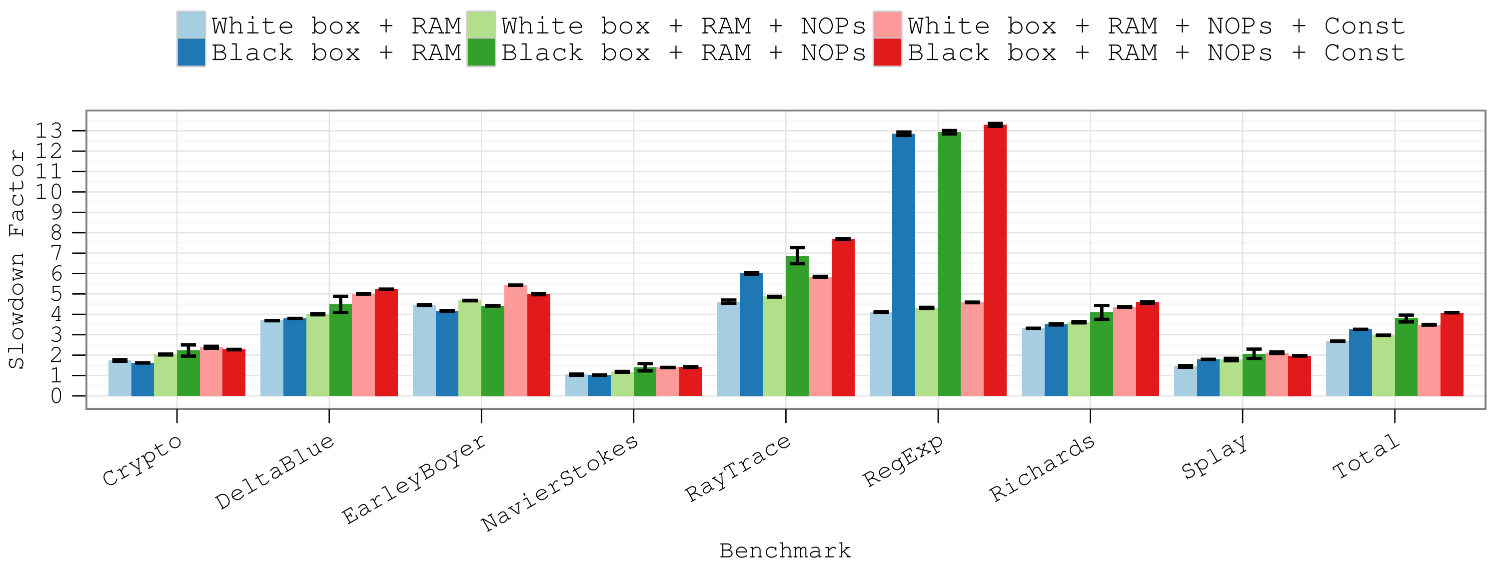

Figure 4.9: V8 benchmark results for librando.

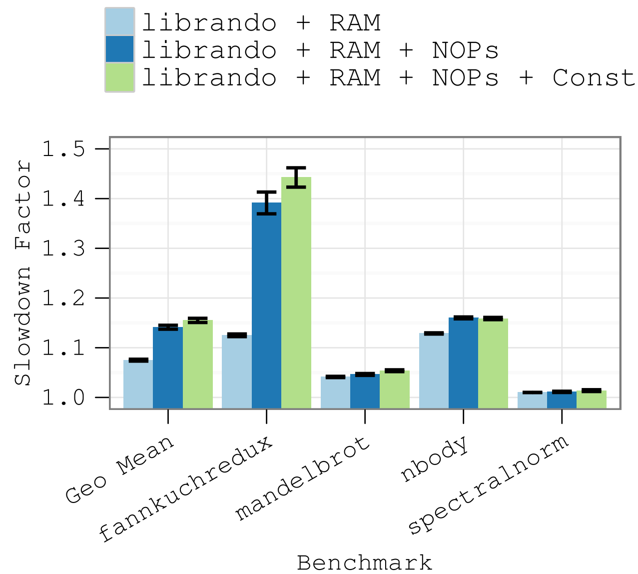

Figure 4.10: HotSpot benchmark results for librando.

The impact on HotSpot is less severe (about 1.6× slowdown without RAM as opposed to 1.08× with it), but still substantial. The Return Address Map optimization is crucial to librando performance. However, we leave further enhancements to our choice and implementation of data structure (cuckoo hash with hand-implemented lookup) to future work, as some of our experiments showed that there is still room for some improvements. We ran the Language Shootout benchmark binarytreesredux (a recursive binary tree implementation in Java that makes extensive use of function calls) on HotSpot and observed a slowdown of approximately 70x, even with the RAM optimization enabled. Further profiling of both this benchmark and all V8 benchmarks showed that around 25% of time spent inside librando takes place inside the RAM lookup function, even with our optimized implementation (28 assembly instructions in total).

Overall, the performance impact depends greatly on the structure and goal of the HLL program, but also on features of the HLL. As we have shown, a statically typed but just-in-time-compiled language has much lower diversification overhead than a dynamically typed program. Another factor to account for is the ratio of time spent in generated code versus time spent in the compiler

or runtime. This technique has higher overhead when new code is generated with high frequency (as is the case with dynamically typed languages), and on other transitions from generated code to the outside. However, synthetic CPU-bound benchmarks are not completely representative of the average workload of a JIT compiler; these benchmarks perform very little input/output operations, and the time spent in the few such operations is small. In contrast, real-world code interacts much more with the rest of the system, and also makes much more use of external libraries. When the execution time of a program is dominated by input and output, or by calls to libraries, the overhead should be much smaller. Our results show an upper-bound for this overhead. This upper-bound is also not necessarily significant in a real-world application; for example, we linked the Chrome browser to librando and used it to visit various web pages, some only containing static content and some using JavaScript heavily. We observed no noticeable degradation in browsing experience.

Chapter 5

Readactor: Execute-only Pages for JIT Compilers

Dynamically generated code, either from a JIT compiler like V8 or from a dynamic rewriter like librando, changes frequently. As Figure 2.2 shows, code generators allocate their code cache with RWX page rights to avoid having to frequently change page permissions. This creates a significant security hole that attackers can exploit [111].

Another problem presents itself in information disclosure attacks. Even if the attacker cannot directly tamper with the code cache, he or she can direct the compiler to compile attacker-controlled source code which generates predictable native code. Applying diversification on this native code, e.g., using librando as presented in Chapter 4, is not sufficient against an attacker who can simply read the code and undo all diversifying transformations (especially when the attacker can run arbitrary HLL code, such as JavaScript). Therefore, the next step to securing JIT-compiled code against attackers is to hide the code from the attacker, allowing the execution of this code but preventing anyone from reading it. Researchers have recently proposed several approaches to hiding executable code from attackers—XnR [6], HideM [49] and Readactor [34]. These approaches

mainly (but not exclusively) focus on hiding statically compiled code, leaving dynamically compiled code as future work. In this chapter, we present the design and implementation of code hiding for the code cache of the V8 JavaScript compiler, as part of Readactor.

5.1 Readactor Design

We implemented our work as part of the Readactor project, described in greater detail in [34]. Readactor is a code-hiding hypervisor that uses hardware virtualization extensions to prevent read access to code pages, while still allowing their execution. Figure 5.1 shows the high-level overview of this project, which consists of a modified ahead-of-time compiler that generates special (readacted) hardened execute-only binaries, and a modified Linux kernel that runs these binaries. Readactor completely separates read-only program data (such as global constants) from executable code, and stores the latter in execute-only pages. The compiler also performs additional hardening transformations, which are beyond the scope of this discussion. In this dissertation, we focus on adding support for dynamically generated code to Readactor. To secure JIT compilers against information disclosure attacks, developers must either manually add support to the compiler for execute-only pages, or use a black-box approach such as librando.

The second component of Readactor, and the one that is relevant to our changes, is the hypervisor. This is the operating system component that enforces execute-only permissions on code pages, preventing attackers from accessing program code. Figure 5.2 shows the structure and operation of the hypervisor. It sits between the operating system and the system memory management unit (MMU), enforcing the execute-only permissions for protected pages on every memory access. When the operating system (OS) loads a readacted program, the OS maps the data pages of the program as regular pages, but maps the code pages as execute-only in collaboration with the hypervisor. We later discuss how the hypervisor and operating system distinguish between regular and readacted pages.

The hypervisor is currently implemented as a Linux kernel module that implements a minimal pass-through hypervisor. This design is required as Readactor uses a modern virtualization feature in Intel processors called *Extended Page Tables* (EPTs)[1]. EPTs allow us to map pages as execute-only, but they are only available under a processor mode known as VMX non-root mode (the mode usually used to run guest virtual machines). At boot time, the processor starts in root (or host) mode, so Readactor switches the entire system to non-root (guest) mode. Alternatively, Readactor can be implemented as a modification to current hypervisors that support EPTs (such as Xen [120] or VMWare [115]).

EPTs add an extra layer of translation during the translation process from virtual to physical addresses. After the MMU translates virtual addresses to physical ones (more specifically, *guest physical addresses*), it performs a second translation from *guest physical addresses* to *host physical addresses*. The EPTs specify the details of this second-level translation. This is generally useful for virtualization, but Readactor uses this feature for code protection. Unlike the regular paging translation mechanism, which does not contain a readable/non-readable permission bit for memory pages and thus force any accessible page to be readable, EPTs provide such a bit and enforce it. If this bit is not set in an EPT for a memory page, then that page is not readable by software. Figure 5.3 illustrates how the guest-to-host physical pages are mapped.

The OS-application interface of Readactor is simple: the operating system provides execute-only pages to applications on demand, e.g., when the application calls `mmap` on Linux, as well as allow the application to change page permissions at will. Existing OS interfaces trivially support this, so no new system calls or other interfaces are required. However, since Readactor split page permissions between the regular page tables and the EPTs, the operating system and the hypervisor must collaborate to keep the two sets of tables synchronized, even when applications frequently change permissions. We investigated two different ways of performing this synchronization:

[1]AMD processors implement a similar feature under the name *Rapid Virtualization Indexing*.

Method A maps each guest physical pages to a single host physical page using the identity mapping. Whenever the OS needs to change permissions on a physical page, it asks the hypervisor to update the EPT for that page and set or clear the readable bit. For every update, there is a transition from the OS into the hypervisor and back (in Intel terminology, the system switches to VMX root mode and back).

Method B maps each host physical page twice as two guest physical pages (one with full access rights, and one execute-only), and let the OS pick between the two. More specifically, we map the entire host physical address space once in the guest with full rights and once as an execute-only space. With this approach, the hypervisor never has to change the EPTs, so no transitions between the OS and hypervisor are required. There is one disadvantage to this approach: it reduces the maximum allowed system memory in half. Modern Intel processors support physical addresses of 36/39/40/48 bits, which correspond to maximum sizes of 64GB/512GB/1TB/256TB. While some older processors still use the smallest address size and there are currently systems with enough physical memory to use all address bits, newer processors support far more memory than any current system contains, so reducing the address space in half is acceptable.

For statically compiled programs, early Readactor experiments showed the two approaches to be identical in performance, as the only time a program changes the permissions on executable pages is at loading time. However, JIT compilers frequently change page permissions on the code cache (switching between writable and executable permissions), so the synchronization strategy can have a large impact on performance. Section 5.3 discusses this impact and shows the results of our evaluation on both approaches.

To support execute-only pages, dynamic code generators can use the existing memory allocation mechanisms, with only a small change in memory allocation policy. The following section discusses all additional challenges we encountered when adding execute-only support to a JIT compiler.

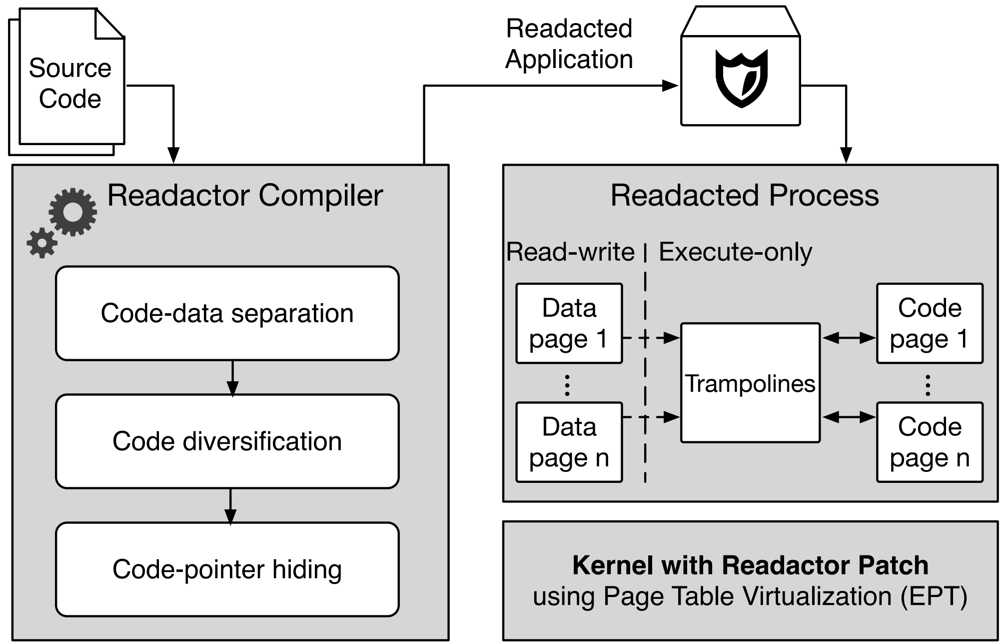

Figure 5.1: Readactor system overview.

Our compiler generates diversified code that can be mapped with execute-only permissions and inserts trampolines to hide code pointers. We modify the kernel to use EPT permissions to enable execute-only pages.

5.2 V8 JIT Compiler Protection

To make Readactor practical and comprehensive, we extended our execute-only memory to support dynamically compiled code. This section describes how this was achieved for the V8 JavaScript engine which is part of the Chromium web browser. We believe that the method we used generalizes to other JIT compilers (this generalization could be performed manually for every JIT, or automatically using a solution like librando).

Modern JavaScript engines are tiered, which means that they contain several JIT compilers. The V8 engine contains three JIT compilers: a baseline compiler that produces unoptimized code quickly and two optimizing compilers called CrankShaft and TurboFan. Having multiple JIT compilers lets the JavaScript engine focus the optimization effort on the most frequently executed code. This matters because the time spent on code optimization adds to the overall running time.

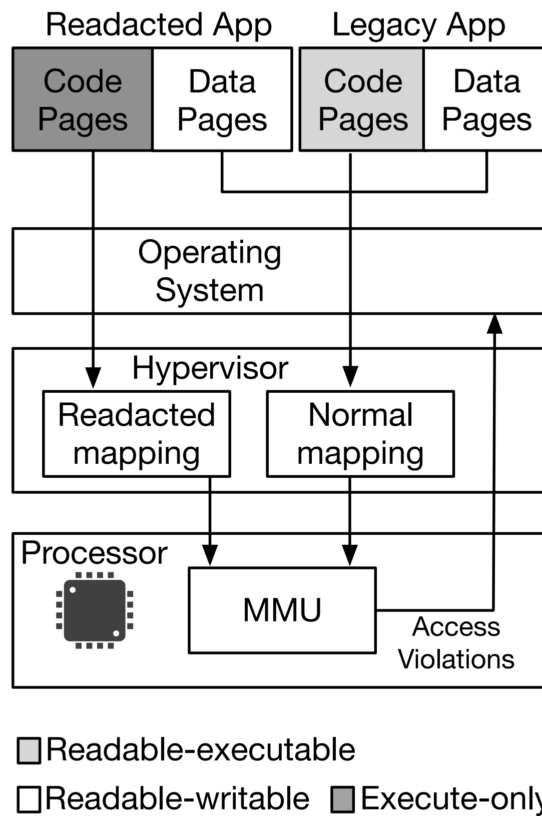

Figure 5.2: Readactor hypervisor overview.

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.

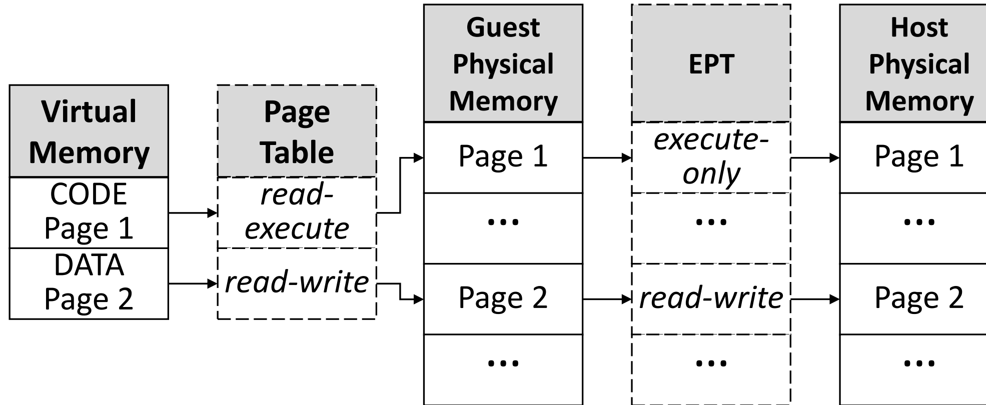

Figure 5.3: Relation between virtual, guest physical, and host physical memory.

Page tables and the EPT contain the access permissions that are enforced during the address translation.

JIT engines cache the generated code to avoid continually recompiling the same methods. Frequently executed methods in the code cache are recompiled and replaced with optimized versions. When the code cache grows beyond a certain size, it is garbage collected by removing the least recently executed methods. Since the code cache is continually updated, JIT compilers typically allocate the code cache on pages with RWX permissions. This means that, unlike statically generated code, there is no easy way to eliminate reads and writes to dynamically generated code without incurring a high performance impact.

We apply the Readactor approach to dynamically generated code in two steps. First, we modify the JIT compilers to separate code and data in their output. Other V8 security extensions [4, 87, 111] require this separation as well to prevent code injection attacks on the code cache, and Ansel et al. [4] also implement it. Second, with code and data separated on different pages, we then identify and modify all operations that require reads and writes of generated code. The following sections discuss these two steps in greater detail. Figure 5.4 shows the permissions of the code cache over time after we modified the V8 engine. Our changes to the V8 JavaScript engine adds a total of 1053 new lines of C++ code across 67 different source files.

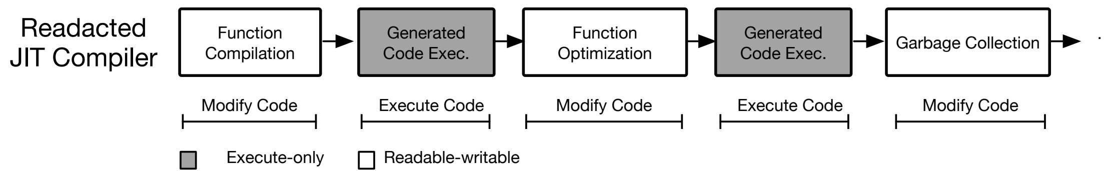

Figure 5.4: Timeline of the execution of a JIT-compiled program.

Execution switches between the JIT compiler and generated code.

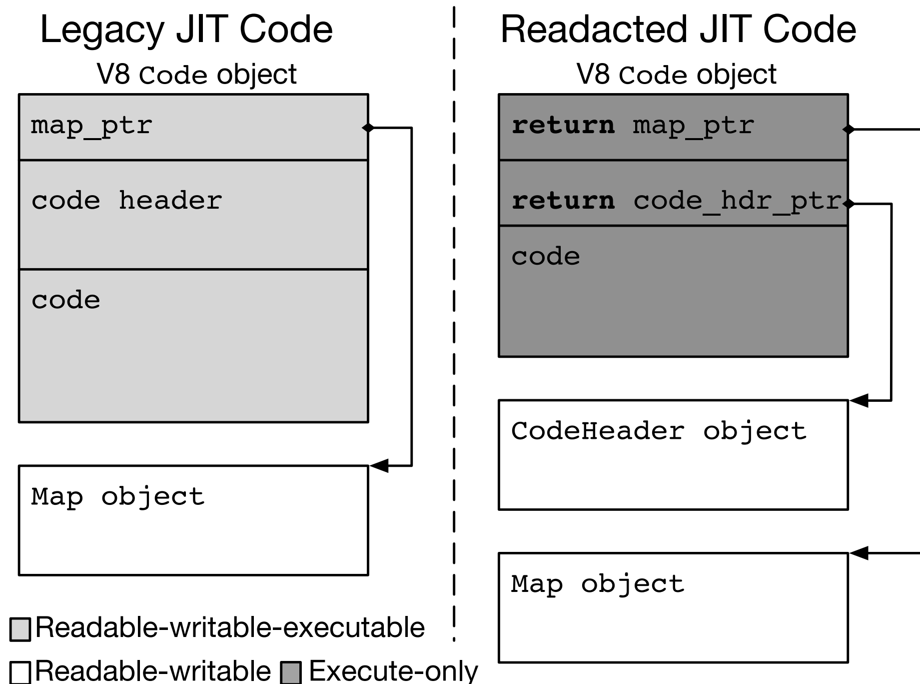

Figure 5.5: Transforming V8 `Code` objects to separate code and data.

5.2.1 Separation of Code and Data

The unmodified V8 JIT compiler translates one JavaScript function at a time and places the results in a code cache backed by pages with RWX permissions. Each translated function is represented by a `Code` object in V8 as shown on the left side of Figure 5.5. Each `Code` object contains the generated sequence of native machine instructions, a code header containing information about the machine code, and a pointer to a `Map` object common to all V8 JavaScript objects.

To store `Code` objects on execute-only memory pages, we move their data contents to separate data pages and make this data accessible by adding new getter functions to the `Code` object. To secure the code header, we move its contents into a separate `CodeHeader` object located on page(s) with RW permissions. We add a getter method to `Code` objects that returns a pointer (`code_hdr_ptr`) to the `CodeHeader` object. Similarly, we replace the map pointer (`map_ptr`) with a getter that returns the map pointer when invoked. These changes eliminate all read and write accesses to `Code` objects (except during object creation and garbage collection) so they can now be stored in execute-only memory; a transformed `Code` object is shown on the right side of Figure 5.5.

5.2.2 Switching Between Execute-Only and RW Permissions

With this separation between code and data inside `Code` objects, we guarantee that no JavaScript code needs to modify the contents of executable pages. However, the JIT compiler still needs to change code frequently. As Figure 5.4 shows, execution alternates between the compiler and JavaScript code, and changes to code can only come from the compiler. We have identified a variety of reasons why V8 would make such changes:

Function optimization occurs when a function is frequently executed and the compiler is able to gather significant execution information about it, V8 compiles a new, optimized version of the function, using the collected information. The unoptimized function is replaced in memory with the new version, and all references to the old function are updated. At some later point, the compiler might have to resume execution of the unoptimized version of the functions if an assumptions made during optimizations did not hold. This step is called *de-optimization*. During optimization and deoptimization, the compiler patches the old version of the function to transfer control to the new one, requiring write access to the former.

Garbage collection is a feature of many programming languages where the execution environment takes care of automatically releasing memory blocks when they are no longer used,

as opposed to requiring the programmer to write code that releases them. For `Code` objects, V8 uses a mark-and-sweep [79] garbage collector. A mark-sweep collector operates in two stages: the marking step iterates through all objects reachable from a known set of roots, marking them as live, while the sweeping step frees all dead objects and relocates the live ones to new code pages. In addition, the V8 collector uses a stop-the-world algorithm, meaning that no JavaScript code can run during collection. This allows us to safely re-map all executable pages as RW before garbage collection, run the collector, then re-map the pages as execute-only.

Code aging is an additional memory optimization that V8 performs allowing it to save some of the memory used by `Code` objects: code aging. Infrequently-run functions that survive several consecutive collection cycles are collected and their entry points replaced with stubs that trigger recompilation. The next time such a function is called, control flow transfers back to the JIT compiler so it may compile the function on-demand. This optimization adds an extra field to the `Code` object header for the code age of a function, and the garbage collection uses this field to keep track of which functions to release.

Inline caches are an optimization frequently used by JIT compilers to speed up slow lookups [40]. Whenever generated code needs to perform a lookup for some property of an object, e.g., its type or the implementation of a function, the JIT compiler optimizes this lookup by caching the parameter and result of the last successful lookup, and reusing the cached value for the next lookup if the parameter matches, avoiding a call to the expensive lookup function. In most implementations of inline caches, both the parameter and result are stored directly in the code (for performance reasons), so on every cache miss the lookup function has to patch the inline cache code with the new values. The lookup functions are all located inside the compiler runtime; we manually modified all of them to re-map the patched code as RW, patch it, then map it back as executable.

Relocation information relocates not only the object itself, but also other objects with addresses embedded inside the current `Code` object. The latter information is modified during garbage collection, as well as function optimization. Some of the relocatable addresses are stored as immediates inside native instructions, so changing them requires modifying the code itself. Whenever an object contains references that must be changes, we re-map it as writable and make the needed changes. In most cases, these changes occur during garbage collection, so all executable objects have already been re-mapped.

Debugging breakpoints are used by the integrated V8 JavaScript debugger and require patching the generated code to stop execution at precise instructions and return control to the JIT compiler.

Completely eliminating code writes from the compiler would require a significant refactoring of V8 (due to extensive use of inline caches, relocations and recompilation, which are hard to completely eliminate), as well as incur a significant performance hit. Instead, we observe that the generated code is, at any point in time, either executed or suspended so that it can be updated. During execution, we map code with execute-only permissions, and when execution is suspended, we temporarily remap it with RW permissions. For both performance and security reasons, we minimize the number of times we re-map pages, as well as the length of time a page stays accessible. Each time a `Code` object becomes writable, it provides a window for the attacker to inject malicious code into that object.

5.3 Evaluation

5.3.1 Performance

We evaluated the performance impact of our changes to the V8 JavaScript engine, as well as the overhead of running the JIT compiler under Readactor. To get a more accurate sample, we benchmarked the V8 engine alone, outside of the browser. As we discussed in Section 5.1, there are two different ways of implementing page permission operations at the hypervisor level. We implemented and evaluated V8 performance for both approaches.

Table 5.1 shows the results of our evaluation when using **Method A** (forwarding changes to the hypervisor, which modifies the EPTs for every change). The performance hit from this approach proved prohibitive, with an average of 97x over all benchmarks and a maximum of 2015x for the DeltaBlue benchmark. Further investigation showed two main sources of the overhead : repeated transitions from the operating system to the hypervisor and back, which required two CPU mode transitions (from non-root mode and back), and TLB flushes, which have a significant performance impact on the entire system. This impact is not as severe for regular page tables, as changing one entry only causes that entry to be flushed. On the other hand, changing an EPT entry causes the entire TLB to be flushed.

After implementing **Method B**, we ran the performance benchmarks again. Figure 5.6 shows the results of this evaluation. We see small to negligible overhead for most benchmarks, with the exception of two benchmarks which put significant pressure on the memory allocator: *EarleyBoyer* and *Splay*. Both benchmarks allocate large numbers of temporary objects and trigger frequent garbage collection cycles, which become much more expensive due to our repeated re-mapping of pages. For another benchmark—*Richards*—we observe a very small speedup of 1%, which we attribute to measurement noise. Overall, the performance penalty of our changes comes to 6.2% when running natively and 7.8% with Readactor enabled.

Benchmark	Vanilla	Readactor	Slowdown
Richards	17669	190.00	93x
DeltaBlue	33449	16.60	2015x
Crypto	18048	283.00	63x
RayTrace	32856	66.90	491x
EarleyBoyer	20776	577.00	36x
RegExp	2397	5.62	426x
Splay	1662	41.70	39x
NavierStokes	16817	7306.00	2.3x
Total Score	12191	125.00	97x

Table 5.1: V8 benchmark performance impact of **Method A**.

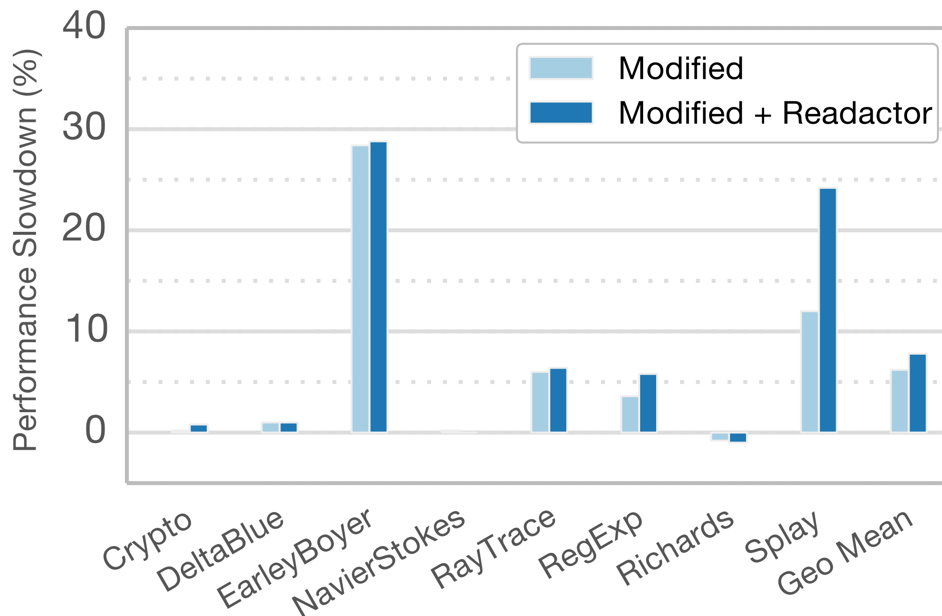

Figure 5.6: Performance of modified V8 running under Readactor.

Performance impact is relative relative to a vanilla build and to a modified build running natively.

5.3.2 Security

Readactor prevents JIT-spraying as well as any attack that attempts to identify useful code in the JIT-compiled code area through direct memory disclosure. We achieve this by marking the JIT code area as execute-only and use V8's built-in coarse-grained randomization (similar to ASLR). Hence, the adversary can neither search for injected shellcode nor find other useful ROP code sequences. On the other hand, given V8's coarse-grained randomization, it is still possible for the adversary to guess the address of the injected shellcode. To tackle guessing, as well as all other attacks, we can either manually implement finer-grained randomization in V8 or use librando (we leave this as future work).

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[2] 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 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.

[2] CVE-2014-1705

Chapter 6

Related Work

6.1 Software Diversity

Software diversity is one strategy to defend against code reuse attacks. Cohen was first to explore program protection using diversity [28]. Forrest at al. [43] later demonstrated stack-layout randomization against stack smashing (an idea most recently revisited by the StackArmor project [25]). DieHard [9] is an implementation of software diversity targeted at memory errors. By randomizing the layout of the application heap, DieHard provides probabilistic protection against attacks like heap buffer overflows and double frees. As memory randomization focuses on a different stage of an attack from code randomization, the two techniques complement each other.

The idea of software diversity was originally explored as a way to obtain fault-tolerance in mission critical software. Approaches to software fault-tolerance are broadly classified as single-version or multi-version techniques. Early examples of the latter kind include Recovery Blocks [98] and N-Version programming [5] that are based on *design diversity*. The conjecture of design diversity is that components designed and implemented differently, e.g., using separate teams, different programming languages and algorithms, have a very low probability of containing similar errors.

When combined with a voting mechanism that selects among the outputs of each component, it is possible to construct a robust system out of faulty components. Since design diversity is only increased at the expense of additional manpower, these techniques are far too costly to see application outside specialized domains such as aerospace and automotive software. The remaining techniques we will discuss aim to provide increased security, and are fully automatic and thus relevant to a greater range of application domains. This means that diversity is introduced later in the software life-cycle by a compiler or binary rewriting system.

The life-cycle of most software follows a similar trajectory: implementation, compilation, linking, installation, loading, executing, and updating. Variations arise because some types of software, typically scripts, are distributed in source form. Some approaches are staged and therefore span multiple life-cycle events; we place these according to the earliest stage.

The most convenient stage at which randomization can be perform is the compilation stage. At that point in the program life-cycle, enough information about the program is available that a wide range of randomizations can be performed, while much of this information is later lost. Jackson et al. [60] discuss the many possible randomizations that a diversifying compiler can perform: NOP insertion, instruction schedule randomization, basic block reordering, randomized register allocation. However, this approach requires that program source code is available, and binaries can be recompiled from this source.

Giuffrida et al. [50] evaluate a comprehensive, compiler-based software diversifier that allows live re-randomization of an operating system micro-kernel. The resulting binaries contain pre-linked LLVM bitcode as well as native machine code. During re-randomization, the host system periodically compiles a new program variant from the bitcode. The newly created variant includes meta-data to migrate the program state from the old to new program variant. Finally, a re-randomization daemon uses the meta-data to transfer the state from the old process to its newly spawned counterpart. Frequent re-randomization may render any knowledge gleaned through memory disclosure useless. However, the entire JIT-ROP attack can run in as little as 2.3 seconds while re-

randomization at 2 second intervals add an overhead of about 20%. Moreover, it remains unknown how well this approach scales to complex applications containing JIT compilers and modern operating systems with monolithic kernels.

In many cases, programs are available only in binary form (the source code may have been lost or simply never received with the binary), so other software diversity implementations randomize the binary itself directly. Address space layout permutation (ASLP) [65] is one such implementation. ASLP randomizes the code layout at function granularity; adversaries must therefore disclose more than one code pointer to bypass ASLP. They locate all program functions using a binary analysis tool, then randomly shuffle program functions inside the on-disk representation of the program.

In-place diversification is an install-time-only approach that sidesteps the problem of undiscovered control flow [90]. Code sequences reachable from the program entry point are rewritten with other sequences of equal length. Unreachable bytes are left unchanged thus ensuring that the topology of the rewritten binary matches that of its original. In-place rewriting preserves the addresses of every direct and indirect branch target and thereby avoids the need for and cost of runtime checks and dynamic adjustment of branch targets. The approach does have two downsides, however: (i) undiscovered code is not rewritten and thus remains available to attackers and (ii) preserving the topology means that return-into-`libc` attacks are not thwarted.

Load-time diversification approaches do not change the on-disk representation of programs. Rather, they perform randomization as these are loaded into memory by the operating system. Several approaches defer diversification by making programs *self-randomizing* [10, 11, 54, 117, 50]. Deferred diversification is typically achieved by instrumenting programs to mutate one or more implementation aspects as the program is loaded by the operating system or as it runs.

Consequently, with deferred diversification, all instances of a program share the same on-disk representation—only the in-memory representations vary. This has several important implications. Deferred approaches remain compatible with current software distribution practices; the program

delivered to end users by simply copying the program or program installer. When diversification is deferred, the program vendor avoids the cost of diversifying each program copy. Instead the cost is distributed evenly among end users. The end user systems, however, must be sufficiently powerful to run the diversification engine. While this is not an issue for traditional desktop and laptop systems, the situation is less clear in the mobile and embedded spaces. Second, when diversification is deferred, an attacker does not improve the odds of a successful attack with knowledge of the on-disk program representation.

However, deferred diversification cannot provide protection from certain attacks. Client-side tampering [30] and patch reverse-engineering [32] remain possible since end users can inspect the program binaries before diversification. Software diversification can also be used for watermarking [41]. If a seed value drives the diversification process and a unique seed is used to produce each program variant, the implementation of each variant is unique, too. If each customer is given a unique program variant, and the seed is linked to the purchase, unauthorized copying of the program binary can be traced back to the original purchase. However, such use of diversity is also hampered by deferred diversification.

Instruction location randomization (ILR) rewrites binaries to use a new program encoding [54]. ILR changes the assumption that, absent any branches, instructions that are laid out sequentially are executed in sequence; instructions are instead relocated to random addresses by disassembling and rewriting programs as they are installed on a host system. A data structure, the *fallthrough* map, contains a set of rewrite rules that map unrandomized instruction locations to randomized ones and to map each randomized instruction location to its successor location. To avoid the need to separate code and data, rewrite rules are generated for all addresses in a program's code section. A process virtual machine—Strata [105]—executes the rewritten programs. At runtime, Strata uses the fallthrough map to guide instruction fetch and reassemble code fragments before execution; fragments are cached to reduce the translation overhead.

Binary stirring [117] is also a hybrid approach that disassembles and rewrites binaries as they are installed such that they randomize their own code layout at load time. Rather than using a process virtual machine, randomization is done via a runtime randomizer that is unloaded from the process right before the main application runs. This ensures that the code layout varies from one run to another. Instead of trying to separate data from code, the text segment of the original binary is treated as data and code simultaneously by duplicating it into two memory regions—one which is executable and immutable and one which is non-executable but mutable. One duplicate is left unmodified at its original location and marked non-executable. The other duplicate is randomized and marked executable. Reads of the text segment go to the unmodified duplicate whereas only code in the randomized duplicate is ever executed. All possible indirect control flow targets in the unmodified duplicate are marked via a special byte. A check before each indirect branch in the randomized duplicate checks if the target address is in the unmodified duplicate and redirects execution to the corresponding instruction in the randomized duplicate.

Several load-time diversification approaches also have runtime components. Barrantes et al. [8], for instance, randomizes the instruction set encoding as code is loaded and uses the Valgrind dynamic binary rewriter [85] to decode the original machine instructions as they are about to execute. Williams et al. [119] similarly implement instruction set randomization atop the Strata process virtual machine. Their approach even has a compile-time component to prepare binaries by adding an extra "hidden" parameter to each function. This parameter acts as a per-function key whose expected value is randomly chosen at load-time and checked on each function invocation. The process virtual machine instruments each function to verify that the correct key was supplied; it also randomizes the instruction set encoding to prevent code injection. Without knowledge of the random key value, function-level code-reuse (e.g., return-into-`libc`) attacks are defeated. Finally, Shioji et al. [108] implement address-randomization atop the Pin [77] dynamic binary rewriter. A checksum is added to certain bits in each address used in control flow transfers and the checksum is checked prior to each such transfer. Attacker injected addresses can be detected due to invalid

checksums. Checksums are computed by adding a random value to a subset of the bits in the original address and hashing it.

In contrast to these hybrid approaches, Davi et al. [37] implement a pure load-time diversification approach that randomizes all segments of program binaries. The code is disassembled and split into code fragments at call and branch instructions. The resulting code fragments are used to permute the in-memory layout of the program. The authors assume that binaries contain relocation information to facilitate disassembly and consequently omit a mechanism to compensate for disassembly errors.

Unfortunately, these defenses remain vulnerable to memory disclosure attacks. The appearance of JIT-ROP attacks convincingly demonstrated that probabilistic defenses cannot tacitly assume that attackers cannot leak code memory layout at runtime [110]. Blind ROP [12], another way to bypass fine-grained code randomization, further underscores the threat of memory disclosure.

In response to JIT-ROP, Backes and Nürnberger proposed Oxymoron [7] which randomizes the code layout at a granularity of 4KB memory pages. This preserves the ability to share code pages between multiple protected applications running 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 hidden table. This prevents direct memory disclosure, i.e., it prevents the recursive-disassembly step in the original JIT-ROP attack. However, Oxymoron can be bypassed via indirect memory disclosure attacks.

Davi et al. [36] also propose 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. Since Isomeron is implemented as dynamic binary instrumentation framework its runtime and memory overheads are substantially greater than compiler-based solutions.

Instead of implementing diversification at the compiler level or in the compiled program itself, the diversification functionality can be included in the operating system [92, 27]. This is exactly how ASLR[1] is implemented. A compiler prepares the code for base address randomization by generating position-independent code; the operating system loader and dynamic linker then adjust the virtual memory addresses at which the code is loaded.

6.2 Integrity-based Defenses

Integrity-based defenses are the primary alternative to software diversity for protection against code reuse attacks. Code reuse attacks (as well as some code injection attacks) redirect the processor program counter, at some point during execution, to an unintended value. In other cases, they also redirect memory reads or writes to unintended addresses. The main idea behind integrity-based defenses is to restrict all such operations (memory accesses and control flow) to intended values. At present, control-flow integrity (CFI) [1, 2] is the most prominent type of integrity-based defense. CFI constrains the indirect branches in a binary such that they can only reach a statically identified set of targets. Since CFI does not rely on randomization, it cannot be bypassed using memory disclosure attacks. However, it turns out that precise enforcement of control-flow prop-

[1] ASLR is an example of a diversification technique that required compiler customization to produce position independent code. All major C/C++ compilers currently support this security feature.

erties invariably comes at the price of high performance overheads on commodity hardware. In addition, it is challenging (if not impossible) to always resolve valid branch addresses for indirect jumps and calls.

G-Free [88], control-flow locking [14], control-flow integrity [1] and software fault isolation (SFI) [121, 78] take a combined compile- and run-time approach to security. The compiler adds code to the program that restricts the control flow to the program's original control flow edges, by instrumenting all branch instructions. While these techniques have proven effective at defending against ROP, any inherent weakness they have can be exploited by attackers. Also, some are specifically targeted at return-oriented programming, so are unable to defend against newer code reuse attacks. This incurs a maintenance cost on whoever implements these techniques, as each implementation must be updated for any new code reuse attack, if at all possible.

As a result, researchers have traded off security for performance by relaxing the precision of the integrity checks. CFI for COTS binaries [123] 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 merely ensures that branch targets are either call-preceded or target an address-taken basic block. Similar policies are enforced by Microsoft's security tool called EMET [80], which builds upon ROPGuard [45].

Compact control-flow integrity and randomization (CCFIR) 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. Our code-pointer hiding technique has similarities with the use of trampolines in CCFIR but the purposes differ. The springboard is part of the CFI enforcement mechanism whereas our trampolines are used to prevent indirect memory closure. Although the layout of springboard entries is randomized to prevent traditional ROP attacks, CCFIR does not include countermeasures against JIT-ROP.

Since CFI does not randomize the code layout, attackers can inspect the code layout ahead of time and carefully choose gadgets that adhere to a coarse-grained CFI policy [51, 52, 20].

SafeDispatch [61] and forward-edge CFI [71] are two compiler-based implementations of fine-grained forward CFI, which is CFI applied to indirect calls. 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. It offers low runtime overhead (2.1%) but only protects virtual method calls. Forward CFI 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 halt 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 do not protect against attacks that use `ret`-terminated gadgets. Moreover, as no code randomization is applied, the adversary can easily invoke critical functions in complex programs that are also legitimately used by the program.

A number of recent CFI approaches focus on analyzing and protecting vtables (used to implement virtual function calls in C++) in binaries created from C++ code [48, 96, 122]. Although these approaches do not require source code access, the CFI policy is not as fine-grained as their compiler-based counterparts. A novel attack technique for C++ applications, *counterfeit object-oriented programming* (COOP), 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 [103].

Code-Pointer Integrity (CPI) was first proposed by Cowan et al. [33], then later revisited as an alternative to CFI by Szekeres et al. [112] and implemented by Kuznetsov et al. [71]. 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, the same is not possible in 64-bit mode so the safe region is merely hidden from attackers whenever segmentation is not fully supported. Evans et al. [42] demonstrated that (in the 64-bit mode of one of five existing implementations that relied on code hiding) the size of the safe region is 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. [109]. Unlike our approach, it remains to be seen whether CPI can be extended to protect dynamically generated code without degrading the JIT compilation latency and performance.

Opaque CFI (O-CFI) [81] is designed to *tolerate* certain kinds of memory disclosure. O-CFI combines code randomization and integrity checks. It tolerates code layout disclosure by bounding the target of each indirect control flow transfer. Since the code layout is randomized at load time, the bounds for each indirect jump are randomized too. The bounds are stored in a small table which is protected from disclosure using x86 segmentation. O-CFI uses binary rewriting and stores two copies of the program code in memory to detect disassembly errors. Apart from the fact that O-CFI requires precise binary static analysis as it aims to statically resolve return addresses, indirect jumps, and calls, the adversary may be able to disassemble the code, and reconstruct (parts of) the control-flow graph at runtime. Hence, the adversary could dynamically disclose how the control-flow is bounded.

6.3 Code Hiding

Memory disclosure attacks, e.g., JIT-ROP [110], significantly raised the bar for software diversity defenses. Whereas a defense solution could previously randomize program code once and assume full security, this is no longer the case. So far, two main alternatives have been proposed to solve this: live re-randomization [50] and code hiding. The latter works by hiding the diversified code from attackers, preventing them from reading it and un-doing the randomization.

One defense against JIT-ROP, Execute-no-Read (XnR) by Backes et al. [6], is conceptually similar to Readactor as it is also based on execute-only pages. However, it only 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 pages in execution are readable. Hence, the adversary can disclose function addresses, 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. It remains unclear how well XnR can protect against indirect memory disclosure, as it only assumes a code randomization scheme without implementing and evaluating one.

HideM by Gionta et al. [49] 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. HideM allows instruction fetches of code but prevents data accesses except whitelisted reads of embedded constants. This is the same technique PaX [92] used to implement W$\oplus$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.

Until Readactor, no code hiding solution explicitly addresses the problem of JIT code cache hiding. This dissertation is, to the best of our knowledge, the first to discuss the challenges encountered when implementing code hiding for a JIT compiler (V8 in this case).

6.4 Other Defenses

Other strategies rely on detection of code reuse attacks, or prevention or using static non-diversifying compile-time techniques. Detection software runs alongside the program and attempts to detect whenever a ROP or other code reuse attack is in progress, by looking for high numbers of return instructions (or other branches) in a small amount of time. DROP [24] dynamically instruments a running program to analyze the frequency of taken returns; whenever the number of returns taken in a short amount of time exceeds a threshold, DROP considers a ROP attack is in progress and takes measures against it. More recent support takes advantage of hardware support for this purpose. In particular, x86 processors contain a last branch record (LBR) register which kBouncer [91] and ROPecker [26] use to inspect a small window of recently executed indirect branches. However, such approaches have been defeated shortly after being introduced [20, 52].

Another approach to preventing ROP attacks is to remove all return instructions (encoded by the `C3` byte in x86 programs), or more generally remove all branch instructions. "Return-less kernels" [74] target ROP attacks specifically during compilation, by reordering program instructions and re-allocating registers so that free branch instructions never appear inside a binary. This defenses successfully defeats the original ROP, but does not work against more generic and recent code reuse attacks. G-Free [88] is a hybrid defense that works similarly: it uses CFI on intended (generated intentionally by the compiler) returns and removes all unintended (accidentally occurring due to instruction encoding) the same way as "return-less kernels".

6.5 JIT Protection

One way of improving JIT compiler security is manually adding security checks and diversity by modifying the compiler. JITDefender [23] hinders JIT spraying attacks by marking all HLL code pages as non-executable; the compiler changes these pages to executable on entry to the generated code, and changes them back on return to the compiler or runtime. This effectively prevents an attacker from redirecting any other branch (other than the intended ones) to dynamically generated code. INSeRT [118] adds code randomization (concretely, it randomizes the operand of every emitted instruction) through a white box approach, requiring manual changes to the native code emitter inside the compiler. Both approaches successfully hinder most JIT spraying and code reuse attacks against HLL code, but require changes to the compiler; they are not feasible when source code is not available, or the modified code cannot be deployed. On the other hand, librando can harden a compiler without requiring any cooperation from the compiler itself.

A similar line of research investigates JIT code generation from inside a sandbox (where control flow is restricted, and code must follow certain restrictions). Native Client [121] is a sandboxed execution environment which adds static checks to native code that enforces restrictions on branch targets (such as 16-byte alignment for all targets). The original implementation of the system did not allow dynamic code generation at all, but later work [4] added this feature. The JIT extensions adds an API to the sandbox that a JIT compiler uses to generate new code; before executing that code, the sandbox verifies that the new code respects all restrictions imposed by Native Client, and therefore cannot break out of the sandbox. Another of their contributions was an in-depth analysis on the different types of NOPs to insert, which they use to enforce basic block alignment; here, we use NOPs for randomization. This is, in some ways, similar to our white box diversification approach: librando can be viewed as the sandbox that the JIT compiler runs inside.

Recently, a new CFI solution has been proposed by Niu and Tan that also applies CFI to JIT-compiled code [87]. Their RockJIT framework enforces fine-grained CFI policies for static code

and coarse-grained CFI policies for JIT-compiled code. Similarly to how we double-map physical host pages in the guest physical space, 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.

In addition to preventing memory disclosure on the V8 code cache, our extensions to Readactor also aim to prevent code injection attacks on the code cache. Ideally, we would achieve this by making the code cache immutable or append-only. However, due to reasons presented in Chapter 5, we are unable to complete remove writes to the code cache from the JIT compiler. These writes create a small window of time when the attacker can inject arbitrary code into the code cache. Song et al. [111] demonstrate that an attack during this window is feasible. They propose a defense based on process separation, where the JIT compiler is located in a separate process from the untrusted browser, and only the JIT process can write to the generated code, and implement this defense in two dynamic code generators—the V8 JavaScript engine used by the Chrome browser, and the Strata virtual machine. LOBOTOMY [62] is another implementation of this approach for the Firefox browser. This successfully protects against code injection attacks against the code cache, but not against disclosure of the generated code. In the untrusted process, the generated code is mapped as read-only and executable, but could instead be mapped as execute-only for use with Readactor. We believe this approach is fully compatible with and complementary to ours, and can be used to protect the JIT from code injection.

6.6 Profile-Guided Optimization

Most compilers use profiling information to optimize frequently used sections of code. This has the goal and effect of increasing program performance and, in some cases, reducing program size. However, there are other applications of profiling where the interest lies in infrequently executed code. One such application is code compression. Debray and Evans [39] analyze the use of profiling in tandem with binary code compression, using basic block execution frequency to decide whether to compress particular regions of code. They showed that focusing compression on cold code produces significant reductions in program size.

Another use case for identifying cold code is intentional code duplication. Recent research uses this technique to identify computational errors due to transient faults in processors [64]. The work by Khudia et al. duplicates values and instructions in a program, to account for transient processor errors. The duplicates perform the same computations as the originals, so their results should match. At specific points in the program, the duplicates are checked against the originals to detect errors. The compiler uses edge profiling to reduce number of duplicated instructions, minimizing the performance impact of this approach. When only duplicating cold instructions, they show a reduction of 41% in overhead from previous techniques.

Profiling information is most often collected by running a special version of the profiled program on carefully selected training input that is representative of most inputs that the program sees in practice. This special version contains instrumentation code that collects the execution profile and saves it when the run finishes. The program is then re-compiled using the collected profile. However, this double-compilation model is difficult to use in practice, and its effectiveness heavily depends on the choice of the training input. An alternative option is static profiling, which estimates the hot/cold regions of the program from its structure, i.e., it assumes that deep loops are hot and the rest of the code is cold. Murphy et al. [82] investigate the use of static profiling to optimize

NOP insertion. Their results show that static profiling is close in effectiveness to dynamic profiling when an ideal training input is available, and significantly better for a bad training input.

Most integrity-based defenses incur a larger overhead than our profile-guided NOP insertion technique. However, many of them rely on inserting expensive code in specific places. Therefore, they can also benefit from profile-guided optimization, by inserting checks and guards selectively to minimize the performance hit. ASAP [116] uses run-time profiling to reduce the performance overhead of various security-focused instrumentations. Contrary to our NOP insertion algorithm, where we profile a vanilla program then use the profile to insert the extra instructions, they first insert the new instrumentation instructions, then use profiling to determine the most expensive instructions. ASAP users specify an overhead budget, and all instrumentation that causes the performance hit to exceed the budget is removed. They report a significant overhead reduction (down to 5% or lower in many cases), while still providing most of the security of vanilla instrumentation.

6.7 Binary Rewriting

There is significant research into dynamic rewriting of program code at run-time. DynamoRIO [17], PIN [76], Valgrind [84], and Strata [105] all intercept and rewrite program code at run-time, for purposes such as instrumentation, profiling, identifying program errors and increasing security The transparency restrictions in their designs are very similar to the restrictions we described for librando. However, one difference that sets librando apart from other rewriters is the limited scope of code rewriting: librando only intercepts and rewrites dynamically generated code, whereas all other rewriters handle all of the application code. For static program code, we envision that a compiler-level diversification solution is used on statically generated code; librando is only meant to diversify code which ahead-of-time diversification solution cannot protect. Our approach allows the JIT compiler and language runtime to run natively, only intercepting dynamically generated code; all other solutions would intercept and rewrite everything, starting with the first instruction

in the JIT compiler. Another difference is that dynamic rewriters are designed as frameworks that allow arbitrary changes to the intercepted code, which increases their complexity significantly, whereas the magnitude of our changes to the code is very small and security-focused (adding NOPs and rewriting instruction operands). For example, Valgrind disassembles executed code and converts it to an intermediate representation (IR), which all code transformations operate on. After all transformations are done, Valgrind converts this IR back to x86 code; this two-way translation is not needed by librando.

Existing dynamic binary rewriters have been used for security. Program Shepherding [66] (implemented using DynamoRIO) and libdetox [94] add security checks before branches, function calls and system calls by intercepting and rewriting program blocks. These checks only allow branches to allowed code addresses, determined algorithmically or from a given list. This effectively defends against code reuse attacks (and more), albeit using a different approach from diversification. Our proposed solution has similar goals, but different methods: we randomize code layout, restricting whatever knowledge the attacker possesses of program code. While deterministic techniques are successful at hardening applications, they are also vulnerable in one regard: if an attacker finds a flaw in such a technique, they are able to attack all hardened targets using this flaw. Randomization, on the other hand, does not assume safety of the defense itself; instead, the goal is to restrict any successful attack to only a small subset of possible targets.

Most current dynamic binary rewriters operate under the assumption that code, once emitted, changes very rarely or not at all. This assumption is not true for dynamically generated code. While rewriters such as DynamoRIO and PIN still support rewriting code emitted by JIT compilers, they do so at significant overhead (as high as 15x on some benchmarks, significantly higher than librando). Hawkins et al. [53] add support for JIT compilers to DynamoRIO and reduce the JIT rewriting overhead to 2-2.5x (bringing DynamoRIO JIT performance close to librando), using a combination of manual JIT instrumentation, static analysis and dynamic detection of code modifications. For the latter, they use a double page mapping similar to ours (which they call

a *Parallel Mapping*), where a physical page is mapped once as readable+writable, and once as readable+executable.

Chapter 7

Conclusions and Future Work

Software diversity, in the form of code layout randomization, has a significant impact on the outcome of code reuse attacks. Attacks against a sufficiently diverse program have a high chance of failure. NOP insertion in particular is an effective form of code layout randomization. However, a naive implementation suffers from a moderate performance overhead. Selective profile-guided insertion of NOPs reduces this overhead by as much as $5\times$. NOP insertion is particularly efficient against return-oriented programming attacks, reducing the number of available gadgets by a factor as high as $2000\times$. The remaining gadgets are, in practice, not sufficient for an attack. This holds true even when using profile-guided NOP insertion. Our evaluation confirmed the positive performance impact of profiling on software diversity, and we are confident that profiling can be successfully applied to other diversifying transformations.

Traditional code injection attacks are becoming increasingly hard due to the widespread deployment of defenses—DEP, W$\oplus$X, NX and XN—against this class of attacks. This prompted the evolution of attacks targeted at JITs: JIT spraying and code reuse attacks. librando is a binary rewriting library that hardens JIT compilers, and generally any software that generates new code at run-time, against these attacks. Our library supports randomization of code from a JIT compiler

without requiring any source code changes to the compiler, giving defenders a quick and comprehensive response. This approach is portable to any existing or new compiler, providing increased security while saving development effort; instead of having to redo the effort of implementing security measures on every JIT, developers may opt to use our library to secure their compiler. The library can also be used as an interim measure, providing security at a temporary performance penalty until security measures are implemented more efficiently in the compiler itself. In cases where compiler source code is not available, or where recompilation and reinstallation are not feasible, this penalty is preferable to the loss in security. If compiler source code is available, compiler developers can improve diversification performance through white box diversification, by adding calls from the compiler to librando. We also presented an optimization with substantial impact on performance: the *Return Address Map*. We successfully tested our work on two industry-strength JIT compilers, both widely used at present. Our evaluation showed that the impact of diversification depends greatly on the workload; librando provides great security benefits at low performance cost (around 15%) for statically typed languages (where it can be enabled at all times), and at a larger cost (a 3-4× slowdown) for dynamically typed languages.

Most implementations of software diversity operate under the assumption of "memory secrecy" and take no steps to prevent code disclosure. For this reason, code disclosure poses a significant threat to such implementations. In the case of dynamic code generators, e.g., JIT compilers, leaking the contents of a code cache can be equally dangerous, but more difficult to secure. We designed and implemented a code hiding technique (execute-only code pages) for JIT code caches on top of an existing execute-only hypervisor—Readactor. Additionally, we modified an existing industry-strength JIT compiler—V8—to use an execute-only code cache. Our evaluation showed a performance hit of about 7.8% from our changes and that our approach successfully thwarts code cache disclosure attacks.

In this dissertation, we have presented several complementary techniques that improve the security, coverage, and practicality of code randomization defenses. We envision that a comprehensive

defense based on software diversity would deploy all our proposed defenses in tandem, mitigating a large variety of attacks.

7.1 Diversifying Transformations

The main code randomization technique in this dissertation is NOP insertion. librando also implements a second randomization specific to JIT compilers: constant blinding. Based on our experience implementing the current randomizations, we believe new ones could be added without much effort to the ahead-of-time compiler, librando and Readactor. Examples of possible randomizations include: code cache address randomization, function reordering, basic block reordering, equivalent instruction substitution, instruction reordering, register re-allocation and many more (related work [50, 60, 90, 117, 59, 72] also discusses and implements some of these for statically generated code diversification).

With the increased use of whole-function reuse attacks (return-into-lib(c), COOP), randomized register re-allocation may prove to be the most important of all transformations, as it is one of the few that can disrupt these attacks. The attacker chains together several functions, calling them with counterfeit parameters. These parameters are often passed in registers, and so are the function return values. By randomizing these registers, this transformation disrupts the function chains used in the attacks, as each function's registers no longer match up with the previous ones. Crane et al. [34] implement this randomization (in tandem with stack randomization) for Readactor and show that it successfully disrupts whole-function code reuse.

7.2 Future Work

The main obstacle preventing librando from supporting other JITs has been multithreading. This also raised some problems for our evaluation of HotSpot, as several benchmarks we attempted to run required multiple threads. Presenting different threads [19, 4] and processes [18] with a consistent view of the code cache is a known and well studied problem, yet it would require a significant effort to implement in librando.

We have presented librando and the Readactor JIT support as two separate implementations with different goals. However, as an alternative to manually modifying every JIT compiler to add code hiding, we could modify librando to hide its code cache after diversifying it, while allowing the attacker to read the original JIT code cache. This would be sufficient to provide the security benefits of Readactor to any JIT compiler without making any changes, but with the downside of the significant performance impact of librando. We leave this integration of the two techniques as future work.

As an extra step in our Readactor security evaluation, we also tested whether indirect disclosure of JIT-compiled code is feasible. Our experiments revealed that V8's JIT code cache contains several code pointers referencing JIT-compiled code. Hence, the adversary could exploit these pointers to infer the code layout of the JIT memory area. To protect against such attacks, these code pointers need to be indirected through execute-only trampolines (our standard jump trampolines). We can store these trampolines in a separate execute-only area away from the actual code. To add support for code-pointer hiding, we would need to modify both the code entry points emitted by JavaScript runtime and all JavaScript-to-JavaScript function calls to call trampolines instead. This would require a significant engineering effort, which we leave as future work.

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. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13:4:1–4:40, 2009.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), 1996. `http://www.phrack.org/issues.html?id=14&issue=49`.

[4] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 355–366, 2011.

[5] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the International Computer Software and Applications Conference*, COMPSAC '77, pages 149–155, 1977.

[6] 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.

[7] M. Backes and S. Nürnberger. Oxymoron - making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the 23rd USENIX Security Symposium*, SEC '14, 2014.

[8] E. Barrantes, D. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, 2005.

[9] E. Berger and B. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, 2006.

[10] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, SEC '03, pages 105–120, 2003.

[11] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, SEC '05, pages 271–286, 2005.

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

[13] D. Blazakis. Interpreter exploitation. In *Proceedings of the 4th USENIX Workshop on Offensive technologies*, WOOT '10, 2010.

[14] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, 2011.

[15] T. Bletsch, X. Jiang, V. 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.

[16] E. Bosman and H. Bos. Framing signals—a return to portable shellcode. In *Proceedings of the 23rd USENIX Security Symposium*, SEC '14, 2014.

[17] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 2003 International Symposium on Code Generation and Optimisation*, CGO '03, 2003.

[18] D. Bruening and V. Kiriansky. Process-shared and persistent code caches. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, 2008.

[19] D. Bruening, V. Kiriansky, T. Garnett, and S. Banerji. Thread-shared software code caches. In *Proceedings of the 4th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '06, 2006.

[20] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, SEC '14, 2014.

[21] S. Checkoway, L. V. 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*, CCS '10, pages 559–72, 2010.

[22] 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 2009 Conference on Electronic Voting Technology/-Workshop on Trustworthy Elections*, EVT/WOTE '09, 2009.

[23] P. Chen, Y. Fang, B. Mao, and L. Xie. JITDefender: A defense against JIT spraying attacks. In *Proceedings of the 26th IFIP TC 11 International Information Security Conference*, SEC '11, pages 142–153, 2011.

[24] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In A. Prakash and I. Sen Gupta, editors, *Information Systems Security*, volume 5905 of *Lecture Notes in Computer Science*, pages 163–177. Springer Berlin / Heidelberg, 2009.

[25] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Proceedings of the 2015 Network And Distributed System Security Symposium*, NDSS '15, 2015.

[26] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 2014 Network And Distributed System Security Symposium*, NDSS '14, 2014.

[27] 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.

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

[29] 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.

[30] 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.

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

[32] B. Coppens, B. De Sutter, and J. Maebe. Feedback-driven binary code diversification. *Transactions on Architecture and Code Optimization*, 9(4), Jan. 2013.

[33] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, volume 2 of *DISCEX '00*, pages 119–129, 2000.

[34] 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.

[35] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, SEC '14, 2014.

[36] 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 2015 Network And Distributed System Security Symposium*, NDSS '15, 2015.

[37] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. 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, pages 299–310, 2013.

[38] L. V. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-oriented programming without returns on ARM. Technical report, System Security Lab, Ruhr University Bochum, Germany, 2010.

[39] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 95–105. ACM, 2002.

[40] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '84, 1984.

[41] R. El-Khalil and A. D. Keromytis. Hydan: Hiding information in program binaries. In *Proceedings of the 6th International Conference on Information and Communications Security*, ICICS '04, pages 187–199, 2004.

[42] 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.

[43] 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.

[44] 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.

[45] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. `http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf`, 2012.

[46] Free Software Foundation, Inc. GCC compiler internals, 2012.

[47] B. Fulgham. The computer language benchmarks game. `http://shootout.alioth.debian.org/`, 2012.

[48] 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.

[49] 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.

[50] 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.

[51] E. Göktas, 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.

[52] 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 *Proceedings of the 23rd USENIX Security Symposium*, SEC '14, 2014.

[53] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao. Optimizing binary translation for dynamically generated code. In *Proceedings of the 2015 International Symposium on Code Generation and Optimization*, CGO '15, 2015.

[54] 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.

[55] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization*, CGO '13, 2013.

[56] 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.

[57] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao – an extensible micro-architectural optimizer. In *Proceedings of the 9th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 1–10, 2011.

[58] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, August 2012.

[59] 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.

[60] 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*, volume 54 of *Advances in Information Security*, pages 77–98. Springer New York, 2011.

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

[62] M. Jauernig, M. Neugschwandtner, C. Platzer, and P. Comparetti. Lobotomy: An architecture for JIT spraying mitigation. In *Proceedings of the 9th International Conference on Availability, Reliability and Security*, ARES '14, 2014.

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

[64] D. S. Khudia, G. Wright, and S. Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 99–108, 2012.

[65] 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, pages 339–348, 2006.

[66] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, 2002.

[67] D. E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[68] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT Numerical Mathematics*, 13:313–322, 1973.

[69] T. Kornau. Return-oriented programming for the ARM architecture. Master's thesis, Ruhr University Bochum, Germany, 2009.

[70] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation techniques, 2005. `http://www.suse.de/~krahmer/no-nx.pdf`.

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

[72] 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.

[73] 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.

[74] 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, 2010.

[75] W. Lian, H. Shacham, and S. Savage. Too LeJIT to quit: Extending JIT spraying to ARM. In *Proceedings of the 2015 Network And Distributed System Security Symposium*, NDSS '15, 2015.

[76] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[77] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[78] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, pages 209–224, 2006.

[79] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, Apr. 1960.

[80] Microsoft. Enhanced Mitigation Experience Toolkit. `https://www.microsoft.com/emet`.

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

[82] M. Murphy, P. Larsen, S. Brunthaler, and M. Franz. Software profiling options and their effects on security based diversification. In *Proceedings of the 1st ACM Workshop on Moving Target Defense*, MTD '14, pages 87–96, 2014.

[83] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11(58), 2001. `http://www.phrack.org/issues.html?issue=58&id=4`.

[84] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 2003.

[85] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, 2007.

[86] A. Neustifter. Efficient profiling in the LLVM compiler infrastructure. Master's thesis, Faculty of Informatics, Vienna University of Technology, 2011.

[87] 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.

[88] 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.

[89] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[90] 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.

[91] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, SEC' 13, pages 447–462, 2013.

[92] PaX. *Homepage of The PaX Team*, 2009. `http://pax.grsecurity.net`.

[93] M. Payer. Too much PIE is bad for performance. Technical report, ETH Zürich, 2012.

[94] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 157–168, 2011.

[95] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, 1990.

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

[97] R. Pucella and F. B. Schneider. Independence from obfuscation: A semantic framework for diversity. *Journal of Computer Security*, 18(5):701–749, 2010.

[98] B. Randell. System structure for software fault tolerance. *SIGPLAN Not.*, 10(6):437–449, 1975.

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

[100] G. F. 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, 2009.

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

[102] J. Salwan. ROPgadget tool, 2012.

[103] 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.

[104] 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.

[105] K. Scott, N. Kumar, S. Velusamy, B. R. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the 2003 International Symposium on Code Generation and Optimisation*, CGO '03, 2003.

[106] 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.

[107] 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.

[108] 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.

[109] J. Siebert, 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.

[110] K. Z. Snow, F. Monrose, L. V. Davi, A. Dmitrienko, C. Liebchen, and A.-R. 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, pages 574–588, 2013.

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

[112] 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, 2013.

[113] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: mitigating contention for QoS in warehouse scale computers. In *Proceedings of the 10th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '12, pages 1–12, 2012.

[114] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. W. 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, pages 121–141, 2011.

[115] VMware, Inc. VMware ESX.
http://www.vmware.com/products/esxi-and-esx/overview.

[116] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. High system-code security with low overhead. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, 2015.

[117] 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.

[118] T. Wei, T. Wang, L. Duan, and J. Luo. INSeRT: Protect dynamic code generation against spraying. In *Proceedings of the 2011 International Conference on Information Science and Technology*, ICIST '11, pages 323–328, 2011.

[119] D. W. Williams, W. Hu, J. W. Davidson, J. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security & Privacy*, 7(1):26–33, 2009.

[120] Xen Project. Xen.
http://www.xenproject.org.

[121] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, S&P '09, pages 79–93, 2009.

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

[123] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, SEC' 13, pages 337–352, 2013.

[124] D. D. Zovi. iOS jailbreak analysis. Technical report, CSAW:THREADS, 2012. http://www.trailofbits.com/resources/ios_jailbreak_analysis_slides.pdf.