CODE-REUSE ATTACKS AND DEFENSES

Vom Fachbereich Informatik (FB 20)
an der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines Doktor-Ingenieurs
genehmigte Dissertation von:

MSc. Lucas Vincenzo Davi
Geboren am 27. April 1984 in Duisburg, Deutschland

Referenten:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (Erstreferent)
Prof. Hovav Shacham (Zweitreferent)

Tag der Einreichung: 22. April 2015
Tag der Disputation: 3. Juni 2015

TECHNISCHE
UNIVERSITÄT
DARMSTADT

CASED/System Security Lab
Intel Collaborative Research Center for Secure Computing
Fachbereich für Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

PHD REFEREES:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (1st PhD Referee)
Prof. Hovav Shacham (2nd PhD Referee)

FURTHER PHD COMMISSION MEMBERS:
Prof. Dr. Dr. h.c. Johannes A. Buchmann
Prof. Dr. Stefan Katzenbeisser
Prof. Dr.-Ing. Matthias Hollick

Darmstadt, Germany April 2015

ABSTRACT

Exploitation of memory corruption vulnerabilities in widely used software has been a threat for almost three decades and no end seems to be in sight. In particular, code-reuse techniques such as return-oriented programming offer a robust attack technique that is extensively used to exploit memory corruption vulnerabilities in modern software programs (*e. g.* web browsers or document viewers). Whereas conventional control-flow attacks (runtime exploits) require the injection of malicious code, code-reuse attacks leverage code that is already present in the address space of an application to undermine the security model of data execution prevention (DEP). In addition, code-reuse attacks in conjunction with memory disclosure attack techniques circumvent the widely applied memory protection model of address space layout randomization (ASLR). To counter this ingenious attack strategy, several proposals for enforcement of control-flow integrity (CFI) and fine-grained code randomization have emerged.

In this dissertation, we explore the limitations of existing defenses against code-reuse attacks. In particular, we demonstrate that various coarse-grained CFI solutions can be effectively undermined, even under weak adversarial assumptions. Moreover, we explore a new return-oriented programming attack technique that is solely based on indirect jump and call instructions to evade detection from defenses that perform integrity checks for return addresses.

To tackle the limitations of existing defenses, this dissertation introduces the design and implementation of several new countermeasures. First, we present a generic and fine-grained CFI framework for mobile devices targeting ARM-based platforms. This framework preserves static code signatures by instrumenting mobile applications on-the-fly in memory. Second, we tackle the performance and security limitations of existing CFI defenses by introducing hardware-assisted CFI for embedded devices. To this end, we present a CFI-based hardware implementation for Intel Siskiyou Peak using dedicated CFI machine instructions. Lastly, we explore fine-grained code randomization techniques.

ZUSAMMENFASSUNG

Laufzeitangriffe nutzen Speicher- und Programmierfehler aus, um beliebiges Schadverhalten auf einem Computersystem zu verursachen. Obwohl diese Angriffe seit über zwei Jahrzehnten bekannt sind, stellen sie immer noch eine große Bedrohung für moderne Software-Programme dar. Dabei benutzen heutige Angriffe eine ausgeklügelte Technik, die sich Return-Oriented Programming (ROP) nennt. Im Gegensatz zu klassischen Laufzeitangriffen, die auf das Einschleusen von Schadcode in den Speicher eines Programmes angewiesen waren, können ROP Angriffe allein über das Zusammensetzen von vorhandenen gutartigen Code Schadverhalten erzeugen. Weil hierbei kein neuer Schadcode explizit eingeschleust wird, umgehen ROP Angriffe weit verbreitete Abwehrmechanismen wie beispielsweise Ausführungsschutz für den Arbeitsspeicher. Insbesondere können ROP Angriffe in Kombination mit sogenannten Speicherlecks –in der englischen Fachliteratur häufig als Memory Disclosure bezeichnet– dazu verwendet werden, um Adress Randomisierung zu umgehen. Um effektiv gegen diese neuartigen Laufzeitangriffe vorzugehen, wurden in den letzten Jahre eine Vielzahl an Abwehrmethoden vorgeschlagen, die meistens entweder auf Kontrollfluss-Integrität oder auf fortgeschrittenen Speicheradressen Randomisierungstechniken basieren.

In dieser Dissertation erforschen wir die Grenzen und Einschränkungen von existierenden Schutzmechanismen gegen ROP Angriffe und demonstrieren praktische Angriffe gegen kürzlich präsentierte Kontrollfluss-Integritätslösungen sowie Speicheradressen Randomisierungstechniken, die nicht selten mit nur minimalen Anforderungen angegriffen werden können. Insbesondere präsentieren wir einen neuartigen ROP Angriff, der ausschließlich indirekte Sprungbefehle missbraucht, um Detektion von Schutzmechanismen zu umgehen, die Integritätsprüfungen für Funktionsrücksprünge ausführen.

Um den Sicherheitsproblemen von vorhandenen Schutzmechanismen effektiv entgegenzutreten, stellen wir in dieser Dissertation die Konzepte und die Implementierungen von mehreren neuartigen Abwehrmethoden vor. Zuerst präsentieren wir eine allgemeine und fortgeschrittene Kontrollflussintegritäts-Lösung für mobile Geräte. Unsere Lösung ist kompatibel zu digitalen Code Signaturen, weil sie nur bereits verifizierten Code dynamisch im Adressspeicher um Integritätsprüfungen erweitert. Zudem erforschen wir einen neuen Hardware-basierten Ansatz zur Kontrollflussintegrität, der im Besonderen die Leistungseinbußen von existierenden Ansätzen löst. Unser Prototyp basiert auf der Intel-basierten Platform Siskiyou Peak, die besonders für eingebettete Systeme geeignet ist. Zuletzt erforschen wir fortgeschrittene Speicheradressen Randomisierungstechniken.

ACKNOWLEDGMENTS

Foremost, I'd like to thank my advisor Prof. Ahmad-Reza Sadeghi for his support, feedback, and guidance. I am very grateful for the intensive and fruitful discussions we had with Ahmad during the five years of my PhD research. His passion and dedication for research has highly influenced my career in academia. I am grateful for the numerous opportunities he has given me, particularly for connecting me with outstanding security researchers worldwide.

Besides my advisor Ahmad, I want to thank several professors for their support and advices. Prof. Fabian Monrose from the University of North Carolina at Chapel Hill has supported me with great feedback and writing tips. I learned a lot from his accurateness and critical reviews. From the beginning of my PhD, I was privileged to collaborate with Prof. Thorsten Holz from Ruhr-Universität Bochum. Our collaboration with Thorsten's team resulted in several publications. I'd also like to thank Prof. N. Asokan (Aalto University) for working with me on a lecture book on mobile security, and Prof. Yier Jin (University of Central Florida) for collaborating on hardware-assisted control-flow integrity. I am very honored to have Prof. Hovav Shacham from University of California, San Diego as second referee of my PhD dissertation. I was also pleased to collaborate with him on advanced return-oriented programming attacks.

This dissertation is a result of collaborations with various co-authors and colleagues. In particular, I'd like to thank all my colleagues at the System Security Lab at Ruhr-Universität Bochum and Technische Universität Darmstadt, Germany. The joint research work with my colleagues Alexandra Dmitrienko, Marcel Winandy, Christopher Liebchen, Steffen Schulz, Thomas Fischer, Matthias Hanreich, Stefan Nürnberger, Sven Bugiel, Bhargava Shastry, Debayan Paul, Mihai Bucicoiu, and Stephan Heuser resulted in several publications.

I have been very lucky to work with several excellent researchers. I'd like to particularly thank Ralf Hund from Ruhr-Universität Bochum. For several months we visited each other, and developed a control-flow integrity framework for mobile devices. Special thanks also go to Kevin Z. Snow from University of North Carolina at Chapel Hill. We presented together just-in-time return-oriented programming attacks at BlackHat USA 2013 and received the best student paper award at IEEE Security & Privacy 2013. There are many other external co-authors I was privileged to work with, particularly Per Larsen, Stephen Crane, Andrei Homescu, Dean Sullivan, Orlando Arias, Mauro Conti, Marco Negro, Stephen Checkoway, Manuel Egele, Blaine Stancill, Nathan Otterness, Felix Schuster, Kari Kostiainen, Elena Reshetova, and Razvan Deaconescu.

I had the privilege to supervise and work with several outstanding students. In the bachelor thesis of Daniel Lehmann, we conducted a security analysis of recently proposed control-flow integrity schemes. The master thesis of Tim Werthmann introduces fine-grained application sandboxing on iOS. For Christopher Liebchen's master thesis, we explored fine-grained code randomization and execution-path randomization.

During my PhD, I had the unique opportunity to do a summer internship at the Security Research Lab at Intel Labs in Hillsboro, Oregon. I'd like to thank my manager Anand Rajan and my mentor Manoj Sastry. Both have greatly supported me during my internship. Because of our newly founded Intel Security Institute in Darmstadt (Intel CRI-SC), I had the opportunity to work with Matthias Schunter, Patrick Koeberl, and Steffen Schulz. In particular, I'd like to thank Patrick for his great feedback on the architectural and implementation aspects of hardware-assisted control-flow integrity.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Computing platforms have become an integral part of our society over the last few decades. The landscape of computing platforms is highly diverse: starting from desktop PCs and laptops for end-users, powerful workstations used to perform highly complex calculations (*e.g.* weather calculations), web servers that need to simultaneously handle thousands of incoming requests, smartphones and tablets enabling on-the-road data access, up to tiny embedded devices deployed in sensors, cars, electronic passports, and medical devices. The inter-connectivity of these devices, *i.e.* the Internet of Everything [93], as well as the sensitive operations and everyday tasks we perform on these devices have made computing platforms an appealing target for various attacks. In particular, the fact that devices are connected with each other and to the Internet has facilitated remote attacks, where the attacker does not require any physical hardware access to compromise the victim's platform.

A prominent entry point of remote attacks is a vulnerable software program executing on the target computing platform. In general, computing platforms execute a large number of software programs to operate correctly and meaningful. Modern platforms typically include an operating system kernel (Windows, UNIX/Linux, Mac OS X) as well as user applications that run on top of that operating system, *e.g.* a web browser, word processor, video player, or a document viewer. Software is written by diverse developers, most of which are not security experts. As a consequence, software indeed operates as expected, but may miss necessary security checks that a sophisticated adversary can exploit to initiate an attack. That is, the adversary provides a malicious input that exploits a program bug to trigger malicious program actions never intended by the developer of the program. This class of software-based attack is generally known as a *runtime exploit* or *control-flow attack*, and needs to be distinguished from conventional *malware* that encapsulates its malicious program actions inside a dedicated executable. This executable simply needs to be executed on the target system and typically requires no exploitation of a program bug. While both runtime exploits and malware are important and prevalent software-based attacks, we mainly focus in this dissertation on runtime exploits, *i.e.* the art of exploiting benign software programs so that they behave maliciously.

In fact, one of the first large-scale attacks against the Internet contained a runtime exploit: in 1988, the Morris worm affected around 10% of all computers connected to the Internet disrupting Internet access for several days [28]. The worm exploited a program bug in `fingerd` – a service (daemon) that allows exchange of status and user information. Specifically, the worm initiated a session with the `finger` server, and sent a special crafted package that overflowed a local buffer on the stack to (i) inject malicious code on the stack, and (ii) overwrite adjacent control-flow information (*i.e.* a function's return address located on the stack) so that the program's control-flow was redirected to the

injected code [177]. Runtime exploits, like the Morris worm, which involve overflowing a buffer are commonly referred to as *buffer overflow attacks*.

The exploitation technique used for the Morris worm is typical for runtime exploits: they first require the exploitation of a program bug to inject malicious code and overwrite control-flow information. In the specific attack against `fingerd`, the problem is caused by a default C library function called *gets()*. This function reads characters, and stores them into a destination buffer until it reaches a newline character. However, it does not validate whether the input it processes fits to the destination buffer. Hence, a large input will exceed the buffer's bounds, thereby overwriting adjacent information such as a function's return address in memory.

Similar severe attacks have threatened end-users over the last years. In 2001, the so-called Code Red worm infected 250,000 machines within 9 hours by exploiting a known buffer overflow vulnerability in Microsoft's Internet Information Server [58]. Another well-known buffer overflow attack is the SQL Slammer worm which infected 75,000 machines in just 10 minutes causing major Internet slowdowns in 2003 [142].

The continued success of these attacks can be attributed to the fact that large portions of software programs are implemented in type-unsafe languages (C, C++, or Objective-C) that do not enforce bounds checking on data inputs. Moreover, even type-safe languages (*e.g.* Java) rely on interpreters (*e.g.* the Java virtual machine) that are in turn implemented in type-unsafe languages. Sadly, as modern compilers and applications become more and more complex, memory errors and vulnerabilities will likely continue to persist, with little end in sight [184, 180].

The most prominent example of a memory error is the stack overflow vulnerability, where the adversary overflows a local buffer on the stack, and overwrites a function's return address [7]. While modern defenses protect against this attack strategy (*e.g.* by using stack canaries [52]), other avenues for exploitation exists, including those that leverage heap [153], format string [86], or integer overflow [27] vulnerabilities.

Regardless of the attacker's method of choice, exploiting a vulnerability and gaining control over an application's control-flow is only the first step of a runtime exploit. The second step is to launch malicious program actions. Traditionally, this has been realized by injecting malicious code into the application's address space, and later executing the injected code. However, with the wide-spread enforcement of data execution prevention such attacks are more difficult to do today [132]. Unfortunately, the long-held assumption that only code injection posed a risk was shattered with the introduction of *code-reuse attacks*, such as return-into-libc [174, 143] and return-oriented programming [169]. As the name implies, code-reuse attacks do not require any code injection and instead use code already resident in memory.

Code-reuse attack techniques are applicable to a wide range of computing platforms: Intel x86-based platforms [169], embedded systems running on an Atmel AVR processor [79], mobile devices based on ARM [117, 16], PowerPC-based Cisco routers [123], and voting machines deploying a z80 processor [40]. Moreover, the powerful code-reuse attack technique return-oriented programming is Turing-complete, *i.e.* it allows an attacker to execute arbitrary malicious code.

In fact, the majority of state-of-the-art runtime exploits leverage code-reuse attack techniques, *e.g.* runtime exploits against Internet Explorer [186, 128], Apple QuickTime [91], Adobe Reader [110], Microsoft Word [46], or the GnuTLS library [195]. Even large-scale cyber attacks such as the popular Stuxnet worm, which damaged Iranian centrifuge rotors, partially deploys code-reuse attack techniques [129].

1.1 GOAL AND SCOPE OF THIS DISSERTATION

The main goals of this dissertation are (1) introducing the design and implementation of new defenses against code-reuse attacks, and (2) advancing the state-of-the-art in code-reuse attack exploitation to demonstrate limitations of existing and recently proposed defenses. In general, we focus on defenses that build on the principle of control-flow integrity (CFI) [4] and code randomization [50]. The former provides a generic approach to mitigate runtime exploits by forcing a program's control-flow to always comply to a pre-defined control-flow graph. In contrast, the latter does not perform any explicit control-flow checks. It mitigates code-reuse attacks by randomizing the code layout so that an attacker can hardly predict where useful code resides in memory. As we will discuss throughout this dissertation, both security principles have been intensively explored over the last few years by academia and industry. For instance, Microsoft started a competition, the Microsoft BlueHat Prize, where three CFI-based proposed defenses against return-oriented programming have been awarded with 250k dollars [181]. Its public security tool Microsoft EMET incorporates some of these defenses to mitigate code-reuse attacks [131].

It is important to note that runtime exploits involve two stages: (1) initial control-flow exploitation, and (2) malicious program actions. In this dissertation, we focus on the second stage of runtime exploits, *i.e.* the execution of malicious computations. Modern stack and heap mitigations (such as stack canaries [52], or heap allocation order randomization [147]) do eliminate categories of attacks supporting stage one, but these mitigations are not comprehensive (*i.e.* exploitable vulnerabilities still exist). Several efforts have been undertaken to achieve memory safety, *i.e.* preventing any code pointer overwrites for type-unsafe languages [53, 139, 140, 119]. However, these solutions necessarily require access to source code and sometimes incur high performance overhead. Thus, in this dissertation, we assume the adversary is able to exercise one of these pre-existing vulnerable entry points. As a result, a full discussion of the first stage of runtime exploits is out of scope for this dissertation.

1.2 SUMMARY OF CONTRIBUTIONS

The main contributions of this dissertation are as follows:

Return-oriented Programming without Returns. We demonstrate an advanced code-reuse attack technique that is solely based on exploiting indirect jump and call instructions on mobile platforms based on an ARM processor. Hence, our attack cannot be prevented by defenses that aim at protecting return addresses such as stack canaries [52]

and return address stacks [81, 61]. To this end, we introduce a generic attack method that emulates the return behavior (without using returns), generate a Turing-complete gadget set, and demonstrate the feasibility of our approach by applying it to an Android-based device.

Code-Reuse Attacks against Coarse-Grained Control-Flow Integrity. We provide the first comprehensive security analysis of various coarse-grained control-flow integrity (CFI) solutions (covering kBouncer [150], ROPecker [47], CFI for COTS binaries [204], ROPGuard [83], and Microsoft EMET 4.1 [131]). A key contribution is in demonstrating that these techniques can be effectively undermined, even under weak adversarial assumptions. More specifically, we show that with bare minimum assumptions, Turing-complete and real-world code-reuse attacks can still be launched even when the strictest of enforcement policies is in use. To do so, we introduce several new code-reuse attack primitives, and demonstrate the practicality of our approach by transforming existing real-world exploits into more stealthy attacks that bypass coarse-grained CFI defenses.

Control-Flow Integrity for Mobile Devices. We introduce the design and implementation of the first generic control-flow integrity (CFI) framework for mobile devices. We realized MoCFI (Mobile CFI) for ARM-based mobile devices running iOS. For this, we had to overcome a number of challenges which are mainly caused due to subtle architectural differences between x86 and ARM, and the unique characteristics of iOS (*e.g.* application signing and closed-source OS). To tackle the latter challenge, we developed a new binary rewriting framework that performs CFI instrumentation for iOS apps at application load-time in memory in order to not break static code signatures. To demonstrate the usefulness of MoCFI, we implemented a fine-grained application sandboxing framework on top of MoCFI that enables user-defined and application-specific sandboxing rules. Our approach tackles the problem of allowing every iOS app to execute under the same sandboxing profile (privilege level).

Hardware-Assisted Control-Flow Integrity. We present a hardware-assisted CFI design and implementation to tackle the efficiency and security limitations of existing defenses. To this end, we implemented and evaluated our hardware-assisted CFI extensions on real hardware platforms: Intel Siskiyou Peak [156] and SPARC LEON3 [157]. As we will demonstrate, our approach significantly reduces the code base for code-reuse attacks and efficiently protects function returns with an overhead of only 2%.

Advanced Code Randomization Techniques and Attacks. To tackle the low randomization entropy of existing base address randomization schemes such as ASLR [151], and effectively mitigate code-reuse attacks via code randomization, we introduce a tool, called XIFER, that provides per-run basic block randomization with an average overhead of only 5%. However, we also show that existing approaches to code randomization can be circumvented with just-in-time code-reuse attacks. Inspired by this advanced code-reuse attack technique, we developed a novel code randomization framework which combines code and execution-path randomization to resist conventional and just-in-time code-reuse attacks.

1.3 OUTLINE

This dissertation is structured as follows: in Chapter 2, we provide comprehensive background on control-flow attacks covering classic code injection attacks and modern code-reuse attack techniques. We also describe widely adopted countermeasures against control-flow attacks such as data execution prevention and base address randomization. Further, we describe in detail the principle of control-flow integrity (CFI). Next, in Chapter 3, we introduce two advanced code-reuse attack techniques. First, we demonstrate a new code-reuse attack targeting mobile devices that is based on exploiting indirect jump and call instructions (Section 3.1). Second, we demonstrate that coarse-grained CFI solutions do not resist sophisticated code-reuse attacks that stitch gadgets from call-preceded code sequences (3.2). In Chapter 4, we turn our attention to CFI-based countermeasures. We present mobile control-flow integrity for ARM-based mobile devices running iOS without requiring source code (Section 4.1), fine-grained application sandboxing for iOS based on our mobile CFI framework (Section 4.2), and hardware-assisted CFI checks for Intel Siskiyou Peak (Section 4.3). In Chapter 5, we present advanced code randomization schemes and attacks: load-time basic block permutation (Section 5.2), just-in-time code-reuse attacks (Section 5.3), and a combination of code and execution-path randomization (Section 5.4). We conclude this dissertation in Chapter 6.

1.4 PREVIOUS PUBLICATIONS

This dissertation is based on several previously published publications as listed below. The full list of publications published by the author of this dissertation can be found in Chapter 7.

Chapter 3: Advanced Code-Reuse Attacks

Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, Marcel Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security* (CCS'10), 2010.

Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

Chapter 4: Advanced Defenses Against Code-Reuse Attacks

Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, Ahmad-Reza Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (NDSS'12), 2012.

Tim Werthmann, Ralf Hund, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz. PSiOS: Bring Your Own Privacy & Security to iOS Devices (Distinguished Paper Award). In *Pro-*

ceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS'13), 2013.

Lucas Davi, Patrick Koeberl, Ahmad-Reza Sadeghi. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Proceedings of the 51st Design Automation Conference - Special Session: Trusted Mobile Embedded Computing* (DAC'14), 2014.

Orlando Arias, Lucas Davi, Matthias Hanreich, Yier Jin, Patrick Koeberl, Debayan Paul, Ahmad-Reza Sadeghi, Dean Sullivan. HAFIX: Hardware-Assisted Flow Integrity Extension. In *Proceedings of the 52nd Design Automation Conference*, DAC'15, 2015.

Chapter 5: Advanced Code Randomization

Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, Ahmad-Reza Sadeghi. Gadge Me If You Can - Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security* (ASIACCS'13), 2013.

Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization (Best Student Paper Award). In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (SP'13), 2013.

Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium* (NDSS'15), 2015.

2 BACKGROUND

In this chapter, we provide basic background information on control-flow attacks and defenses. The introduced concepts are vital to understand the remainder of this dissertation. We first review the most prominent runtime exploit techniques including code injection and code reuse (Section 2.1). Next, we elaborate in Section 2.2 and 2.3 on two important defense strategies: code randomization based on address space layout randomization (ASLR), and control-flow integrity (CFI).

Since control-flow attacks operate at the level of assembler instructions, they are typically tailored to a specific underlying processor architecture. Due to the popularity of x86-based systems and the many exploits available for this architecture, we take x86 as our reference architecture throughout this chapter, and use the Intel assembler syntax, *e. g.* `MOV destination,source`. However, note that the subsequent chapters also include control-flow attacks and defenses on ARM-based mobile devices. Background information on ARM and differences to x86 will be provided in Section 3.1.1.

2.1 CONTROL-FLOW ATTACKS

In general, control-flow attacks or allow an adversary to subvert the intended execution-flow of a program by exploiting a program error. For instance, a buffer overflow error can be exploited to write data beyond the boundaries of the buffer. As a consequence, an adversary can overwrite critical control-flow information which is located close to the buffer. Since control-flow information guide the program's execution-flow, an adversary can thereby trigger malicious and unintended program actions such as installing a backdoor, injecting a malware, or accessing sensitive data.

Control-flow attacks are performed at application runtime. Hence, they are often referred to as runtime exploits. Note that we use both terms interchangeably in this dissertation. In summary, we define a control-flow attack as follows.

> **Control-Flow Attack (Runtime Exploit)**: *A control-flow attack exploits a program error, particularly a memory corruption vulnerability, at application runtime to subvert the intended control-flow of a program. The goal of a control-flow attack is the execution of malicious program actions.*

2.1.1 *General Principle of Control-Flow Attacks*

Loosely speaking, we can distinguish between two major classes of control-flow attacks: (1) code injection, and (2) code-reuse attacks. The former class requires the injection of some malicious executable code into the address space of the application. In contrast,

code-reuse attacks only leverage benign code already present in the address space of the application. In particular, code-reuse attacks combine small code pieces scattered throughout the address space of the application to generate new malicious program codes on-the-fly.

A high-level representation of code injection attacks is given in Figure 1. It shows a sample control-flow graph (CFG) with six nodes. The CFG contains all intended execution paths a program might follow at runtime. Typically, the CFG nodes represent the so-called basic blocks (BBLs), where a BBL is a sequence of machine/assembler instructions with a unique entry and exit instruction. The exit instruction can be any branch instruction the processor supports such as direct and indirect jump instructions, direct and indirect call instructions, and function return instructions. The entry instruction of a BBL is an instruction that is target of a branch instruction.

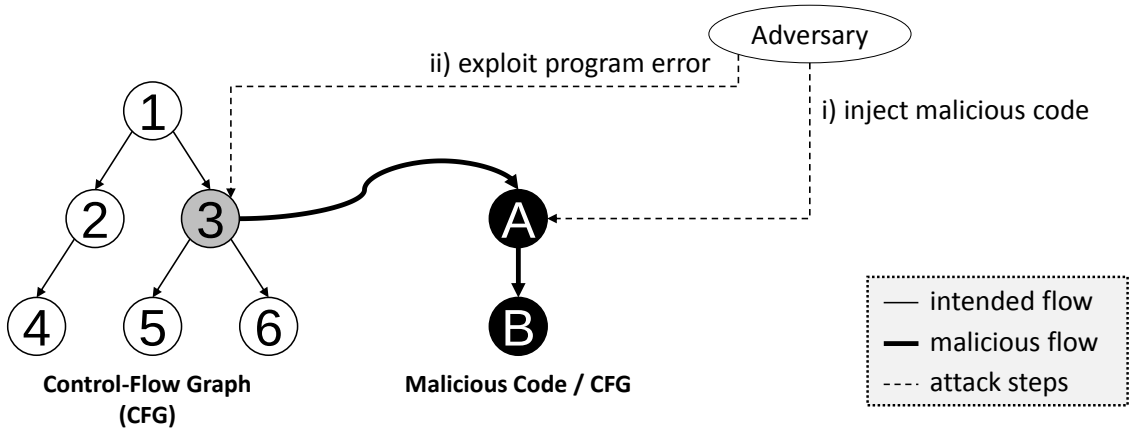


Figure 1: Code injection attacks

As shown in Figure 1, the CFG nodes are connected via directed edges. These edges represent the possible control-flows, *e. g.* there is an intended control-flow from node n_3 to n_5 and n_6, where n simply stands for node.

A code injection attack first requires the injection of malicious code. As programs are residing in a dedicated memory location at runtime, *i. e.* the application's virtual address space, the adversary needs to find a free slot where the code can be injected. Typically, this can be achieved by loading the malicious code into a local buffer that is large enough to hold the entire malicious code. In Figure 1, the malicious code consists of the two nodes n_A and n_B. However, these nodes are not connected to the original CFG. In order to connect the malicious nodes to the intended program nodes, the adversary needs to identify and exploit a program vulnerability. Exploitation of the program vulnerability allows the adversary to tamper with a code pointer, *i. e.* some control-flow information that guides program execution. A prominent example is a function's return address which is always located on the program stack. Other examples are function pointers or pointers to virtual method tables. In the example exploit shown in Figure 1, n_3 is exploited by the adversary to redirect the execution path to node n_A and n_B. In summary, we define code injection attacks as follows.

Code Injection Attack: *A code injection attack is a subclass of control-flow attacks that subverts the intended control-flow of a program to previously injected malicious code.*

Code injection attacks require the injection and execution of malicious code. However, some environments and operating systems deny the execution of code that has just been written into the address space of the program. Consider for instance a Harvard-based computing architecture, where code and data are strictly separated from each other. Hence, a new form of control-flow attacks re-uses existing code. The high-level principle of these so-called code-reuse attacks is shown in Figure 2.

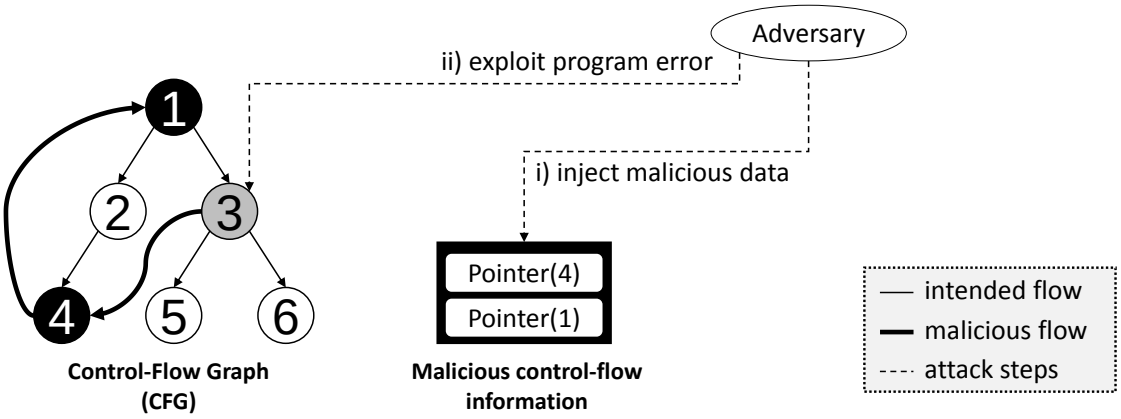


Figure 2: Code-reuse attacks

In contrast to code injection attacks, the adversary only injects malicious data into the address space of the application. Specifically in the example shown in Figure 2, the adversary injects two code pointers; namely pointers to n_4 and n_1. At the time the adversary exploits the program vulnerability in n_3, the control-flow is redirected to the code pointers the adversary previously injected. Hence, the code-reuse attack in our example leads to the unintended execution path: $n_3 \rightarrow n_4 \rightarrow n_1$. In summary, we define a code-reuse attack as follows.

Code-Reuse Attack: *A code-reuse attack is a subclass of control-flow attacks that subverts the intended control-flow of a program to invoke an unintended execution path inside the original program code.*

Note that the internal workflow and memory layout of a control-flow attack depends on the kind of vulnerability that is exploited. For better understanding, we describe the technical details of control-flow attacks based on a classic buffer overflow vulnerability on the program's stack. Hence, we briefly recall the basics of a program's stack memory and the typical stack frame layout on x86. Afterwards, we present the technical details of code injection and code-reuse attacks.

2.1.2 *Program Stack and Stack Frame Elements*

A program stack operates as a last-in first-out memory area. In particular, it is used in today's programs to hold local variables, function arguments, intermediate results, and control-flow information to ensure correct invocation and return of subroutines. The stack pointer which is stored in a dedicated processor register plays an important role as it points to the top of the stack. Typically, the stack is controlled by two operations: (1) a `POP` instruction that takes one data word off the stack, and (2) a `PUSH` instruction which performs the reverse operation, *i. e.* it stores one data word on the top of the stack. Both instructions have direct influence on the stack pointer since they change the top of the stack. That is, for stacks that grow from high memory addresses towards low memory addresses (*e. g.* x86), the `POP` instruction automatically increments the stack pointer by one memory word (on x86: 4 Bytes), while the `PUSH` instruction decrements it by one word.

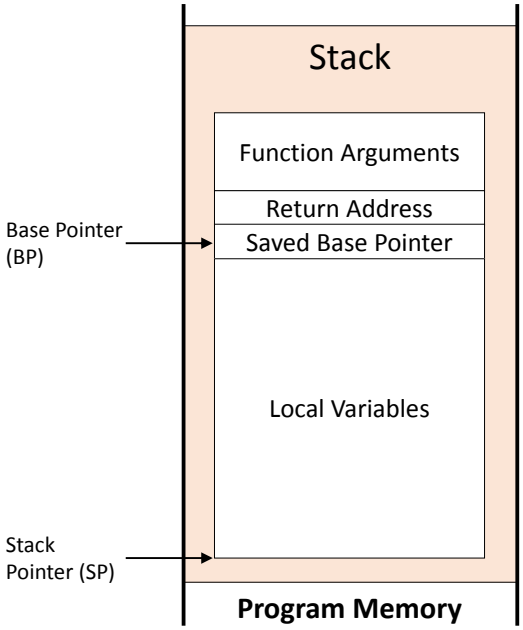


Figure 3: Stack frame memory layout

In general, the stack is divided into multiple stack frames. Stack frames are associated at function-level, *i. e.* for each invoked subroutine one stack frame is allocated. The stack frame has a pre-defined structure for each compiler and underlying processor architecture. An example of a typical x86 stack frame and its elements is shown in Figure 3. The depicted stack frame is referenced by two processor registers: the stack pointer (on x86 `%esp`) and the base pointer register (on x86 `%ebp`). As we already mentioned, the stack pointer always points to the top of the stack. In contrast to the stack pointer, the base pointer is constant and fixed per stack frame: it always points to the saved based pointer. The meaning of the saved base pointer and the other stack frame elements is as follows:

- **Function Arguments**: This field holds the arguments which are loaded on the stack by the calling function.
- **Return Address**: The return address indicates where the execution-flow needs to be redirected to upon function return. On x86, the instruction for calling a function (`CALL`) automatically pushes the return address on the stack, where the return address is the address of the instruction that follows the `CALL`.
- **Saved Base Pointer**: The base pointer of a function is used to reference function arguments and local variables on the stack frame. The function prologue of each subroutine initializes the base pointer. This is achieved in two steps. First, the base pointer of the calling function is pushed onto the stack via `PUSH %ebp`. The base pointer stored onto the stack is then referred to as the saved base pointer. Next, the new base pointer is initialized by loading the current stack pointer value into the base pointer register, *e.g.* `MOV %ebp,%esp`. The function epilogue reverts these operations by first setting the stack pointer to point to the saved base pointer field (`MOV %esp,%ebp`), and subsequently loading the saved base pointer to the base pointer register via `POP %ebp`.
- **Local Variables**: The last field of a stack frame holds the local variables such as integer values or local buffers.

2.1.3 *Code Injection*

In order to perform a code injection attack, the adversary needs to inject malicious code into the address space of the target application. Typically, this can be achieved by encapsulating the malicious code into a data input variable that gets processed by the application, *e.g.* a string, file, or network packet.

Figure 4 depicts a code injection attack, where a local buffer that can hold up to 100 characters is exploited. The adversary has access to the local buffer, *i.e.* the application features a user interface from which it expects the user to enter a data input. On x86, data is written from low memory addresses towards high memory addresses, *i.e.* from the top of the stack towards the saved base pointer in Figure 4.

If the program does not validate the length of the provided data input, it is possible to provide a larger data input than the buffer can actually hold. As a consequence, the stack frame fields which are located above the local buffer are overwritten.

This can be exploited for the purpose of a code injection attack: the adversary first provides a data input which fills the local buffer with arbitrary pattern bytes and the malicious code. As the main goal of many proof-of-concept exploits is to open a terminal (shell) to the adversary, the malicious code is often referred to as shellcode. Second, the adversary overwrites the saved base pointer with arbitrary bytes (here: 0x41414141) and replaces the original return address with the runtime address of the shellcode. For systems that do not apply code and data segment randomization, this address is fixed, and can be retrieved by reverse-engineering the program binary using a debugger.

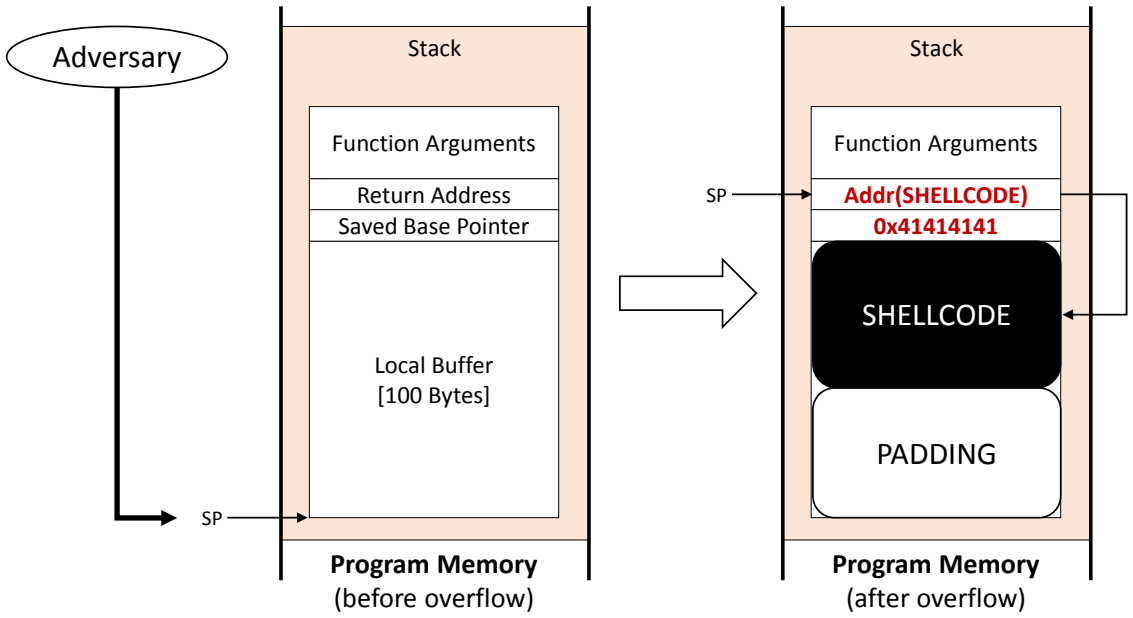


Figure 4: Memory layout of code injection attacks

When the subroutine – where the overflow occurred – completed its task and executes its function epilogue instructions, the stack pointer will be re-set to the location, where the original return address was stored. As the program is not aware of the overflow, it takes the start address of the shellcode as a return address and redirects execution to the beginning of the shellcode. Thus, the shellcode executes and opens a new terminal to the adversary.

2.1.4 *Data Execution Prevention*

One main observation we can make from the code injection attack described above is that malicious code can be encapsulated into a data variable and executed from the program's stack. In fact, code injection attacks were easily possible because data and code got intermixed in memory, and not strictly separated as in Harvard-based processor architectures. Hence, data segments like the stack were marked as readable, writable, and executable (RWX). However, since the main purpose of the stack is still to only hold local variables and control-flow information, we simply need a mechanism to prohibit any code execution from the stack to prevent a code injection attack. As a consequence, kernel patches have been provided to mark the stack as non-executable [173]; *e. g.* enabled in Solaris 2.6 [30].

The concept of marking the stack as non-executable has been later included into a more general security model referred to as Writable XOR eXecutable ($W \oplus X$) or data execution prevention (DEP) [132]. The main idea of $W \oplus X$ is to prevent any memory page from being writable and executable at the same time. Hence, memory pages belonging to data segments are marked as readable and writable (RW), whereas memory

pages that contain executable code are marked as readable and executable (RX). This effectively prevents code injection attacks since an adversary can no longer execute code that has been written via a data variable to a RW-marked memory page. In summary, we define the principle $W \oplus X$ as follows.

> **Writable xor eXecutable ($W \oplus X$)**: *The security model of $W \oplus X$ enforces that memory pages are either marked as writable or executable. This prevents a code injection attack, where the adversary first needs to write malicious code into the address space of an application before executing it.*

Today, every mainstream processor architecture features the so-called no-execute bit to facilitate the deployment of $W \oplus X$ in modern operating systems. For instance, Windows-based operating systems enforce data execution prevention since Windows XP SP2 [132].

2.1.5 *Code-Reuse Attacks*

After non-executable stacks and $W \oplus X$ have been proposed as countermeasures against control-flow attacks, attackers have instantly demonstrated new techniques to launch control-flow attacks. Instead of injecting malicious code into the address space of the application, an adversary can exploit the benign code which is already present in the address space and marked as executable. Such code-reuse attacks have started as so-called return-into-libc attacks and have been later generalized to return-oriented programming attacks. We describe the technical concepts of both attack techniques in the following.

2.1.5.1 *return-into-libc*

The first published exploit that re-uses existing code for a return-into-libc attack has been presented by Solar Designer in 1997 [174]. The exploit overwrites the original return address to point to a critical library function. Specifically, it targets the *system()* function of the standard UNIX C library `libc`, which is linked to nearly every process running on a UNIX-based system. The *system()* function takes as an input a shell command to be executed. For instance, on UNIX-based systems the function call *system("'/bin/sh"')* opens the terminal program. That said, by invoking the *system()* function, the adversary can conveniently reconstruct the operations of previously injected shellcode without injecting any code. In summary, we define a return-into-libc attack as follows.

> **return-into-libc**: *Code-reuse attacks that are based on the principle of return-into-libc subvert the intended control-flow of a program and redirect it to security-critical functions that reside in shared libraries or the executable itself.*

Figure 5 shows the typical memory layout of a return-into-libc attack. A necessary step of our specific return-into-libc attack is the injection of the string `/bin/sh` since *system()* is expecting a pointer to the program path in order to open a new terminal. To

tackle this issue, one could search for the string inside the entire address space of the target application. However, this approach is brittle as the string might not always be present in the address space of the application. A more robust exploitation approach is to inject the string into a data memory page and record its address. At first glance, this might seem a trivial step. However, the challenge stems from the fact that the string needs to be NULL-terminated. Hence, if we attempt to inject the string into the local buffer, we would need to process a NULL Byte which is impossible for many vulnerabilities. For instance, classic buffer overflow vulnerabilities introduced via the *strcpy()* function terminate the write operation if a NULL Byte is processed. The classic return-into-libc attacks overcome this issue by injecting the string as an environment variable. In Figure 5, the adversary defines the `$SHELL` environment variable which contains the string `/bin/sh`.

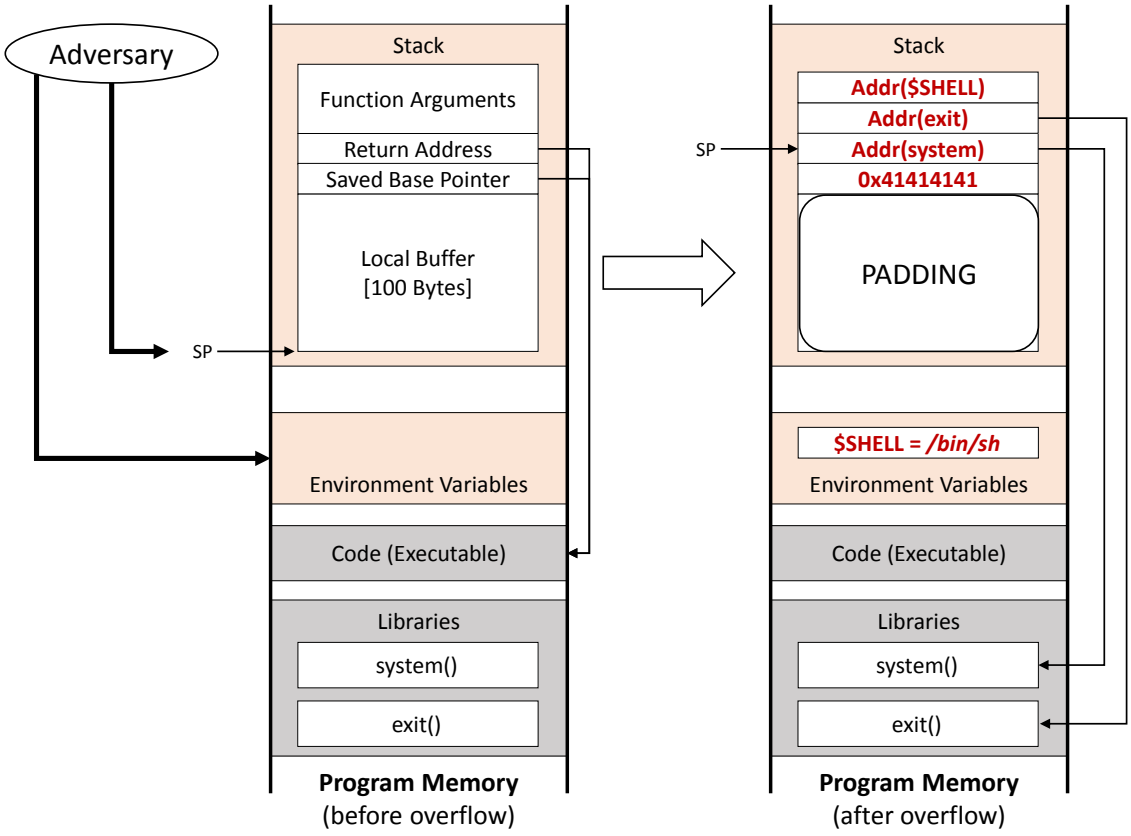


Figure 5: Basic principle of return-into-libc attacks

After the environment variable has been defined, the adversary interfaces to the application by providing a data input that exceeds the local buffer's limits. Specifically, the adversary fills the local buffer with arbitrary pattern bytes. In addition, the saved base pointer is overwritten with 4 Bytes of arbitrary data. Finally, the return address is replaced with the runtime address of the *system()* function. Moreover, two other addresses are written on the stack: the runtime address of the `libc` *exit()* function, and the runtime address of the `$SHELL` variable. The latter resembles the function argument on the stack

frame of the invoked *system()* function. Considering the standard x86-based stack frame layout (see Figure 3), the former will be used as the return address of *system()*. In particular, the *exit()* function will terminate the process upon return of *system()*, *i.e.* at the time the adversary closes the terminal.

This basic return-into-libc attack requires the knowledge of three runtime addresses. In case no code and data randomization is applied, these addresses can be retrieved by reverse-engineering the application using a debugger. Otherwise, an adversary would need to disclose these addresses using memory disclosure techniques which we discuss later in Section 2.2.

The basic return-into-libc attacks only allow invocation of two library library functions, while the second function (in Figure 5 the *exit()* function) needs to be called without any argument. As this poses restrictions on the operations an adversary can perform, several advanced return-into-libc attack techniques have been proposed. For instance, Nergal demonstrated two techniques, called esp-lifting and frame faking, allowing an adversary to perform chained function calls in a return-into-libc attack [143].

2.1.5.2 *Return-Oriented Programming*

The above described return-into-libc attack technique has some limitations compared to classic code injection attacks. First, an adversary is dependent on critical `libc` functions such as *system()*, *exec()*, or *open()*. Hence, if we either instrument or eliminate these functions, an adversary would no longer be able to perform a reasonable attack. In fact, one of the first proposed defenses against return-into-libc is based on the idea of mapping shared libraries to memory addresses that always contain a NULL byte [174]. Second, return-into-libc only allows calling one function after each other. Hence, an adversary is not able to perform arbitrary malicious computation. In particular, it is not possible to perform unconditional branching.

There is also a challenge when applying return-into-libc attacks to Intel 64 Bit based systems (x86-64). On x86-64, function arguments are passed to a subroutine via processor registers rather than on the stack. To tackle this challenge, Krahmer [118] suggested an advanced return-into-libc attack technique called borrowed code chunks exploitation. The main idea is to borrow a function epilogue consisting of several `POP` register instructions. These instructions load the necessary function arguments into processor registers and subsequently redirect the execution-flow to the target subroutine.

Shacham [169] generalizes the idea of borrowed code chunks exploitation by introducing return-oriented programming. This attack technique tackles the previously mentioned limitations of return-into-libc attacks. The basic idea is to execute a chain of short code sequences rather than entire functions. Multiple code sequences are combined to a so-called gadget that performs a specific atomic task, *e.g.* a load, add, or branch operation. Given a sufficiently large code base, an adversary will most likely identify a gadget set that forms a new Turing-complete language. That said, the derived gadget set can be exploited to induce arbitrary malicious program behavior. The applicability of return-oriented programming has been shown on many platforms including x86 [169], SPARC [31], Atmel AVR [79], PowerPC [123], ARM [117], and z80 [40].

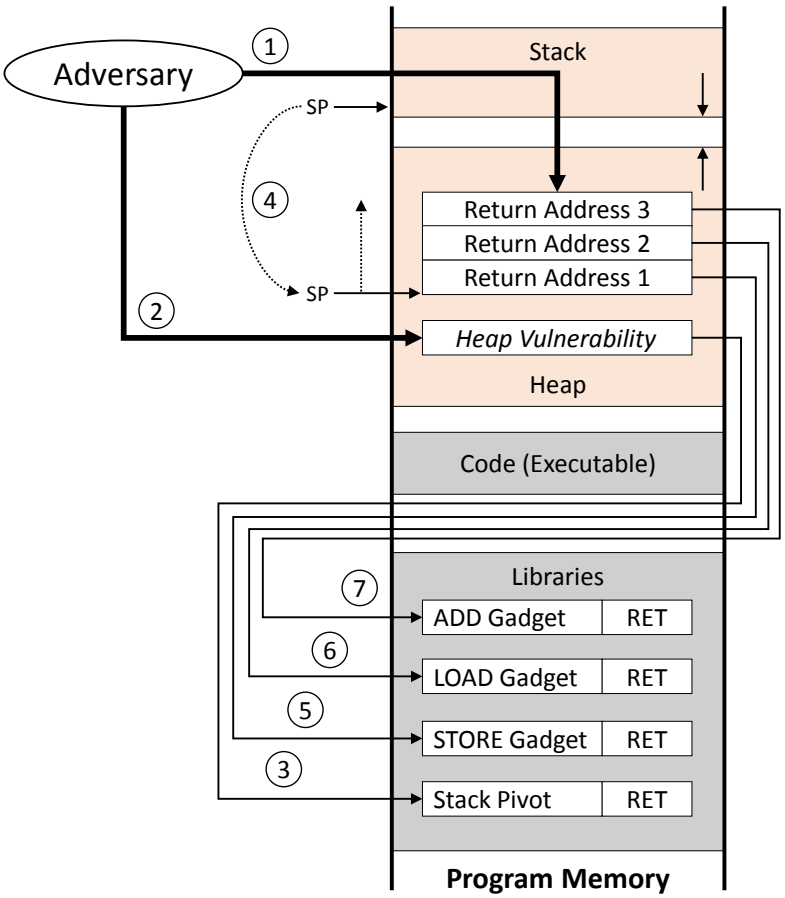


Figure 6: Basic principle of return-oriented programming attacks. For simplicity, we highlight a return-oriented programming attack on the heap using a sequence of single-instruction gadgets.

The basic idea and workflow of a return-oriented programming attack is shown in Figure 6. Note that we discuss a return-oriented programming attack that exploits a heap-based vulnerability to explain all basic attack steps that are taken in modern real-world code-reuse exploits. First, the adversary writes the return-oriented payload into the application's memory space, where the payload mainly consists of a number of pointers (the return addresses) and any other data that is needed for running the attack (Step ①). In particular, the payload is placed into a memory area that can be controlled by the adversary, *i.e.* the area is writable and the adversary knows its start address. The next step is to exploit a vulnerability of the target program to hijack the intended execution-flow (Step ②). In the example shown in Figure 6, the adversary exploits a heap vulnerability by overwriting the address of a function pointer with an address that points to a so-called *stack pivot* sequence [206]. Once the overwritten function pointer is used by the application, the execution-flow is redirected to a stack pivot sequence (Step ③).

Loosely speaking, stack pivot sequences change the value of the stack pointer (`%esp`) to a value stored in another register. Hence, by controlling that register[1], the adversary can arbitrarily change the stack pointer. Typically, the stack pivot directs the stack pointer to the beginning of the payload (Step ④). A concrete example of a stack pivot sequence is the x86 assembler code sequence `MOV %esp,%eax; ret`. The sequence changes the value of the stack pointer to the value stored in register `%eax` and subsequently invokes a return (`RET`) instruction. Notice that the stack pivot sequence ends in a `RET` instruction: the x86 `RET` instruction simply loads the address pointed to by `%esp` into the instruction pointer and increments `%esp` by one word. Hence, the execution continues at the first gadget (`STORE`) pointed to by Return Address 1 (Step ⑤). In addition, the stack pointer is increased and now points to Return Address 2.

It is exactly the terminating `RET` instruction that enables the chained execution of gadgets by loading the address the stack pointer points to (Return Address 2) in the instruction pointer and updating the stack pointer so that it points to the next address in the payload (Return Address 3). Steps ⑤ to ⑦ are repeated until the adversary reaches her goal. To summarize, the combination of different gadgets allows an adversary to induce arbitrary program behavior.

As a result, we define a return-oriented programming attack as follows.

> **Return-Oriented Programming**: *Code-reuse attacks that are based on the principle of return-oriented programming combine and execute a chain of short instruction sequences that are scattered throughout the address space of an application. Each sequence ends with an indirect branch instruction –traditionally, a return instruction– to transfer control from one sequence to the subsequent sequence. In particular, return-oriented programming has been shown to be Turing-complete, i. e. the instruction sequences it leverages can be combined to gadgets that form a Turing-complete language.*

Unintended Instruction Sequences. A crucial feature of return-oriented programming on Intel x86 is the invocation of the so-called *unintended* instruction sequences. These can be issued by jumping into the middle of a valid instruction resulting in a new instruction sequence neither intended by the programmer nor the compiler. Such sequences can be found in large number on the Intel x86 architecture due to unaligned memory access and variable-length instructions. As an example, consider the following x86 code with the given intended instruction sequence, where the byte values are listed on the left side and the corresponding assembly code on the right side:

Listing 1: Intended code sequence

```
b8 13 00 00 00   MOV $0x13,%eax
e9 c3 f8 ff ff   JMP 3aae9
```

1 To control the register, the adversary can either use a buffer overflow exploit that overwrites memory areas that are used to load the target register, or invoke a sequence that initializes the target register and then directly calls the stack pivot.

If the interpretation of the byte stream starts two bytes later the following unintended instruction sequence would be executed by an Intel x86 processor:

Listing 2: Unintended code sequence

```
00 00   ADD %al,(%eax)
00 e9   ADD %ch,%cl
c3      RET
```

In the intended instruction sequence the c3 byte is part of the second instruction. But if the interpretation starts two bytes later, the c3 byte will be interpreted as a return instruction.

2.1.6 *Hybrid Exploits*

Typically, modern systems enforce $W \oplus X$ by default. This forces an adversary to deploy code-reuse attacks. However, a deeper investigation of real-world code-reuse attacks quickly reveals that most exploits today use a combination of code-reuse and code injection. The main idea behind these hybrid exploits is to only use code-reuse attack techniques to undermine $W \oplus X$ protection and launch a code injection attack subsequently. This is possible due to the fact that $W \oplus X$ in its basic instantiation only enforces that a memory page is not writable and executable at the same time. However, a memory page can be first writable (not-executable) and at a later time executable (not-writable).

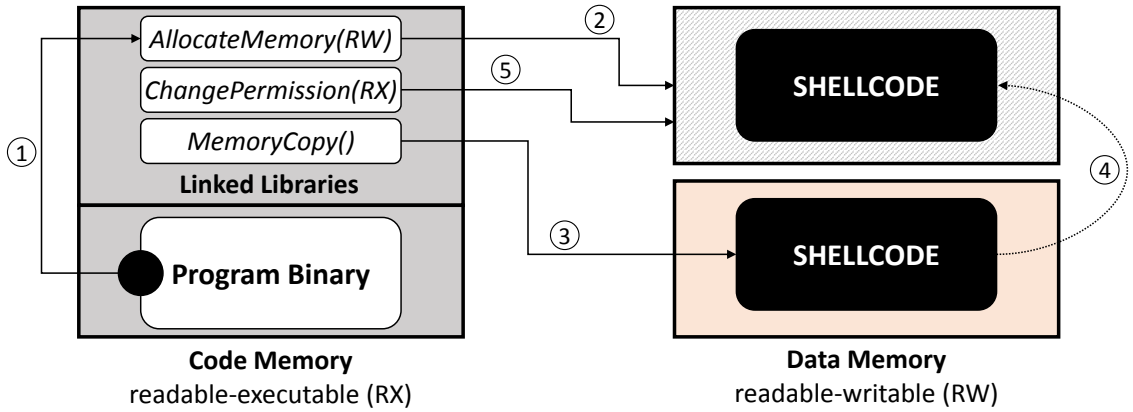


Figure 7: Hybrid exploitation: combination of code reuse with code injection

Figure 7 demonstrates this combined attack technique by example. The shown memory layout is divided into a code and data memory area, where the former one is readable and executable, and the latter one is marked as readable and writable. In particular, the code memory holds the program binary and linked shared libraries. In modern operating systems, several important libraries and their functionality are linked by default into the address space of the application. Consider as an example the UNIX C library `libc`. Although the target application may only require the *printf()* function to print

strings on the standard output (stdout), other `libc` functions such as *system()* or *memcpy()* will be always mapped into the address space of the application as well.

In our example, we assume that the return-oriented payload and the malicious code have been already injected into the data memory area. In Step ①, the payload exploits a program vulnerability and redirects execution to the shared library segment. Specifically, the adversary invokes a default function to allocate a new memory page (*e. g.* the *alloc()* function) marked as readable and writable (Step ②). Typically, the return value will be the runtime address of the newly allocated page. Upon return of the memory allocator, the return-oriented payload invokes a memory copy function (*e. g.* the *memcpy()* function) to copy the injected shellcode to the newly allocated memory page (Step ③ and ④). Finally, the payload invokes a system function (*e. g.* the *mprotect()* function) to change the memory page permissions of the newly allocated page to readable and executable (Step ⑤). Hence, the adversary can now execute the injected shellcode to perform the actual malicious program actions.

This attack can be further optimized. For instance, if the underlying operating system allows the allocation of read-write-execute (RWX) memory pages to support code generation just-in-time, an adversary can skip the *ChangePermission()* function. Further, it is possible to skip the *AllocateMemory()* and *CopyMemory()* function if the adversary knows the address of the memory page where the shellcode has been originally injected to. In that case, we can simply call the *ChangePermission()* function to mark the corresponding memory page as executable.

2.2 ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR)

A well-accepted countermeasure against code-reuse attacks is the randomization of the application's memory layout. The basic idea of address space layout randomization (ASLR) dates back to Forrest et al. [78], wherein a new stack memory allocator was introduced that adds a random pad for stack objects larger than 16 Bytes. Today, ASLR is enabled on nearly all modern operating systems such as Windows, Linux, iOS, or Android. For the most part, current ASLR schemes randomize the base (start) address of segments such as the stack, heap, libraries, and the executable itself. This basic approach is depicted in Figure 8, where the start address of the program executable, its shared libraries, and data segments is relocated between consecutive runs of the application. As a result, an adversary must guess the location of the functions and instruction sequences needed for successful deployment of her code-reuse attack. Hence, we define ASLR as follows.

Address Space Layout Randomization (ASLR): *In order to defend against code-reuse attacks, address space layout randomization randomizes the base address of code and data segments per execution run. Hence, the memory location of code that the adversary attempts to use will reside at a random memory location.*

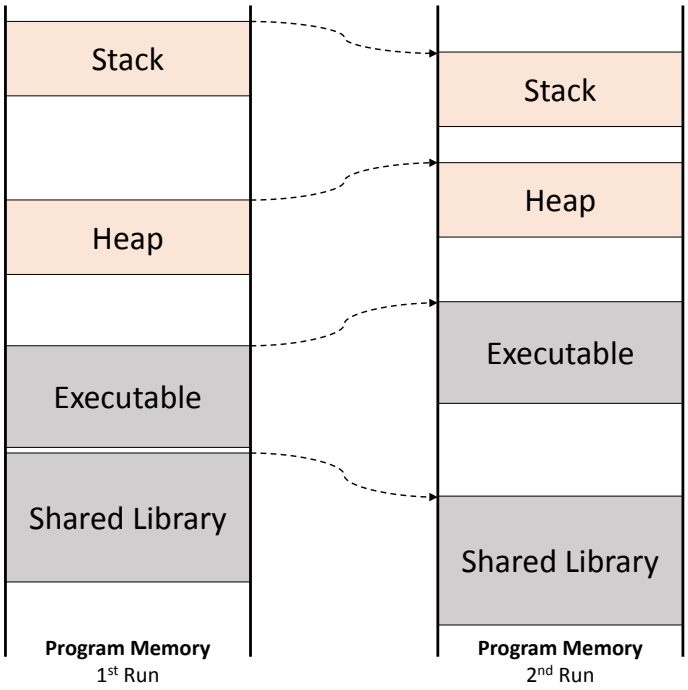


Figure 8: Address space layout randomization (ASLR)

Unfortunately, today's ASLR realizations suffer from two main problems: first, the entropy on 32-bit systems is too low, and thus ASLR can be bypassed by means of brute-force attacks [170, 124]. Second, all ASLR solutions are vulnerable to *memory disclosure* attacks [175, 168] where the adversary gains knowledge of a single runtime address and uses that information to re-enable code reuse in her playbook once again. We remind the reader that this is possible because ASLR only randomizes the base address of the segment meaning that the randomization offset within one segment always remains the same.

For such a memory disclosure attack, consider an application that links to the standard UNIX C library `libc` to invoke the *printf()* function. The adversary's goal is to mount a code-reuse attack using various gadgets from `libc`. At compile-time, the runtime address of *printf()* is not known as ASLR allocates `libc` at a randomized memory location for each run. However, the compiler will add a placeholder for the runtime address of *printf()* into a dedicated data section of the executable that is called global offset table (GOT). At application runtime, the dynamic loader will resolve and allocate the runtime address of *printf()* into the GOT either at load-time of the application or on-demand when *printf()* is called for the first time. As the GOT is readable, an adversary can learn the runtime address of *printf()*. In practice, this can be achieved by using a non-randomized gadget [84], or by exploiting a so-called format string vulnerability which allows arbitrary reads and writes in the address space of an application [86]. Given the runtime address of *printf()*, the adversary can determine the start address of `libc` as the

offset is constant. Once the start address is known, the adversary dynamically adjusts all the pointers inside the return-oriented payload using the constant randomization offset.

2.3 CONTROL-FLOW INTEGRITY (CFI)

In order to defend against code-reuse attacks, code randomization relies on the secrecy of the used randomization offset. In contrast, control-flow integrity (CFI) explicitly enforces that a program's control-flow follows a legitimate path in the application's control-flow graph (CFG) [1, 4].

Figure 9 shows a high-level representation of CFI: first, and prior to program execution, the application's CFG needs to be identified. Next, a CFI check is emitted at the exit instruction of each CFG node. These CFI checks are executed at runtime to prevent any attempt of the adversary to hijack the program's control-flow. For instance, the CFI check at node n_3 validates that the exit instruction only targets either n_5 or n_6. If the adversary aims to redirect execution to n_4, CFI will immediately terminate the program execution.

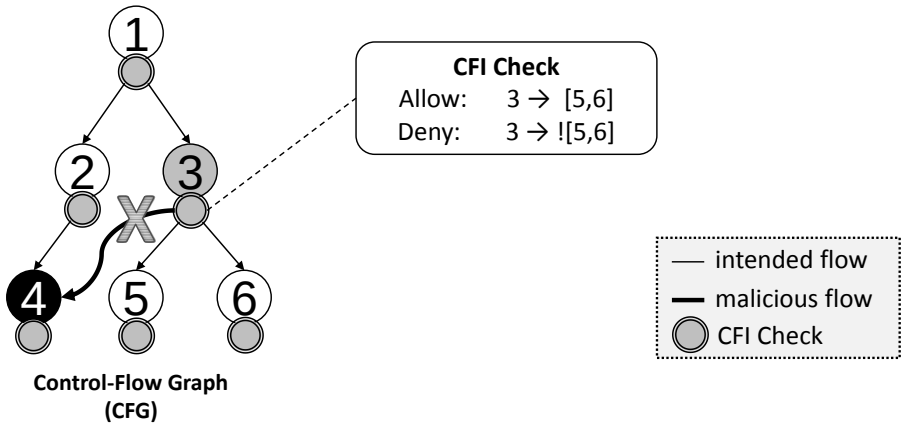


Figure 9: Control-flow integrity (CFI)

In summary, we define CFI as follows.

Control-Flow Integrity (CFI): *CFI offers a generic defense against code-reuse attacks by validating the integrity of a program's control-flow based on a pre-defined control-flow graph (CFG) at runtime.*

In particular, Abadi et al. [1, 4] suggest a label-based CFI approach, where each CFG node is marked with a unique label ID that is placed at the beginning of a BBL. In order to preserve the program's original semantics, the label is either encoded as an offset into a x86 cache prefetch instruction or as simple data word. Inserting labels into a program binary will require moving instructions from their original position. As a consequence, CFI requires adjusting all memory offsets embedded into jump/call and data load/store instructions that are affected by the insertion of the additional prefetch

instructions. Originally, CFI on x86 builds upon the binary instrumentation framework Vulcan which provides algorithms to derive the CFG and a binary rewriting engine to emit labels and CFI checks without breaking the original program-flow.

CFI builds upon several assumptions to effectively defend against code-reuse attacks. Foremost, it assumes that code is not writable, and that an adversary cannot execute injected code from data memory. Both is ensured by enforcing the $W \oplus X$ security model which is enabled by default on modern operating systems (cf. Section 2.1.4). However, this also means that original CFI is not applicable to self-modifying code, or code that is generated just-in-time.

As code is assumed to be immutable, Abadi et al. [1, 4] take an optimization step to increase the efficiency of CFI: they only emit and perform CFI checks for nodes that terminate with an indirect branch instruction. In contrast, direct branch instruction use a statically encoded offset that cannot be altered by an adversary. In the following, we discuss in more detail the general usage scenario for different kinds of indirect branches and how CFI checks are implemented for them.

2.3.1 *CFI for Indirect Jumps*

Typically, indirect jumps are emitted by the compiler for (i) switch-case statements, and (ii) dispatch of subroutine calls to shared libraries. We will describe both usage scenarios before we show how CFI protects this type of indirect branch.

A switch-case statement allows a programmer to execute code based on the content of a variable which is checked against a list of pre-defined values. An example of a switch-case statement that consists of three case branches is shown on the left-hand site of Figure 10. These branches are reached using an indirect jump instruction based on the content of the variable `value` that can hold the numbers 1, 2, or 3.

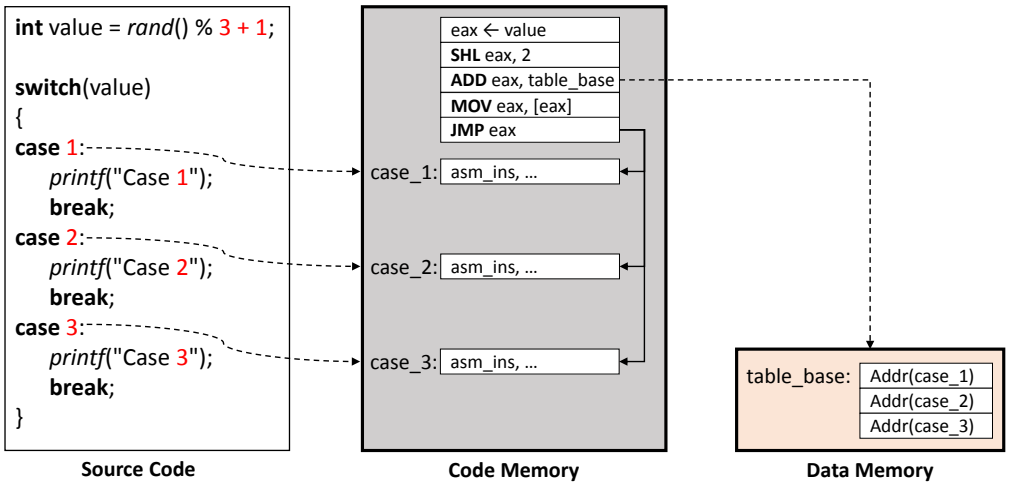


Figure 10: Indirect jumps in switch-case statements

On assembler level, the content of `value` is loaded to register `eax`. The same register is later used in an indirect jump instruction to redirect the control-flow to one of the

case branches. The main idea is to load the correct target address from a dedicated jump table based on the content of `value`. For this, the program performs as follows: first, `eax` is left-shifted via `SHL` by 2 Bits. This is necessary to correctly load the correct target address from the 4-Byte aligned jump table that contains the three possible case branch target addresses. As a result, the base address of the table is added to `eax` after the left shift. Lastly, `eax` is de-referenced to load the target branch address into `eax`. The subsequent indirect jump takes the address stored in `eax` to invoke the correct case statement. Register `eax` can potentially be controlled by an adversary, *e. g.* a buffer overflow is exploited to alter `value`. Hence, it is crucial to perform a CFI check before the indirect jump is executed.

Indirect jumps can also take their target address directly from memory. A prominent example is the dispatch of subroutine calls to shared libraries: consider an application that invokes *printf()* from `libc`. As we explained in Section 2.2, the runtime address of *printf()* will be loaded at a designated memory location in the GOT. The function call to *printf()* in the main application goes through an indirection using the so-called procedure linkage table (PLT). The PLT contains stub code that eventually executes an indirect jump that uses as its target address the runtime address located in the GOT. However, an adversary may exploit a memory-related vulnerability to corrupt the value in the GOT thereby hijacking the intended control-flow.

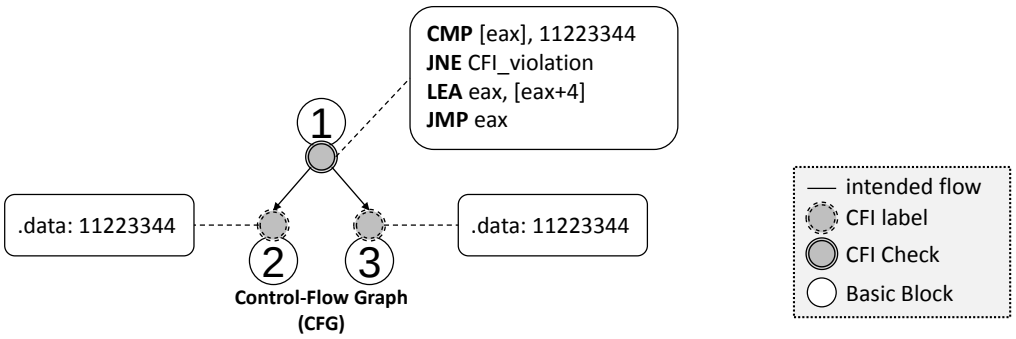


Figure 11: CFI indirect jump check

Figure 11 demonstrates how CFI protects indirect jumps from being exploited. In our example, n_1 invokes an indirect jump based on the value stored in `eax`. Note that n_1 is only allowed to target n_2 and n_3. Hence, CFI needs to emit the unique label 11223344 at the beginning of n_2 and n_3. The exit instruction of n_1 is instrumented in such a way that dedicated CFI code validates whether `eax` targets the label 11223344. Only if this is the case, the program is allowed to take the indirect jump. As the label occupies 4-Byte in memory at n_2 and n_3, it is necessary to update the jump target in `eax` by using the load effective address (`LEA`) instruction with `eax+4` as base offset. This ensures that the indirect jump skips the label.

2.3.2 *CFI for Indirect Calls*

In general, indirect call instructions are emitted by commodity compilers for (i) function calls through function pointers, (ii) callbacks, and (iii) C++ virtual functions. As for indirect jumps, indirect calls on x86 can either take their target address from a processor register, or directly from memory.

In the remainder of this section, we briefly describe the usage and exploitation of indirect call instructions for the invocation of C++ virtual functions. We do so, because C++ virtual table hijacking belongs to one of the most common exploitation techniques to instantiate a runtime attack.

Virtual functions support the concept of polymorphism in object-oriented languages. A prominent example to demonstrate the usefulness of virtual functions is as follows [54]: a base class `shape` contains a generic method *draw()* and declares it as virtual. This allows child classes such as `rectangle` and `oval` to define the same method *draw()* and implement a specific *draw()* method based on the shape's type. In C++, every class that contains virtual functions gets associated to a virtual table (vtable), where the vtable contains pointers to virtual functions. At runtime, a vtable is accessed through a vtable pointer which is stored in the object's data structure. The vtable itself typically resides in read-only memory. Hence, an adversary cannot directly compromise the integrity of the vtable to launch a runtime exploit. On the other hand, the vtable pointer resides in writable memory and is thereby subject to runtime exploits that inject a fake vtable and alter the vtable pointer to point to the injected fake vtable [158]. The next time the compromised program issues a virtual call, it will de-reference the overwritten vtable pointer and redirect the execution to the address stored in the fake vtable.

The so-called use-after-free vulnerabilities are a well-known entry point to instantiate vtable hijacking attacks [5]. The main idea of this attack technique is to exploit a dangling pointer that points to freed memory. In detail the workflow is as follows: a C++ application creates a new object of class `shape`. This leads to the allocation of a pointer that references the data structure of `shape`. In this data structure resides the vtable pointer that points to the vtable of `shape`. At a later time in program execution, the object is deallocated. However, since C++ does not enforce automatic garbage collection, the pointer to the freed `shape` object is still intact. Given a buffer overflow vulnerability, the adversary can inject a fake vtable and overwrite the vtable pointer of the freed `shape` object to point to the fake vtable. Hence, making a virtual call through the dangled pointer of the freed object leads to arbitrary code execution.

Control-flow validation for indirect calls is performed as for indirect jumps: CFI assigns unique labels to valid call targets, and instruments indirect call instructions (as shown for indirect jumps in Figure 11) so that they can only target a valid call site.

2.3.3 *CFI for Function Returns*

As we already described in Section 2.1.5.2, return instructions transfer control to the address located at the top of the stack. Hence, an adversary can potentially overwrite the original return address to perform a code-reuse attack.

Enforcing a label-based CFI approach on return instructions is challenging due to the fact that a single subroutine can be invoked from diverse call sites, or a single indirect function call may target multiple subroutines. The resulting control-flow graph for both cases is shown in Figure 12.

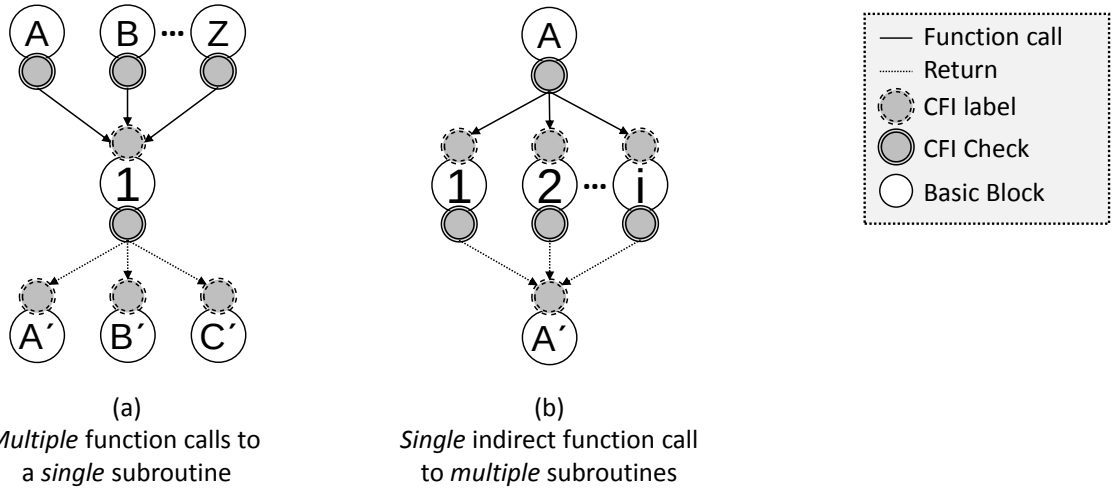


Figure 12: Static label-based CFI checks for function returns

A CFI label-based approach according to the scheme presented by Abadi et al. [1, 4] will eventually lead to coarse-grained protection as a single label needs to be emitted for different call sites. Consider for this the scenario shown in Figure 12-(a): the nodes n_A to n_Z all terminate with a function call instruction that redirects execution to the subroutine at node n_1. The exit of n_1 –implemented as a return instruction– targets the call sites $n_{A'}$ to $n_{Z'}$ depending on which node has called the function, *e. g.* if the function call originated from n_A then the return should target $n_{A'}$. However, since a label-based CFI approach is not aware of the runtime state of the program, it needs to emit a generic label for all call sites $n_{A'}$ to $n_{Z'}$ thereby giving an adversary the possibility to hijack the control-flow, *e. g.* $n_A \rightarrow n_1 \rightarrow n_{Z'}$.

In a similar vein, the scenario shown in Figure12-(b) leads to coarse-grained CFI protection: an indirect function call at node n_A may potentially target many subroutines n_1 to n_i. Obviously, all the function returns at nodes n_1 to n_i need to redirect the control-flow to the call site at node $n_{A'}$. Hence, they all need to check against the same label, namely the label emitted at node $n_{A'}$. If there is any other node in the program that can also target one of the subroutines n_1 to n_i then its corresponding call site needs to be assigned the same label as the one already emitted at $n_{A'}$. In other words, if there is a node n_B that calls n_1 then the call site at node $n_{B'}$ is assigned the same label as $n_{A'}$. Consequently, all calling nodes from n_1 to n_i will be able to target $n_{B'}$ although n_1 is the only node that can be called by n_B.

Hence, a static CFI approach for function returns will probably lead to coarse-grained CFI enforcement where only one generic label is assigned to all call sites [1, 4]. To remedy this situation, Abadi et al. [1, 4] suggest to leverage a shadow stack [48, 81] to allow fine-grained integrity checks for return addresses.

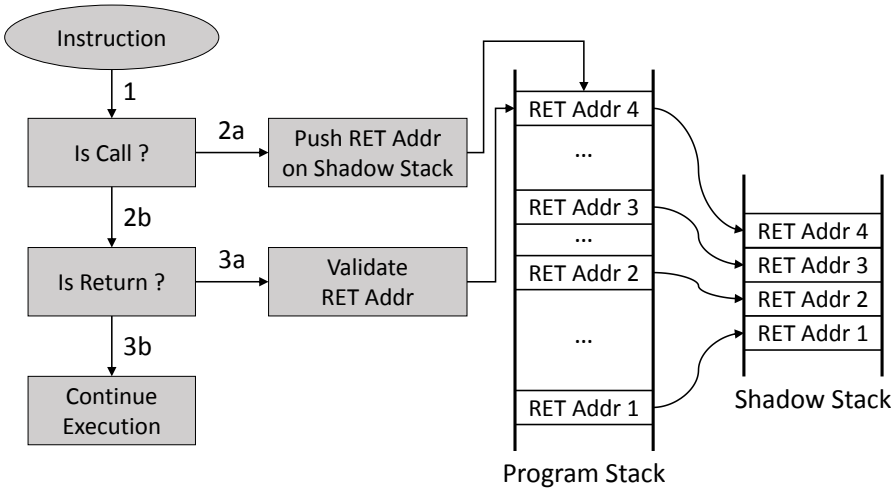


Figure 13: Main principle of shadow stack (return address stack)

Figure 13 depicts the main principle of a shadow stack: at runtime, every call and return instruction is instrumented. Whenever the program issues a call instruction the return address it pushes onto the stack is copied to a dedicated memory area called the shadow stack. Upon function return, we simply need to validate whether the return address the program attempts to use equals the one that is maintained on the shadow stack. This ensures that a return always targets its original caller even when the subroutine is frequently invoked by diverse function calls.

In order to prevent an attacker from tampering with the shadow stack, Abadi et al. [1] leverage memory segmentation which is available on x86-based systems. Alternatively, one could only allow call and return instructions to access the shadow stack using software fault isolation techniques [187].

Although the shadow stack approach allows fine-grained CFI for return instructions, it introduces several practical problems. Foremost, the performance overhead is significant due to the fact that one needs to instrument direct call instructions which occur frequently during program execution. Further, the CFI check needs to load and compare two addresses (one from the program stack and one from the shadow stack). Further, certain programming constructs violate the assumption that a return always needs to target its original caller. Famous examples are setjmp/longjmp, C++ exceptions which rewrite return addresses, or position-independent code that exploits call instructions to locate the current value of the program counter. Lastly, one needs to keep a shadow stack for each execution thread. We refer the interested reader to our previous work on ROPdefender [61], where we leveraged dynamic binary instrumentation to tackle exceptional return cases.

3

ADVANCED CODE-REUSE ATTACKS

In this chapter, we present two advanced code-reuse attacks that undermine control-flow integrity (CFI) based defenses. Our first attack presented in Section 3.1 circumvents defenses that only apply their protection to return instructions. Next, we present in Section 3.2 a novel code-reuse attack which circumvents various well-known coarse-grained CFI defenses (covering kBouncer [150], ROPecker [47], CFI for COTS binaries [204], ROPGuard [83], and Microsoft EMET [131]). Lastly, we thoroughly elaborate on related work on code-reuse attack techniques (Section 3.3), and conclude this chapter in Section 3.4.

3.1 RETURN-ORIENTED PROGRAMMING WITHOUT RETURNS ON ARM

Conventional return-oriented programming attacks are based on exploiting returns, or more generally speaking, on function epilogue sequences. As described in Section 2.1.5.2, return instructions are exploited in a return-oriented programming attack to transfer the control-flow to the subsequent instruction sequence. Hence, defenses that ensure the integrity of return addresses [61, 48, 81, 95, 49] can be deployed to defend against return-oriented programming attacks. In particular, shadow stacks that keep valid copies of return addresses on a separate stack provide fine-grained return address checks (cf. Section 2.3.3). Further, several defenses have emerged that deploy behavioral-based heuristics. For instance, DROP [42] and our tool DynIMA [59] report a return-oriented programming attack when there is an excessive use of return instructions within a short period of time. Further, Li et al. [121] propose a compiler toolchain that eliminates unintended return instructions, and adds a layer of indirection for all intended return instructions to ensure that a return targets an instruction that follows after a function call.

However, Checkoway and Shacham [39] introduced a new code-reuse attack targeting Intel x86-based platforms that requires *no* return instructions. Instruction sequences are instead chained together by indirect jump instructions. This attack cannot be detected in the same way as conventional return-oriented programming attacks since there is no definite convention regarding the target of an indirect jump, *i. e.* return instructions typically redirect execution back to the calling function.

Inspired by the approach introduced in [39], we present in this section a jump-oriented attack method targeting mobile ARM computing platforms. ARM is the standard processor deployed in mobile devices, and more than 50 billion ARM-powered chips have been shipped [159]. Our attack exploits ARM's indirect call instruction *Branch-Load-Exchange (BLX)*. Hence, we call our attack *BLX-Attack*. In contrast to [39], we allow an adversary to exploit an arbitrary processor register to be used as base pointer to jump addresses. This effectively circumvents those protection mechanisms that specifically trace the stack

pointer to detect code-reuse attacks. We instantiated our attack on an Android device allowing us to send unauthorized text messages.

Contributions. We present a jump-based attack method on ARM platforms that bypasses integrity checks for return addresses. Our attack allows us to change program behavior without code injection. Although in principle we adopt the jump-based attack presented in [39], developing such an attack on an ARM platform is not straightforward: due to memory alignment enforced by ARM the code base available to an attacker is significantly smaller compared to x86. Moreover, Checkoway and Shacham [39] requires a special dispatcher gadget based on the so-called BYOPJ (Bring your own pop jump) paradigm. Such a gadget was not available in our target libraries.

To demonstrate the effectiveness of our attack, we mount our BLX-Attack on a Google Android device by exploiting a heap-overflow vulnerability. We show that it is possible to attack an Android application to send unauthorized text messages via SMS.

Section Outline. After providing background information on the ARM architecture in Section 3.1.1, we present in Section 3.1.2 our adversary model and assumptions. We give an overview of our BLX-Attack in Section 3.1.3, and explain the technical details of our gadget set in Section 3.1.4. We show how our BLX-Attack can be mounted on an Android device in Section 3.1.5, and conclude in Section 3.1.6.

3.1.1 *Background on ARM*

ARM is a 32-Bit processor[1] and features 16 general-purpose registers `r0` to `r15` as depicted in Table 1. All these registers can be accessed/changed directly. In contrast to Intel x86, machine instructions are allowed to directly operate on the program counter `pc` (`eip` on x86). Additionally, ARM processors feature a current program status register (`cpsr`), which holds the current state of the system. It contains condition flags, interrupt enable flags, and the current mode.

Register	Purpose
`r0 - r3`	Arguments into function; function results
`r4 - r11`	Register variables (must be preserved)
`r12`	Scratch register (used for long jumps)
`r13 (sp)`	Stack pointer
`r14 (lr)`	Link register (for return address)
`r15 (pc)`	Program counter
`cpsr`	Control program status register

Table 1: ARM registers

1 Recently, ARM also launched a 64-Bit version for its Cortex-A50 Series [15], which is deployed in Apple's A7 processor for iPhone 5s.

In general, ARM follows the *Reduced Instruction Set Computer* (RISC) design philosophy, *e. g.* it features dedicated load and store instructions, enforces aligned memory access, and offers instructions with a fixed length of 32 bits. However, since the introduction of the ARM7TDMI microprocessor, ARM provides a second instruction set called THUMB which usually has 16 bit instructions. The THUMB instruction set is a subset of the ARM instruction set and is in particular suitable for embedded systems which often suffer from greater memory restrictions as PCs. Moreover, THUMB code provides better performance than ARM for systems shipped with a 16-bit memory. If instructions have to be fetched from a 16-bit memory then it will take two cycles to fetch an ARM instruction, whereas only one cycle is needed to fetch a THUMB instruction. In particular, the Android libraries *libc.so* and *libwebcore.so* which we use as the code base for our BLX-Attack contain mainly THUMB instructions.

Calling Convention. The ARM Architecture Procedure Call Standard (AAPCS) [14] suggests that function calls should be performed either through a `BL` or through a `BLX` instruction. The `BL` instruction performs a branch-with-link operation, *i. e.* it enforces a branch to the specified routine by writing the destination address to the program counter `pc`, and by writing the return address to the link register `lr`. The `BLX` instruction additionally allows interworking between ARM and THUMB code. Further, only the `BLX` instruction allows indirect function calls (*i. e.* the target address of the branch is hold in a register). Note that, in practice, not all function calls follow the AAPCS calling convention: Instead of transferring the return address to `lr`, the ARM C compiler may enforce the return address to be pushed onto the stack and afterwards performs a direct branch to the function through a `B` or `BX` instruction.

Arguments to a function are provided in the registers `r0` to `r3`. If a function requires more than four arguments then these must be passed on the stack. Additionally, the output values of a function are returned via these registers. Registers `r4` to `r8`, `r10`, and `r11` are used for holding local variables of the called function, but THUMB-compiled code usually uses only `r4` to `r8`. According to the AAPCS, a function must preserve the callee-save registers `r4` to `r8`, `r10`, `r11`, and `sp`. ARM also supports a base pointer register to facilitate access to local variables: `r11` in ARM mode, and `r7` in THUMB mode.

A function return is completed by writing the return address to the program counter `pc`. For this, the ARM architecture provides no dedicated return instruction. Instead, any instruction that is able to write to the program counter can be applied as return instruction. For instance, one common return instruction is the `BX lr` instruction that branches to the address stored in the link register `lr`. Further, it is also possible to use the `LDM` (load multiple) or `POP` instructions that load the return address from the stack, *e. g.* `POP r4-r7,pc` loads `r4` to `r7` and the program counter with new values from the stack.

3.1.2 *Assumptions*

We define a strong adversary model. For our attack, we assume the availability of standard protection mechanisms against code injection and return address corruption attacks.

1. We assume that the target platform enforces the $W \oplus X$ security model (cf. Section 2.1.4). Thus, an adversary cannot use well-known code injection attacks. This is reasonable because the ARM architecture provides the XN bit (*i. e.* similar to Intel's non-executable bit) which facilitates the enforcement of $W \oplus X$. The new generations of Apple's smartphone iPhone make use of the XN bit for each memory page [104]. At the time of developing our jump-based attacks, Android did not yet leverage the XN bit, thus allowing code injection attacks. However, we assumed a stronger Android architecture, and in fact, since Android version 4, $W \oplus X$ is enabled by default.

2. We assume that the target platform deploys countermeasures to defend/detect conventional return-oriented programming attacks, *e. g.* by using [61, 48, 81]. We believe that fine-grained return address checks that were implemented for the Intel x86 architecture can be adopted to ARM architectures and the ARM C compiler. In fact, as we will show later in this dissertation, we realized and evaluated a shadow stack on ARM (cf. Section 4.1).

3. We assume that the target platform provides an application with some bug allowing to instantiate a heap-based buffer overflow attack. The reason for instantiating our attack by means of a heap overflow is that we want to avoid the use of any return instruction, so that our attack circumvents integrity checks for return instructions. This is reasonable since attackers have moved to heap-based buffer overflow exploitation since several years [153].

3.1.3 *Overview on BLX-Attack*

In this section, we present the high-level idea of our BLX-Attack method. First, we describe the main aspects of the ARM `BLX` instruction and how this instruction can be exploited for our attack. Next, we present the general design of the BLX-Attack such as the memory layout and the main attack steps.

3.1.3.1 *Attack Components*

The `BLX` instruction stands for *Branch-Load-Exchange* and is usually used for indirect function calls. A *branch* is enforced to jump to an address stored in a particular register, while the return address is *loaded* into the link register `lr`, and (if necessary) an instruction set *exchange* from ARM to THUMB and vice versa can be enforced. In the following, we will show how indirect branch instructions such as `BLX` can be exploited to launch a code-reuse attack.

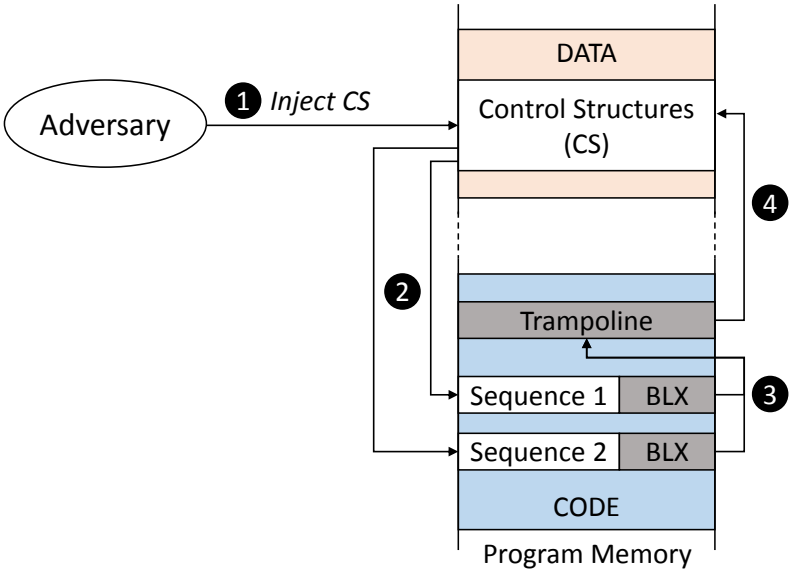


Figure 14: Basic layout of BLX-Attack on ARM

The principle of the BLX-Attack method is depicted in Figure 14. It shows an abstract view of a program's memory. The adversary cannot inject own malicious code due to enabled $W \oplus X$ protection (cf. Assumption 1). However, an adversary is still able to reuse existing code of the target program and its libraries. To do so, the adversary corrupts the control structure (CS) section thereby maliciously redirecting program execution to a unintended piece of code in the code section. Usually, control structures (such as return and jump addresses) are located on the program's stack or on the heap. The instruction sequence of the linked library is executed until an indirect branch instruction has been reached which redirects the execution to the next sequence of instructions by using a *trampoline*. The trampoline is also part of the code section and is responsible for loading the address of the subsequent instruction sequence from the control structure (CS) section.

In contrast to a conventional return-oriented programming attack (see, *e. g.* [169]), our BLX-Attack does not use the return instruction as connector for the instruction sequences. Instead, it exploits ARM's indirect call instruction `BLX`.

Checkoway and Shacham [39] already demonstrated that indirect jump instructions can be exploited to circumvent control-flow checks that solely target return instructions. However, the attack presented in [39] targets Intel x86 and cannot be applied straightforward to ARM-based systems. The code base available to an attacker on ARM is significantly smaller compared to Intel x86. Recall that Intel x86 code provides a large code base for return-oriented attacks due to the presence of many unintended instruction sequences (cf. Section 2.1.5.2). In our work, we exclusively focus on original and intended instruction sequences. Nevertheless, a recent work by Lian et al. [122] demonstrates that the interworking of 2-Byte/4-Byte THUMB instructions with 4-Byte ARM instructions can be exploited to execute unintended instructions sequences.

Moreover, the attack on Intel x86 requires a POP-JMP sequence to realize the trampoline in Figure 14. However, such a sequence is rarely available in modern programs. Hence, Checkoway and Shacham [39] specified the "*Bring Your Own Pop Jump (BYOPJ)*" paradigm which requires the target program or one of its libraries to include a POP-JMP. However, for the typical libraries used on ARM such a trampoline sequence does not exist. Although typical libraries on ARM do not include POP-JMP sequences[2], we show how to design a Turing-complete attack method for ARM platforms *without* requiring the BYOPJ paradigm.

Reasons for using BLX. The BLX instruction is typically not a part of a function epilogue. Instead, it is typically leveraged as function call instruction inside the function body. Hence, an attack based on BLX instructions evades detection from defenses that protect return instructions. Moreover, in contrast to Intel's x86 indirect call instruction, the BLX instruction does not impact values on the stack (or generally on the memory), which makes the BLX instruction very suitable for our attack. However, since the program counter pc can be accessed as a general purpose register, any instruction that uses the program counter pc as a destination register could also be exploited for our attack. We selected the BLX instruction because most of the instruction sequences we identified in our code base end with BLX.

For extraction of a Turing-complete gadget set we inspected *libc.so* and *libwebcore.so* libraries of an Android 2.0 platform (the most recent version at the time we conducted this project). Android's libc version is very compact, hence, we included Android's Web Browser library *libwebcore.so* to enlarge the code base. On Android 2.0, both libraries are linked by default into the address space of an application to fixed addresses. That said, no code randomization (cf. Section 2.2) is applied to these libraries.

3.1.3.2 *Attack Method Design*

In the following, we present the memory layout and each attack step of our BLX-Attack.

Memory Layout. Figure 15 depicts the memory layout and the steps of our BLX-Attack. The memory area under control of the adversary contains jump addresses and arguments which are clearly separated from each other. Each jump address points to a specific instruction sequence, where each sequence ends with a BLX instruction in order to allow chaining of multiple sequences. We misuse the stack pointer R_A as a pointer to arguments and need a second register (denoted with R_{JA}) as a pointer to jump addresses. The order of jump addresses and arguments highly depends on the appropriate instruction sequences found on a platform. For instance, if the instruction sequence which updates R_{JA} adds a positive constant then jump addresses have to go from lower to higher memory addresses. In Figure 15, jump addresses go from lower to higher memory addresses and arguments are ordered vice versa. Of course, if jump addresses are not separated from arguments then one register could be saved. This is actually the

2 Sometimes such a sequence can be found in a function epilogue. However, these sequences can be protected by enforcing integrity checks for return instructions.

preferred way proposed by Checkoway and Shacham [39]. However, on Intel x86, arguments are mainly loaded by a POP instruction from the stack which directly updates the stack pointer. Unfortunately, the typical libraries we examined load arguments without updating the stack pointer. That is the reason why we use R_{JA} as pointer to jump addresses which is updated after each instruction sequence.

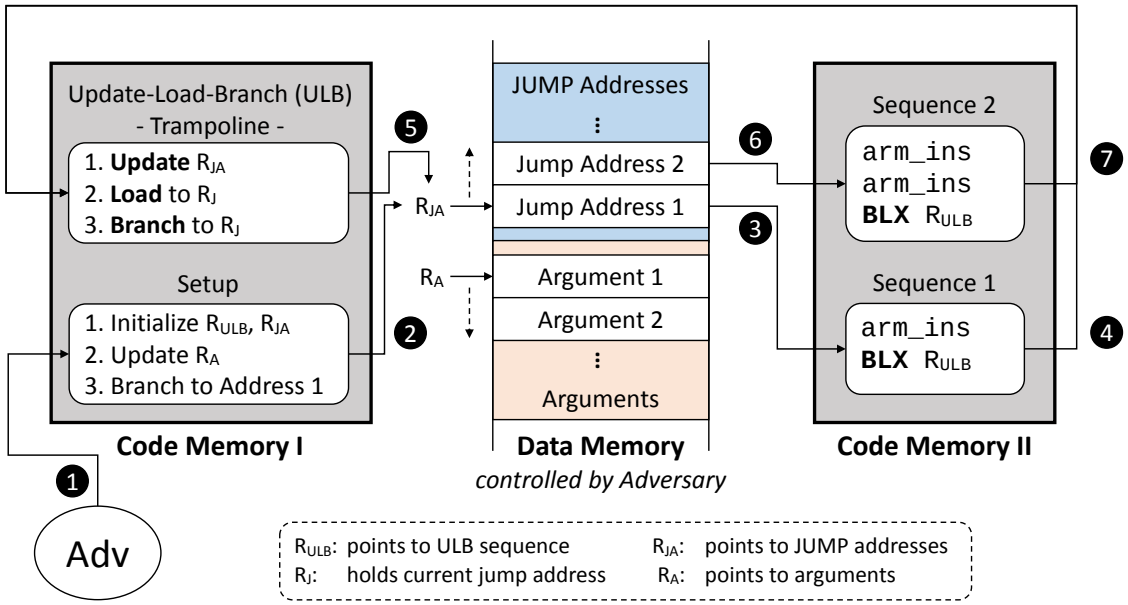


Figure 15: BLX-Attack method

BLX-Attack Steps. First, the adversary injects jump addresses and arguments to the stack or the heap (cf. Section 3.1.5 for a concrete example). Our attack method consists mainly of three parts: (i) **setup**, (ii) **Update-Load-Branch (ULB) sequence**, and (iii) **gadgets** which consist of several instruction sequences. By subverting the control-flow, the adversary is able to initialize several registers. We refer to this process as a *setup* (Step ❶). The setup initializes three registers: R_{JA}, R_{ULB} and R_A. R_{JA} and R_A are used as pointers to jump addresses and arguments, respectively. Register R_{ULB} is loaded with the address of our ULB sequence (see below). Finally, the last action of our setup phase is to redirect execution to Sequence 1 (Step ❷ and ❸ in Figure 15). After Sequence 1 completes its task, the BLX instruction (located at the end of the sequence) redirects execution to our ULB sequence using register R_{ULB} (Step ❹). The ULB sequence is responsible for *updating* register R_{JA}, *loading* the jump address R_{JA} points to (here Sequence 2) into the free register R_J, and *branching* to Sequence 2 by using register R_J (Step ❺ and ❻). That is, our ULB sequence is the connecting link (the trampoline in Figure 14) for all sequences of instructions the adversary aims to invoke. In fact, Sequence 2 terminates in a BLX instruction that transfers control back to our ULB sequence (Step ❼) allowing the adversary to call the next sequence.

Principally, the instruction sequences themselves could enforce the branch to the next sequence. However, most of them do not contain instructions for loading the next jump address. Hence, we select to load the next jump address by using our ULB sequence.

3.1.4 *Turing-Complete Gadget Set*

In this section, we present the Turing-complete gadget set for our BLX-Attack allowing an adversary to generate arbitrary malicious program behavior. The gadgets range from simple gadgets that load a value into a register up to sophisticated gadgets that enforce conditional branching.

In general, gadgets consist of several instruction sequences. For our purposes the instruction sequences have to end with a `BLX` instruction to redirect execution to our ULB sequence. Thus, useful instruction sequences must be first extracted form libraries linked to an application. Previous work [31, 102] has shown how to automate the identification of gadgets.

A Turing-complete gadget set for a BLX-Attack should at least consists of gadgets for (i) **memory operations** (load/store), (ii) **data processing** (data moving and arithmetic/logical operations), (iii) **control-flow** (conditional/unconditional branching), and (iv) **system and function calls**. We could construct all these gadgets using the sequences in our code base, namely the libraries *libwebcore.so* and *libc.so* of an Android 2.0 device. In the following, we will present the technical details for all classes of gadgets.

3.1.4.1 *Details of Setup and ULB Sequence*

First, we describe the details of our setup and the ULB sequence which are necessary to successfully initiate and execute our attack. Since our concrete BLX-Attack directly initializes register `r4` to `r15` by exploiting a setjmp buffer overflow vulnerability on the heap [51], we assume for the moment that the adversary can directly initialize these registers.

In Section 3.1.3, we introduced the registers R_{JA}, R_{ULB}, and R_A as the fundamental basis for our attack. The allocation of these registers highly depends on the identified instruction sequences in our code base and involves technical challenges because these registers must be preserved during the execution of the gadget chain. For our code base, we decided for the following allocation: R_{JA} = `r6`, R_{ULB} = `r3`, and R_A = `sp`. We use the stack pointer (`sp`) for R_A because many sequences we identified in our code base contain load/store operations, where `sp` is used as base register. However, our attack does not force the adversary to control the stack pointer. Instead, any register (R_A) can be used as pointer to arguments and data.

Further, we use the following sequences for the setup and the ULB sequence:

```
LDR  r3,[sp,#0]; BLX r3    /* Setup sequence */
ADDS r6,#4; LDR r5,[r6,#124]; BLX r5  /* ULB sequence */
```

We use `r3` for R_{ULB} because most of the sequences in our code base end with a `BLX r3` instruction. Our setup sequence initializes `r3` (*i. e.* R_{ULB}) by loading the address of the ULB sequence from the stack through a `LDR` (load register) instruction. We describe the

role of the LDR instruction in more detail in Section 3.1.4. Note, since our adversary is able to directly initialize r4 to r15 by the setjmp vulnerability, we require no additional setup sequences for R_{JA} and R_A.

Recall that the ULB sequence acts as connector for all executed instruction sequences by updating R_{JA} after each sequence. Since registers r0 to r3 are often used as destination registers before a BLX instruction, we decided to use r6 as R_{JA} register. The ULB sequence first increases register r6 by 4 Bytes (Update). Afterwards, it loads the next jump address (by an offset of 124 Bytes to r6) into r5 (Load). That said, our ULB sequence leverages r5 for R_J. Finally, we branch to the loaded address (Branch).

In summary, we use r3, r5, r6, and sp as basic registers to ensure execution of our ULB sequence and actual return-oriented gadgets. Hence, registers r0 to r2 as well as r7 to r13 (in total ten registers) can be freely used by any of our invoked instruction sequences to perform exploit operations.

One technical problem we have to address stems from the fact that most of our sequences use the pre-indexed addressing mode meaning that sp does not change its value after it is used as base register in a load operation. It would be desirable to directly load sp as typically done in stack pivot sequences (cf. Section 2.1.5.2). Unfortunately, we have no such load operation in the sequences of our code base. Hence, we use the following sequence to update sp:

```
SUB sp,#12; ADDS r0,r4,#0; BLX r3    /* Updating sp */
```

This sequence decreases the value of the stack pointer by 12 Bytes and as a side-effect overwrites the value of register r0 with the content stored in r4. To preserve register r0, its value could be stored to memory or moved to a free register before.

3.1.4.2 *Memory Operations*

Memory operation gadgets are needed for loading and storing values from and to memory. Due to the RISC architecture of ARM processors load and store operations are only permitted through dedicated load and store instructions. The ARM instruction set offers for this two instructions, LDR and STR.[3] A general-purpose register can be loaded through a LDR instruction. Storing a register to memory is performed through the STR instruction. For instance, to load a word from the stack (with zero Bytes offset) to r1, the following sequence could be used:

```
LDR r1,[sp,#0]; BLX r3
```

Loading an Immediate. Typically, memory operations also include a gadget that loads an immediate value into a general-purpose register. For instance, to load NULL into register r2 the following sequence could be used:

```
MOVS r2,#0; BLX r3
```

3 Despite these two instructions, ARM provides the LDM and STM instructions for a multiple load and store operation.

Storing to Memory. For a store operation, we need at least two registers: one holding the word to be stored, and one holding the target address. Figure 16 depicts our store gadget which stores the content of several registers (`r1`, `r3`, and `r4`) to a memory address pointed to by `r2`.

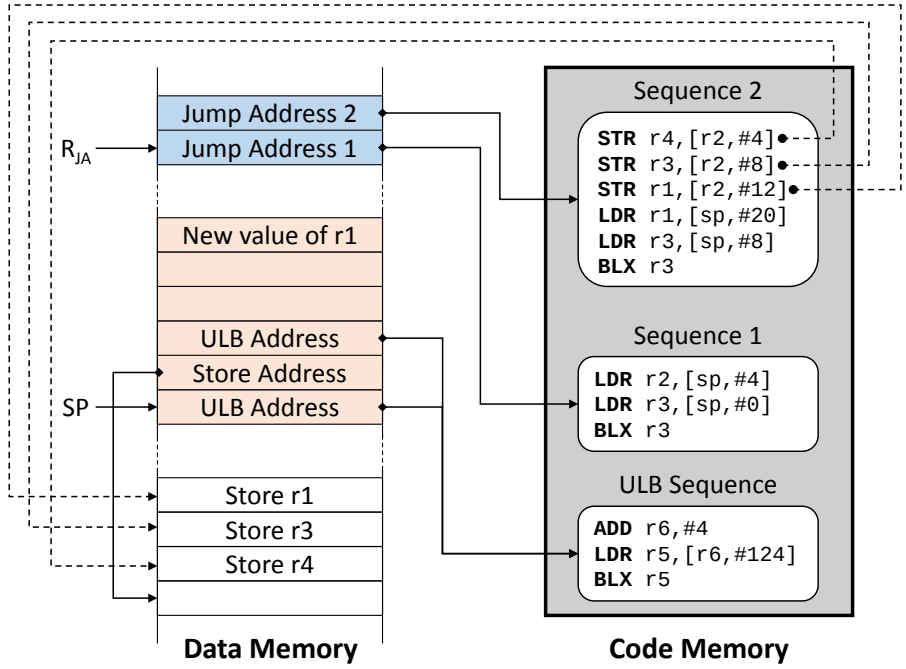


Figure 16: Store gadget

Sequence 1 consists of two load instructions. The first one loads the target address for the store operation to register `r2`. The target address is located in the argument memory space at `[sp,#4]`. Unfortunately, the second load instruction overwrites register `r3` (R_{ULB}). Therefore, the address stored at `[sp,#0]` must be the address of our ULB sequence to preserve R_{ULB}. Afterwards, Sequence 2 stores the registers `r1`, `r3`, and `r4` to the memory area pointed to by `r2`. However, Sequence 2 once again overwrites register `r3`. Hence, the address of our ULB sequence must also be placed at address `[sp,#8]`. In addition, register `r1` is assigned a new value located at `[sp,#20]`.

3.1.4.3 *Data Processing*

Data processing gadgets include gadgets for moving data among registers, logical (AND, OR, NOT, EOR), and arithmetic (ADD, SUB, MUL, DIV) operations. Basically, data processing gadgets need first memory load gadgets to initialize the source registers. Afterwards, the desired operation is performed on the source registers.

Data Movement Gadgets. THUMB compiled code uses for data movement the arithmetic add instruction `ADDS`[4], where the second operand is simply set to zero:

4 An add instruction with the "S" suffix updates also the CPSR flag register.

```
ADDS r0,r1,#0; ADDS r1,r4,#0; BLX r3
ADDS r5,r1,#0; ADDS r7,r2,#0; BLX r3
```

For instance, the first sequence moves `r1` to `r0` and `r4` to `r1`.

Arithmetic Gadgets. The ADD gadget is realized with the above mentioned arithmetic add instruction `ADDS`:

```
ADDS r0,r0,r2; BLX r3
```

This sequence adds the register `r0` and `r2`, and stores the result in register `r0`.

Our SUB gadget is based on the arithmetic sub instruction `SUBS` as depicted in Figure 17. This gadget subtracts `r0` from `r4`. Sequence 1 and 2 load the first operand into `r4` through `r0`. Note that the conditional branch in Sequence 2 is never taken, because `r3` holds the address of the ULB sequence (which does obviously not equal to zero). Next, Sequence 3 loads `r0` with the second operand. The fourth sequence loads into register `r2` the address where the result of the subtraction will be stored. Lastly, Sequence 5 performs the subtraction and stores the result at memory position `[sp,#32]` and in register `r1`.

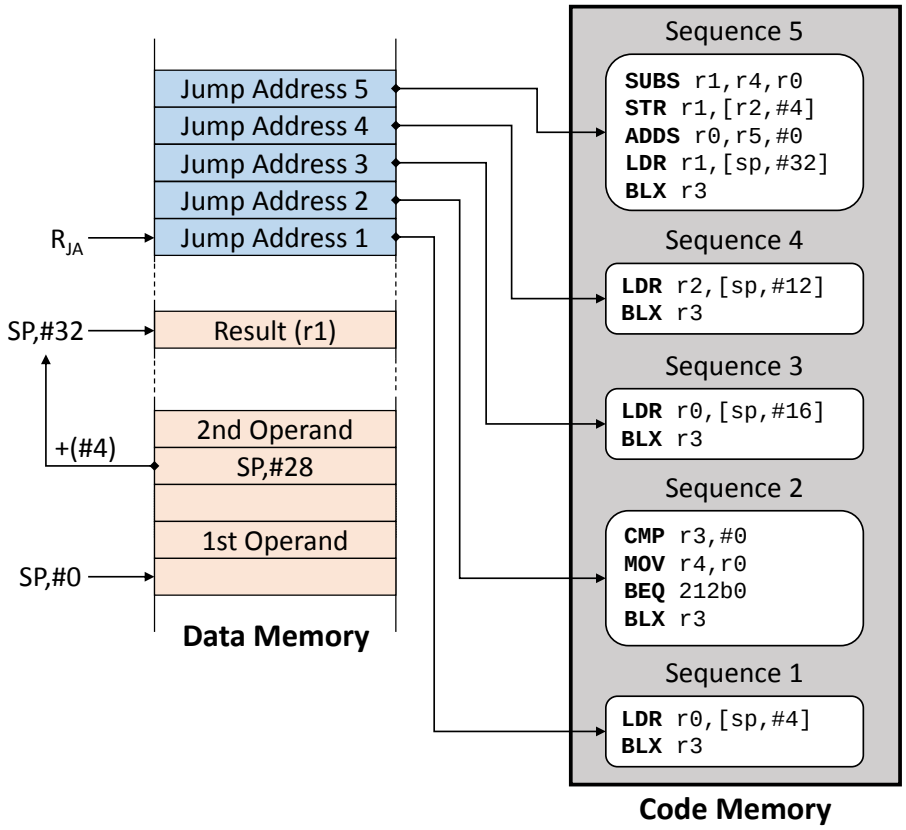


Figure 17: Subtract gadget

The remaining MUL and DIV gadgets can be realized by invoking the ADD and SUB gadget in a loop.

Logical Gadgets. As an example for a logical operation gadget, we present the AND gadget. In general, logical and arithmetic operation gadgets must first load the operands into source registers. Subsequently, the desired logical/arithmetic operation is performed on the loaded registers. Our AND gadget is depicted in Figure 18.

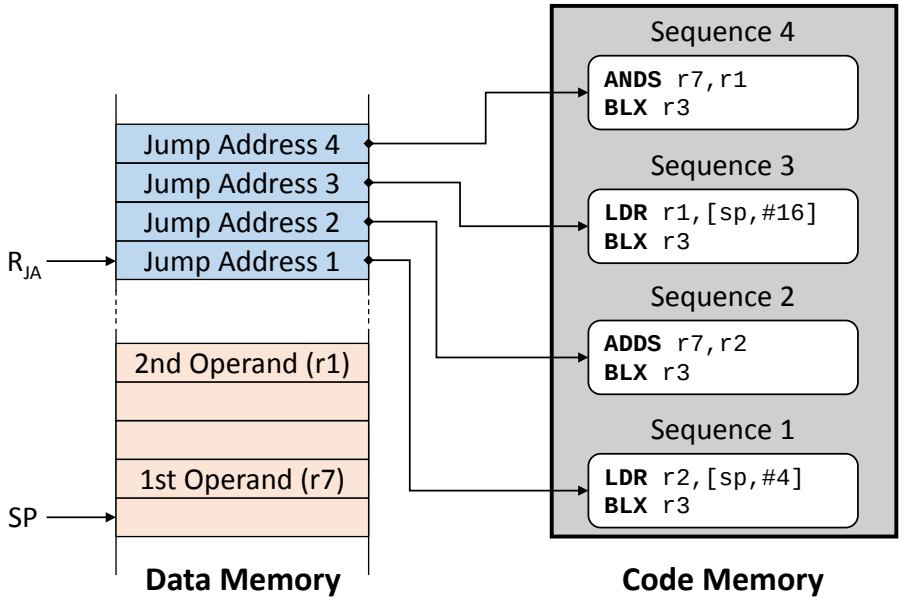


Figure 18: AND gadget

Sequence 1 and 2 are responsible for loading the first operand into register `r7`. Afterwards, Sequence 3 loads the second operand into register `r1` and Sequence 4 performs the AND operation on register `r1` and `r7`. Lastly, the result is stored into register `r7`.

One important logical gadget to mention is the NOT gadget that computes the two's complement of a specific value. We realize the NOT (based on the ideas presented in [39]) by subtracting the source register (through a SUB gadget) from (-1). The AND and NOT gadget can be combined to a NAND gadget. All other logical operations (such as OR, EOR) can be emulated through our NAND gadget.

Similarly, the negate gadget can be simulated through a subtract gadget by subtracting the source register from NULL.

Shift Gadgets. Although shift gadgets are not always included in Turing-complete gadget sets (*e. g.* [169]), we show how these can be realized by the `ASRS` (arithmetic right shift) and `LSRS` (logical left shift) instructions, as follows:

```
ASRS r0,r0,#1; ADDS r0,r2,r0; BLX r3
LSLS r2,r2,#2; ADDS r2,r1,r2; BLX r3
```

For instance, the first sequence performs an arithmetic right shift on `r0` by one bit. To preserve the result of the shift operation, `r2` has to be loaded with NULL (*e. g.* by the load immediate gadget explained in Section 3.1.4.2). Otherwise, the second instruction would overwrite `r0` by adding `r2` to `r0`.

3.1.4.4 *Control-Flow*

In contrast to ordinary programs, branching in the context of our BLX-Attack implicates changing the R_{JA} (`r6`) register rather than the instruction pointer. The unconditional branching gadget can be realized by adding an offset to register R_{JA}, or by directly loading R_{JA} with a new value.

Our conditional branching gadget is based on the ideas presented in [169]: We compare two values and depending on the result, R_{JA} is either changed by an unconditional branch gadget or remains as before. To realize this gadget, we need a compare operation. This can be simulated through a SUB gadget updating the carry flag in the `cpsr` register. The updated carry bit is subsequently added to the constant `0xFFFFFFFF`. Hence, the result will be either NULL or `0xFFFFFFFF`. Finally, the result must be ANDed with the desired branch offset. The result of this last operation will be either NULL (Carry Bit = 1) or the offset (Carry Bit = 0), which is finally added to R_{JA}.

3.1.4.5 *System and Function Calls*

System calls are highly important for runtime exploits. They allow an attacker to invoke special services of the operating system (like opening a file or executing a new program). System calls are typically implemented as subroutines in `libc`. Thus, a program only needs to invoke the appropriate function for the system call. A common alternative to this scheme consists of passing arguments in registers and in storing the system call number in a dedicated register (*e. g.* on ARM `r7`, and on Intel `eax`). The system call is then invoked through a software interrupt (*e. g.* on ARM `SVC 0x0` (Supervisor Call), and on Intel `INT 0x80`).

The `libc` version of the Android OS implements system calls by transferring the system call number to `r7`. Hence, all system call functions only differ in the `MOVS r7,#SYS_NR` instruction. For instance, the *execve* function is implemented as follows:

```
PUSH {r4, r7};
MOV r7,#11; oxb
SVC 0x00000000
POP {r4,r7}
MOVS r0,r0
BXPL lr
```

Hence, we invoke a system call by calling its appropriate `libc` function. It is noteworthy to mention that our system call gadget can be leveraged for calling any other function.

The memory layout and implementation of our system call gadget is depicted in Figure 19. We have to take into account that the `BLX` instruction loads the return address into the link register `lr`. Since the `BXPL lr` (located at the end of the *execve* function) redirects execution back to the value stored in the link register, we have to ensure that `lr` points at that time to a valid instruction sequence. However, when the `BLX` instruction is invoked, `lr` will be automatically loaded with the address of `[pc,#2]` (for Thumb compiled code). Hence, we use an instruction sequence with two `BLX` instructions (Sequence 1).

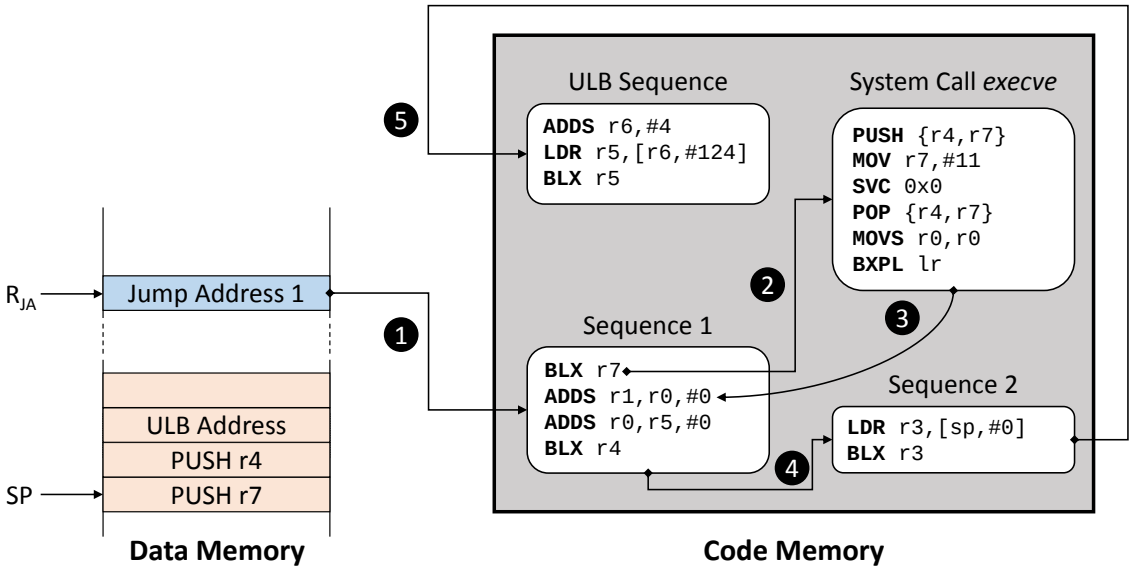


Figure 19: System call gadget

The arguments for the system call must be initialized by load gadgets (not depicted in Figure 19). Usually, registers `r0-r3` hold arguments for a system call. If a system call expects an argument in `r3` then our R_{ULB} will be overwritten. Thus, we must temporarily change the R_{ULB} to a different register if `r3` is used as argument.

First, Sequence 1 invokes the system call function (Step ❶ and ❷), where the address of the system call function is stored in `r7`. After the system call returns, the `BXPL lr`[5] instruction redirects execution back to Sequence 1 (Step ❸). Next, Sequence 1 performs two data movement instructions and then redirects execution to Sequence 2 (Step ❹). This sequence re-initializes our R_{ULB} register `r3` with the address of the ULB sequence. Finally, Sequence 2 redirects execution to the ULB sequence which loads the next jump address (Step ❺).

As can be seen in Figure 19, the system call function pushes two values onto the stack. Since we separated arguments from jump addresses, push instructions are not as critical as they are in the original return-oriented programming attack[6] [169]. However, a push instruction could overwrite arguments pointed to by the stack pointer. If this is the case, the adversary has to use our store and load gadgets to backup the two arguments and to restore them after the system call returns.

5 The condition flag PL means that the branch will only be executed if the N flag in the `cpsr` register is not set. The N flag will be set if ro holds a negative value. This will only be the case if an error occurred during the system call.

6 If return addresses and arguments are both located on the stack, a push instruction may overwrite a return address.

3.1.5 *Proof-of-Concept Exploit on Android*

To demonstrate the effectiveness of our BLX-attack, we exploited an own developed proof-of-concept application on Android. Typically, Android applications are written in Java, but can also access C/C++ libraries via the Java Native Interface (JNI). Application developers may use JNI to incorporate C/C++ libraries into their applications. Moreover, many C libraries are mapped by default to fixed memory addresses in the program's memory space. This provides a large C/C++ code base that we exploit for our attack. In particular, we successfully launched our attack on Android's device emulator (Android version 2.0) as well as on the Android Dev Phone 2 Android version 1.6.

3.1.5.1 *Vulnerable Application*

Our vulnerable application is a standard Android Java application which uses the Java native interface (JNI) to include our C/C++ code. The included C/C++ code is shown in the listing below and is mainly based on the ideas presented in [39].

```
struct foo
{
  char buffer[460];
4 jmp_buf jb;
};
jint Java_com_example_hellojni_HelloJni_doMapFile (JNIEnv* env, jobject thiz)
{
  // open binary file
9       sFile = fopen("/data/local/binary", "r");
  ...
  struct foo *f = malloc(sizeof * f);
  i = setjmp(f->jb);
  if (i!=0) return 0;
14 fgets (f->buffer, sb.st_size, sFile);
  longjmp (f->jb,2);
}
```

The application suffers from a so-called setjmp vulnerability [51]. In general, setjmp and longjmp are system calls which allow non-local control transfers. For this, setjmp creates a special data structure (referred to as `jmp_buf`). The register values from `r4` to `r15` are stored in `jmp_buf` once setjmp has been invoked. When longjmp is called, registers `r4` to `r15` are restored to the values stored in the `jmp_buf` structure. If the adversary is able to overwrite the `jmp_buf` structure before longjmp is called then control can be transferred to arbitrary code sequences without corrupting a single return address.

In Line 14, the *fgets()* function inserts data provided by a file, called `binary`, into a buffer (located in the structure `foo`) without checking the bounds of the buffer. Note that the structure `foo` also contains the `jmp_buf` structure. Hence, if the binary is larger than 460 Bytes it will overwrite the contents of the adjacent `jmp_buf` structure.

However, while reverse-engineering our Android application, we recognized that Android enables heap protection for setjmp by storing a fixed canary directly after the local buffer and lets the `jmp_buf` structure start 52 Bytes after that canary. The canary is hard-coded into `libc`. Thus, it is device and process-independent. As a consequence, we have

to take into account the value of the canary and add 52 Bytes space between the canary and the `jmp_buf`.

3.1.5.2 *Exploitation*

In order to mount a BLX-Attack against the vulnerable program, we aim at subverting the intended execution-flow of the program, and invoke a gadget chain to execute a malicious Tcl script. This script sends text messages to an adversary-chosen number (potentially, a high-premium rate number). At C source code level, we desire to execute the following *execve* command:

```
execve ("/data/data/com.example.hellojni/Send_SMS", NULL, NULL)
```

Specifically, this command launches the malicious `SEND_SMS` binary, which embeds our target Tcl script to send text messages. For the purpose of our BLX-Attack, we need to convert the *execve* command to assembler instructions. Figure 20 shows the corresponding gadgets and the memory layout of our attack.

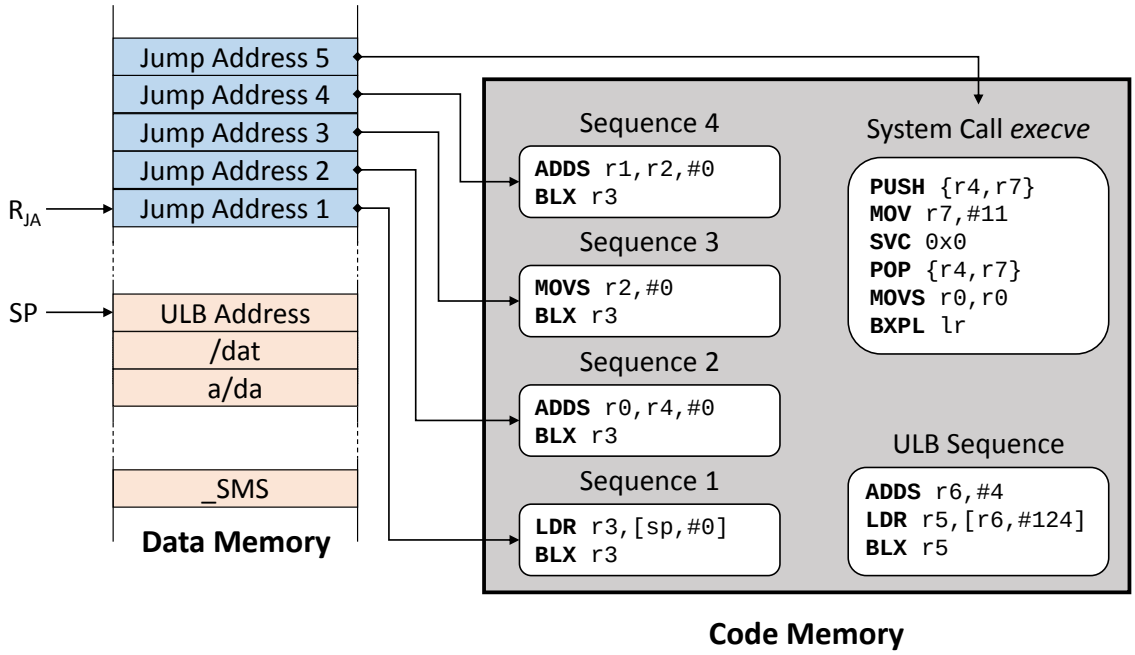


Figure 20: Gadgets used in our BLX-Attack on Android

The attack chain is as follows:

1. Exploit setjmp vulnerability to initialize register `r6` and `sp` for our ULB sequence.
2. Load `r3` with the address of our ULB sequence (Sequence 1)
3. Load in `r0` the address of the string `"/data/data/com.exploit.example/Send_SMS"` (Sequence 2)
4. Load `r1` and `r2` with NULL (Sequence 3 and 4)

5. Invoke the *execve* system call (Sequence 5)

This gadget chain successfully exploited our vulnerable application and sends unauthorized text messages.

Remark. In fact, our attack launches a privilege escalation attack on Android since the vulnerable application has no permission to send any text messages. For this, we exploited the Android scripting environment (ASE) application as a confused deputy to perform privileged operations on behalf of our gadget chain. More details on the nature of the privilege escalation attack we have identified can be found in [60].

3.1.6 *Summary and Conclusion*

Our BLX-Attack method demonstrates that Turing-complete return-oriented programming attacks can be leveraged on ARM-based mobile devices *without* using any return instructions. This allows our attack to evade detection from shadow stack based defenses. As a result, defenses against return-oriented programming need to enforce fine-grained protection for both function returns and indirect jumps/calls, *i. e.* by enforcing control-flow integrity (CFI) [1, 4]. In fact, to tackle the shortcomings of existing defenses, we will introduce in Section 4.1 the first fine-grained CFI solution for mobile devices. However, before turning our attention to advanced defenses against code-reuse attacks, we explore the limitations of recently introduced efficient but coarse-grained CFI defenses against code-reuse attacks.

3.2 ON THE INEFFECTIVENESS OF COARSE-GRAINED CONTROL-FLOW INTEGRITY

Control-flow integrity (CFI) has been proposed as a general and fine-grained defense approach to thwart code-reuse attacks. As we described in Section 2.3, it derives the application's control-flow graph (CFG) prior to execution to determine the valid set of branch targets for indirect jumps and calls. At runtime, it performs control-flow checks based on the derived set of valid targets and the shadow stack which contains valid return addresses a function is allowed to return to. Although CFI requires no source code of an application, it suffers from practical limitations that impede its deployment in practice, including performance overhead of 21%, on average [4, Section 5.4], when function returns are validated based on a return address (shadow) stack. To date, several CFI frameworks have been proposed that tackle the practical shortcomings of the original CFI approach. ROPecker [47] and kBouncer [150], for example, leverage the branch history table of modern x86 processors to perform a CFI check on a short history of executed branches. More recently, Zhang and Sekar [204] demonstrate a new CFI binary instrumentation approach that can be applied to commercial off-the-shelf (COTS) binaries.

However, the benefits of these state-of-the-art solutions comes at the price of relaxing the original CFI policy. Abstractly speaking, coarse-grained CFI allows for CFG relaxations that contains dozens of more legal execution paths than would be allowed under the approach first suggested by Abadi et al. [4]. The most notable difference is that the coarse-grained CFI policy for return instructions only validates if the return address points to an instruction that follows after a call instruction. In contrast, Abadi et al. [4]'s policy for fine-grained CFI ensures that the return address points to the original caller of a function (based on a shadow stack).

Surprisingly, even given these relaxed assumptions, all recent coarse-grained CFI solutions we are aware of claim that their relaxed policies are sufficient to thwart return-oriented programming attacks[7]. In particular, they claim that the property of Turing-completeness is lost due to the fact that the code base which an adversary can exploit is significantly reduced. Yet, to date, no evidence substantiating these assertions has been given, raising questions with regards to the true effectiveness of these solutions.

Contribution. We revisit the assumption that coarse-grained CFI offers an effective defense against return-oriented programming. For this, we conduct a security analysis of the recently proposed CFI solutions including kBouncer [150], ROPecker [47], CFI for COTS binaries [204], ROPGuard [83], and Microsofts' EMET tool [131]. In particular, we derived a combined CFI policy that takes for each indirect branch class (*i. e.* return, indirect jump, indirect call) and behavioral-based heuristics (*e. g.* the number of instructions executed between two indirect branches), the most restrictive setting among these policies. Afterwards, we use our combined CFI policy and a weak adversary having access to only a *single* — and common used system library — to realize a Turing-complete gadget set. The reduced code base mandated that we develop several new return-oriented

7 Some of the mechanisms used in kBouncer and ROPGuard (both awarded by Microsoft's BlueHat Prize [181]) have already been integrated in Microsoft's defense tool called EMET [131].

programming attack gadgets to facilitate our attacks. To demonstrate the power of our attacks, we show how to harden existing real-world exploits against the Windows version of Adobe Reader [110] and mPlayer [36] so that they bypass coarse-grained CFI protections. We also demonstrate a proof-of-concept attack against a Linux-based system.

Section Outline. After elaborating on CFI challenges in Section 3.2.1, we categorize recently proposed CFI schemes in Section 3.2.2, and derive a combined policy in Section 3.2.3. In Section 3.2.4, we present our Turing-complete gadget set. Next, we present in Section 3.2.5 several useful gadgets for real-world exploitation, and a new gadget type, denoted as long-NOP, to undermine coarse-grained CFI policies that are based on behavioral-based heuristics. To prove the effectiveness of our attack, we leverage our Turing-complete gadget set in Section 3.2.6 to transform existing return-oriented exploits into more stealthy attacks that cannot be prevented by coarse-grained CFI schemes. Lastly, we conclude in Section 3.2.7.

3.2.1 *Control-Flow Integrity Challenges*

There are several factors that have impede the deployment of CFI in practice, such as CFG coverage, performance, robustness, and ease of deployment. In this section, we elaborate on each of these factors in more detail.

In their seminal work on CFI, Abadi et al. [4] also included a formal security proof for the soundness of their solution. The key argument put forth in this work is that "despite attack steps, the program counter always follows the CFG." [4]. In other words, every control-flow is permitted as long as the CFG allows it. Consequently, the security level highly depends on the level of CFG coverage. And that is exactly where recent CFI solutions have deviated (substantially) from the original work, primarily as a means to address performance overhead.

Recall that in the original CFI proposal, the CFG was obtained a priori using binary analysis techniques supported by a proprietary framework called Vulcan. This framework required debug symbols of the application in order to generate the CFG. However, debug symbols are rarely available for today's software programs. Since the CFG is created ahead of time, it does not capture the dynamic state of the call stack. That is, with only the CFG at hand, one can not enforce that functions return to their most recent call site, but only that they return to any of the possible call sites. This limitation is tackled by adding a shadow stack to the statically created CFG to validate if the function's return address equals the one pushed onto the safe shadow stack. In this way, many control-flow transfers are prohibited, largely reducing the gadget space available for a return-oriented programming attack.

Given the power of CFI, it is surprising that it has not yet received widespread adoption. The reason lies in the fact that extracting the CFG is not as simple as it may appear. To see why, notice that (i) source code is not readily available (thereby limiting compiler-based approaches), (ii) binaries typically lack the necessary debug or relocation information, as was needed for example, in the Vulcan framework, and (iii) the approach

Category	Policy	x86 Example	Description
①	CFI_{RET}	`RET`	returns
	CFI_{JMP}	`JMP reg|mem`	indirect jumps
	CFI_{CALL}	`CALL reg|mem`	indirect calls
②	CFI_{HEU}		heuristics
③	CFI_{TOC}		time of CFI check

Table 2: Our target CFI policies to validate coarse-grained CFI

induces performance overhead due to dynamic rewriting and runtime checks. Much of the academic research on CFI in the last few years has focused on techniques for tackling these drawbacks.

3.2.2 *Categorizing Coarse-Grained Control-Flow Integrity Approaches*

As noted above, a number of new control-flow integrity (CFI) solutions have been recently proposed to address the challenges of good runtime performance, high robustness and ease of deployment. The most prominent examples include kBouncer [150], ROPecker [47], CFI for COTS binaries [204], and ROPGuard [83]. To aide in better understanding the strengths and limitations of these proposals, we first provide a taxonomy of the various CFI policies embodied in these works. Later, to strengthen our own analyses, we also derive a combined CFI policy that takes into account the most restrictive CFI policy.

3.2.2.1 *CFI Policies*

Table 2 summarizes the five CFI policies we use throughout this paper to analyze the effectiveness of coarse-grained CFI solutions. Specifically, we distinguish between three types of policies, namely ① policies used for indirect branch instructions, ② general CFI heuristics that do not provide well-founded control-flow checks but instead try to capture general machine state patterns of return-oriented attacks and ③ a policy class that covers the time CFI checks are enforced.

We believe this categorization covers the most important aspects of CFI-based defenses suggested to date. In particular, they cover polices for each indirect branch the processor supports since all control-flow attacks (including code reuse) require exploiting indirect branches. Second, heuristics are used by several coarse-grained CFI approaches (*e. g.* [150, 83]) to allow more relaxed CFI policies for indirect branches. Finally, the time-of-check policy is an important aspect because it states at which execution state return-oriented attacks can be detected. We elaborate further on each of these categories below.

① **– Indirect Branches.** Recall that the goal of CFI is to validate the control-flow path taken at *indirect* branches, *i. e.* at those control-flow instructions that take the target ad-

dress from either a processor register or from a data memory area[8]. The indirect branch instructions present on an Intel x86 platform are indirect calls, indirect jumps, and returns. Since CFI solutions apply different policies for each type of indirect branch, it is only natural that there are three CFI policies in this category, denoted as CFI_{CALL} (indirect function calls), CFI_{JMP} (indirect jumps), CFI_{RET} (function returns).

② **– Behavior-Based Heuristics (HEU).** Apart from enforcing specific policies on indirect branch instructions, CFI solutions can also validate other program behavior to detect return-oriented programming attacks. One prominent example is the number of instructions executed between two consecutive indirect branches. The expectation is that the number of such instructions will be low (compared to ordinary execution) because return-oriented programming attacks invoke a chain of short code sequences each terminating in an indirect branch instruction (cf. Section 2.1.5.2).

③ **– Time of CFI Check (TOC).** Abadi et al. [4] argued that a CFI validation routine should be invoked whenever the program issues an indirect branch instruction. In practice, however, doing so induces significant performance overhead. For that reason, some of the more recent CFI approaches reduce the number of runtime checks, and only enforce CFI validation at critical program states, *e. g.* before a system or API call.

Next, we turn our attention to the specifics of how these policies are implemented in recent CFI mechanisms.

3.2.2.2 *kBouncer*

The approach of Pappas et al. [150], called kBouncer, deploys techniques that fall in each of the aforementioned categories. Under category ①, Pappas et al. [150] leverage the x86-model register set called last branch record (LBR). The LBR provides a register set that holds the last 16 branches the processor has executed. Each branch is stored as a pair consisting of its source and target address. kBouncer performs CFI validation on the LBR entries whenever a Windows API call is invoked. Its promise resides in the fact that these checks induce almost no performance overhead, and can be directly applied to existing software programs.

With respect to its policy for returns, kBouncer identifies those LBR entries whose source address belong to a return instruction. For these entries, kBouncer checks whether the target address (*i. e.* the return address) points to a *call-preceded* instruction. A call-preceded instruction is any instruction in the address space of the application that follows a call instruction. Internally, kBouncer disassembles a few bytes before the target address and terminates the process if it fails to find a call instruction.

While kBouncer does not enforce any CFI check on indirect calls and jumps, Pappas et al. [150] propose behavioral-based heuristics (category ②) to mitigate return-oriented attacks. In particular, the number of instructions executed between consecutive indirect

8 Typically, CFI does not validate direct branches because these addresses are hard-coded in the code of an executable and cannot be changed by an adversary when $W \oplus X$ is enforced.

branches (*i. e.* "the sequence length") is checked, and a limit is placed on the number of sequences that can be executed in a row.[9]

A key observation by Pappas et al. [150] is that even though pure code-reuse payloads can perform Turing-complete computation, in actual exploits they will ultimately need to interact with the operating system to perform a meaningful task. Hence, as a time-of-CFI check policy (category ③) kBouncer instruments and places hooks at the entry of a WinAPI function. Additionally, it writes a *checkpoint* after CFI validation to prohibit an adversary from simply jumping over the hook in userspace. In their implementation, the central Windows handler verifies whether the correct checkpoint has been written before transferring the control-flow to the WinAPI function is executed.

3.2.2.3 *ROPGuard and Microsoft EMET*

Similar to Pappas et al. [150], the approach suggested by Fratric [83] (called ROPGuard) performs CFI validation when a critical Windows function is called. However, its policies differ from that of Pappas et al. [150].

First, with respect to policies under category ①, upon entering a critical function, ROPGuard validates whether the return address of that critical function points to a call-preceded instruction. Hence, it prevents an adversary from using an instruction sequence terminating in a return instruction to invoke the critical Windows function. In addition, ROPGuard checks if the memory word before the return address is the start address of the critical function. This would indicate that the function has been entered via a return instruction. ROPGuard also inspects the stack and predicts future execution to identify gadgets. Specifically, it walks the stack to find return addresses. If any of these return addresses points to a non-call-preceded instruction, the program is terminated.

Interestingly, there is no CFI policy for indirect calls or indirect jumps. Furthermore, ROPGuard's only heuristic under category ② is for validating that the stack pointer does not point to a memory location beyond the stack boundaries. While doing so prevents return-oriented payload execution on the heap, it does not prevent traditional stack-based return-oriented attacks; thus the adversary could easily reset the stack pointer before a critical function is called.

Lastly, similar to kBouncer, ROPGuard's CFI validation is executed whenever a critical Windows function is called. ROPGuard allows to define these functions in a configuration file, whereas in EMET a pre-defined list is used.

Remarks. ROPGuard and its implementation in Microsoft EMET [17] use similar CFI policies as in kBouncer. One difference is that kBouncer checks the indirect branches executed in the past, while ROPGuard only checks the current return address of the critical function, and for future execution of gadgets. ROPGuard is vulnerable to code-reuse attacks that are capable of jumping over the CFI policy hooks, and cannot prevent attacks that do not attempt to call any critical Windows function. To tackle the former problem (*i. e.* bypassing the policy hook), EMET adds some randomness in the length and structure of the policy hook instructions. Hence, the adversary has to guess the

9 Specifically, kBouncer reports a return-oriented attack when a chain of 8 short sequences has been executed, where a sequence is referred to as "short" whenever the sequence length is less than 20 instructions.

right offset to successfully deploy her attack. However, recent memory disclosure attacks show that such randomization approaches can be circumvented [172].

3.2.2.4 *ROPecker*

ROPecker is a Linux-based approach suggested by Cheng et al. [47] that also leverages the last branch record register set to detect past execution of gadgets. Moreover, it speculatively emulates the future program execution to detect gadgets that will be invoked in the near future. To accomplish this, a static offline phase is required to generate a database of all possible instruction sequences. To limit false positives, Cheng et al. [47] suggest that only code sequences that terminate after at most n instructions in an indirect branch should be recorded.

For its policies in category ①, ROPecker inspects each LBR entry to identify indirect branches that have redirected the control-flow to a gadget. This decision is based on the gadget database that ROPecker derived in the static analysis phase. ROPecker also inspects the program stack to predict future execution of gadgets. There is no direct policy check for indirect branches, but instead, possible gadgets are detected via a heuristic. More specifically, the robustness of its behavioral-based heuristic (category ②) completely hinges on the assumption that instruction sequences will be short and that there will always be a chain of at least some threshold number of consecutive instruction sequences.

Lastly, its time of CFI check policy (category ③) is triggered whenever the program execution leaves a sliding window of two memory pages. Specifically, ROPecker sets all memory code pages to non-executable except (i) the page where program execution is currently performed on, and (ii) the most previously executed page. When the program aims to execute code from a non-executable page, ROPecker adjusts the sliding and performs CFI validation. In addition, ROPecker hooks into some critical Linux functions, namely *mprotect()*, *mmap2()*, and *execve()*. Before executing those functions, CFI validation is enforced as well.

Remarks. Clearly, ROPecker performs more frequently CFI checks than both kBouncer and ROPGuard. Hence, it can detect return-oriented attacks that do not necessarily invoke critical functions. However, as we shall show later, the fact that there is no policy for the target of indirect branches is a significant limitation.

3.2.2.5 *CFI for COTS Binaries*

Most closely related to the original CFI work by Abadi et al. [4] is the proposal of Zhang and Sekar [204] which suggest an approach for commercial-off-the-shelf (COTS) binaries based on a static binary rewriting approach, but without requiring debug symbols or relocation information of the target application. In contrast to all the other approaches we are aware of, the CFI checks are directly incorporated into the application binary. To do so, the binary is disassembled using the Linux disassembler `objdump`. However, since that disassembler uses a simple linear sweep disassembly algorithm, Zhang and Sekar [204] suggest several error correction methods to ensure correct disassembly.

Moreover, potential candidates of indirect control-flow target addresses are collected and recorded. These addresses comprise possible return addresses (*i. e.* call-preceded instructions), constant code pointers (including memory locations of pointers to external library calls), and computed code pointers (used for instance in switch-case statements). Afterwards, all indirect branch instructions are instrumented by means of a jump to a CFI validation routine.

Like the aforementioned works, the approach of Zhang and Sekar [204] checks whether a return or an indirect jump targets a call-preceded instruction. Furthermore, it also allows returns and indirect jumps to target any of the constant and computed code pointers, as well as exception handling addresses. Hence, the CFI policy for returns is not as strict as in kBouncer, where only call-preceded instructions are allowed. On the other hand, their approach deploys a CFI policy for indirect jumps, which is largely unmonitored in the other approaches. However, it does not deploy any behavioral-based heuristics (category ②).

Lastly, CFI validation (category ③) is performed whenever an indirect branch instruction is executed. Hence, it has the highest frequency of CFI validation invocation among all discussed CFI approaches.

Similar CFI policies are also enforced by CCFIR (compact CFI and randomization) [202]. In contrast to CFI for COTS binaries, all control-flow targets for indirect branches are collected and randomly allocated on a so-called springboard section. Indirect branches are only allowed to use control-flow targets contained in that springboard section. Specifically, CCFIR enforces that returns target a call-preceded instruction, and indirect calls and jumps target a previously collected function pointer. Although the randomization of control-flow targets in the springboard section adds an additional layer of security, it is not directly relevant for our analysis. Recall that memory disclosure attacks can reveal the content of the entire springboard section [172]. The CFI policies enforced by CCFIR are in principle covered by CFI for COTS binaries. However, there is one noteworthy policy addition: CCFIR denies indirect calls and jumps to target pre-defined sensitive functions (*e. g. VirtualProtect*). We do not consider this policy for two reasons: first, this policy violates the default external library call dispatching mechanism in Linux systems. Any application linking to such a sensitive (external) function will use an indirect jump to invoke it.[10] Second, as shown in detail by Göktas et al. [89] there are sufficient direct calls to sensitive functions in Windows libraries which an adversary can exploit to legitimately transfer control to a sensitive function.

Remarks. The approach of Zhang and Sekar [204] is most similar to Abadi et al. [4]'s original proposal in that it enforces CFI policies each time an indirect branch is invoked. However, to achieve better performance and to support COTS binaries, it deploys less fine-grained CFI policies. Alas, its coarse-grain policies allow one to bypass the restrictions for indirect call instructions (CFI_{CALL}). The main problem is caused by the fact that only one label is used for all indirect call targets. Given the large set of indirect call targets in modern applications, an adversary can overwrite a valid function pointer with

10 The target address of an external function is dynamically allocated in the global offset table (GOT) which is loaded by an indirect memory jump in the procedure linkage table (PLT).

the address of another function. A typical target is the Linux global offset table (GOT) which holds branch addresses for library calls. This leaves the solution vulnerable to so-called GOT-overwrite attacks [35] that overwrite pointers (in the GOT) to external library calls. We return to this vulnerability in Section 3.2.6. Moreover, even if one would ensure the integrity of the GOT, we are still allowed to use a valid code pointer defined in the external symbols. Hence, the adversary can invoke dangerous functions such as *VirtualAlloc()* and *memcpy()* that are frequently used in modern applications and libraries.

3.2.3 *Deriving a Combined Control-Flow Integrity Policy*

In our analysis that follows, we endeavor to have the best possible protections offered by the aforementioned CFI mechanisms in place at the time of our evaluation. Therefore, our combined CFI policy (see Table 3) selects the most restrictive setting for each policy. Nevertheless, despite this combined CFI policy, we then show that one can still circumvent these coarse-grained CFI solutions, construct Turing-complete return-oriented programming attacks (under realistic assumptions), and launch real-world exploits.

At this point, we believe it is prudent to comment on the parameter choices in these prior works — and that adopted in Table 3. In particular, one might argue that the prerequisite thresholds could be adjusted to make code-reuse attacks more difficult. To that end, we note that Pappas et al. [150] performed an extensive analysis to arrive at the best range of thresholds for the recommended number of consecutive short sequences (s) with a given sequence length of $n <= 20$. Their analysis reveals that adjusting the thresholds for s beyond their recommended values is hardly realistic: when every function call was instrumented, 975 false positives were recorded for $s <= 8$.

An alternative is to increase the sequence length n (*e. g.* setting it to $n <= 40$). Doing so would require an adversary to find a long sequence of 40 instructions after each seventh short sequence (for $s <= 7$). However, increasing the threshold for the sequence length will only exacerbate the false positive issue. For this reason, Pappas et al. [150] did not consider sequences consisting of more than 20 instructions as a gadget in their analyses. We provide our own assessment in Section 3.2.6.3.

The approach of Cheng et al. [47], on the other hand, uses different thresholds for s and n than in kBouncer. Making the thresholds in ROPecker more conservative (*e. g.* reducing s and increasing n) will lead to the same false positives problems as in kBouncer. Moreover, the problem would be worse, since ROPecker performs CFI validation more frequently than kBouncer. Nevertheless, we show that regardless of the specific choice of parameter chosen in the recommended ranges, our attacks render these defenses ineffective in practice (cf. Section 3.2.6).

3.2.4 *Turing-Complete Gadget Set*

We now explore whether or not it is possible to derive a Turing-complete gadget set even when all state-of-the-art coarse-grained CFI protections are enforced. In particular,

Control-Flow Integrity (CFI) Policies	CFI for COTS [204]	kBouncer [150]	ROPecker [47]	ROPGuard [83]	EMET 4.1 [131]	*Combined Policy*
CFI_{RET} : destination has to be call-preceded	✓	✓	◯	✓	✓	✓
CFI_{RET} : destination can be taken from a code pointer	✓	✗	◯	✗	✗	✗
CFI_{JMP} : destination has to be call-preceded	✓	◯	◯	◯	◯	✓
CFI_{JMP} : destination can be taken from a code pointer	✓	◯	◯	◯	◯	✓
CFI_{CALL} : destination can be taken from an exported symbol	✓	◯	◯	◯	◯	✓
CFI_{CALL} : destination can be taken from a code pointer	✓	◯	◯	◯	◯	✓
CFI_{HEU} : allow only s consecutive short sequences,	◯	$s <= 7$	$s <= 10$	◯	◯	$s <= 7$
CFI_{HEU} : where *short* is defined as n instructions	◯	$n <= 20$	$n <= 6$	◯	◯	$n <= 20$
CFI_{TOC} : check at every indirect branch	✓	◯	◯	◯	◯	Indirect branch
CFI_{TOC} : check at critical API functions or system calls	◯	✓	✓	✓	✓	
CFI_{TOC} : check when leaving sliding code window	◯	◯	✓	◯	◯	

Table 3: Policy comparison of coarse-grained CFI solutions: ✓ indicates that the CFI policy is applied and enforced. ✗ means that the CFI policy is prohibited (corresponding execution flows would lead to an attack alarm) . ◯ indicates that the CFI policy is not applied/enforced. The combined policy takes the most restrictive setting for each CFI policy.

we desire a gadget set that still allows an adversary to undermine the combined CFI policy (see Table 3).

Assumptions. To be as pragmatic as possible, we assume that the adversary can only leverage the presence of a single shared library to derive the gadget set. This is a very stringent requirement placed on ourselves since modern programs typically link to dozens of libraries.

Note also that we are not concerned with circumventing other runtime protection mechanisms such as ASLR or stack canaries. The reasons are twofold: first, coarse-grained CFI protection approaches do not rely on the presence of other defenses to mitigate against code-reuse attacks. Second, in contrast to CFI, ASLR and protection mechanisms that defend against code pointer overwrites (*e. g.* stack canaries, bounds checkers, and pointer encryption) do not offer a general defense, and moreover, are typically bypassed in practice. In particular, ASLR is vulnerable to memory disclosure attacks [175, 172]. That said, the attacks and return-oriented programming gadgets we present in the following can be also leveraged to mount memory disclosure attacks in the first stage.

Methodology and Outline. Our analysis is performed primarily on Windows as it is the most widely deployed desktop operating system today. Specifically, we inspect `kernel32.dll` (on x86 Windows 7 SP1), a 848kb system library that exposes Windows API functions and is by default linked to nearly every major Windows process (*e. g.* Adober Reader, IE, Firefox, MS Office). It is also noteworthy to mention that our results do not only apply to Windows. Although we did not perform a Turing-complete gadget analysis for Linux's default library (`libc.so`), to demonstrate the generality of our approach, we provide a shellcode exploit that uses gadgets from `libc` (cf. Section 3.2.6.2). To facilitate the gadget finding process, we developed a static analysis python module in `IDA Pro` that outputs all call-preceded sequences ending in an indirect branch. We also developed a sequence filter in the general purpose `D` programming language that allows us to check for sequences containing a specific register, instruction, or memory operand. Note that in the subsequent discussions, we use the Intel assembler syntax, *e. g.* `MOV destination, source`, and use a semicolon to separate two consecutive instructions.

We first review the basic gadgets that form a Turing-complete language [169, 41]. Similar to our code-reuse attack described in Section 3.1, we require gadgets that realize memory load and store operations, arithmetic/logical gadgets, as well as a gadget to realize a conditional branch to achieve Turing-completeness. Afterwards, in Section 3.2.5, we present two new gadget types, called the Call-Ret-Pair gadget, and the Long-NOP gadget. Constructing the latter was a non-trivial engineering task and the outcome played an important role in "stitching" gadgets together, thereby bypassing coarse-grained CFI defenses. It should also be noted that we only present a *subset* of the available sequences. Eliminating the specific few sequences presented in this section will not prevent our attack, since `kernel32.dll` (and many other libraries) provides a multitude of other sequences we could have leveraged.

3.2.4.1 *Loading Registers*

Load gadgets are leveraged in nearly every return-oriented programming exploit to load a value from the stack into a CPU register. Recall that x86 provides six general-purpose registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`), a base/frame pointer register (`ebp`), the stack pointer (`esp`), and the instruction pointer (`eip`). All registers can be directly accessed (read and write) by assembler instructions except the `eip` which is only indirectly influenced by dedicated branch instructions such as `RET`, `CALL`, and `JMP`.

Typically, stack loading is achieved on x86 via the `POP` instruction. The call-preceded load gadgets we identified in `kernel32.dll` are summarized in Table 4. Except for the `ebp` register, we are not able to load any other register without inducing a side-effect, *i. e.* without affecting other registers. That said, notice that the sequence for `esi`, `edi`, and `ecx` only modifies the base pointer (`ebp`). Because traditionally `ebp` holds the base pointer and no data, and ordinary programs can be compiled without using a base pointer, we consider `ebp` as an *intermediate register* in our gadget set. The astute reader would have noticed that the sequences for `edi` and `ecx` modify the stack pointer as well through the `LEAVE` instruction, where `LEAVE` behaves as `MOV esp,ebp; POP ebp`. However, we can handle this side-effect, since the stack pointer receives the value from our intermediate register `ebp`. Hence, we first invoke the load gadget for `ebp` and load the desired stack pointer value, and afterwards call the sequence for `edi/ecx`.

More challenges arise when loading the general-purpose registers `eax`, `ebx`, and `edx`. While `ebx` can be loaded with side-effects, we were not able to find any useful stack pop sequence for `eax` and `edx`. This is not surprising given the fact that we must use call-preceded sequences. Typically, these sequences are found in function epilogues, where a function epilogue is responsible for resetting the caller-saved registers (`esi`, `edi`, `epb`). We alleviate the side-effects for `ebx` by loading all the caller-saved registers from the stack.

Register	**Call-Preceded Sequence (ending in `RET`)**
EBP	`POP ebp`
ESI	`POP esi; POP ebp`
EDI	`POP edi; LEAVE`
ECX	`POP ecx; LEAVE`
EBX	`POP edi; POP esi; POP ebx; POP ebp`
EAX	`MOV eax,edi; POP edi; LEAVE`
EDX	`MOV eax,[ebp-8]; MOV edx,[ebp-4];`
	`POP edi; LEAVE`

Table 4: Register load gadgets

For `eax` and `edx`, data movement gadgets can be used. As can be seen in Table 4, `eax` can be loaded using the `edi` load gadget in advance. The situation is more complicated for `edx`, especially given our choice to only use `kernel32.dll`. In particular, while there is a sequence that allows one to load `edx` by using the `ebp` load gadget beforehand, it is

challenging to do so since the adversary would need to save the state of some registers. That said, other default Windows libraries (such as `shell32.dll`) offer several more convenient gadgets to load `edx` (*e. g.* a common sequence we observed was `POP edx; POP ecx; JMP eax`), and so this limitation should not be a major obstacle in practice.

3.2.4.2 *Loading and Storing from Memory*

In general, software programs can only accomplish their tasks if the underlying processor architecture provides instructions for loading from memory and storing values to memory. Similarly, return-oriented programming attacks require memory load and store gadgets. Although we have found several load and store gadgets, we focus on the gadgets listed in Table 5.

Type	**Call-Preceded Sequence (ending in `RET`)**
LOAD (eax)	`MOV eax,[ebp+8]; POP ebp`
STORE (eax)	`MOV [esi],eax; XOR eax,eax;`
	`POP esi; POP ebp`
STORE (esi)	`MOV [ebp-20h],esi`
STORE (edi)	`MOV [ebp-20h],edi`

Table 5: Selected memory load and store gadgets

In particular, we discovered load gadgets that use `eax` as the destination register. The specific load gadget shown in Table 5 loads a value from memory pointed to by `[ebp+8]`. Hence, the adversary is required to correctly set the target address of the memory load operation in `ebp` via the register load gadget shown in Table 4.

We also identified a corresponding memory store gadget on `eax`. The shown gadget stores `eax` at the address provided by register `esi`, which needs to be initialized by a load register gadget beforehand. The gadget has no side-effects since it resets `eax` (which was stored earlier) and loads new values from the stack into `esi` (which held the target address) and `ebp` (our intermediate register).

Given a memory store gadget for `eax` and the fact that we have already identified register load gadgets for each register, it is sufficient to use the same memory load on `eax` to load any other register. This is possible because we use the `eax` load gadget to load the desired value from memory, store it afterwards on the stack, and finally use one of the register load gadgets to load the value into the desired register. Finally, we also identified some convenient memory store gadgets for `esi` and `edi` only requiring `ebp` to hold the target address of the store operation.

3.2.4.3 *Arithmetic and Logical Gadgets*

For arithmetic operations, we utilize the sequence containing the x86 `SUB` instruction shown in Table 6. This instruction takes the operands from `eax` and `esi` and stores the result of the subtraction into `eax`. Both operands can be loaded by using the register

load gadgets (see Table 4). The same gadget can be used to perform an addition: one only needs to load the two's complement into `esi`. Based on addition and subtraction, we can realize multiplication and division as well. Unfortunately, logical gadgets are not as commonplace. There is, however, a XOR gadget that takes its operands from `eax` and `edi` (see Table 4).

Type	Call-Preceded Sequence (ending in `RET`)
ADD/SUB	`SUB eax,esi; POP esi; POP ebp`
XOR	`XOR eax,edi; POP edi; POP esi;` `POP ebp`

Table 6: Arithmetic and logical gadgets

3.2.4.4 *Branching Gadgets*

We remind the reader that branching in return-oriented programming attacks is realized by modifying the stack pointer rather than the instruction pointer [169]. In general, we can distinguish two different types of branches: unconditional and conditional branches. `kernel32.dll`, for example, offers two variants for a unconditional branch gadget (see Table 7). The first uses the `LEAVE` instruction to load the stack pointer (`esp`) with a new address that has been loaded before into our intermediate register `ebp`. The second variant realizes the unconditional branch by adding a constant offset to `esp`. Either one suffices for our purposes.

Conditional branch gadgets change the stack pointer iff a particular condition holds. Because load, store, and arithmetic/logic computation can be conveniently done for `eax`, we could place the conditional in this register. Unfortunately, because a direct load of `esp` (that depended on the value of `eax`) was not readily available, we realized the conditional branch in three steps requiring the invocation of only four instruction sequences. That said, our gadget is still within the constraints for the number of allowable consecutive sequences in the Combined CFI-enforcement Policy (see $s <= 7$ for CFI_{HEU} in Table 3).

First, we use the conditional branch gadget (see Table 7) to either load `0` or a prepared value into `eax`. In this sequence `NEG eax` computes the two's complement and, more importantly, sets the carry flag to zero if and only if `eax` was zero beforehand. This is nicely

Type	Call-Preceded Sequence (ending in `RET`)
unconditional Branch 1	`LEAVE`
unconditional Branch 2	`ADD esp,0Ch; POP ebp`
conditional LOAD(eax)	`NEG eax; SBB eax,eax;` `AND eax,[ebp-4]; LEAVE`

Table 7: Branching gadgets

used by the subsequent `SBB` instruction, which subtracts the register from itself, always yielding zero, but additionally subtracting an extra one if the carry flag is set. Because subtracting one from zero gives `0xFFFFFFFF`, the next `AND` instruction masks either none or all the bits. Hence, the result in `eax` will be exactly the contents of `[ebp-4]` if `eax` was zero, or zero otherwise. One might think that it is very unlikely to find sequences that follow the pattern `NEG-SBB-AND`. However, we found 16 sequences in `kernel32.dll` that follow the same pattern and could have been leveraged for a conditional branch gadget.

We then use the ADD/SUB gadget (see Table 6) to subtract `esi` from `eax` so that the latter holds the branch offset for `esp`. Finally, we move `eax` into `esp` using the stack as temporary storage. The STORE(eax) gadget (see Table 5) will store the branch offset on the stack, where `POP ebp` followed by the unconditional Branch 1 gadget loads it into `esp`.

3.2.5 *Extended Gadget Set*

For those readers who have either written or analyzed real-world code-reuse exploits before, it would be clear to them that several other gadgets are useful in practice. For example, modern exploits usually invoke several WinAPI functions to perform malicious actions, *e. g.* launching a malicious executable by invoking *WinExec()*. Calling such functions within a return-oriented programming attack requires a function call gadget (Section 3.2.5.1). It is also useful to have gadgets that allow one to conveniently write a NULL word to memory (Section 3.2.5.2) or the stack pivot gadget [206] which is used by return-oriented attacks exploiting heap overflows (Section 3.2.5.3).

Additionally, to provide a generic method for circumventing the behavioral-based heuristics of the Combined CFI Policy, we present a new gadget type, coined Long-NOP, containing long sequences of instructions which do not break the semantics of an arbitrary return-oriented programming chain (Section 3.2.5.4).

3.2.5.1 *Call-Ret-Pair Gadget*

CFI policies raise several challenges with respect to calling WinAPI functions within a code-reuse attack. First, one cannot simply exploit a `RET` instruction because the CFI_{RET} policy states that only a call-preceded sequence is allowed — clearly, the beginning of a function is not call-preceded. Second, the adversary must regain control when the function returns. Hence, the return address of the function to be called must point to a call-preceded sequence that allows the code-reuse attack to continue.

To overcome these restrictions, we utilize what we coined a *Call-Ret-Pair* gadget. The basic idea is to use a sequence that terminates in an indirect call but provides a short instruction sequence afterwards that terminates in a `RET` instruction. Among our possible choices, the Call 1 sequence shown in Table 8 was selected.

To better understand the intricacies of this gadget, we provide an example in Figure 21. This example depicts how we can leverage our gadget to call *VirtualAlloc()*. We start with a load register gadget which first loads the start address of *VirtualAlloc()* into `esi`. Further, it loads into `ebp` an address denoted as `ADDR`. At this address is stored RET 3,

Type	Call-Preceded Sequence
Call 1	`LEA eax,[ebp-34h]; PUSH eax;`
	`CALL esi; ret`
Call 2	`CALL eax`
Call 3	`PUSH eax; CALL [ebp+0Ch]`

Table 8: Function call gadgets

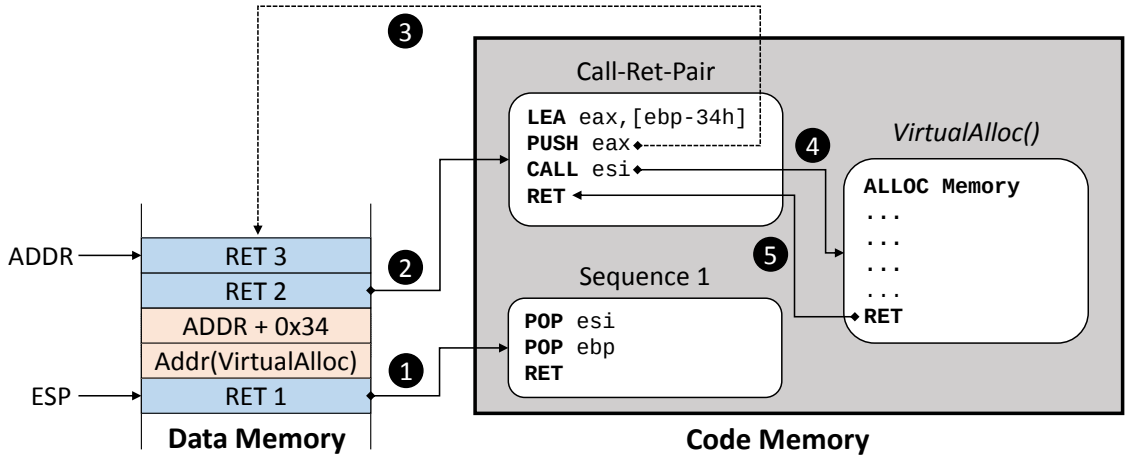


Figure 21: Example for Call-Ret-Pair gadget

the pointer to the instruction sequence we desire to call after *VirtualAlloc()* has returned. The next sequence is our Call-Ret-Pair gadget, where the first instruction effectively loads RET 3 pointed to by `[ebp-34h]` into `eax`. Next, RET 3 is stored at `ADDR` onto the stack using a `PUSH` instruction before the function call occurs. The `PUSH` instruction also decrements the stack pointer so that it points to RET 2. The subsequent indirect call invokes *VirtualAlloc()* and automatically pushes the return address onto the stack, *i. e.* it will overwrite RET 2 with the return address. This ensures that the control-flow will be redirected to the `RET` instruction in our Call-Ret-Pair gadget when *VirtualAlloc()* returns. Lastly, the return will use RET 3 to invoke the next sequence.

Note that this Call-Ret-Pair gadget works for subroutines following the `stdcall` calling convention. Such functions remove their arguments from the stack upon function return. For functions using `cdecl`, we use a Call-Ret-Pair gadget that pops after the function call, the arguments of the subroutine from the stack. The details of the gadget we use for a `cdecl` compiled function can be found in [64, Appendix D].

For code-reuse attacks that terminate in a function call, we leverage the Call 2 and Call 3 gadgets shown in Table 8. The difference resides in the fact that Call 2 requires the target address to be loaded into `eax`, whereas Call 3 loads the branch address from memory.

Recall that the CFI policy for indirect calls (CFI_{CALL} in Table 3) only permits the use of branch addresses taken from an exported symbol or a valid code pointer place. However,

as we already described in Section 3.2.2.5, modern applications typically make use of many WinAPI functions by default. We can indirectly call any one of these functions using the external symbols as the integrity of code pointers is not guaranteed.

3.2.5.2 *NULL-Byte Write Gadget*

In real-world exploits it is useful to have gadgets that allow one to conveniently write a NULL word to memory. This is important as real-world vulnerabilities typically do not allow an adversary to write a NULL byte in the payload, but such functionality is indeed needed to write a 32-bit NULL word on the stack when required as a parameter to function calls.

A prominent example is the traditional *strcpy(dest,src)* vulnerability, which can be exploited to write data beyond the boundaries of the *src* variable. However, *strcpy()* stops copying input data after encountering a NULL byte.

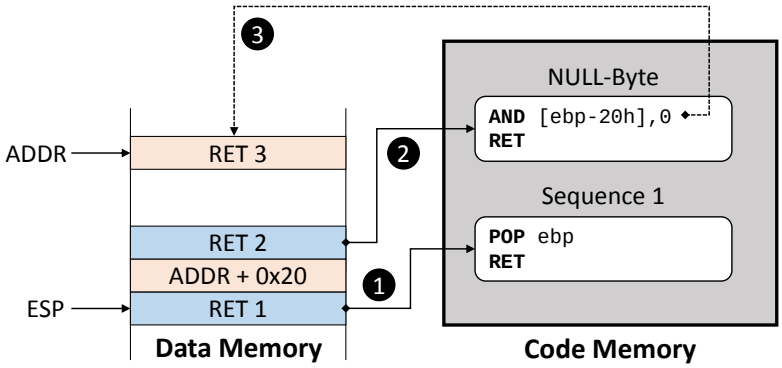


Figure 22: Details of NULL-Byte write gadget

Our choice for such a gadget is shown in Figure 22. This gadget first loads the target address into `ebp` with the first sequence (Step ❶). The next sequence exploits the `AND` instruction to generate a NULL word at the memory location pointed to by `[ebp-20h]` (Step ❷ and ❸).

3.2.5.3 *Stack Pivot Gadget*

We take advantage of two distinct stack pivot gadgets shown in Table 9. The first one is our unconditional branch gadget, which moves `ebp` via the `LEAVE` instruction to `esp`. The other sequence takes the value of `esi` and loads it into `esp`. In both sequences, the adversary must control the source register `ebp` and `esi`, respectively. This is achieved by invoking a load register gadget beforehand. Note also that several vulnerabilities allow an adversary to load these registers with the correct values at the time the buffer overflow occurs.

3.2.5.4 *Long-NOP Gadget*

Our final gadget is needed to thwart the restriction that after $s = 7$ short sequences in a row is used, another sequence of $n >= 20$ instructions must follow (see CFI_{HEU} in

Type	Call-Preceded Sequence (ending in RET)
Pivot 1	`LEAVE`
Pivot 2	`MOV esp, esi; POP ebx; POP edi;`
	`POP esi; POP ebp`

Table 9: Stack pivot gadgets

Table 3). For this task, we developed a new gadget type that we refer to as the long no-operation (long-NOP) gadget. Constructing long-NOP in a way that does not break the semantics of an arbitrary return-oriented programming chain was a non-trivial task.

To identify possible sequences for this gadget type, we let our sequence finder filter those call-preceded sequences that contain more than 20 instructions. To ensure that the long sequence does not break the semantics of the return-oriented programming chain, we further reduced the set of sequences to those that (i) contain many memory-write instructions, and (ii) make use of only a small set of registers. While the latter requirement is obvious, the former seems counter-intuitive as it can potentially change the memory state of the process. However, if we are able to control the destination address of these memory writes, we can write arbitrary values into the data area of the process outside the memory used by our code-reuse attack.

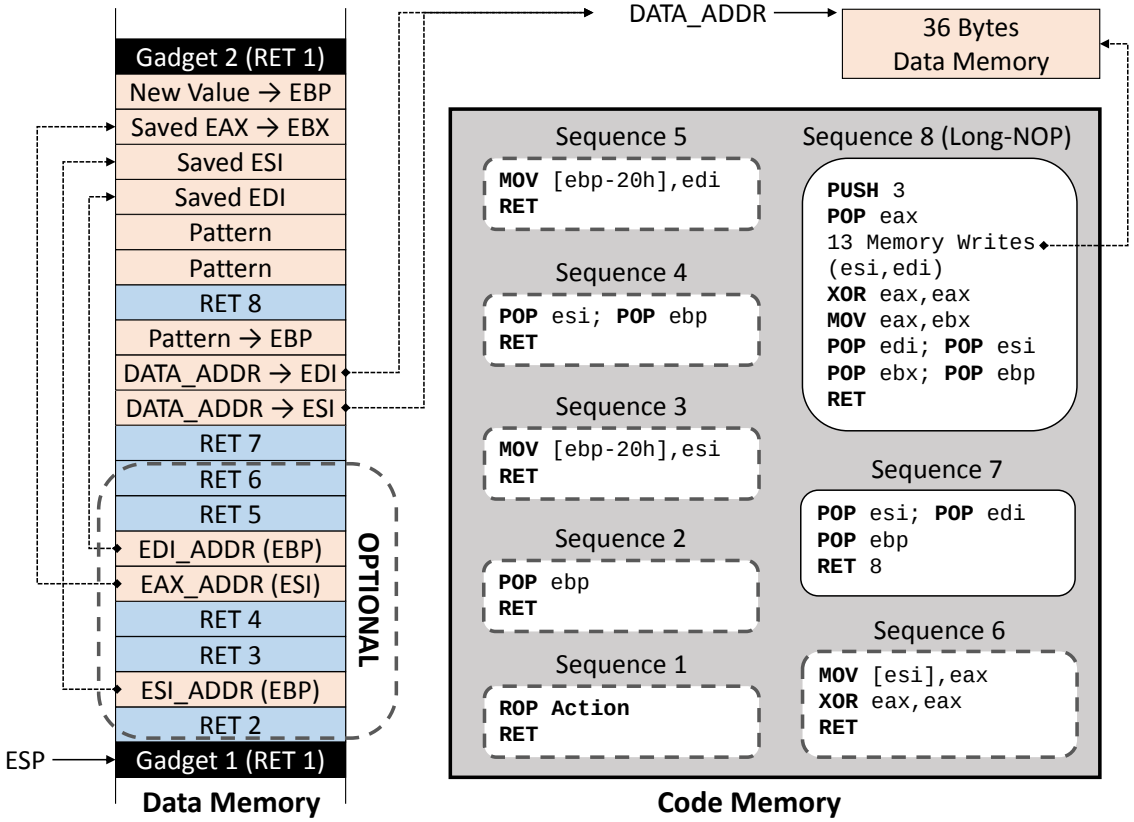


Figure 23: Flow of Long-NOP gadget

Among the sequences that fulfill these requirements, we chose a sequence that is (abstractly)[11] shown in Figure 23. It contains 13 memory write instructions using only the registers `esi` and `edi`. We stress that the entire gadget chain for long-NOP does not induce *any* side-effects, *i.e.* the content of all registers and memory area used by the code-reuse attack is preserved.

We distinguish between mandatory and optional sequences used for long-NOP. The latter sequences are only required if the content of all registers needs to be preserved. We classify them as optional, since it is very unlikely that code-reuse attacks need to operate on all registers during the entire execution phase. If all registers need to be preserved (worst-case scenario), we require 6 sequences before the long-NOP gadget sequence is invoked. Since all registers are preserved, we can issue in each round another sequence until all desired sequences have been executed.

Mandatory Sequences. The mandatory sequences are those labeled Sequence 7 and 8 (in Figure 23). Sequence 7 is used to set three registers: `esi`, `edi`, and `ebp`. We load in `esi` and `edi` the same address, namely `DATA_ADDR`, which points to an arbitrary data memory area in the address space of the application, *e.g.* stack, heap, or any other data segment of an executable module. Due to the `RET 8` instruction, the stack pointer will be incremented by 8 more bytes leaving space for pattern values. Afterwards, our long-NOP sequence uses `esi` and `edi` to issue 13 memory writes in a small window of 36 bytes. In each round, we use the same address for `DATA_ADDR`, and hence, we always write the same arbitrary values in a 36 byte memory space not affecting memory used by our code-reuse attack. The long-NOP sequence also destroys the value of `eax` and loads new values via `POP` instructions in other registers. However, these register changes are resolved by our optional sequences discussed next.

Optional Sequences. Sequence 2 to 6 are the optional sequences, and are responsible for preserving the state of all registers. The optional sequences shown in Figure 23 represent those already presented in our basic gadget arsenal in Section 3.2.4. Depending on the specific goals and gadgets of a code-reuse attack, the adversary can choose among the optional sequences as required.

Sequence 2 and 3 store the value of `esi` on the stack in such a way that the `POP esi` instruction in long-NOP resets the value accordingly. Sequence 4 to 6 store the content of `eax` and `edi` on the stack. Similar to the store for `esi`, the content is again re-loaded into these registers via `POP` instructions at the end of the long-NOP sequence. However, the content of register `eax` and `ebx` is exchanged after the long-NOP sequence since `MOV eax,ebx` stores `ebx` to `eax`, and the former value of `eax` is loaded via `POP` into `ebx`. However, we can compensate this switch by invoking the Long-NOP gadget twice so that `eax` and `ebx` are exchanged again.

11 For the interested reader, we have placed the specific assembler implementation of the long-NOP sequence in [64, Appendix C].

3.2.6 *Hardening Real-World Exploits*

We now elaborate on the hardening of two real-world exploits against 32-bit Windows 7 SP1 and a Linux proof-of-concept exploit. Specifically, we transform publicly available code-reuse attacks against Adobe PDF reader [110] and the GNU mediaplayer mPlayer [36]. We used the gadget set derived in Section 3.2.4 and 3.2.5 to perform the transformation. Furthermore, our attacks are executed with the *Caller*, *SimExecFlow*, *StackPivot*, *LoadLib*, and *MemProt* option for ROP detection in Microsoft EMET 4.1 enabled. The source code for both attacks is given in the Appendix of our technical report [65].

3.2.6.1 *Windows Exploits*

Our Adobe PDF attack exploits the integer vulnerability CVE-2010-0188 in the TIFF image processing library `libtiff`. The vulnerability originally targeted Adobe PDF versions 9.1-9.3 running on Windows XP SP2/SP3. Likewise, the mPlayer attack exploited a buffer overflow vulnerability that allows the adversary to overwrite an exception handler pointer. Since we perform our analyses on Windows 7, we ported both exploits from Windows XP to Windows 7. The exploits are provided as a malicious `pdf` and `m3u` video file, respectively.

Exploit Requirements. For both exploits, we need to (1) allocate a new read-write-execute (RWX) memory page with *VirtualAlloc()*, (2) copy malicious shellcode into the newly allocated page by using *memcpy()*, and (3) redirect the control-flow to the shellcode. Originally, the exploits made use of non-call-preceded gadgets, and used a long chain of short instruction sequences. For mPlayer 18 consecutive short sequences are executed, while for Adobe PDF 11 sequences are executed until the first system call is issued. Hence, both exploits clearly violate CFI_{RET} and CFI_{HEU} of the combined CFI policy. That is, these exploits are prevented by Microsoft EMET because of CFI_{RET}, and are detected by both kBouncer and ROPecker due to violation of the CFI_{HEU} policy.

Replacing Code Sequences. A simplified view of the gadget chain we use for our hardened exploits in the PDF exploit is shown in Figure 24. We first replaced all non-call-preceded sequences with one of our call-preceded sequences in our gadget set identified in Section 3.2.4. Both exploits mainly use load register and memory gadgets to set the arguments for *VirtualAlloc()* and *memcpy()*, and function call gadgets to invoke both functions. By leveraging only call-preceded sequences, our attacks comply to the CFI policy for returns (CFI_{RET}).

Since both exploits make use of WinAPI calls, we utilized our Call-Ret-Pair gadget to invoke *VirtualAlloc()* and *memcpy()*. As both functions are default routines used in a benign execution of Adobe PDF and mPlayer, we are allowed to leverage indirect calls to invoke these functions (addressing CFI_{CALL}). Lastly, we need to tackle the CFI policies for behavioral heuristics (addressing CFI_{HEU}) by ensuring that we never execute more than 7 short sequences in a row before calling our long-NOP gadget.

Putting-It-All-Together. Sequence ❶ in Figure 24 loads the target address of *VirtualAlloc()* into `esi`. The arguments to this function (`Arg1-Arg4`) are set on the stack. They are

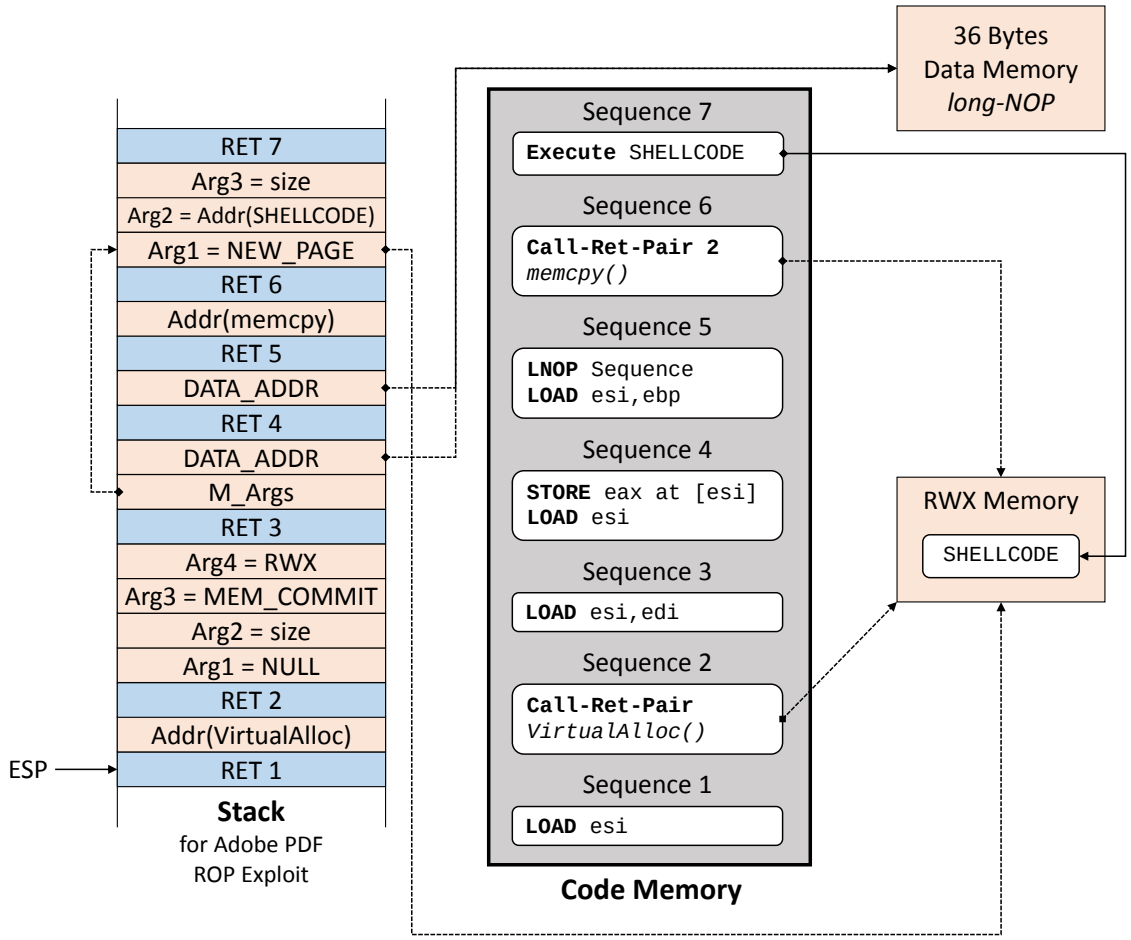


Figure 24: Simplified view of our hardened PDF exploit. See Appendix in [65] for the full source code.

chosen in such a way that *VirtualAlloc()* allocates a new RWX memory page. Sequence ❷ leverages our Call-Ret-Pair gadget to call *VirtualAlloc()*. The start address of the page is placed by *VirtualAlloc()* into `eax`.

Sequence ❸ and ❹ facilitate two goals: first they store the start address of the new RWX page on the stack. Second, they prepare the execution of the long-NOP gadget. In particular, they set `esi` and `edi` to DATA_ADDR. This address points to an arbitrary data section of one of the linked libraries. Our long-NOP sequence (Sequence ❺) will later perform 13 memory writes on this data region, thereafter setting `esi` to the start address of *memcpy()*. Sequence ❻ invokes *memcpy()* to copy the malicious shellcode onto the newly allocated RWX page. Lastly, our return-oriented programming chain transfers the control-flow to the copied shellcode via Sequence ❼, which in both exploits opens the Windows calculator.

For the Adobe PDF attack, we used 7 sequences with 52 instructions executed. In the hardened version of the mPlayer exploit, we used 49 sequences with 380 instructions executed. Note that the 49 sequences include the interspersed long-NOP sequences to

adhere to the CFI policy CFI_{HEU}. We used a writable memory area of 36 Bytes for the long-NOP gadget. The requirement of more sequences for the mPlayer attack can be attributed to the fact that this exploit did not allow for the use of any NULL bytes in the payload and so we needed to leverage a NULL-Byte gadget (cf. Section 3.2.5.2) in this exploit. The mPlayer exploit also required a stack pivot gadget (cf. Section 3.2.5.3). This attack also required a specific stack pivot gadget adding a large constant to `esp`. Unfortunately, our stack pivot sequences in `kernel32.dll` did not use large enough constants, and the original sequence exploited a non call-preceded one in `avformat-52.dll`. However, we identified another useful call-preceded stack pivot sequence in the same library which allowed us to instantiate the exploit.

The above strategies can be used to easily transform other return-oriented attacks to bypass current coarse-grained CFI defenses. Furthermore, given our routines for finding and filtering useful call-preceded instruction sequences, the process of transforming exploits could be fully automated. We leave that as an exercise for future work.

A final remark concerns the control transfer to the injected shellcode. In both exploits, we invoke a call-preceded sequence terminating in an indirect jump. While this approach works for kBouncer, ROPecker, and ROPGuard, it might raise an alarm for CFI for COTS binaries if the shellcode is placed at an address that is not within the set of valid function pointers (*i. e.* indirect jump targets). However, there are several ways to tackle this issue. A very effective approach has been shown by Göktas et al. [89], where the code section is simply set to be writable, the shellcode copied to an address which resembles a valid function pointer, and after which the code section is reset back to be executable.

3.2.6.2 *Linux Shellcode Exploit*

Since the approach of Zhang and Sekar [204] targets Linux specifically, we also developed a proof-of-concept exploit that shows how our attack bypasses the CFI policies for indirect calls. To do so, we use a sample program that suffers from a buffer overflow vulnerability allowing an adversary to overwrite a return address on the stack. The goal of our attack is to call *execve()*, which is a standard system function defined in `libc.so` to execute a new program. The challenge, however, is that the example program does not include *execve()* in its external symbols, and consequently, we are not allowed to redirect the control-flow to *execve()* using an indirect call.

To overcome this restriction, we leverage a well-known attack technique called global offset table (GOT) overwrite [35]. The basic idea is to write the address of *execve()* at a valid code pointer location. A well-known location for doing so is the GOT table, which contains pointers to library calls such as *printf()*. We reiterate that the weakness here is that CFI for COTS binaries deploys only one label for indirect call targets. Given the fact that the GOT is typically writable in the current design of Linux due to dynamic address resolution of external library functions and lazy binding, we can invoke gadgets to overwrite the pointers placed in the GOT. Specifically, we first find useful sequences from the Linux standard library `libc.so` and use gadgets that perform the GOT overwrite while using only call-preceded sequences.

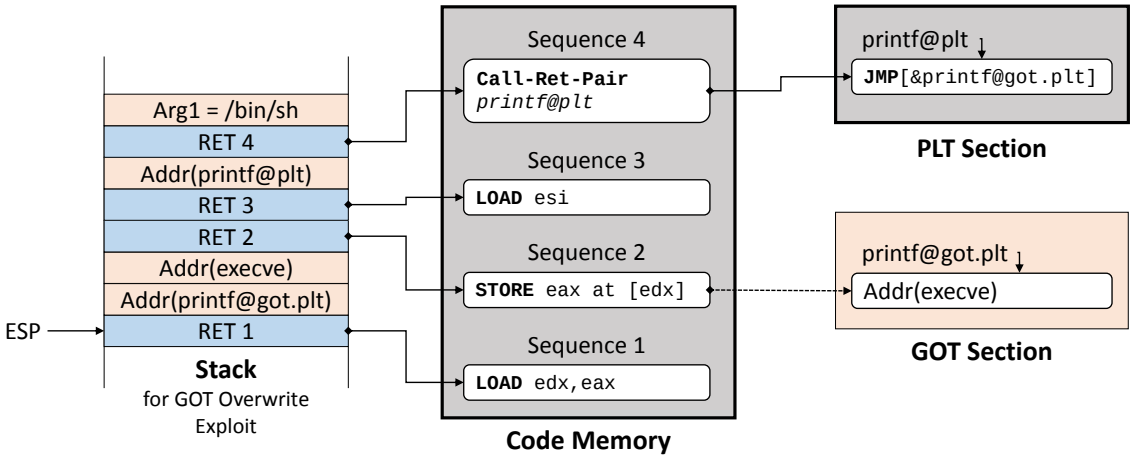


Figure 25: GOT overwrite attack

Putting-It-All-Together. An example on how we bypass the CFI policy for indirect calls is shown in Figure 25. The approach is as follows: first, Gadget ❶ loads the address of the GOT entry we want to modify into `edx`, and loads `eax` with the address of *execve()*. Next, Gadget ❷ overwrites the address of *printf()* with the address of *execve()* in the GOT. Finally, Gadget ❸ loads the address of the *printf()* stub into `esi`, and Gadget ❹ uses a Call-Ret-Pair gadget to invoke *execve()*. At this point, the attack succeeds without violating any of the CFI policies.

3.2.6.3 *On Parameter Adjustment*

As alluded to in Section 3.2.3, adjusting the parameters for the CFI_{HEU} policy beyond the recommended settings will negatively impact the false positive rate. To assess that, we extended the analysis beyond what Pappas et al. [150] originally performed in order to analyze the impact of increasing n to 30 or 40 instructions — thereby rendering our Long-NOP gadget (which is only 23 instructions long) stitching ineffective. Specifically, we performed an experiment using three benchmarks of the SPEC CPU 2006 benchmark suite: `bzip2`, `perlbench`, and `xalancbmk`. The first two are programmed in C, while the latter in C++. We developed an Intel Pintool that counts the number of instructions issued between two indirect branches, and the number of consecutive short instruction sequences. Whenever a function call occurs, we check how many short sequences (s) have been executed since the last function call.

As Figure 26 shows, increasing the thresholds for n induces many potential false positives (y-axis). In particular, for each benchmark (x-axis), we observe that for $s > 10$ there are about 20,000 potential false positives, *i. e.* 20,000 times we detected a function call that was preceded by more than 10 short sequences[12].

12 We also simulated the analysis performed in [150] by setting $n = 20$. However, we arrive at a significantly higher false positive rate than in [150]. This is likely due to the fact that we perform our analysis on industry benchmark programs, while their analysis is based on opening web-browsers or document readers. Furthermore, their focus is on WinAPI calls, whereas in Figure 26 we instrument every call.

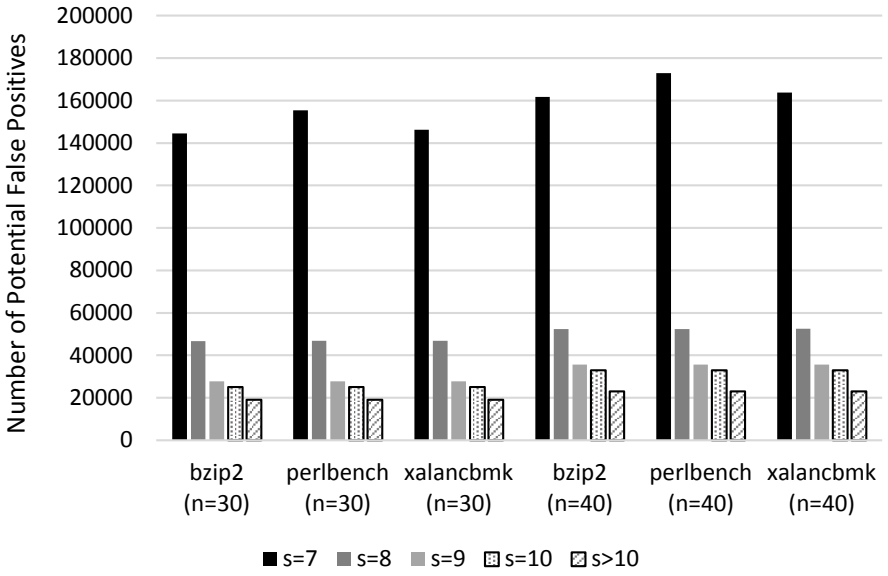


Figure 26: Potential false positives when the parameters for the consecutive sequences (s) and sequence length (n) are adjusted.

3.2.7 *Summary and Conclusion*

Without question, control-flow integrity offers a strong defense against runtime exploits. Its promise lies in the fact that it provides a general defense mechanism to thwart such attacks. Unfortunately, several pragmatic issues (most notably, its relatively high performance overhead), have limited its widespread adoption.

To better tackle the performance trade-off between security and performance, several coarse-grained CFI solutions have been proposed to date [150, 204, 202, 83, 47]. These proposals all use relaxed policies, *e. g.* allowing returns to target any instruction following a call instruction.

While many advancements have been made along the way, all to often the relaxed enforcement policies significantly diminish the security afforded by Abadi et al. [4]'s seminal work. This realization is a bit troubling, and calls for a broader acceptance that we should not sacrifice security for small performance gains. Doing so simply does not raise the bar high enough to deter skillful adversaries. Indeed, our own work shows that even if coarse-grained CFI solutions are combined, there is still enough leeway to mount reasonable and Turing-complete code-reuse attacks. Our hope is that our findings will raise better awareness of some of the critical issues when designing robust CFI mechanisms, all-the-while re-energizing the community to explore more efficient solutions for empowering CFI.

3.3 RELATED WORK

In this section, we provide a comprehensive overview of related code-reuse attacks. Note that we focus in this section our related work discussion on attack techniques and only partially discuss countermeasures. We turn our attention to defensive techniques in the subsequent chapters and elaborate on related work that aims at defending against code-reuse attacks in Section 4.4 and Section 5.5.

3.3.1 *Evolution of Code-Reuse Attacks*

Originally, code-reuse attacks have been introduced by Solar Designer in 1997 [174] as a consequence to the non-executable stack patch. The main idea is to transfer control to a critical function such as *execve()* or *system()* residing in the standard UNIX C library libc to bypass a non-executable stack. As we already described in Section 2.1.5.1, this attack technique undermines data execution prevention and is usually referred to as return-into-libc attack. Whereas the original return-into-libc attack allowed only invocation of a single function, Nergal demonstrated that techniques such as esp lifting and frame faking can be leveraged to allow chained function calls inside a return-into-libc attack [143]. However, chained function calls are challenging on processor platforms where arguments are passed via registers rather than on the stack. A prominent example is the 64-Bit Intel processor version x86-64. To tackle this challenge, Krahmer [118] introduced a technique, coined borrowed code chunk exploitation, allowing an attacker to execute just a short sequence inside a function to initialize the necessary arguments.

Shacham [169] generalizes the idea of borrowed code chunk exploitation and demonstrated for the first time that code-reuse attacks allow arbitrary and Turing-complete malicious program actions by combining code sequences residing in shared libraries. The principle of return-oriented programming [160] has been applied and explored on many processor architectures. The original attack targeted Intel x86 architectures [169] and exploited mainly the so-called unintended instruction sequences which can be invoked on Intel due to variable-length instructions and unaligned memory access (cf. Section 2.1.5.2). Subsequent work demonstrated that Harvard-based architectures — where code and data is strictly separated from each other — cannot prevent return-oriented programming attacks. To this end, Francillon and Castelluccia [79] leverage return-oriented programming to inject arbitrary malware on an Atmel AVR-powered sensor. Further, Buchanan et al. [31] apply return-oriented programming to the RISC-based architecture SPARC, where no unintended code sequences exist by design. In particular, they introduce a compiler that automatically constructs return-oriented exploits. In a similar vein, return-oriented programming has been shown on other architectures including PowerPC-based Cisco routers [123] and ARM-based mobile devices [117, 104]. As real-world example, Checkoway et al. [40] even demonstrate a return-oriented programming exploit on z80-powered voting machines (Harvard architecture) to shift votes.

Hund et al. [102] go one step further: they present the first compiler that automatically identifies return-oriented gadgets in a given binary and compiles (based on the gadget set) return-oriented programs. In particular, they construct a kernel rootkit that entirely

leverages return-oriented programming to undermine kernel integrity protection mechanisms. Interestingly, the evaluation of the return-oriented compiler reveals that quicksort executes more than 100 times slower when entirely implemented as return-oriented program. Unfortunately, none of the countermeasures proposed to date has further investigated the tremendous performance overhead of return-oriented programming which could potentially be exploited to detect return-oriented programming execution.

On the one hand, return-oriented programming raised a lot of academic and industrial research. On the other hand, no real-world exploits using return-oriented programming have been discovered until 2010. We believe that this is due to the fact that many PC platforms still did not strictly enforce data execution prevention thereby allowing attackers to launch conventional code injection attacks. However, in 2010, the first return-oriented exploit targeting Adobe PDF has been discovered [110]. From there on, a number of return-oriented exploits have appeared [91, 46, 128, 195].

3.3.2 *Jump-Oriented Programming*

All conventional return-oriented programming attacks discussed so far are based on return instructions and thus can be defeated by return address checkers. These tools or compiler extensions ensure the integrity of return addresses, which are corrupted through the conventional return-oriented programming attack [81, 48, 95, 49, 61]. However, Checkoway and Shacham [39] propose a new code-reuse attack that does not require any return instruction. Instead, the attack exploits indirect jump instructions. The attack is based on the *"Bring your own pop jump (BYOPJ)"* paradigm which assumes that a special `POP-JMP` sequence is either available in the target program or in one of its libraries. However, such a sequence is rarely found in ordinary program code. Our BLX-attack demonstrates that similar attacks can be mounted on ARM-powered platforms through the `BLX` instruction. In addition, we generalize the attack technique presented in [39] by introducing our update-load-branch sequence avoiding the presence of a `POP-JMP` sequence. Independent to our research, Bletsch et al. [26] introduce jump-oriented programming (on x86), a code-reuse attack that requires no return instructions and no `POP-JMP` sequence. For the latter, Bletsch et al. [26] leverage a generic dispatcher gadget to transfer control to the subsequent code sequence. In fact, the dispatcher gadget is similar to our update-load-branch sequence but targets Intel x86. Chen et al. [43] leverage jump-oriented programming to construct a rootkit that does not execute any return instruction.

More distantly related to return-oriented programming is the concept of JIT-spraying attacks [24]. These attacks allow an adversary to return to benign code the she injected via a script. This is achieved by forcing a JIT-compiler to allocate new executable memory pages with attacker-defined code that encapsulates dangerous unintended instruction sequences. Since scripting languages do not permit an adversary to directly program x86 shellcode, the attacker must carefully construct the script so that it contains useful gadgets in the form of unintended instruction sequences. For instance, Blazakis [24] suggests using XOR operations where the immediate operand to the XOR instruction embeds the

malicious instructions. In a recent work, Lian et al. [122] demonstrate that JIT-spraying attacks are also applicable to architectures that are based on an ARM processor.

3.3.3 *Gadget Compilers*

Gadget compilers ease the adversary's job of identifying gadgets in a given binary, and constructing a return-oriented exploit thereof. We already mentioned two of these gadget compilers: Buchanan et al. [31] introduce a return-oriented exploit compiler for SPARC, and Hund et al. [102] a gadget compiler that automatically identifies gadgets and compiles a return-oriented exploit for x86. However, these two gadget compilers focus on code sequences ending in a return instruction. A gadget compiler that entirely focuses on constructing jump-oriented exploits is presented by Chen et al. [44]. The compiler targets x86-compiled code and leverages the so-called combinational gadget terminating in a `CALL-JMP` sequence to invoke a system call in a jump-oriented attack. Whereas the previously discussed gadget compilers focused on a particular processor platform, Dullien et al. [70] introduce a gadget discovery tool that operates platform-independent by decompiling assembler instructions to an intermediate language called REIL.

Lastly, Schwartz et al. [164] present the Q compiler. This compiler automates the entire process of identifying gadgets, assembling a return-oriented exploit, and hardening existing exploits that fail due to code randomization or data execution prevention. Interestingly, the Q compiler is based on semantic definitions, *e. g.* it considers $\texttt{reg}_1 \leftarrow \texttt{reg}_2 * 1$ as a data movement gadget rather than a multiplication gadget allowing Q to compensate missing gadget types. This technique allows Q to generate return-oriented exploits on a small code base (*e. g.* 20KB code) that does not per-se contain all gadget types. Related to the small code base leveraged by Schwartz et al. [164], Homescu et al. [98] demonstrate that a Turing-complete gadget set can be derived from so-called micro-gadgets, *i. e.* code sequences that only consist of 2 to 3 Bytes. The probability of finding these very short sequences among different binaries is higher than compared to complex gadgets.

3.3.4 *Code Reuse in Malware*

Code-reuse attack techniques have been also leveraged to hide malicious program behavior from static analysis tools or non-ASCII filters. Lu et al. [125] leverage a return-oriented decoder that only consists of code pointers that represent valid ASCII printable characters. The decoder takes as an input the encoded code-reuse exploit and outputs the actual code-reuse exploit at application runtime. Similarly, [188] successfully deployed code-reuse attack techniques to undermine the application vetting process conducted by Apple. To this end, they developed an application that contains an intended buffer overflow vulnerability which gets exploited under certain conditions (*e. g.* after the app is installed on the user's device). Once the control-flow is hijacked, the exploit payload leverages return-oriented programming to invoke private iOS APIs. Lastly, Vogl

et al. [185] introduce persistent data-only malware, *i. e.* malware that leverages code-reuse attack techniques to realize a persistent rootkit. With respect to malware detection, Stancill et al. [178] present an analysis system that detects return-oriented programming payloads in malicious files. Their system, efficiently analyzes incoming documents (PDF, Office, or HTML files), and detects whether they contain a return-oriented programming payload.

In a different domain, code-reuse techniques have been deployed to hide secret program actions: Lu et al. [126] introduce program steganography that is based on executing unintended code sequences to hide program actions from static analysis tools. However, leveraging code-reuse attack techniques for a legitimate and benign purpose will eventually lead to false alarms in tools that aim at detecting and preventing code-reuse attacks.

3.3.5 *Code Reuse and Code Randomization*

A well-known approach to defend against code-reuse attacks is to randomize the code layout of an application. As a consequence, an adversary can only guess where useful code resides in memory. Today's operating systems enforce address space layout randomization (ASLR) to randomize the base address of code and data segments (cf. Section 2.2). However, Shacham et al. [170] demonstrate that the entropy on 32-Bit systems is too low allowing brute-force attacks. The main idea of such attacks is to guess the address of a single library function residing in `libc`. For instance, the attacker attempts to execute the *sleep()* function to halt program execution for a pre-defined time. However, such attacks only succeed if no re-randomization is performed after each guess. Recently, Bittau et al. [23] demonstrate that even 64-Bit based systems – where the randomization entropy is significantly larger – can be compromised by means of brute-force attacks on code randomization. The introduced technique, denoted as BROP (blind ROP), probes on byte granularity the stack to eventually disclose gadgets based on whether the target server crashed or execution has continued. However, similar to the attack presented in [170], BROP only applies to server applications that do not re-randomize after a crash.

As Fresi Roglia et al. [84] and Schwartz et al. [164] demonstrate, code randomization is only effective if it is applied to every code section mapped into the address space of an application. Both attacks leverage position-dependent (non-randomized) code of the main (Linux) executable binary to instantiate a code-reuse attack. Whereas Schwartz et al. [164] directly derive a Turing-complete gadget set from the non-randomized code, Fresi Roglia et al. [84] invoke a couple of code sequences to disclose and overwrite function pointers in the global offset table (GOT) [35], and start a conventional return-into-libc attack subsequently. However, since the latter attack relies on static offsets, it can be prevented by randomizing the order of functions residing in `libc`. Wang et al. [190] tackle this challenge and dynamically identify function pointers in the GOT. In particular, their goal is to disclose function pointers that point to a function that includes a direct call to a system call. Hence, even if the order of functions is randomized, the single disclosed function pointer already allows invocation of a dangerous system call. However, the shown attack still requires non-randomized code.

Fine-grained code randomization [120] applied to all code segments defends against these attacks by randomizing function order [114], basic blocks [191, 63], instructions [97], and register allocation [149]. In particular, until recently, it was widely believed that fine-grained code randomization prevents an attacker from determining a chain of useful code sequences based on a single leaked function pointer. However, our novel just-in-time code-reuse attack demonstrates that an attacker can exploit scripting facilities to disclose the randomized code of a large number of memory pages on-the-fly based on a single leaked function pointer [172]. In Chapter 5, we elaborate in detail on fine-grained code randomization schemes, their limitations, and emerging technologies to defend against just-in-time code reuse.

3.3.6 *Code-Reuse Attacks against Coarse-Grained CFI*

Concurrent and independent to our work, several research groups have investigated the security of coarse-grained CFI solutions [108, 89, 90, 37, 162]. However, our analysis differs from these works as we examine the security of a combination of coarse-grained CFI policies irrespective of when the CFI check occurs. For instance, the attacks shown in [89, 37, 162] are prevented by our combined CFI policy which monitors the sequence length at any time in program execution. Furthermore, unlike these works, we systematically show the construction of a Turing-complete gadget set based on a weak adversary that has only access to one standard shared Windows library. On the other hand, concurrent work also investigates some other interesting attack aspects: Göktas et al. [89] demonstrate attacks against CCFIR [202] using call-preceded gadgets to invoke sensitive functions via direct calls; Carlini and Wagner [37] and Schuster et al. [162] show flushing attacks that eliminate return-oriented programming traces before a critical function is invoked.

3.4 SUMMARY AND CONCLUSION

In this chapter, we presented a new runtime exploitation technique to compromise software running on an ARM-based device. Our attack is based on the principles of return-oriented programming but does not use any return or function epilogue sequences. Instead, our attack chains together instruction sequences from existing libraries by means of the indirect subroutine call instruction `BLX` (Branch-Load-Exchange). Hence, it cannot be detected by return address checkers. In fact, we showed that such attacks are Turing-complete allowing an attacker to induce arbitrary malicious computation. Return-oriented programming without returns underlines the need that software programs need to be protected in a more generic approach such as control-flow integrity (CFI) which restricts the program's execution path to a pre-defined control-flow graph. Unfortunately, several pragmatic issues (most notably, its relatively high performance overhead), have limited its widespread adoption. To better tackle the performance trade-off between security and performance, several coarse-grained CFI solutions have been proposed. However, all too often the relaxed enforcement policies significantly diminish the security afforded by Abadi et al. [4]'s seminal work. Indeed, our own work shows that even if coarse-grained CFI solutions are combined, there is still enough leeway to mount reasonable and Turing-complete code-reuse attacks.

4

ADVANCES IN CONTROL-FLOW INTEGRITY DEFENSES

In this chapter, we turn our attention to defensive techniques against code-reuse attacks. In particular, we focus on defenses that are based on the principle of control-flow integrity (CFI) and resist the attacks that we presented in Chapter 3. In contrast to a variety of ad-hoc solutions that tackle a specific vulnerability or only a certain class of runtime exploits, CFI provides a general solution against code-reuse attacks. First, we present the design and implementation of our CFI framework for mobile devices (Section 4.1). Second, we demonstrate that our CFI framework enables fine-grained application sandboxing on iOS (Section 4.2). Next, we present the design, implementation, and evaluation of HAFIX, our hardware-assisted flow integrity extensions enabling fine-grained CFI on Intel Siskiyou Peak and SPARC LEON3 (Section 4.3). Lastly, we elaborate on related work on CFI (Section 4.4) and conclude this chapter (Section 4.5).

4.1 MOBILE CONTROL-FLOW INTEGRITY

Smartphones and tablet computers are becoming ubiquitous, and the sales figures for both types of devices are growing rapidly. Probably the most important reason for this popularity is the availability of a large number of mobile applications, ranging from simple games over messaging apps to office apps. Consumers can easily download these apps from app stores, and then install and use them. The recent numbers of published apps have already reached impressive thresholds, e.g. in July 2014 more than one billion apps were available on Google Play and Apple's App Store [179].

On the other hand, privacy and security concerns arise because apps can also access personal/sensitive information or trigger critical services such as starting a phone call. In addition, many current systems offer a large attack surface because they still run software programs implemented in unsafe languages such as C or C++. In particular, modern mobile platforms like Apple's iOS and Google's Android have recently become appealing attack targets (*e. g.* [104, 137, 105, 113, 192, 135]), and increasingly leak sensitive information to remote adversaries (*e. g.* the SMS or contacts database [105, 135]).

A general approach to mitigate runtime exploits is the enforcement of control-flow integrity (CFI) [4] (cf. Section 2.3). Surprisingly, and to the best of our knowledge, there exist no CFI framework for mobile platforms.

In this section, we present the design and implementation of MoCFI (Mobile CFI), a CFI enforcement framework for mobile platforms. Specifically, we focus on the ARM architecture since it is the standard platform for smartphones and tablets. The implementation of CFI on ARM is often more involved than on desktop PCs due to several subtle architectural differences that highly influence and often significantly complicate a CFI solution: (1) the program counter is a general-purpose register, (2) the processor

may switch the instruction set at runtime, (3) there are no dedicated return instructions, and (4) control-flow instructions may load several registers as a side-effect.

Although our solution can be deployed to any ARM-based mobile platform, we chose Apple's iPhone for our reference implementation because of three challenging issues: First, the iPhone platform is a popular target of control-flow attacks due to its use of the Objective-C programming language. In contrast, Android is not as prone to runtime exploits, because applications are mainly written in the type-safe Java programming language. Second, iOS is closed-source meaning that we can neither change the operating system nor can we access the applications' source code. Third, applications are encrypted and signed by default.

Contribution. To the best of our knowledge, MoCFI is the first general CFI enforcement framework for mobile platforms. Solutions like NativeClient (NaCl) for ARM [166] only provide a compiler-generated sandbox. NaCl needs access to source code, and currently does not support 16-Bit THUMB code which is typically used in modern mobile applications. In contrast, our solution operates on binaries and can be transparently enabled for individual applications. MoCFI allows us to apply CFI onto mobile applications with commonly unavailable source code.

To this end, we first implemented a system to recover the control-flow graph (CFG) of a given iOS application in binary format. In particular, we extend PiOS [72] (a data-flow analysis framework) to generate the CFG. Based on this information, we perform control-flow validation routines that are used during runtime to check if instructions that change the control-flow are valid. Our prototype is based on library injection and in-memory patching of code which is compatible to memory randomization, static code signing, and encryption. Finally, our approach only requires a jailbreak for setting a single environment variable, installing a shared library, and allowing our library to rewrite the application code during load-time.

For performance evaluation, we measured the overhead MoCFI introduces as well the average overhead for typical applications and worst-case scenarios. The evaluation shows that our implementation is efficient and prevents return-oriented exploits.

Section Outline. The remainder of this section is organized as follows: after recalling the basics of the iOS smartphone operating system and its security architecture in Section 4.1.1, we present the technical challenges when applying CFI to mobile platforms in Section 4.1.2. Next, we present the design and implementation of MoCFI in Section 4.1.3 and 4.1.4. In Section 4.1.5, we provide a security discussion and explain current limitations. Lastly, we evaluate the performance of MoCFI in Section 4.1.6, and conclude in Section 4.1.7.

4.1.1 *Background on iOS*

Apple iOS is a closed and proprietary operating system designed for mobile Apple devices such as iPhone, iPad, and iPod Touch. iOS applications and the main iOS system libraries are implemented in Objective-C which is a C-based object-oriented language.

As iOS defers many decisions from compile-time to runtime, a so-called Objective-C runtime system is required which is linked to every process.

In general, iOS provides several security features such as application sandboxing, mandatory code signing, data encryption, data execution prevention, and memory randomization. In the following, we elaborate on the underlying security architecture and describe each of the enforced security features.

A well-known security feature deployed in iOS is code signing. It ensures that only Apple-signed software can be executed on an iOS device. To bypass this restriction, users can jailbreak (root) their devices to install arbitrary non-approved Apple software. Apple approves signed applications after a vetting process. Although the implementation details of application vetting are not public, Apple states that it *reviews all apps to ensure they are reliable, perform as expected, and are free of offensive material* [8]. Apple also deploys an AES-256 hardware crypto engine to encrypt the file system of an iOS device.

An abstract view of the security architecture to realize application sandboxing on iOS is shown in Figure 27. iOS deploys sandboxing to isolate applications from each other, and to control access of applications to the operating system. In particular, we distinguish components on three software layers: (1) the kernel layer which provides basic system services (file system and network) and a kernel module to realize application sandboxing, (2) the Objective-C framework layer and a privacy setting service, and (3) the application layer where third-party and built-in apps are executing.

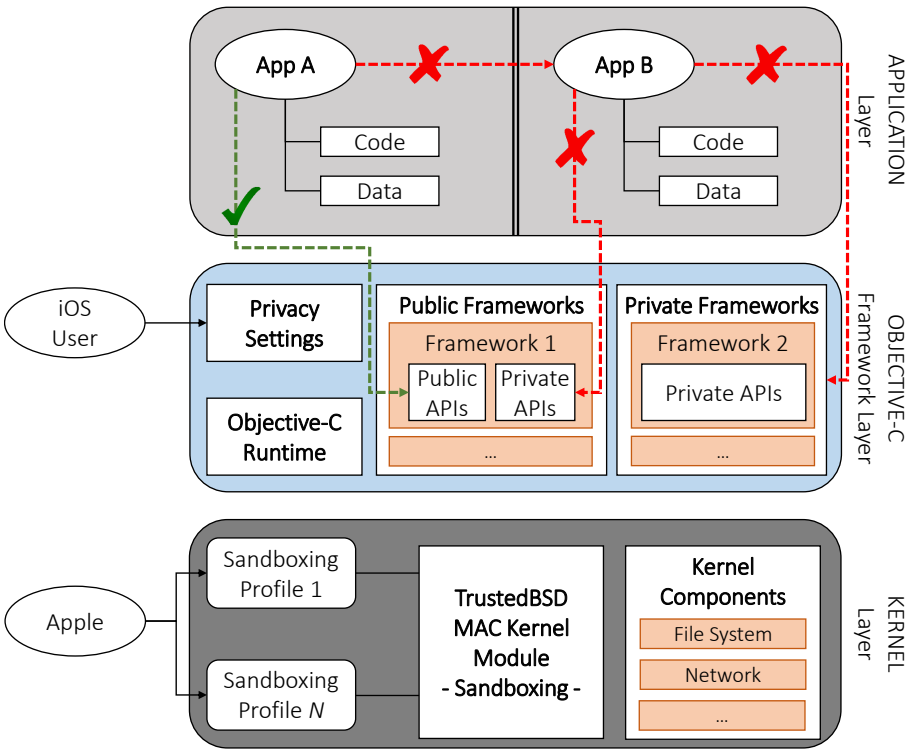


Figure 27: Basic iOS architecture to enforce application sandboxing

The main component to enforce application sandboxing resides in the iOS kernel, namely a TrustedBSD mandatory access control (MAC) module. This kernel module enforces sandboxing at the level of system calls and directory paths. Further, sandboxing is driven by sandboxing profiles which are pre-defined by Apple. The profiles consist of access control lists (ACLs) that either deny or grant access to certain system calls and file paths.

Apple defines a single sandboxing profile for third-party apps. Hence, all apps execute with the same privilege level. In particular, this profile prohibits App A to access code or data from other applications like App B (see Figure 27).

Apart from the TrustedBSD kernel module, there are several restrictions imposed by Apple indirectly related to application sandboxing. Apple distinguishes between public and private frameworks. A framework refers to a shared library and associated resources to support the framework (*e. g.* images, header files). Of particular interest are the frameworks located in the Objective-C framework layer as they provide access to main phone facilities (such as Location, SMS, Calendar, Contacts, etc.). The private frameworks are only accessible by system applications. On the other hand, the public frameworks can be accessed by every third-party application.

Although third-party applications are only allowed to access public APIs of a public framework, there is no fundamental operating system mechanism that prevents the use of private APIs. Instead, Apple relies on the application vetting process to discover such unauthorized access requests.

Finally, since iOS version 6, iOS allows users to specify privacy settings on a per-app basis. Typically, iOS apps have by default access to private information such as contacts, device IDs, keyboard cache, or location. In order to restrict the access to this information, iOS users can arbitrarily configure privacy settings. In fact, this allows users to specify restrictions on some selected privacy-related *public* APIs. However, there is yet no option to enforce access control on non-privacy related public APIs as well as pure private APIs.

Runtime Protection Mechanisms. Since iOS v2.0, Apple enables the $W \oplus X$ security model (cf. Section 2.1.4) to prevent code injection. Furthermore, iOS deploys dynamic code signing enforcement (CSE) at runtime [207]. In contrast to systems that only enable $W \oplus X$ (*e. g.* Windows or Linux), CSE on iOS prevents an application from allocating new memory marked as executable. On the other hand, CSE at runtime in conjunction with $W \oplus X$ has practical drawbacks because it does not support self-modifying code and code generated by just-in-time (JIT) compilers. Therefore, iOS provides the so-called *dynamic-codesigning* entitlement that allows applications to generate code at runtime. At the time of writing, only the Mobile Safari Browser and full-screen web applications are granted the *dynamic-codesigning* entitlement. However, neither CSE at runtime nor $W \oplus X$ can prevent return-oriented programming attacks that only leverage existing and signed code pieces. Since iOS v4.3, address space layout randomization (cf. Section 2.2) is supported by default.

4.1.2 *Challenges*

The adoption and adaption of control-flow integrity (CFI) [4] to mobile platforms involves several difficulties and challenges.

The technical challenges are due to the architectural differences between ARM (RISC design) and Intel x86 (CISC design), and because of the specifics of mobile operating systems. These highly influence and often complicate a CFI solution as we argue in the following.

- **No dedicated return instruction**: As already mentioned in Section 3.1.1, ARM does not provide dedicated return instructions. Instead, any branch instruction can be leveraged as a return. Moreover, returns may have side-effects, meaning that the return does not only enforce the return to the caller, but also loads several registers within a single instruction. Hence, in contrast to Intel x86, a CFI solution for ARM has to handle all different kinds of returns, and has to ensure that all side-effects of the return are properly handled.

- **Multiple instruction sets**: CFI on ARM is further complicated by the presence of two instruction sets (ARM and THUMB), which can even be interleaved. Hence, it is necessary to distinguish between both cases during the analysis and enforcement phase, and to ensure the correct switching between the two instruction sets at runtime.

- **Direct access to program counter**: Another difference is that the ARM program counter `pc` is a general-purpose register which can be directly accessed by a number of instructions, *e. g.* arithmetic instructions are allowed to load the result of an arithmetic operation directly into `pc`. Furthermore, a load instruction may use the current value of `pc` as base address to load a pointer into another register. This complicates a CFI solution on ARM, since we have to consider and correctly handle all possible control-flow changes, and also preserve the accurate execution of load instructions that use `pc` as base register.

- **Missing binary rewriter**: Moreover, in contrast to Intel x86, no binary instrumentation framework exists for the ARM platform which allows the automatic generation of the control-flow graph and the memory adjustment when new instructions are inserted into a binary. For instance, the CFI proposal for Intel x86 [4] depends on the non-public binary instrumentation framework Vulcan [71] which enforces these operations automatically.

- **Application signing and encryption**: Mobile operating systems typically feature application encryption and signing. Since the traditional CFI approach [1] performs changes on the stored binary, the signature of the application cannot be verified anymore.

- **Closed-source OS**: Several mobile operating systems (such as iOS) are closed-source. Hence, we cannot change the actual operating system to deploy CFI on

smartphones. Moreover, end-users and even App Store maintainers (*e. g.* Apple's App Store) have no access to the applications' source code.

4.1.3 *Design of MoCFI*

In this section, we present the design and implementation of MoCFI, our generic framework to mitigate code-reuse attacks on mobile platforms based on the principle of control-flow integrity (CFI). The general architecture and workflow of MoCFI is shown in Figure 28. In general, we distinguish in our design two phases: static analysis and runtime enforcement. The static analysis phase comprise tools to perform an initial analysis of the compiled application binary. In particular, MoCFI deploys a preprocessor that decrypts and disassembles the encrypted application (Step ❶). Afterwards, we perform a thorough static analysis on the application binary (Step ❷): we generate the control-flow graph (CFG) of the application and employ a branch detector to identify all branches contained in the binary and extract all information that is necessary to enforce CFI at runtime. Note that the static analysis needs to be performed only *once* after compilation and can be integrated as an additional step in the deployment phase of a typical mobile application.

In the runtime enforcement phase, we monitor the application while it is executing using our novel MoCFI shared library that rewrites the application binary in memory at load-time (Step ❸), and enforces control-flow integrity (CFI) at execution-time (Step ❹).

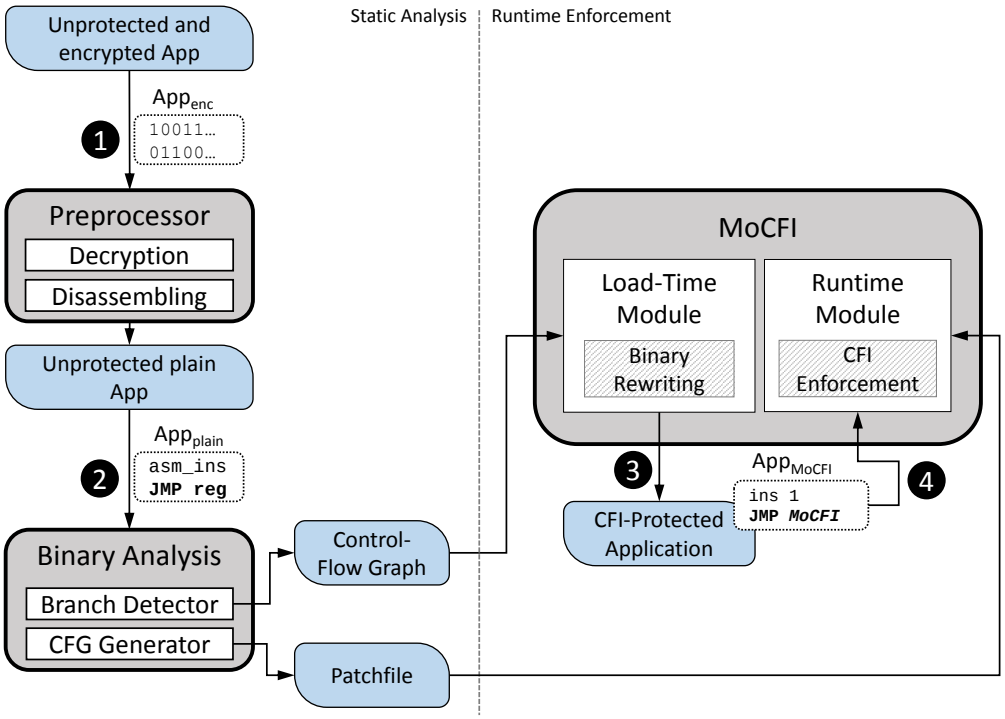


Figure 28: General design and workflow of MoCFI

Although the depicted design applies in general to all CFI solutions, our design requires a number of changes, mainly due to (i) the architectural differences between ARM and Intel x86, (ii) the missing binary rewriter and automatic graph generation for ARM, and (iii) the specifics of mobile operating systems. In the following, we describe each involved system component and our approach in more detail.

Preprocessor. The first step of our static analysis phase is performed within the preprocessor component, which has mainly two tasks: (1) decrypting, and (2) disassembling the target application binary. In particular, we faced the challenge that smartphone applications are often encrypted by default (*e. g.* iOS applications). We thus obtain the unencrypted code of a binary through *process dumping* [72]. For disassembling the application binary we deploy standard disassembler tools that support the ARM architecture.

Binary Analysis. The original CFI work for Intel x86 [1] employs the binary instrumentation framework Vulcan [71] to automatically derive the CFG and to statically rewrite an application binary. However, such a framework does not exist for ARM. Hence, we developed own techniques to accurately generate the CFG. After our preprocessor decrypted and disassembled the application binary, we identify all relevant branches contained in the binary. By relevant branches, we refer to branch instructions that an adversary may exploit for a control-flow attack. These mainly comprise indirect branches, such as indirect jumps and calls, and function returns. Moreover, we include direct function calls to correctly validate function returns, *i. e.* to be able to check if a function return targets the original caller. We do not instrument direct jump instructions for obvious reasons: the target address for these are fixed (hard-coded), and cannot be manipulated by an adversary. Finally, we store meta information for each indirect branch and function call (*e. g.* instruction address, length, type, etc.) in a separate patchfile.

Based on the result of the branch detector, we generate the CFG by static tools that we developed ourselves. In particular, our static tools calculate possible target addresses for each indirect branch. Finally, a binary representation of the CFG is stored in a separate file (denoted as Control-Flow Graph), which is linked to the smartphone application at runtime.

MoCFI Load-Time Module: Binary Rewriting. The binary rewriting engine is responsible for binding additional code to the binary that checks if the application follows a valid path of the CFG. Typically, one replaces all branch instructions in the binary with a number of new instructions that enforce the control-flow checks [1]. However, replacing one instruction with multiple instructions requires memory adjustments because all instructions behind the new instructions are moved downwards. The Intel x86 approach uses the Vulcan binary instrumentation framework [71] which automatically accomplishes this task. However, memory adjustment without a full binary rewriting framework requires high implementation efforts.

Due to the limited possibilities to change smartphone binaries (due to code signing) and the missing full binary rewriter, we opted for the following rewriting approach (which has been originally proposed by Winwood et al. [196]). At *load-time* we replace

all relevant branches (based on the extracted rewriting) with a single instruction: the so-called *dispatcher instruction*. The dispatcher instruction redirects the program's control-flow to a code section where the CFI checks reside, namely to the runtime module of our MoCFI shared library.

This approach also raises several problems: First, accurate branch instructions have to be implemented that are able to jump to the correct CFI check. Second, the CFI checks require information from where the dispatch originated. As we will demonstrate, our solution efficiently tackles the above mentioned problems.

MoCFI Runtime Module: Control-Flow Integrity Enforcement. The key insight of CFI is the realization of control-flow validation routines. These routines have to validate the target of every branch to prevent the application from targeting an instruction sequence that violates CFI. Obviously, each branch target requires a different type of validation. Whereas the target address of an indirect jump or call can be validated against a list of valid targets, the validation of function returns requires special handling because return addresses are dynamic and cannot be predicted ahead of time. To address this issue, MoCFI reuses the concept of shadow stacks that hold valid copies of return addresses [48, 81], while the return addresses are pushed onto the shadow stacks when a function call occurs.

4.1.4 *Implementation*

In this section, we present the implementation details of MoCFI. Our reference implementation of MoCFI targets iOS version 4.3.1, which was at the time we conducted this research project the most recent iOS version. The static analysis tools comprise 842 lines of code written in the IDC language, which is a scripting language that is deployed by the well-known disassembler IDA Pro. The runtime module of MoCFI has been developed in Objective-C and assembly language using Xcode 4. Specifically, the runtime module of MoCFI is implemented as a shared library consisting of 1,430 lines of code in total.

Note that our prototype implementation currently protects the application's main code, but no dynamic libraries that are loaded into the process. We leave support for shared libraries open to future work. However, it is straightforward to extend MoCFI accordingly, there are no new conceptional obstacles to overcome.

We now describe how we generate the CFG and the patchfile of an iOS binary (Section 4.1.4.1). Next, we present the implementation details of our MoCFI load-time and runtime module (Section 4.1.4.2 and 4.1.4.3).

4.1.4.1 *Static Analysis*

As we mentioned in Section 4.1.2, one challenge for enforcing CFI on iOS stems from the fact that Apple restricts access to source code. Recall that even developers only submit the applications' binary files when uploading applications on the App Store. To tackle this challenge, we apply our static analysis techniques directly on iOS binaries. This allows us to generate the application's CFG and identify all branch instructions inside

the binary. We need the former one to validate if a branch follows a valid execution path in the CFG, while the latter one is required to guarantee accurate binary rewriting. To perform these tasks, we use the IDA Pro v6.0 Disassembler that reliably disassembles ARM and THUMB code. On top of IDA Pro, we implemented scripts to automate the analysis and extract the necessary information from a given iOS binary.

Patchfile Generation. As shown in Figure 28 in Section 4.1.3, Step ❷ involves the generation of rewriting information for each individual binary. This information is required by the load-time module of MoCFI to instrument each branch instruction with a new instruction that redirects execution to a CFI validation routine residing in the runtime module. In order to identify all relevant branch instructions, we evaluate each instruction belonging to the text segment and check if the instruction is relevant in the context of CFI. Next, we perform a fine-grained instruction analysis and store the derived meta information (e.g. instruction address, mode, length, type, etc.) in the patchfile. By bundling the patchfile with the application, we can protect its integrity, as all application bundle contents are code-signed.

Generation of the Control-Flow Graph (CFG). Typically, a CFG consists of (i) nodes that represent basic blocks of an application, and (ii) edges connecting the various nodes. The edges are due to any kind of branch instruction. However, as we argued in Section 4.1.3, for our purposes it is sufficient to derive a CFG for those basic blocks that terminate in an indirect branch. Furthermore, function returns cannot be resolved statically as their destination addresses depend on the program's dynamic state. Hence, for our CFG, we first focus on basic blocks that terminate in an indirect call or jump instruction.

To determine the valid set of destination addresses for an indirect jump, we trace back all registers the indirect jump depends on. As an example, on ARM an indirect branch could implemented as follows: `LDR pc,[r0,r1,LSL #2]`. The resulting target address is calculated at runtime by adding `r0` to `r1` multiplied by four. Hence, we need to determine the possible values of `r0` and `r1` to identify the valid set for this indirect jump. Similarly, we trace back the possible values of a register that an indirect call depends on (*e. g.* `r3` for `BLX r3`).

Indirect jumps are typically emitted by the compiler when a switch-case statement is used, *e. g.* depending on the runtime value of a variable, code of a specific case-statement is executed. The control-flow is redirected to the case-statement by means of an indirect jump instruction. In our static analysis, we identify such indirect jumps and record all valid target addresses in our CFG file.

Another challenge with regards to CFG generation arises from the fact how iOS and particularly Objective-C realizes method calls: at the machine-level, any method call on an Objective-C object is dispatched via a direct call to a a generic message handling routine called `objc_msgSend`. The name of the method to be called (the so-called selector) and the class instance that should receive the request are provided as parameters to `objc_msgSend`. As an adversary might be able to alter these parameters to call an arbitrary method of her choice, we need to track valid parameters for each method call and record them in our CFG file. For this, we built upon PiOS [72] and former

reverse-engineering work on iOS [69, 134] to generate the necessary CFG information for `objc_msgSend` calls.

Limitations. Resolving indirect jumps and calls is challenging as the value they depend on needs to be determined during static analysis. Unfortunately, recovering all indirect branches is not always feasible. That said, we need to have a fallback mechanism in case an indirect branch is not per-se resolvable. Note that this is not a specific limitation of MoCFI, but affects all CFI solutions.

To tackle this challenge, we decided to enforce reliable heuristics that still constrain the possible set of valid control-flow destinations but at the same time ensure correct program execution. In particular, we observed that most indirect jumps operate inside the scope of the function they are executed. In other words, an indirect jump targets a destination address inside the currently executing function. Hence, for indirect jumps we cannot resolve, we can restrict the target address to the bounds of the function. Similarly, a reliable heuristic can be applied to unresolvable indirect calls: we simply restrict the target address of an indirect call to the beginning of a function. This still allows an adversary to redirect execution to an arbitrary function, but prevents the adversary from jumping into the middle of a function as typically done in return-oriented programming attacks.

4.1.4.2 *Load-Time Module*

As shown in Figure 28, the MoCFI runtime enforcement consists of two components: the load-time and runtime module. We present implementation details for both components. However, we first start our implementation discussion on how we instantiate MoCFI as a runtime monitor to iOS applications.

MoCFI Instantiation. Recall that one implementation challenge stems from the fact that mobile operating system apply code signing to application (cf. Section 4.1.2). Hence, we need a mechanism to instrument an iOS application after the application's signature has been verified.

We tackle this challenge by implementing MoCFI as a shared library that instruments an application on-the-fly in memory (rather than on disk). To inject our shared library, we leverage standard library injection methods already provided by the operating system. On UNIX-based system, this mechanism is provided through the environment variable `LD_PRELOAD`. If this environment variable is set then the loader guarantees that the library to be injected is loaded before any other program binaries. On iOS, the same mechanism is supported via the `DYLD_INSERT_LIBRARIES` environment variable [9]. Hence, we simply set this environment variable to point to our MoCFI shared library. In particular, we set the environment variable for the iOS Springboard process so that every iOS application launched by the user is automatically protected by MoCFI.

After being loaded into the application's address space, MoCFI starts to implement the CFI enforcement by rewriting the code of the application in memory. We call this part of MoCFI the *load-time module* in contrast to the *runtime* module, which enforces the actual CFI checks.

In-Memory Instrumentation. In order to rewrite the target iOS application in memory, the load-time module needs to locate and read the associated patchfile (cf. Section 4.1.4.1). However, MoCFI cannot simply rewrite the application as iOS enforces data execution prevention, *i.e.* the target memory pages are non-writable. Typically, we only would need to invoke the *mprotect()* system call to change the permissions of the target memory pages. However, iOS prohibits us from invoking *mprotect()*. Fortunately, iOS still allows us to re-map memory pages using the *mmap()* [10] system call. Hence, we instrument an application using *mmap()* and set the page permission to non-writable after our load-time module has completed its rewriting. Note that the presence of *mmap()* does not allow an adversary to bypass MoCFI by reverting the MoCFI instrumentation. In order to do so, the control-flow of the application needs to be altered beforehand. However, this would lead to a CFI violation, which is in turn detected by MoCFI.

Trampoline Approach. Our binary rewriting engine overwrites the relevant control-flow instructions with the so-called *dispatcher instructions*. The dispatcher redirects the program flow to a short piece of assembler code, namely to the *trampoline*, which in turn transfers the execution to our MoCFI library (see Figure 29). Hence, the trampolines are used as bridges between the application we aim to protect and our MoCFI library.

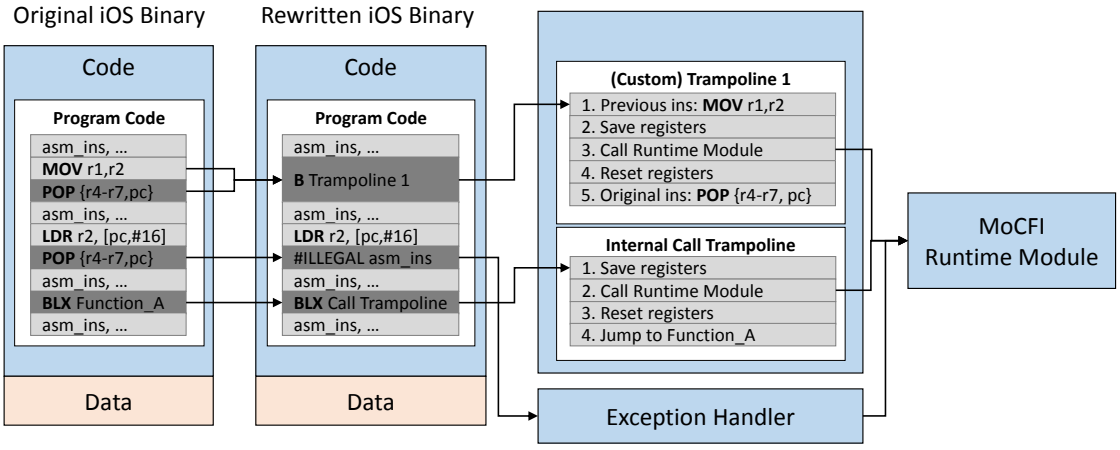


Figure 29: Trampoline approach

Specifically, we allocate dedicated trampolines for each indirect branch (*i.e.* indirect jumps/calls and returns), one generic trampoline for direct internal function calls (*i.e.* calls within the same code segment), and one generic trampoline for external function calls. Two example trampolines are shown in Figure 29: the first one (Trampoline 1) is used for a return instruction, while the second one (Internal Call Trampoline) handles a (direct) internal function call. In general, each trampoline saves the current execution state, invokes the appropriate MoCFI validation routine, resets the execution state, and issues the original branch. Due to the last step, we ensure that all registers are loaded correctly, even if the branch loads several registers as a side-effect, *e.g.* the replaced return `POP {r4-r7,pc}` is copied by our load-time module at the end of Trampoline 1.

Hence, we ensure that `r4` to `r7` are correctly loaded with values from the stack before the return address is loaded to `pc`.

Note that, depending on the replaced branch instruction, we allocate a THUMB or ARM trampoline to ensure the correct interworking between the two instruction sets. In the following, we present the different kinds of dispatcher instructions our solution utilizes. The specific assembler implementation of the different ARM/THUMB trampolines are described in [62, Appendix A].

ARM Dispatcher Instruction. As mentioned in Section 3.1.1, each ARM instruction has a fixed size of 32 bits. Hence, we can simply replace each relevant ARM control-flow instruction with a generic dispatcher branch (`B`) instruction. Effectively, this branch instruction allows us to address any location in the range of ±16MB. Consequently, our trampolines (see Figure 29) are allocated close to the code section we are instrumenting so that we never exceed the range of ±16MB. Theoretically, even though we have never noticed this case for the iOS applications we have tested, an application's code section may be larger than 16MB which would impede us from allocating trampolines in ±16MB memory range. In practice, this is very unlikely, but we could still search for empty code regions (padding bytes) to enlarge the branch target address range.

Note that direct function calls require a different dispatcher instruction as we need to ensure that the return address is correctly set. Hence, we use instead the `BLX` instruction as dispatcher which offers the same target address range of ±16MB. As mentioned in Section 3.1.1, `BLX` automatically loads the return address into the `lr` (link) register. On the other hand, for indirect function calls MoCFI uses again the `B` instruction as dispatcher instruction. In fact, we correctly set the corresponding return address into `lr` inside the CFI runtime module based on the fact whether the call is an external (library function) or internal call (main code function). For external calls, we let `lr` point to a specialized code piece of MoCFI to recognize when an external library call returns.

THUMB Dispatcher Instructions. THUMB instructions pose a challenge in our design as indirect branches such as returns only reserve 2 Bytes in memory. Replacing these instructions with direct 16-bit THUMB branch `B` instructions only allows us to address code in the area of ±512 Bytes. As this range is surely too small to allocate trampolines, we need to take a different approach. Specifically, we tackle this issue in MoCFI by replacing a 16 Bit indirect THUMB branch with a 32 Bit THUMB dispatcher instruction. However, this has the effect that we overwrite 2 Thumb instructions, *e. g.* in Figure 29 the original branch (`POP {r4-r7,pc}`) and the instruction preceding the branch (`MOV r1,r2`). To still preserve the program's semantics, we execute the latter one at the beginning of our trampolines (Step 1 in Trampoline 1).

Dispatching through Exception Handling. However, replacing 2 THUMB instructions is not possible if the instruction preceding the branch references the program counter or is itself a branch. For instance, `LDR r2,[pc,#16]` in Figure 29 uses the current value of `pc` to load a pointer. In such scenarios, we replace the indirect branch instruction with a pre-defined illegal instruction. This illegal instruction triggers an exception when executed. To catch this exception, we register an iOS exception handler inside MoCFI,

and subsequently invoke our CFI validation routines. As exception handling induces additional performance overhead, we only use it for exceptional cases. To further reduce the use of the exception handler, one could calculate the address from which `pc` is loaded in the static analysis phase, and replace the relevant load instruction with a new memory load instruction which could be placed at the beginning of the trampoline.

4.1.4.3 *Runtime Module*

An abstract view of the runtime module is shown in Figure 30: it mainly consists of dedicated validation routines for each branch type, where each type is represented with a rectangle in Figure 30. The validation routines have to validate the target of every branch to prevent the application from targeting a basic block beyond the scope of the CFG and the current execution path. Obviously, each branch target requires a different type of validation, as we will describe in the following.

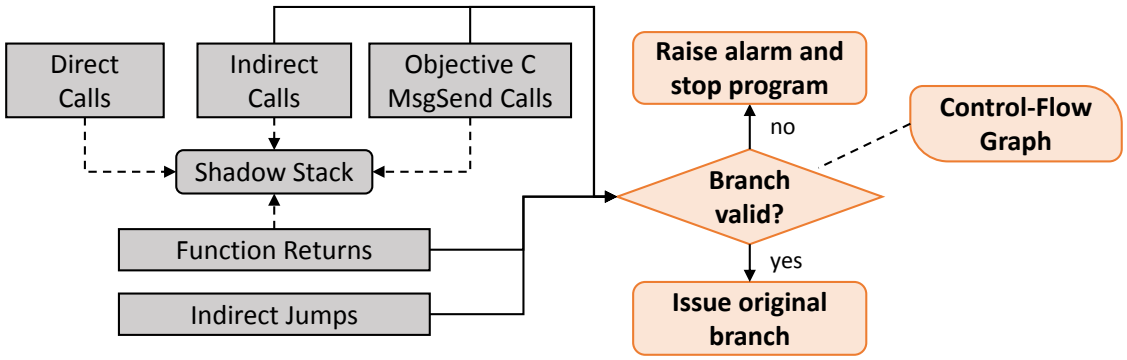


Figure 30: Overview of the runtime module

Function Calls and Returns. To prevent code-reuse attacks, we monitor all function calls and returns, and apply the shadow stack paradigm: whenever the program invokes a subroutine (through a direct, indirect, or dispatcher call), we copy the return address on a dedicated shadow stack. Upon function return, we compare the return address the program wants to use to the address stored on our shadow stack. Since function calls (through `BL` or `BLX`) automatically store the return address in `lr`, we simply need to push `lr` onto the shadow stack. Further, we maintain a separate shadow stack for each execution thread to support multi-threaded programs. Upon function return, we determine the return address the program aims to use and retrieve the required stack pointer offset from the patchfile.

As mentioned in Section 4.1.2, return instructions can be implemented in many different ways on ARM, and often involve the loading of several general-purpose registers. We ensure that all side-effects are correctly handled by issuing the original return at the end of the trampoline (as described in Section 4.1.4.2).

Indirect Jumps and Calls. As already discussed in Section 4.1.4.1, the set of valid target addresses for indirect jumps and calls has been either determined at static analysis

time or remains unknown. For those indirect jumps/calls whose set of target addresses is known, we simply check whether the target address the program attempts to use is in the set of valid addresses. If not, we abort program execution, and report a CFI violation. We enforce this check within the MoCFI runtime module: for simple indirect branch instructions such as `MOV pc,r0` or `BLX r0`, MoCFI simply reads out the target register (here: `r0`), and compares it to the valid set of target addresses. For complex indirect branch instructions such as the `LDR pc,[r0,r1,LSL #2]` instruction mentioned in Section 4.1.4.1, MoCFI accurately performs the calculation $r0 + r1 * 4$ at runtime, and subsequently checks the output to the set of valid addresses. Recall that the target registers to perform the latter calculation can be directly retrieved from the patchfile. Finally, for those indirect branches whose set of target addresses could not be determined at static analysis time, we apply the heuristics mentioned in Section 4.1.4.1.

Objective-C msgSend Calls. Dispatcher calls via *objc_msgSend()* require special handling as the method name (*selector*), and the target *class instance* are provided as parameters to the generic dispatcher routine of *objc_msgSend()*. As our experiments revealed, checking the parameters to *objc_msgSend()* is definitely necessary as the parameters are loaded from writable memory which an adversary can overwrite to hijack the execution path of the application. Based on the valid selectors and class instances we collected with our static analysis tools (cf. Section 4.1.4.1), we check at runtime whether the parameters provided to *objc_msgSend()* contain combinations of selectors and class instances that are in our set of valid combinations.

4.1.5 *Discussion and Security Considerations*

Our solution effectively detects code-reuse attacks on mobile devices by validating all indirect branch instructions executed by an iOS app. In particular, we developed static analysis tools to identify sets of valid branch targets for each indirect branch instruction. In addition, we also maintain a shadow stack for return addresses to enforce fine-grained control-flow validation upon function return.

In order to demonstrate that MoCFI detects advanced code-reuse attacks, we adopted a return-oriented programming attack presented by Iozzo and Miller [104] (developed for iOS v2.2.1) to iOS v4.3.1. When protecting the vulnerable application with MoCFI, the attack fails, and we successfully prevent an exploitation attempt. For the interested reader, the full exploit details can be found in [62, Appendix B].

As we have discussed above, it is not always possible to identify the full set of valid targets for each indirect branch. If this is the case, we fall-back to heuristics that still prevent an adversary from arbitrarily changing the control-flow. Moreover, our static tools could be extended by enhanced backtracking techniques to limit the set of possible branch targets. However, the design of sophisticated static tools goes beyond the scope of this project. Instead, our goal is to introduce the first framework that provides system-wide and efficient CFI enforcement for mobile applications executing on an ARM processor.

Since MoCFI performs binary rewriting after the iOS loader has verified the application signature, our scheme is compatible to application signing. On the other hand,

our load-time module is not directly compatible to the iOS CSE (code signing enforcement) runtime model. CSE prohibits any code generation at runtime on non-jailbroken devices, except if an application has been granted the *dynamic-signing* entitlement. To tackle this issue, one could assign the *dynamic-signing* entitlement to applications that should be executed under the protection of MoCFI. On the one hand, this is a reasonable approach, since the general security goal of CFI is to protect benign applications rather than malicious ones. Further, the *dynamic-signing* entitlement will not give an adversary the opportunity to circumvent MoCFI by overwriting existing control-flow checks in benign applications. In order to do so, one would have to mount a control-flow attack beforehand that would be detected by MoCFI. On the other hand, when *dynamic-signing* is in place, benign applications may unintentionally download new (potentially) malicious code, or malicious applications may be accidentally granted the *dynamic-signing* entitlement (since they should run under protection of MoCFI), and afterwards perform malicious actions. To address these problems, we can constrain binary rewriting to the load-time phase of an application, so that the *dynamic-signing* entitlement is not needed while the application is executing. Further, new sandbox policies can be specified that only allow the MoCFI library to issue the *mmap()* call to replace existing code, *e. g.* the internal page flags of the affected memory page are not changed, or their values are correctly reset after MoCFI completed the binary rewriting process.

Limitations. MoCFI does currently not protect shared libraries. In other words, it only applies CFI to the main application code. Hence, an adversary may exploit a vulnerability in a shared library to instantiate a code-reuse attack. We leave CFI protection on shared libraries as future implementation work. In fact, there are no new conceptual obstacles to solve for applying MoCFI to shared libraries.

As a consequence, we currently disable the return check if an external library calls a function that resides in the main application. Therefore, MoCFI registers when execution is redirected to a library and disables the return address check for functions that are directly invoked by the shared library. However, note that this can be easily fixed by either applying our trampoline approach to function prologues (*i. e.* pushing the return address on the shadow stack at function entry) or by applying MoCFI to shared libraries.

4.1.6 *Evaluation of MoCFI*

In order to evaluate the performance of MoCFI, we applied it to an iOS benchmark tool (called Gensystek Lite[1]), applied it to a full-recursive own developed quicksort algorithm, and several real-world iOS applications. As we described in Section 4.1.5, we apply MoCFI to the main application code, and not to the libraries. However, the benchmark tools we apply perform most part of the computation within the application.

Figure 31 shows the results for the Gensystek Lite application, where the slowdown factor for each individual benchmark is shown below the x-axis. Remarkably, the FPU/ALU, PI calculation, and the RAM (memory read/write) benchmarks add the highest over-

1 http://www.ooparts-universe.com/apps/app_gensystek.html

head (3.85x and 5x, respectively). The overhead for the remaining benchmarks is very low and ranges between 1% to 21%.

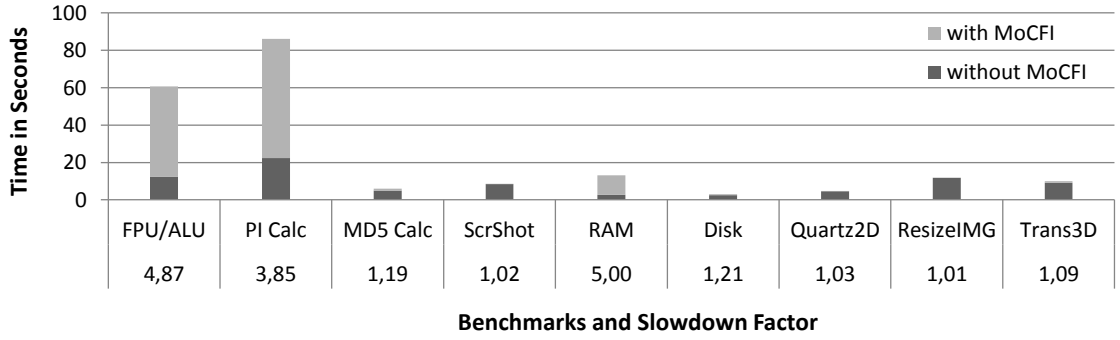


Figure 31: MoCFI performance on Gensystek Lite benchmarks

In order to approximate an upper boundary for performance penalties, we evaluated MoCFI by running a quicksort algorithm. Our implemented algorithm makes use of recursion and continuously calls a compare function which consists of only 4 instructions and one *return*. Hence, MoCFI frequently performs a control-flow check in this worst-case scenario. Nevertheless it performs quite well and needs 81ms for $n = 10,000$ (see Table 10).

n	Without MoCFI	With MoCFI
100	0.047 ms	0.432 ms
1000	0.473 ms	6.186 ms
10000	6.725 ms	81.163 ms

Table 10: Performance of MoCFI on quicksort

Moreover, we applied MoCFI to several popular iOS applications, among others Facebook, Minesweeper, TexasHoldem, and Gowalla. Our experiments showed that MoCFI does not induce any notable overhead while the applications execute. Further, MoCFI induces an acceptable overhead at load-time: *e. g.* for the Facebook application (code size 2.3MB; 33,647 calls; 5,988 returns; 20 indirect jumps) and TexasHoldem (2.8MB; 62,576 calls; 4,864 returns; 1 indirect jump) our rewriting engine required less than half a second to rewrite the entire application.

4.1.7 *Conclusion and Summary*

In this section, we focus on the problem of mitigating runtime exploits on modern mobile devices. We showed for the first time how the principle of control-flow integrity (CFI) enforcement can be applied to ARM-powered devices. Our solution tackles several unique challenges of ARM and mobile operating systems, which we discussed in detail. We solved all challenges and implemented a complete CFI enforcement framework for Apple iOS that instruments applications dynamically in memory.

In particular, MoCFI resists our latest attacks against coarse-grained CFI proposals (cf. Section 3.2) because it deploys fine-grained CFI for returns. Moreover, it also provides effective prevention against the latest so-called Jekyll attacks which hide malicious behavior encapsulated into a benign-looking app by leveraging return-oriented programming [188]. On the other hand, there are some open implementation aspects which need to be tackled in future work such as extending our CFI protection to shared libraries, and integrating MoCFI into the compiler toolchain to avoid jailbreaking the device.

4.2 PSIOS: APPLICATION SANDBOXING FOR IOS BASED ON MOCFI

As we have shown in Section 4.2, MoCFI defends against code-reuse attacks on mobile devices by instrumenting and validating indirect branch instructions. On the other hand, many attacks launched against mobile devices are based on injection of malicious apps rather than compromising benign apps. These apps directly access the device's system resources to induce malicious behavior or steal private user information. However, the taken control-flow adheres to the application's control-flow graph, and thereby does not violate CFI.

We will show throughout this section that our framework MoCFI can still be leveraged to protect a user from being compromised by malicious apps. Our key idea is to leverage MoCFI to enforce policy checks when branch instructions are executed. At the same time, MoCFI additionally ensures that our policy checks cannot be bypassed by means of a code-reuse attack.

4.2.1 *Motivation and Contributions*

Apple iOS is one of the most popular mobile operating systems. As its core security technology, iOS provides application sandboxing but assigns a generic sandboxing profile to every third-party application (cf. Section 4.1.1). However, recent attacks and incidents with benign applications demonstrate that this design decision is vulnerable to crucial privacy and security breaches, allowing applications (either benign or malicious) to access contacts, photos, and device IDs. Moreover, the dynamic character of iOS apps written in Objective-C renders the currently proposed static analysis tools less useful.

Our goal is to address the open problem of not only detecting privacy leaks for iOS apps, but actually *preventing* them. The key idea of our approach is to assign specific sandboxing profiles to each application to enforce a given fine-grained privacy policy. Such a profile may either be defined by a user at installation time, or centrally provided by a system administrator or an enterprise. Logically, we generate a protection layer between applications and the iOS Objective-C Runtime environment. Further, we monitor applications at execution time, and ensure that they only perform actions that adhere to the given sandboxing profiles. Our solution operates *directly* on the application binary. Hence, it neither requires recompilation nor access to the source code, which enables enforcement of policies for arbitrary applications.

In the remainder of this section, we present the design and implementation of PSiOS, a tool that features a novel policy enforcement framework for iOS. Our reference implementation deploys control-flow integrity based on our MoCFI framework that protects applications against runtime exploits (as described above in Section 4.2). PSiOS enables user-driven and fine-grained application sandboxing: on the one hand, a user can dynamically update sandboxing profiles without the need to recompile or reinstall the application. Hence, end-users can easily revoke or assign privileges. On the other hand, our framework enables very fine-grained policies in which the user or system administrator can precisely specify which privileges are assigned to an application. This is possible, since our sandboxing profiles cover the entire Objective-C runtime, and allows

argument validations for each API call. Since our framework is based on MoCFI, we also prevent attackers from exploiting vulnerabilities in the application code to hijack its assigned rights [105, 135]. We demonstrate that PSiOS effectively prevents privacy breaches by testing our tool with SpyPhone [138], an iOS app specifically designed to steal sensitive information from an iOS device.

Note that our approach differs from Apple's recently introduced entitlement keys (*i. e.* permissions). Specifically, Apple provides 25 keys to confine the privileges of apps [11]. However, these keys are specified by the app developer or directly by Apple. Hence, neither the end-user nor an enterprise can apply custom sandboxing policies to their apps. Instead, one needs to rely on Apple to apply the appropriate entitlements to each app, while malware writers will avoid to confine their apps for obvious reasons. Moreover, as we will elaborate in Section 4.2.2, these entitlements are enforced on the basis of the built-in iOS sandboxing framework, which (in contrast to PSiOS) cannot enforce fine-grained sandboxing rules.

It is noteworthy to mention, that independent from our research work on PSiOS, Apple has recently introduced privacy settings options (starting from iOS 6) where end-users can disable or enable access to private information on an app-by-app basis [12]. We believe that this new feature is a step in the right direction since iOS devices suffer from lack of privacy protection. However, we stress that our tool PSiOS does not only cover the same access rules, but also features argument validation (*e. g.* to allow access to a subset of private information), enables fine-grained access control beyond access to private information (*e. g.* any system call an application may invoke), and at the same time prevents runtime exploits.

4.2.2 *Problem Description*

Recall that iOS sandboxing is realized by a kernel module which has been adopted from the TrustedBSD kernel. This module mediates and validates every system call and its arguments according to sandboxing profiles already *pre-defined* by Apple. As already mentioned, iOS assigns a generic sandboxing profile to *every* third-party application which enables every application to access the public frameworks, and specifically grants access to contacts, location, device information, call history, keyboard cache, recent searches, e-mail account configurations, and photos. Recently, several attacks were reported, where applications abused their privilege set to steal, for instance, the user's address book [165]. Moreover, whenever an application is exploited by a runtime exploit, the adversary can misuse the application's privileges to steal sensitive information as well [105, 135].

Note that iOS already supports sandboxing at the kernel-level, but not within the Objective-C runtime. The design decision taken by Apple leads to coarse-grained sandboxing because the Core OS layer misses the semantics of the Objective-C runtime. Instead of enforcing access control on a specific API call, iOS has to enforce access control based on invoked system calls. In particular, the Core OS layer cannot enforce access control on the main Objective-C constructs such as used classes, objects, variables, and methods, which are extensively used by iOS applications and involve a chain of diverse system calls, files, and memory structures.

4.2.3 *High-Level Idea*

To address the mentioned security and design weaknesses of the current iOS sandboxing scheme, we aim towards a framework that allows access control for the Objective-C runtime, and the enforcement of the least-privilege principle. Note that realizing such a system for iOS is highly involved since iOS is closed-source. Hence, we cannot simply replace or extend existing modules as typically performed in recent Android security research proposals, *e. g.* Kirin [73] or TaintDroid [74].

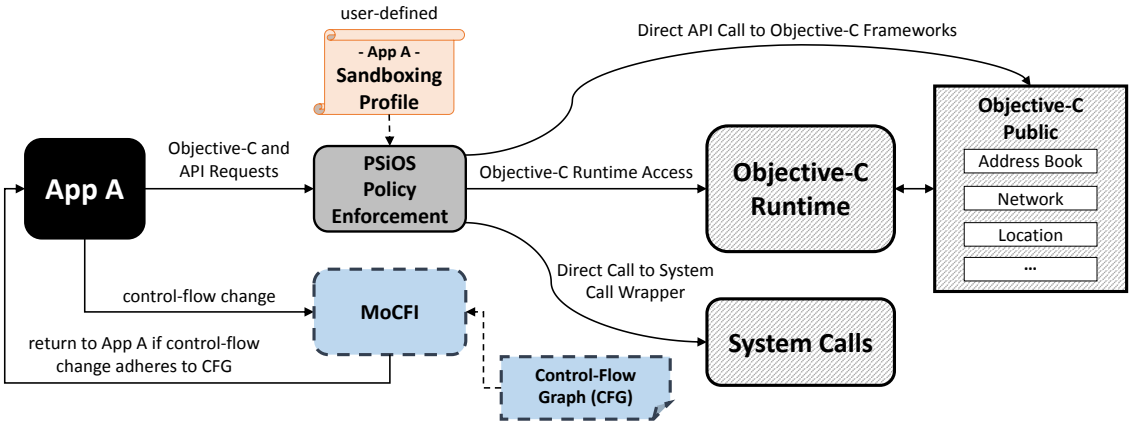


Figure 32: High-level idea of PSiOS

The high-level idea of PSiOS (**P**rivacy and **S**ecurity for **iOS** devices) is shown in Figure 32. In contrast to Apple's approach where sandboxing profiles are generic and pre-defined, PSiOS allows a different and user-defined sandboxing profile for each application. Basically, we add a new module that operates between the Application and Objective-C framework Layer (cf. Section 4.1.1), which we call *policy enforcement* component. As shown in Figure 32, this component mediates every access request to the Objective-C runtime, the frameworks, and the system call wrapper. It enforces access control rules on each access request based on the user-defined sandboxing profiles. Only when the policy has not been violated, we forward the request to its original destination. Note that the current iOS system does not enforce *any* access control mechanism to the Objective-C runtime and frameworks. On the other hand, iOS already enforces access control on system calls, but our approach allows an individual enforcement policy for each iOS application.

To monitor if an application adheres to the given sandboxing profile, we instrument the application so that all access requests are redirected to the policy enforcement. Alternatively, one could directly extend the Objective-C runtime and frameworks with dedicated interfaces that validate whether the caller has the appropriate privileges (similar to the Android permission system [92]). However, the iOS Objective-C runtime and frameworks are closed-source. Hence, extending them directly is infeasible.

Furthermore, the Objective-C runtime operates on the same level as the application code. Hence, policy checks inside the runtime can be bypassed by simply jumping over the policy code using either a control-flow attack or by dynamically calculating the

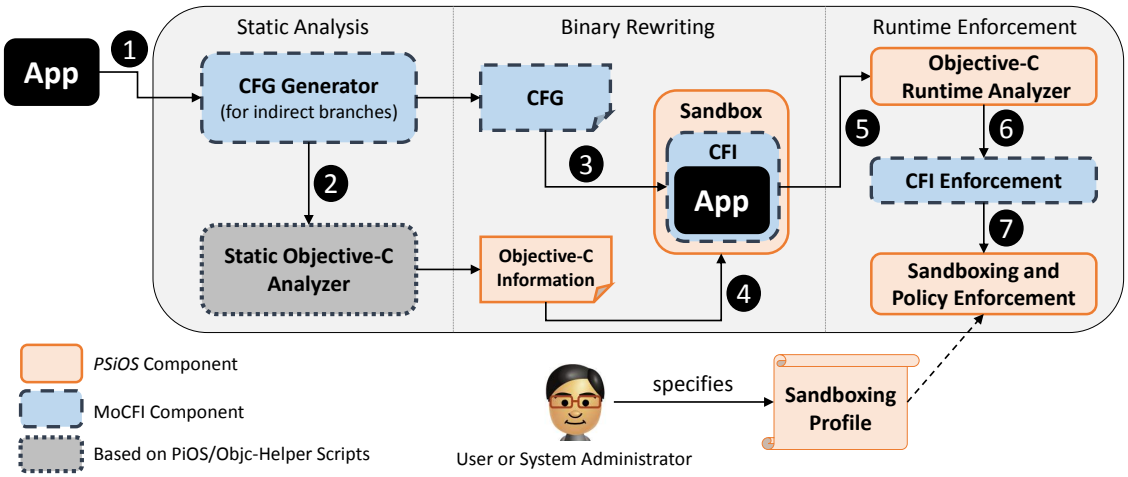


Figure 33: Architecture of PSiOS

relevant offset. Our framework does not suffer from this shortcoming as we have full control over the application code. Further, an adversary cannot mount a code-reuse attack to bypass our policy checks as we leverage MoCFI to thwart any CFI violation.

4.2.4 *Design of PSiOS*

In this section we introduce the design of PSiOS (**P**rivacy and **S**ecurity for **iOS** devices) which enforces our high-level idea presented in Section 4.2.3.

4.2.4.1 *Architecture*

The general architecture of PSiOS is shown in Figure 33. Basically, our design can be divided into three distinct phases: (1) static analysis (*offline*), (2) binary rewriting at *load-time*, and (3) runtime CFI and policy enforcement at *execution-time*. While the static analysis phase needs to be performed only once, the binary rewriting and runtime enforcement phase are performed whenever the application is launched by the user.

The general workflow is as follows: First, we reverse-engineer the application binary by using automated tools to derive the application's structure. In particular, we leverage for Step ❶ the MoCFI's static analysis components to derive the application's CFG (cf. Section 4.1.4.1). Further, we implemented a static Objective-C analyzer that reuses techniques of existing tools such as PiOS [72] and Objective-C helper scripts [69] to identify used Objective-C classes and methods (Step ❷). Note that we extended these tools to also identify calls to the system call wrapper. In Step ❸, we instrument the application at load-time to embed CFI checks using MoCFI. Second, we leverage binary rewriting to insert checkpoints into the application that will be triggered whenever an application aims to access the Objective-C runtime, the public frameworks, or the system call wrapper (Step ❹).

At execution-time, we first use a novel runtime Objective-C analyzer that tackles the incompleteness of the static analysis, and retrieves important runtime information on

Objective-C constructs such as registered parent and child classes, and runtime addresses of invoked methods (Step ❺). Afterwards, CFI ensures that the control-flow of an application always follows the legitimate paths of the CFG (Step ❻). Further, for all access requests to the Objective-C environment and the system call wrapper, our policy enforcement component validates if the request adheres to the given policy rules (Step ❼).

PSiOS supports three enforcement types: *Log*, *Exit*, and *Replace*. The *Log* enforcement only records all policy-related events, but does not take any actions when a policy is violated. This option is useful for training phases allowing a system administrator to identify all relevant Objective-C and system calls which facilitates the specification of sandboxing profiles.

The *Exit* option immediately terminates an iOS application when a policy violations occurs. Due to legacy compliance, we also support a *Replace* enforcement optin. This allows an application to securely continue to execute even though a policy violation occurred. In that specific case, PSiOS replaces the return values of the Objective-C runtime with fake data. For instance, if the sandboxing profile prohibits access to the address book, the return value will be either fake contacts or an empty list.

4.2.4.2 *Internals of* PSiOS *Policy Checks*

We implemented the design of PSiOS (see Figure 33) in a prototype that supports iOS version 4.3.2, 4.3.3, 5.0.1, and 5.1.1. The static Objective-C analyzer is realized as a new Python module for the reverse-engineering tool IDA Pro 6.x. For rewriting the application binary and enforcing CFI, we re-use our MoCFI framework. However, we need to extend MoCFI to introduce the policy enforcement and the runtime Objective-C analyzer. In particular, the latter component enables runtime analysis of Objective-C constructs. The entire runtime tools are realized as one shared iOS library that is developed in the Objective-C++ language. Since Apple prohibits any user from installing a new shared library, we had to jailbreak our test devices to inject our library to every iOS application, and to enable binary rewriting at runtime.

The internal workflow of a PSiOS-instrumented application is depicted in Figure 34. The application to be instrumented consists of several arbitrary machine instructions (denoted as INS) and branches. Each indirect branch and direct call instruction is instrumented using MoCFI. To this end, we simply replace these instructions with a dispatcher instruction which redirects the control-flow of an application to our runtime module.

The runtime module first saves the current program state by storing all processor registers. Subsequently, it validates the control-flow based on the information stored in the control-flow graph (CFG). In Step 3, our modified runtime module invokes our new PSiOS policy checking component. This component involves two main actions: (i) retrieving static and runtime information on API as well as Objective-C calls, and (ii) validating whether the desired control-flow adheres to the application-specific sandboxing profile.

Sandboxing Rules. The sandboxing rules are internally encoded in XML, and contain blacklisted Objective-C and API calls. Each rule contains the following fields/attributes:

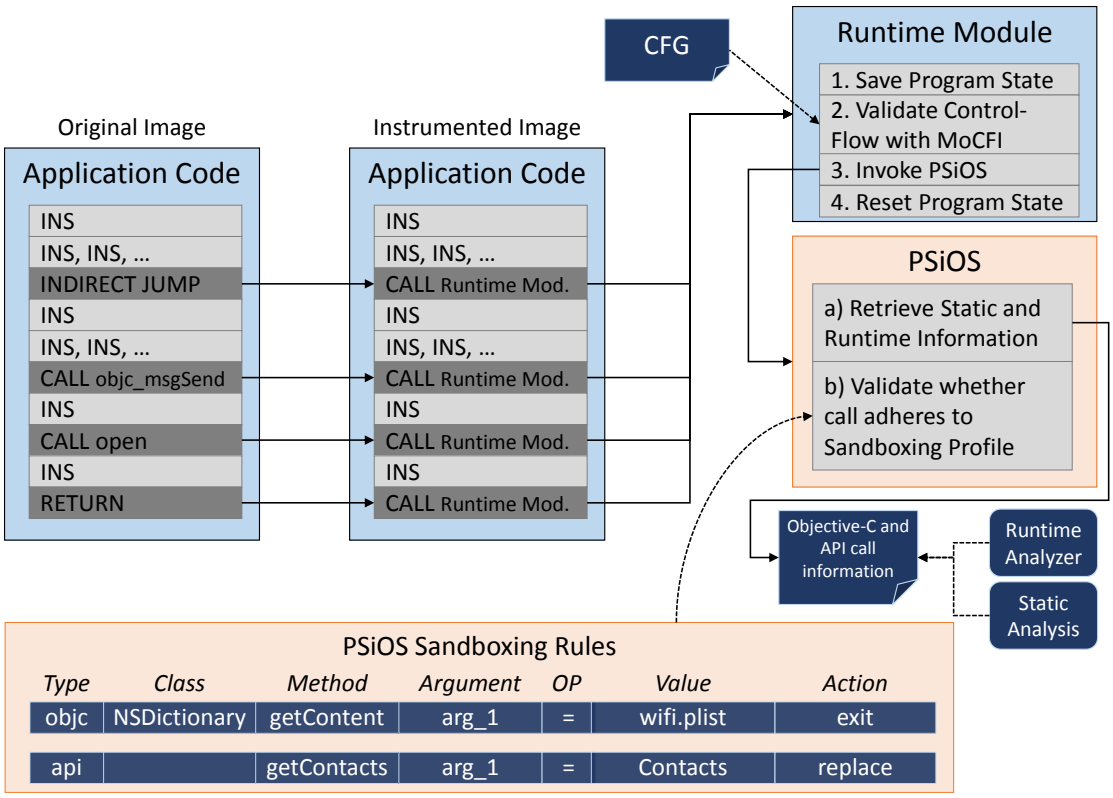


Figure 34: Internal workflow of PSiOS

- **Type**: The rule type indicates whether a rule applies to an Objective-C or API call.
- **Class**: For rules that are of type Objective-C, this field specifies the target class name.
- **Method**: For Objective-C calls, this field holds the target selector. Similarly, for API calls, this field provides the name of the API to be validated.
- **Argument**: The Argument field holds the number of the argument to be checked. Note that we support sandboxing rules for each argument. For this, one only needs to specify one rule per argument. Moreover, it is also possible to apply different checks to a single argument. This can be realized by defining one rule per check.
- **Operand (OP)**: The operand holds the compare method which can be of form: $(=, !, <, >, >=, <=)$
- **Value**: This field provides the value to compare with.
- **Action**: The action field holds the enforcement type *Log*, *Exit*, or *Replace*.

Specifically, the two sandboxing rules given in Figure 34 prohibit an iOS app to access the WiFi configuration file `wifi.plist` (Rule 1), and return an empty list whenever the

application attempts to access the address book (Rule 2). Note that both is allowed by default for all iOS apps. Rule 1 applies to Objective-C type calls. In particular, the class of interest is `NSDictionary` with the selector *getContent()*. For this specific class-selector pair, our rule targets the first argument which in case it is equal to `wifi.plist` results in a policy violation, and termination of the application.

In contrast, Rule 2 applies to the standard iOS API call to retrieve contacts information. For the API call *getContacts()*, we validate whether the first argument equals to Contacts. If so, we replace the actual contact list with an empty list.

4.2.5 *Evaluation of PSiOS*

In this section, we analyze the effectiveness and efficiency of PSiOS by using the SpyPhone application [138] as proof-of-concept since it demonstrates which private data can be accessed by every iOS application. SpyPhone is an open-source proof-of-concept application that demonstrates which data can be collected by iOS third-party applications. As SpyPhone is able to retrieve a significant amount of private and sensitive data about the user and the device, we thoroughly analyzed how this is achieved and how the data harvesting can be prevented using PSiOS.

Basically, SpyPhone retrieves Wi-Fi configurations, location, call histories, and e-mail account information by accessing property lists that are stored as XML files on any iOS device at the Core OS layer. For instance, the information about a user's email accounts is retrieved from the file `com.apple.accountsettings.plist`. In general, access to these files can be denied by restricting the Objective-C *dictionaryWithContentsOfFile()* method of the `NSDictionary` class, which is used to parse the XML file into an Objective-C data structure. For each file, we defined one rule that restricts the first parameter of the method, namely the file name.

SpyPhone also accesses sensitive information by calling appropriate Objective-C methods at the Core Services Layer. For instance, this applies to the user's address book or when requesting device information (*e. g.* the user's phone number). In particular, the address book is used to store private phone numbers, email addresses, and home addresses. We specified policy rules that prohibit SpyPhone from using these Objective-C methods (*e. g.* `ABAddressBookCopyArrayOfAllPeople` to access the user's address book). For the interested reader, the entire sandboxing profile (including all policy rules) for SpyPhone can be found in [194, Appendix B].

Applying PSiOS to iOS Apps. To demonstrate the effectiveness of our approach, we also applied PSiOS to a number of popular iOS applications such as Facebook, WhatsApp, ImageCrop, BatteryLife, Flashlight, ImageCrop, InstaGram, MusicDownloader, MyVideo, NewYork (Game), Quickscan, LinPack, Satellite TV and the Audi App. For each app, we defined a specific sandboxing profile that prevents the app from accessing private user information. In particular, PSiOS successfully prevents access to the address book (for Quickscan, Facebook, and Whatsapp), to personal photos (for ImageCrop and InstaGram), and to the iOS universal unique identifier, abbr. UUID (for Quickscan, BatterLife, Flashlight, MusicDownloader, MyVideo, NewYork, and Audi).

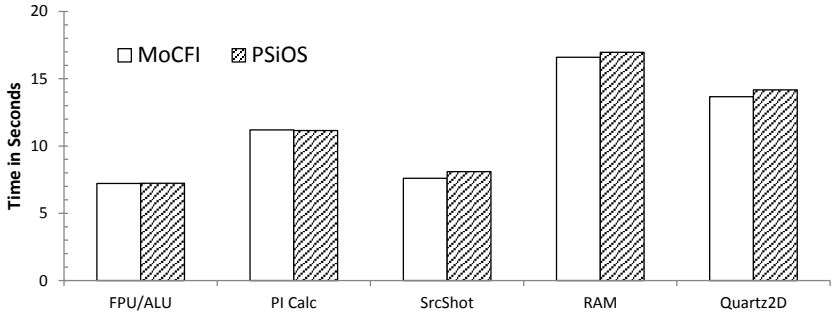


Figure 35: Comparing performance overhead of MoCFI and PSiOS

Performance Overhead. We performed runtime measurements based on the iOS Gensystek benchmark app to directly compare the overhead introduced by PSiOS compared to native CFI protection with MoCFI. For PSiOS, we applied several policies on Objective-C selectors (such as `NSString` and `NSBundle`) which frequently trigger a policy validation for each of the individual benchmarks. The results of our measurements are shown in Figure 35. Compared to native CFI execution with MoCFI, PSiOS only adds a negligible overhead ranging from 0.28% to 6.58%.

4.2.6 *Discussion*

The runtime components of PSiOS are completely implemented inside our MoCFI shared library. We opted for this implementation approach because it enables a system-centric CFI-based sandboxing solution, and allows every iOS app to immediately benefit from our tools. As for MoCFI, to install and push our library on an iOS device, we require a jailbreak of the device, since Apple is closed-source and strictly prohibits any installation of a new shared library. For MoCFI and PSiOS, we *only* require a jailbreak for setting a single environment variable, installing a shared library, and allowing our library to rewrite the application code during load-time. Setting the variable as well as installing and signing our library can be easily done by Apple for future iOS releases. Further, for binary rewriting, our library only needs to be assigned the dynamic code signing entitlement which allows an application generate code just-in-time.

Note that similar approaches targeting Android require the rooting of an Android-powered device as well (*e. g.* TaintDroid [74] or AppFence [101]). Moreover, PSiOS does not necessarily require a jailbreak. For instance, PSiOS could be provided as a static rewriting tool that Apple applies to all app binaries before releasing them on the App Store. The static analysis (which requires IDA Pro at the moment) could be incorporated using cloud services. Further, app developers could run PSiOS before submitting their applications to the App Store. Both approaches are compatible to Apple's signature scheme since PSiOS rewrites the app before it is signed by Apple.

4.2.7 *Previous Work*

In this section, we elaborate on previous work in the area of mobile security and application sandboxing.

Research on iOS. The closest work to our framework is PiOS [72] which is a static analysis tool to detect privacy leaks of iOS applications. PiOS generates an application's control-flow graph by backtracking all Objective-C calls and by reconstructing the class inheritance relationships. However, PiOS suffers from some shortcomings: First, its analysis does not cover embedded metaclasses (*i. e.* root classes of Objective-C classes), which frequently occur in iOS applications. Second, it does not support class clusters. Third, it mainly uses a backtracking of ARM processor registers to determine used classes and selectors. However, as our experiments have shown this approach often fails to resolve the used class. In contrast, PSiOS tackles this problem by including the Objective-C sections in its analysis and by retrieving runtime information. To summarize, PiOS is constrained in its analysis because it fails to cover the full picture of the Objective-C runtime, particularly when an adversary deploys obfuscation techniques to circumvent static analysis tools.

MobileSubstrate [161] is a framework for jailbroken iOS devices that provides runtime patching of existing programs. To this end, application code can be rewritten to install hooks for Objective-C message handlers and C/C++ functions. The rewriting engine works similar to our approach, however, MobileSubstrate merely provides hooking support. Further, it does not provide protection against runtime exploits that can undermine policy validation code.

Research on Android. In the last years Android has been an appealing subject of research. Kirin [73] is an extended application installer that checks application's permission combinations according to a given policy. Apex [141] goes a step further and allows end-users to choose permissions at install-time. However, since iOS is closed-source, these approaches are not feasible on iOS.

TaintDroid [74] is a framework to detect data leakage attacks on Android. It uses dynamic taint analysis, and warns the user whenever sensitive data leaves the device at a taint sink (*e. g.* the network interface). The AppFence [101] framework builds upon TaintDroid and enables fine-grained privacy rules, and enables the return of shadow data when a policy rule has been violated. However, TaintDroid does not fully cover native code, and could be subverted by runtime exploits. Moreover, it is directly implemented into Android's Java virtual machine (Dalvik). Since iOS use Objective-C rather than the interpreted Java language, it remains open how such a system could be integrated in iOS. Further, in parallel to our work, several security extensions have been proposed to enable fine-grained sandboxing rules (Aurasium [198]) or fine- grained privacy controls [34], but on Android, while we focus on iOS.

Finally, there are several works on Android that tackle privilege escalation attacks at application-level [77, 68, 33]. These attacks are based on the observation that two applications merge their permissions (either directly or indirectly) resulting in a larger sandbox. However, inter-app communication is still an exceptional event on iOS. Nevertheless, in

our future work we aim to investigate the feasibility and detection of privilege escalation attacks on iOS with PSiOS.

4.2.8 *Summary and Conclusion*

PSiOS is a novel policy enforcement framework for the closed-source mobile operating system iOS providing fine-grained, application-specific, and user-driven sandboxing for third-party applications without requiring access to source code. It demonstrates the usefulness of MoCFI which provides secure execution of the policy enforcement framework. We implemented a fully working prototype and demonstrated that PSiOS effectively prevents privacy breaches by testing our reference implementation with *SpyPhone* [138], a tool specifically designed to steal data from an iOS device. PSiOS significantly raises the bar of application sandboxing attacks. In particular, its policy framework prevents recent attacks that have successfully undermined Apple's vetting process [96, 188].

4.3 HAFIX: HARDWARE-ASSISTED CONTROL-FLOW INTEGRITY EXTENSION

Although control-flow integrity (CFI) offers high protection against code-reuse attacks, it either suffers from performance overhead or its implementations deploy too coarse-grained policies that a sophisticated adversary can bypass (cf. Section 3.2). Up to now, the majority of research on CFI has focused on software-based solutions.

In this section, we take a hardware-based CFI approach that has several advantages over software-based counterparts: First, it is significantly more efficient, as we demonstrate. Second, compiler support is simplified by reducing complex CFI checking code to single CFI instructions. Third, dedicated CFI hardware instructions and separate CFI memory provide strong protection of critical CFI control-flow data.

Our Goal and Contributions. We present the design and implementation of a hardware-assisted CFI solution, called HAFIX (Hardware-Assisted Flow Integrity eXtension). Our CFI proposal is based on a state model and a per-function CFI label approach. We focus on backward-edge CFI, that is, CFI for indirect branches through function return instructions in the program's CFG. In contrast, forward-edge CFI handles indirect branches in the CFG that are caused by jumps and function calls [182].

In terms of performance, protecting backward edges is more challenging, simply due to the fact that function returns occur far more frequently than indirect calls and jumps.[2] Existing backward-edge CFI schemes either suffer from large performance degradation (when using a shadow stack for return addresses), or they deploy too coarse-grained CFI policies that can be bypassed as mentioned before.

We developed and extensively evaluated for the first time a fine-grained backward-edge CFI system in hardware. To develop HAFIX, we had to tackle several challenges: (i) efficient and secure access to CFI control-flow data (*i. e.* label memory), (ii) specifying and realizing new CFI instructions that perform efficiently in one cycle, and (iii) automatically emitting these instructions to program binaries during compilation. On top of this, we also tackled the challenge of providing CFI protection for recursive function calls.

We present real hardware implementations of backward-edge CFI targeting bare metal code. Specifically, we extend the processors' instruction set with new CFI assembler instructions, and also modify the commodity compilers to automatically emit our new instructions into applications. We developed proof-of-concept implementations on two different processor architectures: The first is the Intel Siskiyou Peak platform which primarily targets embedded applications [156]. The second is the open source LEON3 microprocessor which is compatible with the SPARC V8 instruction set and has been developed by the European Space Agency for avionic applications [157]. In this dissertation, we focus on the implementation on Intel Siskiyou Peak.

In our implementation of HAFIX, we focus on embedded systems which have become pervasive, and are built into a vast number of devices such as sensors, vehicles, mobile and wearable devices. However, due to resource constraints, they fail to provide

2 Recall that most jumps and calls are direct, and require no CFI protection as their branch target is statically encoded.

sufficient security, and are particularly vulnerable to runtime exploits. Establishing security and trust in embedded systems introduces specific security challenges that go beyond that of traditional PC platforms [116]. As a consequence, security is usually only introduced if the corresponding resource usage is minimal. A second challenge is that embedded systems are usually programmed using native (unsafe) programming languages such as C or assembly language. Hence, they frequently suffer from vulnerabilities that can be exploited by means of code-reuse attacks. For instance, Francillon and Castelluccia [79] demonstrated that return-oriented programming can be leveraged on an embedded Atmel AVR-powered sensor to persistently inject a malicious firmware based on code reuse.

One significant aspect of our work is the performance and security evaluation of fine-grained hardware-based (backward-edge) CFI that we conducted on embedded systems under different security scales. We evaluated our HAFIX implementation using standard embedded benchmarks (including Dhrystone and CoreMark), and show that it only adds 2% of performance overhead on average. Moreover, we provide a detailed security evaluation to demonstrate that HAFIX reduces the available gadget space to 19.82% on average compared to recently proposed coarse-grained CFI defenses in a worst-case setup.

4.3.1 *System Model*

In this section, we present our target threat model, main assumptions, and requirements.

Threat Model. Our main goal is to thwart code-reuse attacks launched through CFG backward edges (function returns). The adversary i) has full control over the program's stack and heap to inject new and overwrite existing return addresses, ii) has access to the application's code including linked libraries, iii) can exploit a memory corruption error to instantiate a code-reuse attack, and iv) even can bypass any deployed code randomization (*e.g.* ASLR), *i.e.* the adversary has full knowledge of the application's memory layout.

Assumptions. As the main scope of this paper are CFG backward edges, we assume that the target system deploys software-based CFI protection for forward edges. Recently, it has been shown by Tice et al. [182] that compiler-based forward-edge CFI protection can be efficiently implemented in LLVM and GCC. We also assume that the target hardware platform enforces protection against code injection (*e.g.* data execution prevention) currently deployed by default on many platforms.

Requirements. The main objectives of our CFI solution are efficiency/practicability, as well as enforcing fine-grained CFI protection on CFG backward edges. As mentioned before, fulfilling both requirements is highly challenging, as function returns occur frequently at runtime. Recent research has shown that coarse-grained CFI policies for returns, *i.e.* restricting returns to target a call-preceded instruction, are insufficient and can be bypassed (cf. Section 3.2).

4.3.2 *Design*

In this section, we introduce the design and implementation of our novel security hardware architecture that provides fine-grained protection against code-reuse attacks based on control-flow integrity (CFI).

Our basic design modeled as a state machine is shown in Figure 36. The main idea of our protection mechanism is to enforce CFI based on label *state*, and decouple source from destination labels. For this, we distinguish between three states: (0) for ordinary program execution, (1) function entry, and (2) function exit. To enforce our CFI label approach, we introduce two new CFI label instructions, namely `CFIBR` and `CFIRET`. As the names imply, we use one label instruction for function calls and another one for returns. This distinction allows us to deploy different policies for calls and returns, and at the same time ensures that an indirect call cannot target a label instruction used for returns and vice versa.

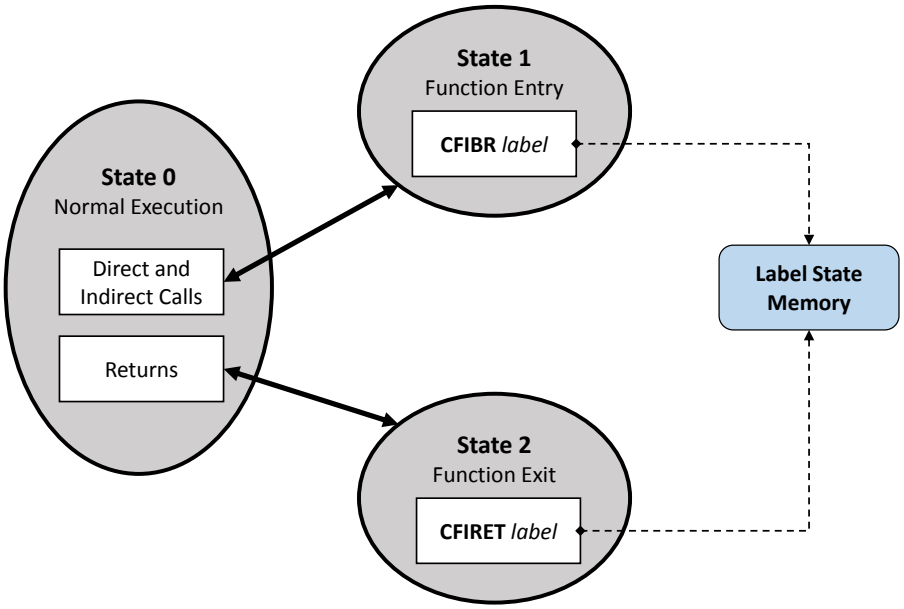


Figure 36: HAFIX CFI state machine

Inspired by the state model proposed by Budiu et al. [32], we enforce that after a call or return instruction, the next instruction to be executed has to be a CFI instruction. That said, any other instruction will immediately terminate the program execution. Further, valid execution of CFI instructions in State 1 and 2 trigger state transitions to State 0 allowing the program to continue ordinary execution.

In the following, we describe in detail our CFI policies enforced in HAFIX.

4.3.2.1 *Call Instrumentation*

Direct and indirect calls will lead to a transition from State 0 to State 1 (function entry). In this state, we only allow the program to use a `CFIBR` instruction. Any other instruction

will lead to a CFI violation, and subsequently to a program termination (kill). Each `CFIBR` instruction contains a *label*, which is hard-coded as an immediate. Specifically, we use labels on a per-function level. In other words, every function in the program is assigned one unique *label*.

The effect of a `CFIBR` is twofold: first, it loads the used label in a dedicated and isolated memory storage that we refer to as *Label State Memory*. This operation effectively activates a label, *i. e.* it indicates that a function has been entered. The second effect is that after `CFIBR` has been executed, the processor changes back to State 0. Our mechanism ensures that an indirect call must target a `CFIBR` instruction. Since these are placed at each function entry, we prevent the adversary from jumping into the middle of a function. Note that we elaborate on fine-grained call checks in Section 4.3.5.

4.3.2.2 *Return Instrumentation*

Fine-grained software-based CFI approaches validate function returns based on the *shadow stack* (or return address stack) paradigm (cf. Section 2.3.3): all return addresses that are pushed on the program's stack through call instructions are backed-up on a separate protected shadow stack. Upon function return, CFI verifies whether the program uses a return address that is held on the shadow stack. Although shadow stacks provide fine-grained protection, they (i) significantly decrease performance and (ii) lead to false positives for certain programming constructs (C++ exceptions with stack unwinding, setjmp/longjmp). Recent work demonstrates that performance can be increased by leveraging a parallel shadow stack [57]. However, the parallel stack still resides in the same address space of the target application.

In our solution HAFIX, we force a return to target a call-preceded instruction inside a function that is currently executing. As we will show, this CFI policy can be efficiently implemented in hardware and requires only minimal changes to the compiler.

The underlying design to enforce this CFI policy is depicted in Figure 37. In order to monitor functions that are currently executing, HAFIX requires the compiler to assign unique labels to each function. Further, it forces the first instruction of each function to be a `CFIBR`. This instruction loads the label of the function into a our label state memory, to indicate that the function is active (Step ❶). Internally, direct and indirect call instructions lead to a processor state switch in which the processor only accepts `CFIBR`. To deactivate a function, HAFIX uses the `CFIDEL` instruction which effectively removes the label from the label state memory (Step ❷). Hence, `CFIDEL` instructions are executed just before a function return instruction.

The critical point of backward-edge CFI is the final return instruction of the subroutine. This indirect branch instruction can be exploited by the adversary to hijack the program's control-flow based on a malicious return address. However, in HAFIX return instructions need to target an active call site. To enforce this, only returns to the `CFIRET` instruction are permitted, in particular those `CFIRET` instructions that define a currently active label in the label state memory (Step ❸). The dashed line in Figure 37 indicates that `CFIRET` does not change the label state, but only checks whether a label is active. We will give a concrete code example in Section 4.3.3.2.

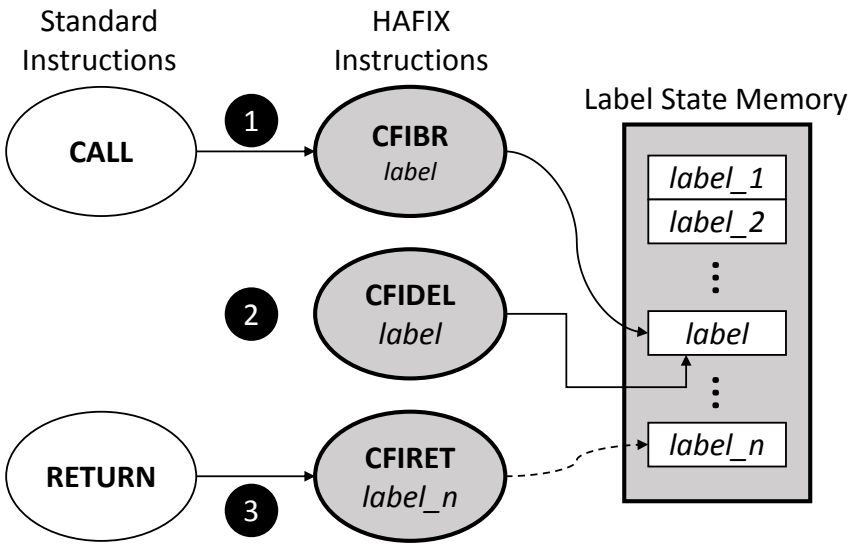


Figure 37: Call-Return policy check in HAFIX

Implementation Requirements. We need to define new hardware instructions `CFIBR`, `CFIDEL`, and `CFIRET`. Further, we need to implement a state model that switches states on function call and returns to only accept as next instructions `CFIBR` and `CFIRET` respectively. On the compiler side, we need to emit these instructions at their corresponding places: `CFIBR` at function start, `CFIRET` at all call sites, and `CFIDEL` at function return.

4.3.2.3 *Special Case of Recursive Functions*

Our design and implementation needs to tackle an important challenge of handling recursive function calls. They lead to a number of store operations of the same label since a recursive function invokes itself several times before each instance returns. To solve this problem, we only store the label once and record the number of invocations in a separate (hidden) register. For this, we introduce a new CFI instruction called `CFIREC` and a new shadow register called `CFIREC_CNTR`.

The abstract workflow of a recursive call under our new CFI instrumentation is shown in Figure 38. First, the compiler emits `CFIREC` at the beginning of all recursive functions (rather than a standard `CFIBR`). Upon execution, `CFIREC` activates the label of the recursive function but only if the `CFIREC_CNTR` is set to zero (Step ❶). In addition, and also in case the label is already activated, `CFIREC` increments the `CFIREC_CNTR` (Step ❷). Internally, we also associate the label of the recursive function to the `CFIREC_CNTR` register.

Upon function return, the `CFIDEL` instruction validates whether the current label is associated to `CFIREC_CNTR`. If the link exists, HAFIX knows that a recursive function attempts to return. Hence, we subsequently check whether `CFIREC_CNTR` > 1. If so, we only decrement `CFIREC_CNTR` (Step ❸) as more returns from the same function are expected. Only if `CFIREC_CNTR` is equal to 1, we remove the label from our memory (Step ❹) as the last function instance of the recursive function returns. Note that our mechanism targets

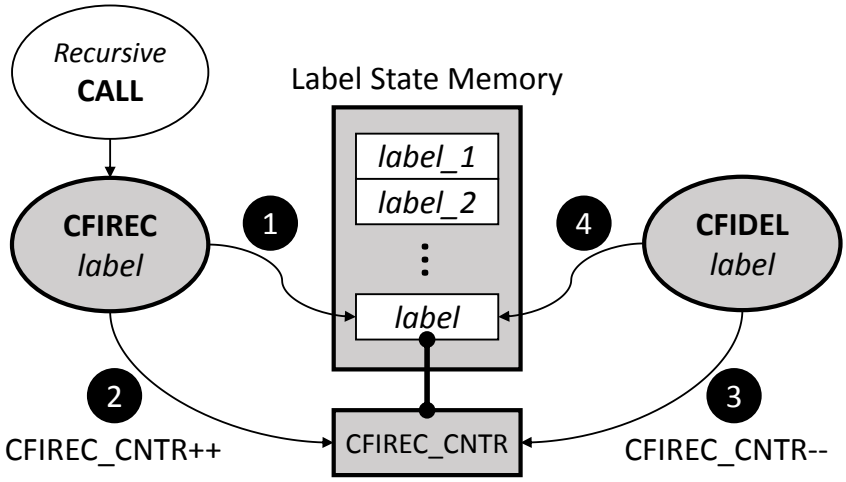


Figure 38: Recursion handling in HAFIX

non-nested recursive functions. Nested recursive functions are rare in embedded applications, *e. g.* none of our benchmarks contained nested recursive calls (cf. Section 4.3.4).

4.3.3 *Implementation*

In this section, we present the implementation of HAFIX on our target architecture Intel Siskiyou Peak. We refer the interested reader to [13] for the SPARC implementation details. We give a short overview on Siskiyou Peal, present HAFIX-instrumented code, and finally describe the implementation of the CFI instructions and the label memory in hardware.

4.3.3.1 *Intel Siskiyou Peak*

Intel's Siskiyou Peak is a 32-bit, fully synthesizable core intended for deeply embedded applications [156]. The core is highly configurable and features a 5-stage, single-issue processor pipeline. Major configuration options include a Memory Protection Unit, various branch predictors, I&D caches and multiplier performance options. Siskiyou Peak also includes a variety of micro-architectural options to trade off clock frequency for improved instructions-per-cycle. The processor is organized as a Harvard architecture with separate busses for instruction, data and memory mapped IO spaces. The instruction set is a small subset of the 32-bit x86 instruction set and shares the same variable-length binary encoding.

4.3.3.2 *Code Instrumentation*

Figure 39 shows HAFIX-instrumented assembler code targeting Intel Siskiyou Peak. The example shows two sample functions with their function prologue and epilogue instructions, where *funct_A* simply calls *funct_B*. The emitted CFI instructions are highlighted with bold font.

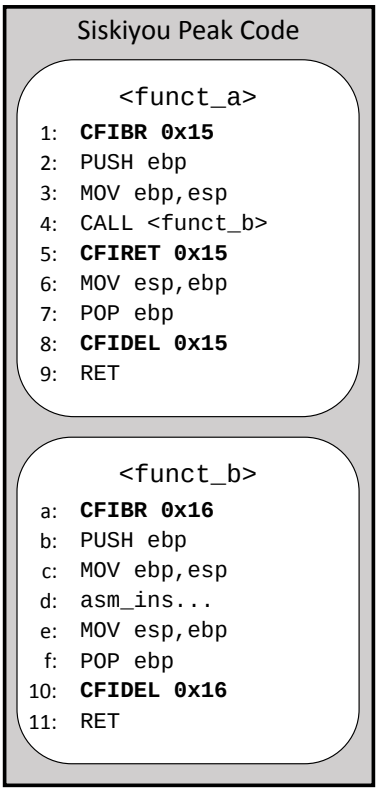


Figure 39: HAFIX-instrumented code

Our instrumented compiler prepends `CFIBR` to function prologues that activate the unique label of the function in the label state memory. Further, `CFIDEL` is appended to function epilogues to mark the end of a function resulting in a label de-activation (see also Figure 37). Lastly, `CFIRET` is inserted after a `CALL` instruction.

Unique labels are added to these instructions within their context with a program that is executed after the linker. We modified the customized LLVM compiler toolchain (shipped already with Siskiyou Peak) to generate HAFIX-instrumented code. Moreover, during a post-compilation processing phase, we identify recursive function calls and replace the corresponding `CFIBR` with `CFIREC` instructions.

Detailed Code Example for Siskiyou Peak. A detailed flow of the CFI protection offered by HAFIX is shown in Figure 40. It shows a sample program consisting of three functions: A, B, and C. Each function is assigned a label, *e. g.* label `0025` is used for Function A. As described above this label is written to our CFI label state memory through the `CFIBR` instruction at the beginning of each function (Step ①).

Note that Function A performs a subroutine call to Function B (Step ②). Hence, the processor switches to State 1, and the `CFIBR` of Function B activates label `0050` in the label state memory (Step ③). The critical point with regards to CFI is the function return in Function B. Potentially, the adversary could have manipulated the return address to hijack the execution-flow. To prevent this attack, we apply our CFI policy for returns

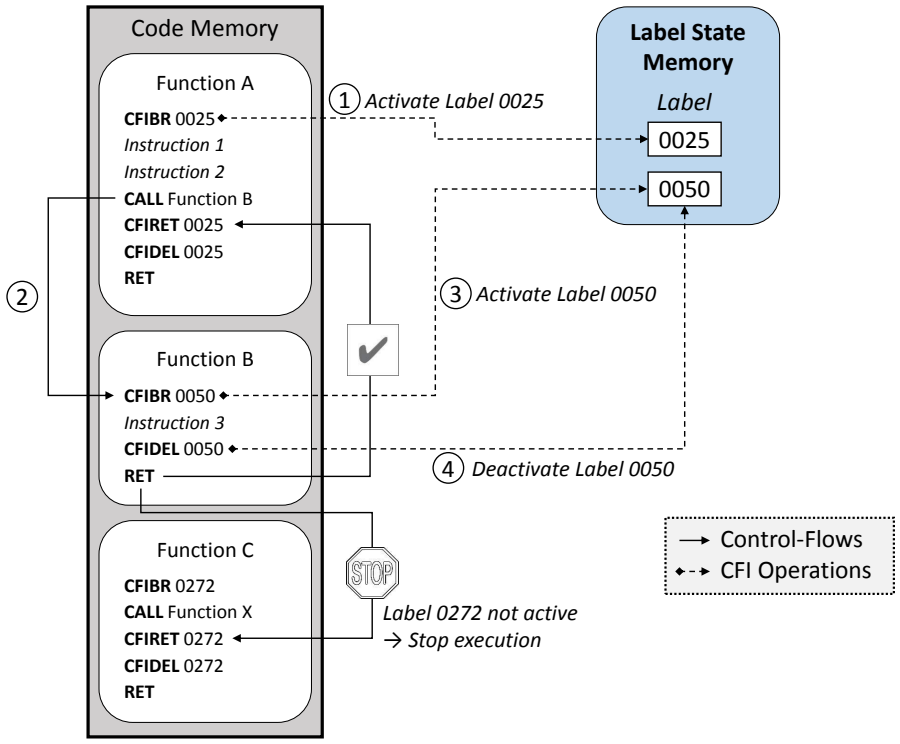


Figure 40: Workflow of call/return CFI checks in HAFIX

ensuring that a return can only target a `CFIRET` instruction using an *active* label. In the example shown in Figure 40, our CFI policy is preserved when the program returns to Function A, because at its call side the CFI compiler has emitted a `CFIRET 0025` instruction. Consequently, the adversary has no chance to redirect the control-flow to the call side of Function C, since label `0272` is not active. Recall that coarse-grained CFI protection schemes that only deploy one label for returns would have not prohibited this malicious execution-flow.

4.3.3.3 *Hardware CFI and Label Memory*

Hardware CFI enforcement on Siskiyou Peak is achieved by augmenting the execution stage of the processor pipeline with a CFI control unit and associated label state memory, see Figure 41. The CFI control unit monitors CFI instruction sequencing and manages CFI label state. In the event of a CFI violation being detected, a processor exception is issued allowing a software handler to take appropriate action.

Label state memory is implemented as a tightly-coupled 16384x1 memory with the CFI label employed as the index. This facilitates highly efficient CFI instructions: For a given label, `CFIBR` sets the memory location indexed by the label while a `CFIDEL` clears it. The `CFIRET` instruction reads the location indexed by the label and raises an exception if not set. In the target platform of Xilinx Spartan-6 this approach allows CFI label state to be efficiently mapped onto two synchronous Block RAMs. Due to the low logic complexity of the indexing mechanism, it is feasible to clock the Block RAM on the opposite

clock edge removing a cycle of read latency and enabling single-cycle performance for all CFI instructions.

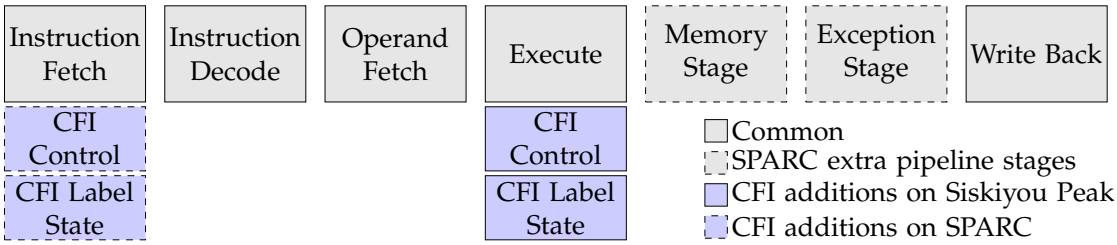


Figure 41: CFI pipeline integration

4.3.4 *Evaluation*

In this section, we evaluate the performance and security of HAFIX.

4.3.4.1 *Performance*

The system impact was evaluated using a suite of microprocessor benchmarks including CoreMark, Dhrystone, cover, crc, matmult and recursion. The performance overhead for the HAFIX-enhanced Siskiyou Peak and LEON3 cores is shown in Figure 42 with the respective unmodified stock core used as the baseline. The overall performance overhead is around 2% for both architectures with backward-edge CFI enabled. The largest increases, as expected, are seen in those benchmarks that include many short function calls such as Dhrystone and recursion. None of our benchmarks programs raised a false CFI violation. As discussed by Dang et al. [57] several shadow stack implementations require special handling of certain programming constructs (e.g., setjmp/longjmp) to avoid a false alarm.

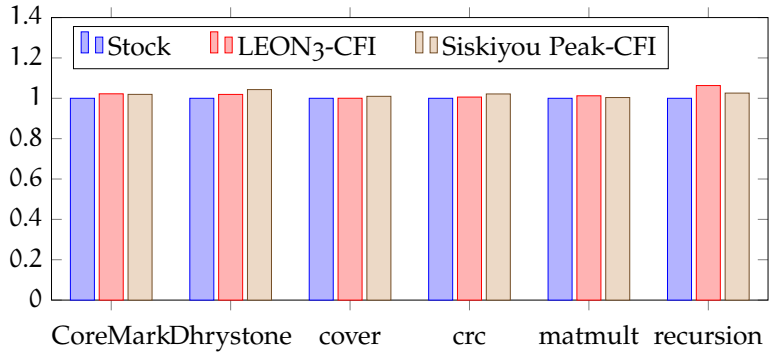


Figure 42: CFI extension overhead w.r.t stock core for LEON3 and Siskiyou Peak

The performance of the implemented architecture with respect to area was evaluated using results from the Xilinx place and route (PAR) tools. The HAFIX enhanced Siskiyou

Peak core consumes an additional 2.49% registers and less than 1% additional LUTs (Look Up Tables). The CFI label state memory is implemented using 2 Block RAMs.

4.3.4.2 *Security*

To measure to what extent HAFIX reduces the set of valid branch addresses, we record the label state memory at each function return for all of our benchmark programs under the CFI implementation for Siskiyou Peak. The chart shown in Figure 43 demonstrates that on average only 0.70% of all program instructions are addressable by a function return; with a maximum of 2.2% (CoreMark).

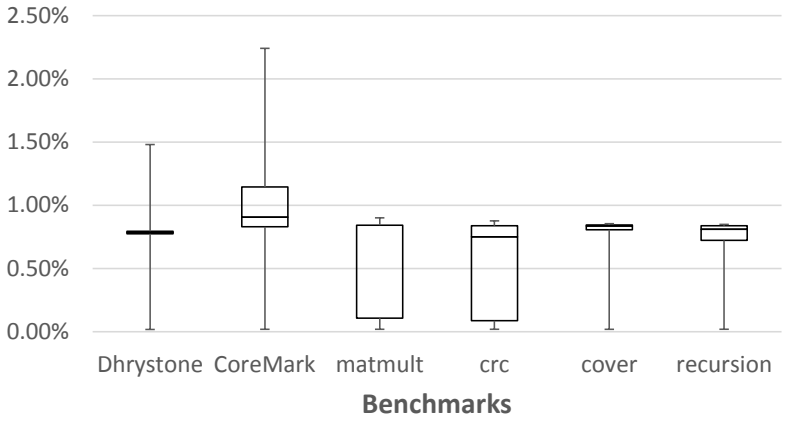


Figure 43: Percentage of program instructions that a function return is allowed to target

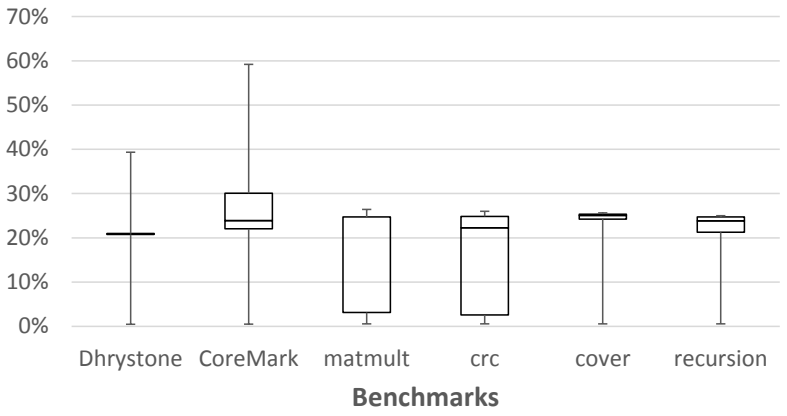


Figure 44: Percentage of `CFIRET` instructions that a function return is allowed to target

In order to give our evaluation more meaning, we directly compare our backward-edge CFI realization to recent CFI-based approaches that restrict returns to target a call-preceded instruction [150, 204, 83]. For this, we validate how many `CFIRET` instructions (i.e., call-preceded instructions) a function can target on average. Figure 44 shows that the median percentage of valid `CFIRET` instructions for the individual benchmarks

ranges from 3.13% (matmult) to 25.36% (cover). Hence, HAFIX significantly reduces the gadget space (to 19.82% on average) compared to recent CFI-based approaches.

Note that using static-linked benchmark programs for our security evaluation resembles a worst-case scenario. In fact, all the numbers reported would be tremendously lower for dynamically-linked programs for two reasons. First, shared libraries introduce a large amount of code that is never used during program execution. In coarse-grained CFI, all call sites inside the shared library are valid targets, but in HAFIX only those call sites of the invoked shared library function are valid. Second, benchmark programs typically contain a large (main) function that invokes a number of subroutines. As the main function remains active almost throughout the entire program execution, all its call sites (*i. e.* `CFIRET` instructions) are valid targets for function returns. As an example, Dhrystone contains a large main function with 87 call sites out of 419 call sites in total. However, even in this circumstance, HAFIX still reduces the gadget space to 20% compared to recent CFI-based approaches. In order to further reduce the space, we can emit multiple labels into the main function, e.g., splitting the Dhrystone main function into four parts reduces the set of addressable `CFIRET` instructions to 5%.

Note that an adversary cannot undermine HAFIX by modifying the CFI label state or the `CFIREC_CNTR` register simply due the fact that both are not directly accessible by software, but only by our CFI instructions. Since all modern platforms as well as our target architectures prevent code injection attacks (using data execution prevention), an adversary can neither modify the CFI-protected code nor inject malicious CFI instructions.

Practical Exploits. We also evaluated the effectiveness of HAFIX using code-reuse exploits against self-developed vulnerable programs. Our attack on SPARC is initiated by overflowing a buffer, which results in the eventual overwrite of the register holding the return address, `%i7`. Similar to a conventional return-into-libc attack, our malicious return address points to the start of a payload. However, upon returning from the function, HAFIX reports a control-flow violation since the exploit jumps to an address that does not match a valid `CFIRET` site. Similarly, on Siskiyou Peak our exploit returns to an invalid `CFIRET` site. Once the HAFIX invalidation occurs, a CPU reset trap terminates code execution in the exception detection stage.

4.3.5 *Extensions*

In the following, we discuss possible extensions to enforce hardware-assisted CFI for indirect call and jump instructions.

4.3.5.1 *Fine-grained Indirect Call Instrumentation*

To prevent code-reuse attacks from jumping into the middle of a function, HAFIX enforces that call instructions target a valid function start. This would still allow an adversary to launch pure return-into-libc attacks, in which only entire functions are invoked by exploiting indirect calls. Note that such attacks are not specific to our design, but apply to previous CFI solutions as well. However, return-into-libc attacks typically

still need to invoke at least some instruction sequences to prepare function arguments, which is detected by our approach. Moreover, assuming a precise control-flow graph as in [4, 32], our design can be easily leveraged to enforce fine-grained CFI checks for indirect calls. We would only need to load all valid labels of an indirect call in a second label memory area (*Call Labels*) and, restrict the subsequent `CFIBR label` instruction to use a label that can be found in *Call Labels*.

4.3.5.2 *Indirect Jump Detection*

CFI protection for indirect jumps is challenging due to the fact that many indirect jumps cannot be resolved prior to execution, *i. e.* their set of target addresses are hard to predict. Given the imperfection of static analysis tools to resolve indirect branches, we investigated a hardware-assisted behavioral-based approach. In our experimental setup, we keep track of several counters in the window of five indirect jumps. One counter keeps track of the number of direct branches executed between our sliding window. In fact, direct branches are rarely used in return-oriented exploits since these target hard-coded addresses. Another two counters are deployed to keep track of stack pushes and pops. Typically, return-oriented programming attacks do not use push instructions [169] as they potentially overwrite return addresses of the return-oriented payload. That said, if many pop but only a few push instructions are issued in our window, we have another indicator that a return-oriented programming attack is currently executing.

It is envisaged that the heuristics associated with indirect jump detection will be implemented in an fully autonomous manner with software only responsible for configuring the appropriate indirect jump window size and thresholds for direct jumps and push and pop stack operations. More specifically, indirect jumps will be counted, and when the window size, *e. g.* 5, is reached the direct jump, push, and pop counters will be compared against configurable thresholds. If these thresholds are exceeded, a CFI violation is reported. Conversely, if no thresholds have been exceeded the counters will be reset and monitoring will continue.

For our indirect jump heuristics, we tested several thresholds using SPEC CPU2006 benchmarks. Our experiments reveal that between a sliding window of five indirect jumps there are never execution traces that contain less than 3 direct jumps and 3 push instructions. Although heuristic-based approaches will never provide an ultimate solution to runtime exploitation prevention, they can significantly raise the bar for control-flow attacks.

Another alternative we are currently investigating is a new CFI policy for indirect jumps that ensures indirect jumps can only target a function whose label is active. Since indirect jumps typically remain in the boundaries of the currently executing functions[3], it seems to be a promising direction that we will explore. In particular, we will investigate whether such a strict CFI policy will raise false positives.

3 There are some known exceptions such as stub code for calls to external library functions in the Linux PLT (Procedure Linkage Table) section. To support these indirect jumps, we could enforce that indirect jumps are allowed to target functions whose label is active or whose next instruction is a `CFIBR L`.

4.3.6 *Conclusion and Future Work*

For the first time, we present the implementation and evaluation of a fine-grained hardware-assisted CFI scheme HAFIX that provides integrity checks for backward edges (returns). Our implementation of HAFIX on Intel Siskiyou Peak and SPARC LEON3 provides new dedicated CFI instructions that efficiently perform CFI checks in a single cycle. We require minimal changes to the compiler toolchain to emit our new CFI instructions. Our security evaluation demonstrates that HAFIX significantly reduces the code base an adversary can leverage to perform code-reuse attacks. Compared to recently proposed software-based CFI approaches, HAFIX reduces the gadget space to 19.82% with an average performance overhead of only 2%.

In our reference implementation of HAFIX we target bare metal code. Ideally, HAFIX needs to be applied to all code running on the target system, including the operating system. We are currently working on an operating system CFI support module that handles label states for different processes. Specifically, this extension will store and restore labels whenever a context switch occurs.

Another ongoing work concerns the label space for programs that link to several shared libraries. In this case, we need to ensure that our compiler emits unique labels per library. We are also currently exploring new CFI instructions that facilitate hardware-assisted forward-edge CFI.

4.4 RELATED WORK

In this section, we survey related work on control-flow integrity (CFI). We investigate both software-only and hardware-assisted CFI solutions.

4.4.1 *Software-only CFI Schemes*

The basic principle of monitoring the control-flow of an application in order to enforce a specific security policy has been introduced by Kiriansky et al. [115] in their seminal work on program shepherding. This technique allows arbitrary restrictions to be placed on control transfers and code origins to confine a given application. Program shepherding employs dynamic binary instrumentation based on the DynamoRIO framework [29]. In its prototype implementation, it provides return address protection allowing a return only to target an instruction that is preceded by a call instruction. Thus, its implementation is vulnerable to the advanced code-reuse attacks we presented in Section 3.2.

A more fine-grained analysis was presented by Abadi et al., who proposed control-flow integrity enforcement [1, 4]. This technique asserts the basic safety property that the control-flow of a program follows only the legitimate paths determined in advance (cf. Section 2.3). If an adversary hijacks the control-flow, CFI enforcement can detect this deviation and prevent the attack. In contrast to a variety of ad-hoc solutions, CFI provides a general solution against runtime exploits. In particular, Abadi et al. [2] develop a framework that is based on an abstract machine model and an instruction set to prove that CFI enforcement is sound. Lastly, they present a hardware-based implementation of CFI enforcement [32] which we discuss in detail in Section 4.4.4. We use CFI as the basic technique and show that this principle can be applied on the ARM processor architecture to protect mobile devices against runtime exploits (cf. Section 4.1), and enforce fine-grained application sandboxing in iOS (cf. Section 4.2). To do so, we had to overcome several obstacles due to the subtle architectural differences between x86 and ARM, and the specifics of mobile operating systems. In particular, we demonstrate how to emit CFI checking code on-the-fly at application load-time.

A number of binary-instrumentation based CFI schemes have appeared to tackle the performance overhead incurred by original CFI for x86 [4] and our tool MoCFI [62] for ARM. We described in detail kBouncer[150], ROPGuard (Microsoft EMET) [83, 131], CFI for COTS binaries [204], ROPecker [47], and CCFIR [202] in Section 3.2.2. In particular, we demonstrate in Section 3.2 that even if these CFI schemes and their policies are combined with each other, they can be bypassed by advanced code-reuse attacks under weak adversarial assumptions.

The CFI schemes discussed so far employ binary instrumentation to enforce CFI. In contrast, Pewny and Holz [152] present a CFI compiler, called Control-Flow Restrictor (CFR), targeting iOS. Similar to MoCFI, CFR checks at each indirect branch whether it follows a legitimate path in the control-flow graph (CFG). However, CFR deploys a coarse-grained policy for returns which makes it vulnerable to advanced code-reuse attacks.

Onarlioglu et al. [148] propose a compiler extension, called G-Free, for Intel x86 to eliminate unintended instruction sequences of a program binary. Further, it encrypts return addresses with a random key, and constrains indirect jumps to a local function.

Control-flow locking (CFL) proposed by Bletsch et al. [25] also follows the compiler-based approach. Their technique is inspired by the hardware-assisted CFI solution presented by Budiu et al. [32]: it inserts lock code before each indirect branch, and unlock code at each possible branch destination. Whereas the lock code sets a label, the unlock code checks if the correct label has been set. That is, the control-flow checks occur after the indirect branch has been taken. This allows the adversary to hijack the control-flow once. However, before the next indirect branch is executed, CFL checks whether an unlock operation has been executed. To prevent the adversary from calling a system call in-between, CFL inserts lock checks before each system call. On the other hand, CFL deploys coarse-grained policies: indirect jumps/calls can target every function entry and any code location whose symbol is used as data. Further, returns from indirectly callable functions can return to any instruction that follows after a call. Lastly, checking against a static label before a return from a directly callable function will eventually lead to coarse-grained CFI (cf. Section 2.3.3): all possible call sites to the target function need to share the same label.

CFI enforcement has been also leveraged to ensure benign execution in hypervisor code. Wang and Jiang [189] introduce HyperSafe which protects x86 hypervisors by enforcing hypervisor code integrity and CFI. With respect to returns, HyperSafe only validates if the return address is within a set of possible return addresses which has been calculated offline. However, the dynamic nature of mobile applications prevents us from calculating return addresses offline. A more recent approach by Criswell et al. [56] explores CFI for operating system kernels, but their system only enforces coarse-grained CFI policies.

Classic CFI schemes do not support separate compilation meaning that they do not support separate instrumentation of program modules that are later linked to one single program. As a consequence, CFI typically requires all program modules to be available at instrumentation time to assign unique labels. Niu and Tan [145] tackle this shortcoming by encoding CFG information (including labels) into dedicated tables that are consulted at runtime to perform CFI validation. In particular, MCFI merges these CFG tables when program modules need to be linked dynamically. Their tool, called MCFI, enforces fine-grained CFI based on a compiler-based approach. However, MCFI does not enforce fine-grained return address checks, *i. e.* a static call-graph is used to determine possible call sites for a given return instruction. In a follow-up work, Niu and Tan [146] expand MCFI protection to just-in-time (JIT) compilers. Their approach, called RockJIT, takes a hybrid approach: it enforces fine-grained CFI for the JIT compiler code, and coarse-grained CFI for code generated just-in-time.

Lastly, Prakash et al. [154] aim at enforcing system-wide CFI for kernel code and user processes. Their solution, called Total-CFI, deploys a shadow stack to enforce fine-grained checks for returns, and validates indirect jumps and calls based on a whitelist of valid targets taken from relocation tables. In its prototype implementation, Total-CFI employs dynamic translation based on QEMU to virtualize an entire guest operating sys-

tem. Hence, Total-CFI incurs a large performance overhead since it requires a software emulator for the entire operating system.

4.4.2 *Non-Control Data Attacks*

Our focus in this dissertation are attacks and defenses against code-reuse attacks. In particular, we consider attacks that hijack the intended control-flow of an application (cf. Section 1.1). On the other hand, Chen et al. [45] demonstrate that non-control-data attacks are realistic and can be launched against modern applications. The main idea of these attacks is to alter application data such as user input, configuration data, user ID, or authentication data to execute privileged code without violating control-flow integrity. Although non-control-data attacks are beyond the scope of this dissertation, we briefly review some schemes that aim at mitigating these attacks because they also include mechanisms to prevent runtime exploits.

Data-flow integrity (DFI) proposed by Castro et al. [38] aims at preventing non-control-data attacks. To do so, DFI derives the static data-flow graph (DFG) of an application, and inserts runtime checks to validate if read and write operations follow the legitimate data flow given in the DFG. However, this compiler-based approach incurs significant performance overhead. Write Integrity Testing (WIT) proposed by Akritidis et al. [6] extends the original DFI work. It leverages interprocedural points-to analysis which outputs the CFG, and computes the set of objects that can be written by each instruction in the program. Based on the result of the points-to analysis, WIT assigns a color to each object and each write instruction. WIT enforces write-integrity by only allowing the write operation if the originating instruction and the target object share the same color. As a second line of defense, it also enforces CFI to check if an indirect call targets a valid execution path in the CFG. However, WIT does not prevent return-oriented attacks, because it does not check function returns.

4.4.3 *Inlined Reference Monitors*

In general, one can classify CFI as an instantiation of an inlined reference monitor (IRM). To insert an IRM into an application, a rewriter or compiler produces — based on a given security policy and the original program's binary or source code — a secured application [75]. The IRM ensures that program execution does never violate the security policy. In CFI [4], the IRM consists of CFI label checks and the shadow stack for returns to ensure that the control-flow always follows a legitimate path in the CFG. In fact, the CFG resembles the security policy.

Apart from CFI, there are several other well-known IRM schemes such as software fault isolation (SFI) [187] and data-flow integrity [38]. Although their goals slightly differ from CFI, they share many similarities to CFI. Moreover, many SFI and DFI schemes are combined with some form of CFI to ensure that the IRM is not compromised by runtime exploits. As an example, recall that our tool PSIOS (cf. Section 4.2) employs CFI

to enforce fine-grained application sandboxing on iOS. In the following, we provide a brief overview of SFI and DFI schemes.

The basic idea of SFI is to isolate code and data of an untrusted module in a separate fault domain [187]. Afterwards, the untrusted module is instrumented to ensure that the code cannot jump or reference data beyond its fault domain. The original SFI proposal targeted RISC architectures, and is not applicable to architectures that feature variable-length instructions. McCamant and Morrisett [130] tackle this shortcoming and presented a SFI scheme, called PittSFIeld, for the CISC-based architecture Intel x86. PittSFIeld aligns branch instructions to a 16-Byte boundary. To do so, it emits NOP instructions to ensure that all possible branch targets are aligned to 16 Bytes. NativeClient (NaCl) [199, 166] builds upon SFI and enables a sandboxed environment for native code plugins in web browsers. Our iOS policy framework is closely related to these works. In particular, we enable SFI for iOS applications, by instrumenting all calls to the Objective-C runtime. However, note that the existing works either focus on entirely different computing platforms or are incompatible to Objective-C.

Abadi et al. [3] propose XFI as an extension to their original x86-based CFI scheme [4]. In particular, XFI allows enforcement of fine-grained memory access control rules. For this, it employs a XFI rewriter and verifier to insert software guards that perform runtime checks on control-flow and memory access. However, the additional integrity constraints on memory and the stack incur a higher performance overhead.

Zeng et al. [200] showed that CFI combined with static analysis enables the enforcement of efficient data sandboxing. In particular, the presented scheme provides confidentiality of critical memory regions by constraining memory reads to uncritical data regions. This is achieved by placing guard zones before and after the uncritical data area. The solution has been implemented in the LLVM compiler infrastructure (similar to the NaCl compiler [199, 166]), targets the Intel x86 platform, and slightly increases performance compared to XFI at the cost of relaxing the integrity constraints for return addresses.

The aforementioned schemes have a practical limitation: they target a specific hardware architecture and porting them to other architectures involves significant effort. To address this limitation, Zeng et al. [201] propose Strato which is a framework that enables IRM implementations at the compiler intermediate-representation (IR) level.

Although CFI offers a viable defense to protect IRM schemes, it typically does not allow separate compilation. Niu and Tan [144] tackles this limitation by dividing code into chunks and restricting indirect branches to only target the first instruction of a chunk. The boundaries of code chunks are stored in the module's chunk table. The tool, called Monitor Integrity Protection (MIP), maintains a bitmap per chunk table, and supports static as well as dynamic combination of program modules by merging chunk tables on-demand. However, with respect to CFI, MIP only deploys coarse-grained CFI policies. Hence, an adversary can leverage our advanced code-reuse attack techniques to undermine MIP.

4.4.4 *Hardware-Assisted CFI Schemes*

Several approaches leveraged (or introduced new) hardware-based mechanisms to mitigate runtime exploits. For instance, kBouncer and ROPecker use the Last Branch Recording (LBR) history table of recent Intel processors [150, 47]. They add hooks into API call sites, and once these are triggered at runtime, validate the LBR entries based on a CFI policy. However, the deployed CFI policies and behavioral-based heuristics are too coarse-grained and can be bypassed as we have shown in Section 3.2.

Similarly, Xia et al. [197] use performance counters and Intel's Branch Trace Store (BTS) to detect control-flow deviations. Zhang et al. [205] aim at detecting program execution anomalies based on a new hardware architecture that validates all branch instructions [205]. However, both approaches require an offline training phase [197, 205], and assume a precise and static control-flow graph (CFG). Deriving a complete CFG is hardly possible given the complexity and dynamic nature of modern programs. In particular, statically determining valid return addresses leads to coarse-grained CFI policies as described in [4].

Our hardware-assisted CFI design presented in Section 4.3 is closely related to the design presented by Budiu et al. [32]. In their work, new hardware CFI instructions are introduced to enforce label checks on each indirect branch. For this, each branch target is annotated with a label instruction (`CFILABEL L`), and every indirect branch is replaced by a corresponding CFI instruction, *e.g.* `JMPC reg,L`. The latter CFI instruction jumps to the address specified in `reg` and at the same time sets a label `L` in a dedicated (new) CFI register. After the indirect branch has been executed, the processor changes state such that `CFILABEL` is the only permissible next instruction. In particular, the state will only change back to the ordinary execution state after a `CFILABEL` instruction has been executed using exactly the label `L` that has been stored in the CFI register by the indirect branch. In fact, our CFI state model in HAFIX is inspired by Budiu et al. [32]. However, the CFI scheme proposed by Budiu et al. [32] leads to coarse-grained CFI policies. Consider a subroutine that can be called by different callers. Since the return of the subroutine can only use one label, *e.g.* `RETC L1`, the compiler needs to insert a label instruction using `L1` for each possible call site. This allows the adversary to choose to which call side she wants to return to. Moreover, another related problem arises for those indirect calls that can potentially target many diverse functions. As an example, consider an indirect call that can possibly target 200 functions (which is not an artificial scenario even if source code is available, see *e.g.* [6]). To ensure fine-grained CFI, one would need to assign a unique label for each of the 200 function return instructions. As such, the compiler needs to add 200 corresponding label instructions at the call side (the instruction after the indirect call), *i.e.* 200 `CFILABEL L1-L200` instructions. This leads to a significant space and performance overhead. Given these problems, using the approach presented in [32] will with high probability lead to a coarse-grained CFI policy, where identical labels will be used for a subset of returns.

Branch regulation as proposed by Kayaalp et al. [111] requires identifying function bounds and a shadow stack to enforce fine-grained CFI. However, their approach suffers from the basic problems of shadow stacks.

Francillon et al. [80] introduce an embedded microprocessor that includes memory access control for the stack, which is split into data-only and call/return addresses-only parts. The processor enforces access control that does not allow to overwrite the call/return stack with arbitrary data. This effectively prevents stack-based return-oriented attacks. However, it requires major software changes to support a system-wide split-stack scheme.

A hardware-facilitated solution has been proposed by Frantzen and Shuey [81]. Their scheme, called StackGhost, targets SPARC-based systems. It leverages stack cookies that are XORed with return addresses at function entry, and XORed again upon function return. The design of StackGhost also includes a return address stack (*i. e.* shadow stack), but to the best of our knowledge, this has not been implemented and benchmarked. Further, StackGhost depends on specific features, which are unique to SPARC, and which, according to Frantzen and Shuey [81], cannot be easily adopted to other hardware platforms.

4.4.5 *Backward-Edge CFI Schemes*

Several CFI schemes focus exclusively on enforcing integrity checks for return instructions. These backward-edge CFI schemes aim at mitigating conventional return-oriented programming attacks (cf. Section 2.1.5).

Chiueh and Hsu [48] present a compiler-based implementation of a shadow stack. Several approaches aim to detect malicious change of return addresses by using instrumentation techniques without requiring source code. Gupta et al. [95] and Chiueh and Prasad [49] rewrite function prologue and epilogue instructions to incorporate a return address check on each function return. However, both approaches are not able to detect return-oriented attacks that use unintended instruction sequences, because they only instrument intended function epilogues.

In contrast, TRUSS (Transparent Runtime Shadow Stack) [171] and our tool ROPdefender [61] are based on just-in-time based instrumentation. These approaches leverage a shadow stack to perform integrity checks for return addresses. Due to just-in-time instrumentation, TRUSS and ROPdefender are able to detect execution of unintended sequences issued.

Recently, Dang et al. [57] review several shadow stack-based CFI schemes and demonstrate that performance can be increased by leveraging a parallel shadow stack. However, in contrast to HAFIX, the parallel stack still resides in the same address space of the target application.

4.4.6 *Forward-Edge CFI Schemes*

As there exist CFI schemes that focus on return instructions (backward-edge CFI), there exist schemes whose target are indirect jumps and calls (forward-edge CFI). For instance, the approaches of Tice et al. [182] and Jang et al. [109] focus on protecting indirect calls to virtual methods in C++. Both approaches have been implemented as a compiler exten-

sion and ensure that an adversary cannot manipulate a virtual table (vtable) pointer so that it points to an adversary-controlled (malicious) vtable. Unfortunately, these schemes do not protect against classical return-oriented attacks which exploit return instructions.

The aforementioned approaches require the source code of the application which might not be always readily available. In order to protect binary code, a number of forward-edge CFI schemes have been presented recently [85, 203, 155]. Although these approaches require no access to source code, they are not as fine-grained as their compiler-based counterparts. A novel attack technique, called COOP (counterfeit object-oriented programming), undermines the CFI protection of these binary instrumentation-based defenses by invoking a chain of virtual methods through legitimate call sites to induce malicious program behavior [163].

4.5 SUMMARY AND CONCLUSION

Defending against code-reuse attacks is challenging. In particular, it is highly challenging to design and implement a defense that is efficient, effective, and practical at the same time. In this chapter, we take a software-only and hardware-assisted control-flow integrity (CFI) approach to prevent these attacks, where we focus on fine-grained protection to mitigate our advanced code-reuse attacks presented in Chapter 3. Our software-only approach MoCFI targets mobile platforms running an ARM processor. It keeps compatibility to application signatures by instrumenting applications in memory after the loader has verified the signature. Further, it resists advanced code-reuse attacks by enforcing fine-grained checks for each indirect branch, and can be leveraged as an inlined reference monitor to enable fine-grained application sandboxing on iOS. On the other hand, we observed performance bottlenecks when enforcing software-only based fine-grained CFI. To tackle the performance problem, we explore a hardware-assisted CFI scheme. Our approach HAFIX introduces dedicated hardware CFI instructions and isolated storage for CFI data. In fact, HAFIX demonstrates that a hardware-software co-design allows fine-grained and efficient CFI enforcement.

5 ADVANCES IN CODE RANDOMIZATION

Code randomization defends against code-reuse attacks by randomizing the memory location of code segments. The most well-known instantiation of code randomization is address space layout randomization (ASLR) which randomizes the base address of shared libraries and the main executable. Unfortunately, ASLR is often bypassed in practice due to its low randomization entropy. Advanced code randomization schemes that randomize the internal code layout of a program aim at tackling this limitation. We elaborate on these so-called fined-grained randomization approaches and present our tool XIFER [63] which enables basic block permutation per application run. We also present a novel just-in-time code-reuse attack that demonstrates the limits of existing fine-grained randomization approaches [172]. As we will show, such advanced attacks can be tackled by combining code randomization with execution-path randomization [66].

5.1 BACKGROUND ON FINE-GRAINED CODE RANDOMIZATION

A widely deployed countermeasure against code-reuse attacks is the randomization of the applications' memory layout. The idea of software diversity (or program evolution) has been introduced by Cohen [50] in his seminal work on how to protect computer systems and their running software programs against software exploits. The basic observation is that an adversary typically generates an attack vector and aims to simultaneously compromise as many systems as possible using the same attack vector (*i. e.* one attack payload). To mitigate this so-called ultimate attack, Cohen proposes to diversify a software program into multiple and different instances while each instance still covers the entire semantics of the root software program. The goal is to force the adversary to tailor a specific attack vector/payload for each software instance and computer system making the attack tremendously expensive. Different approaches can be taken for realizing software diversity, *e. g.* memory randomization (ASLR) [78, 151], based on a compiler [50, 82, 106], or by means of binary rewriting and instrumentation [149, 97, 191, 63].

As we described in Section 2.2, base address randomization via ASLR [151] cannot resist control-flow attacks that exploit a memory disclosure vulnerability beforehand. To tackle this limitation, a number of so-called fine-grained ASLR, *i. e.* fine-grained code randomization schemes[1], have recently appeared in the academic literature [22, 114, 149, 97, 191, 63]. The underlying idea in these works is to randomize the code structure, for instance, by shuffling functions, basic blocks, or instructions (ideally for each program run [191, 63]).

Figure 45 demonstrates how fine-grained ASLR is applied to an application with three code blocks. For each execution, base address randomization ensures that the executable

1 Note that we use both terms interchangeably.

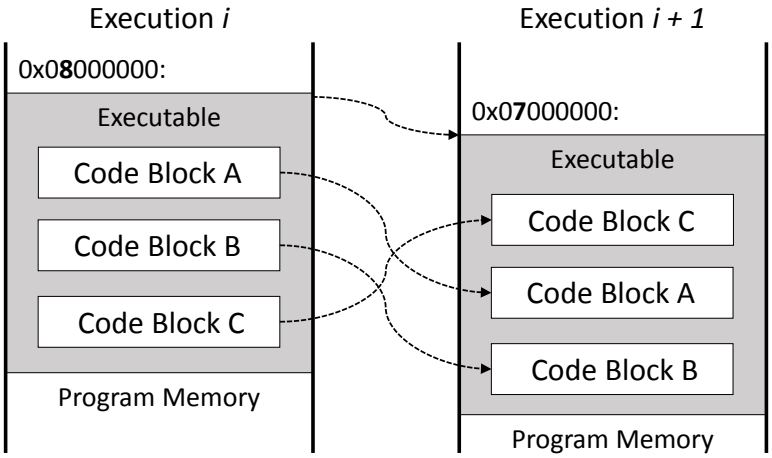


Figure 45: Fine-grained code randomization

is loaded at a start address different from the previous execution. In addition, fine-grained ALSR permutes the code blocks such that an adversary can no longer deploy a code-reuse attack by disclosing a single runtime address. Obviously, changing the order of code blocks requires adjusting memory offsets used in the code blocks. For instance, if Code Block B in Figure 45 issues a branch instruction to redirect execution to Code Block C, it will be necessary to adjust the branch target as Code Block C has been moved to the beginning of the executable.

Chapter Outline. The remainder of this chapter is organized as follows. In Section 5.2, we present our fine-grained ASLR scheme XIFER which enables basic block permutation per application run. Next, we present in Section 5.3 a just-in-time code-reuse attack that poses a severe threat to all fine-grained ASLR solutions proposed so far. In Section 5.4, we present our advanced fine-grained ASLR scheme called ISOMERON to tackle just-in-time code-reuse attacks. Lastly, we elaborate on related work in Section 5.5, and conclude this chapter in Section 5.6.

5.2 XIFER: A RANDOMIZATION TOOL FOR BASIC BLOCK PERMUTATION

In this section, we present our randomization tool, called XIFER, to defend against code-reuse attacks based on fine-grained ASLR. Our tool performs fine-grained memory diversification for *each* program run. Basically, we transform the control-flow graph (CFG) of an application at load-time by means of binary rewriting. Our CFG transformations include permutation of small code pieces at basic block (BBL) level, splitting of BBLs, and injection of new instructions. The novelty of our approach resides in the fact that all three techniques are entirely enforced at runtime in memory without requiring source code.

In summary, our contributions are the following:

- **Mitigation of code-reuse attacks:** Our diversification techniques adequately mitigate code-reuse attacks such as return-into-libc and return-oriented programming

because all instructions are moved from their original location. In contrast to recent compiler-based [50, 82] and static rewriting-based approaches [149, 97], we rediversify the program at *each* run. Further, our permutation can be applied down to a single instruction.

- **Prototype for x86/ARM:** Our prototype implementation targets both mainstream processor architectures Intel x86 and ARM, and can be applied to all binaries that use the standard Linux ELF executable format. Intel's IA-32 (x86) platform is widespread in the desktop market, and ARM processors are deployed in most well-established tablets and smartphones.
- **Compliance to code signing:** The design of our tool is compliant to code signing, which is a standard feature in modern app store distribution models. In contrast to static rewriting approaches [149, 97, 114, 191], our tool rewrites the application after the OS loader has already verified the signature.
- **Evaluation:** We evaluated our prototype for x86 and ARM by using the standard benchmark suite SPEC CPU2006. Our evaluation results demonstrate that our tool efficiently performs software diversity with an average runtime overhead of only 5%, and a load-time overhead of only 1s for a 5MB executable.

Section Outline. After presenting the high-level idea of XIFER in Section 5.2.1 and its design in Section 5.2.2, we elaborate in detail on technical challenges we had to overcome to implement basic block permutation in Section 5.2.3. Lastly, in Section 5.2.4 we present our evaluation results.

5.2.1 *High-Level Idea*

In general, an application is represented by its corresponding control-flow graph (CFG) which covers all valid execution paths. Due to the linear program memory layout, the CFG will be represented flattened in memory when the program is loaded (see Figure 46). After the program has been loaded into the memory, and its signature has been verified, we transform (including random code permutation) the layout of the control-flow graph (CFG) G to G′. The transformed CFG G′ is isomorphic to G. Hence, it covers the entire control-flows and semantics of the original CFG G.

As core diversification techniques we leverage BBL *permutation* [67], *splitting* [50], and *injection* [50], while the novelty of our approach resides in the fact that we entirely perform these operations at runtime directly after the OS linker has loaded the application into memory:

- **Permutation:** We permute BBLs in memory to move them from their original memory position. For instance, in Figure 46, BBL E is moved from the end to the middle of the memory space. Moreover, we distribute BBLs belonging to a single function across the entire memory space. This introduces a highly randomized memory layout, and tremendously increases the randomization entropy compared to proposals that only reorder functions [22, 114].

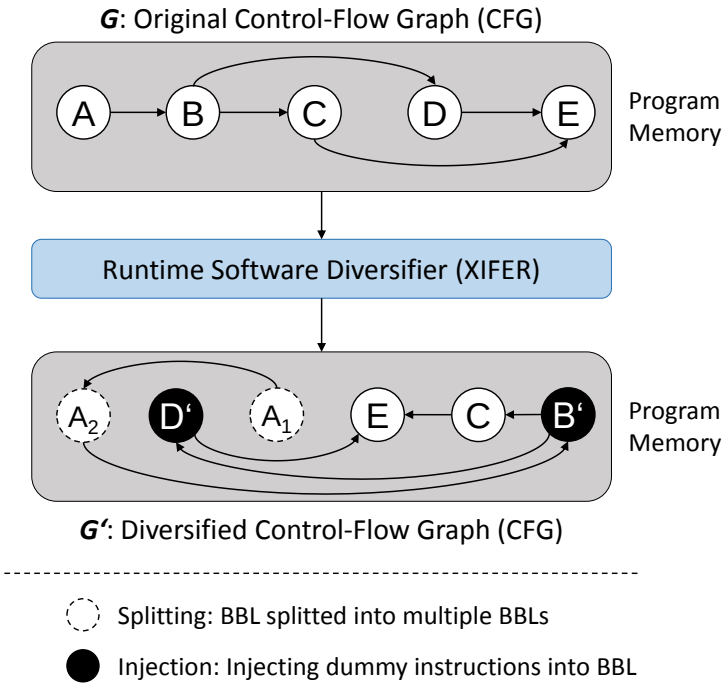


Figure 46: High-level idea of XIFER

- **Splitting:** In addition, we further increase the entropy, by splitting a single BBL in multiple BBLs, and distributing these across the memory space with the same permutation. Hence, our permutation can be applied from BBL down to instruction granularity. The number of those artificial splittings is based on a security parameter. For instance, BBL A has been split into two BBLs A_1 and A_2.

- **Injection:** Finally, we inject new instructions within a BBL (e.g., BBL B is transformed to BBL B'), and insert new (dummy) BBLs into the application. To preserve the program's semantics, the new code will perform only NOP operations.

Note that G' only represents one possible control-flow graph, and the number of possible graph transformations is extremely high: only for the BBL permutation we already achieve an entropy of $n!$, while n denotes the number of BBLs within an application.

Even if the adversary knows that the application suffers from a vulnerability, she cannot launch a code-reuse attack, since the location and structure of all BBLs has been randomized. In addition, our approach is secure against disclosure attacks, where the address of a known function is leaked to the adversary. This is due to the fact that all offsets between functions and BBLs have been randomly changed. Even if the permutation and the memory layout of one specific instance is known, the adversary cannot assume that the target device is using this instance, since our diversification is applied for each application run.

5.2.2 *Design of XIFER*

The design of XIFER is depicted in Figure 47. The workflow is as follows: after the app has been loaded by the OS linker into the memory, we first disassemble the application on-the-fly (Step ❶). Next, we identify all BBLs that belong to the application and derive the basic control-flow graph (CFG) of the application (Step ❷). The next steps (Step ❸ to ❺) involve the diversification of the application which is performed within our runtime rewriter. More precisely, we *inject* new BBLs and NOP instructions into the application, *split* BBLs to multiple BBLs, and finally *permute* the BBLs in memory based on the output of a Pseudo Random Number Generator (PRNG). Since we move and inject BBLs in memory, our rewriter needs to adjust all relative memory offsets and branch targets.

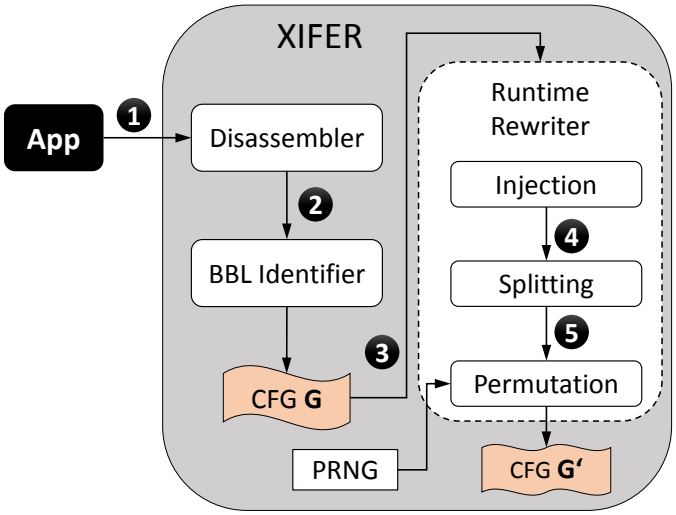


Figure 47: Design of XIFER

5.2.3 *Implementation and Technical Challenges*

The core component of XIFER is the binary rewriter which we implemented for the Android/Linux version for Intel x86 and ARM.

In summary, the main steps involved in the rewriting process of XIFER are (1) loading the executable, (2) disassembling the bytecode on-the-fly, (3) building a reference graph of the executable, (4) applying code transformation, and (5) finally writing the executable back to memory so that it can start executing. To accurately rewrite the application, XIFER only requires relocation information which is also required to enable conventional ASLR protection. We implemented our rewriter and our runtime software diversifier for the popular Linux ELF executable format.

Recall that our goal is to achieve runtime diversification at load-time. This poses several technical challenges as code does no longer reside at its original position. For better understanding, we describe these challenges using an abstract intermediate notation:

the CFG of the application is a directed graph $G = (V, E)$, where V denotes the set of vertices and E the set of edges. Each vertex $v \in V$ denotes a basic block (BBL) of the application. Each vertex $v \in V$ consists of a sequence of ordered instructions $v = (i_1, i_2, ..., i_n)$, where i_{index} denotes a single instruction and n indicates the length of the BBL. The last instruction of the BBL i_n is any branch instruction the processor supports (e.g., direct/indirect jump or call, and function return) and allows the transition from one BBL to another. Finally, an edge $e \in E$ is an ordered tuple $e = (i, j)$, where $i; j \in V$.

Function Returns. BBL permutation is technically more involved than function permutation because the semantics of each function return and conditional branch changes, whenever BBLs belonging to one function are no longer located next to each other.

Consider the example shown in Table 11, where for simplicity we assume that every instruction reserves one byte in memory: in the original program, BBL v_1 terminates with a call instruction that targets BBL v_3 located at address 0x06. Typically, the call instruction will automatically set the return address to the succeeding instruction (i.e., i_3 of BBL v_2). The return instruction of BBL v_3 will exactly use this return address to transfer the execution back to the caller; in this case to i_3 at 0x03.

Addr	Original Program G	Diversified Program G′
0x00:	$v_1 = (i_1, i_2, \text{CALL}\{0x06\})$	...
0x03:	$v_2 = (i_3, i_4, i_5)$	...
0x06:	$v_3 = (i_6, i_7, \text{RET})$	...
...	...	...
0x20:	...	$v_1 = (i_1, i_2, \textbf{CALL}\{\textbf{0x40}\})$
0x23:	...	$v_4 = (\textbf{JMP}\{\textbf{0x60}\})$
...	...	...
0x40:	...	$v_3 = (i_6, i_7, \text{RET})$
...	...	...
0x60:	...	$v_2 = (i_3, i_4, i_5)$

Table 11: Support of BBL permutation for function returns

After moving and permuting BBLs in memory, BBL v_1 is located at 0x20, BBL v_2 at 0x60, and BBL v_3 at 0x40. Hence, the BBLs are no longer next to each other. To preserve the program's semantics, our tool rewrites the call instruction of BBL v_1 so that it uses 0x40 as target address. However, the call instruction would automatically set the return address to 0x23, but i_3 is now located at 0x60. To tackle this issue, we insert a new branch ($jmp\{0x60\}$) after the call instruction. Hence, the return instruction of BBL v_2 will still return to 0x23, but the new inserted branch at this address will redirect the execution to i_3.[2]

Conditional Branches. A similar challenge is caused by conditional branches. In general, these branches are only enforced if a specific condition is met (e.g., one operand

2 Note that it is not possible to replace the return with a fixed branch to address 0x60, because BBL v_3 may be called by other BBLs as well.

is greater than the other). If the condition is not met, the control-flow will *automatically* continue at the instruction which succeeds the conditional branch. To tackle this challenge we use the same technique as for function returns: we insert an (unconditional) branch directly after the conditional branch, which redirects the execution to the original intended instruction (see Table 12).

Addr	Original Program G	Diversified Program G′
0x00:	$v_1 = (i_1, i_2, \textbf{JMPEQ}\{\textbf{0x06}\})$	...
0x03:	$v_2 = (i_3, i_4, i_5)$	...
0x06:	$v_3 = (i_6, i_7)$	...
...	...	...
0x20:	...	$v_1 = (i_1, i_2, \textbf{JMPEQ}\{\textbf{0x40}\}, \textbf{JMP}\{\textbf{0x60}\})$
...	...	...
0x40:	...	$v_3 = (i_6, i_7)$
...	...	...
0x60:	...	$v_2 = (i_3, i_4, i_5)$

Table 12: Support of BBL permutation for conditional branches

Position-Independent Code (PIC). Code that is compiled as position-independent can be executed at an arbitrary memory address. In order to correctly de-reference data from memory, PIC usually looks-up the current value of the program counter. As we will describe in the following, this look-up mechanism will fail if BBLs are no longer next to each other, raising another challenge for our tool.

Table 13 shows a typical example for this look-up: BBL v_1 terminates in a call instruction that simply targets the subsequent instruction ($\mathrm{POP}\{r_0\}$) at address 0x03. On x86, the call instruction pushes the return address (0x03) on the stack, and $\mathrm{POP}\{r_0\}$ will load this address in register r_0. Hence, the program knows (by looking-up r_0) that it is currently executing at memory address 0x03. The second instruction of BBL v_2 leverages this information to de-reference data (AA) located at address 0x10 and loading it into r_1.

In the diversified program, BBL v_1, BBL v_2, and the data are at different memory locations. Hence, the former "call-pop" sequence will no longer de-reference the correct data. To tackle this challenge, our tool performs two operations: it rewrites the call instruction so that it correctly transfers the execution to BBL v_2, and it changes the offset used in the load instruction from 0x07 to 0x3F. Hence, the load instruction will de-reference data from $r_0 + 0x3D$ which equals $0x23 + 0x3D = 0x60$, exactly the address where the intended data (AA) resides in the diversified program.

5.2.4 *Evaluation*

To measure the runtime overhead induced by XIFER, we applied the well-known industry benchmark tool SPEC CPU2006. We performed the evaluation on an Intel Core i5

Addr	Original Program G	Diversified Program G′
0x00:	$v_1 = (i_1, i_2, \text{CALL}\{0x03\})$	...
0x03:	$v_2 = (\text{POP}\{r_0\},$	...
	$\text{LOAD}\{r_1, (r_0 + 0x07)\}, i_3)$	
...	...	...
0x10	AA	...
...	...	...
0x20:	...	$v_1 = (i_1, i_2, \textbf{CALL}\{\textbf{0x40}\})$
...	...	...
0x40:	...	$v_2 = (\text{POP}\{r_0\},$
		$\text{LOAD}\{r_1, (r_0 + \textbf{0x3D})\}, i_3)$
...	...	...
0x60:	...	AA

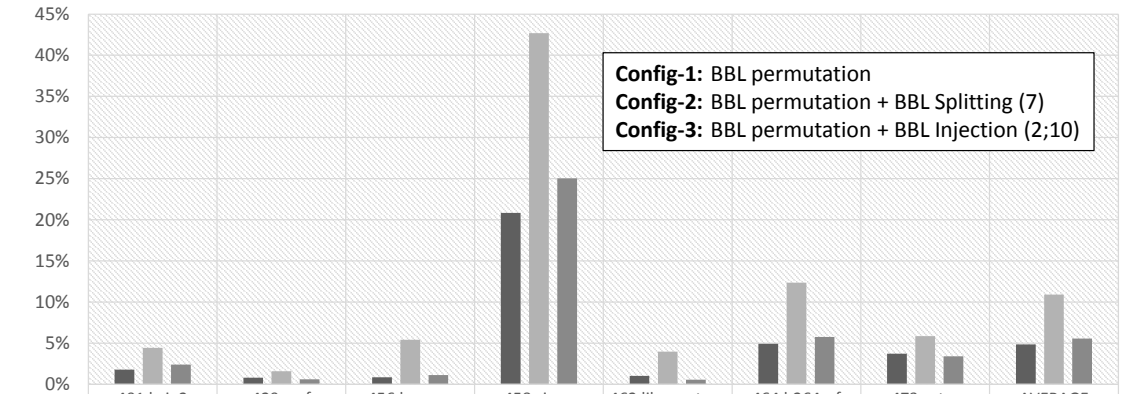
Table 13: Support of BBL permutation for PIC code

(Intel Q57 Chipset) PC that was equipped with 4GB DDR-3 RAM and a 500GB hard disk. On our evaluation PC, we installed Xubuntu 12.04. In particular, we chose the SPEC integer benchmarks *401.bzip2*, *429.mcf*, *456.hmmer*, *458.sjeng*, *462.libquantum*, *464.h264ref*, and *473.astar*. We compiled all benchmarks using gcc-4.5.3 and the uClibc C library. Since our tool does not currently diversify shared libraries we linked all libraries statically to the executable.

We also specified three diversification configurations: (Config-1) only BBL permutation, (Config-2) BBL permutation and BBL splitting after every 7th instruction, and (Config-3) BBL permutation and insertion of two NOPs that are inserted after every 10th instruction.

The results of our evaluation are summarized in Figure 48: for Config-1 the average overhead is only 5%. This basic configuration already has a much higher entropy than most existing software diversity tools and demonstrates the efficiency of our approach. BBL permutation in conjunction with BBL splitting (Config-2) induces a slightly higher overhead (11%). On the other hand, this configuration provides a higher diversification entropy compared to Config-1. Finally, Figure 48 demonstrates that our BBL injection method (Config-3) induces almost the same overhead as Config-1. Hence, BBL injection only adds negligible overhead to BBL permutation.

In contrast to our evaluation on x86, we conducted micro benchmarks for ARM. For this, we used an Android Nexus S device running Android version 4.0.3. To perform precise measurements, we leveraged the ARM hardware clock cycle counter (CCNT) which is part of the system co-processor (CP15). To measure the runtime overhead on ARM, we developed an application that calculates 100 times a SHA-1 hash of a 1K buffer. In addition, we apply a standard bubble sort algorithm on an array of 1024 elements. In average, the runtime overhead for the diversified executable is only 1.52% for the SHA-1 benchmark, and 1.92% for the bubble sort algorithm.



	401.bzip2	429.mcf	456.hmmer	458.sjeng	462.libquantum	464.h264ref	473.astar	AVERAGE
Config-1	1.78%	0.79%	0.86%	20.84%	1.01%	4.93%	3.72%	4.85%
Config-2	4.42%	1.59%	5.41%	42.70%	3.97%	12.35%	5.84%	10.90%
Config-3	2.40%	0.60%	1.11%	25.03%	0.54%	5.76%	3.40%	5.55%

Figure 48: Performance measurements for XIFER

We also measured the performance of the entire rewriting process which is always performed at application load-time. For a 5MB large binary, our XIFER rewrites the entire application after 1.2s.

The XIFER code only reserves 100kb code on disk. Since XIFER instruments an application at load-time, it requires no additional disk space. If only BBL permutation is applied the memory overhead is negligible. Obviously, BBL splitting and injection add new instructions to the address space of the application (depending on the chosen security parameter).

5.3 JUST-IN-TIME CODE REUSE: ON THE LIMITATIONS OF CODE RANDOMIZATION

Fine-grained code randomization schemes, including our tool XIFER, have been introduced to tackle the limitations of base address randomization. Indeed, with fine-grained code randomization enabled, an adversary cannot reliably determine the addresses of interesting gadgets based on disclosing a single runtime address. This is mainly due to the fact that existing code-reuse attacks require the adversary to perform static and offline analysis on the application's binary files to identify useful gadgets. Since ideal fine-grained code randomization performs code randomization at application load-time (*e. g.* as performed by our tool XIFER and STIR [191]), an adversary can hardly mount any code-reuse attack due to the lack of runtime information.

In order to perform a code-reuse attack on randomized code, an adversary needs to defer gadget discovery to application runtime. This is challenging since an adversary needs to locate and read valid code pages in memory. In addition, it requires a disassembly engine and a gadget finding tool which both need to be executed at runtime. On the other hand, many of the modern applications today feature a scripting engine to support client-side scripting, *e. g.* JavaScript for web browsers, VBScript for Microsoft Office applications, or ActionScript for Adobe Flash Player. That is, an adversary can execute some well-defined code (embedded into a script) on the target platform. Although

scripting languages do not provide low-level interfaces to directly access and modify memory pages, they allow us to instantiate a new class of code reuse that we denoted as just-in-time return-oriented programming (JIT-ROP).

5.3.1 *Assumptions and Adversary Model*

We now turn to our assumptions and adversarial model. In general, an adversary's actions may be enumerated in two stages: (1) exercise a vulnerable entry point, and (2) execute arbitrary malicious computations. Similar to previous work on runtime exploits (*e. g.* the original paper on return-oriented programming by Shacham [169]), our assumptions cover defense mechanisms for the second stage of runtime exploits, *the execution of malicious computations*.

In what follows, we assume that the target platform uses the following mechanisms to mitigate the execution of malicious computations:

- **Data execution prevention:** We assume that the security model of data execution prevention is applied to the stack and the heap (cf. Section 2.1.4). Hence, the adversary is not able to inject code into the program's data area. Further, we assume that the same mechanism is applied to all executables and native system libraries, thereby preventing one from overwriting existing code.
- **JIT mitigations:** We assume a full-suite of JIT-spraying mitigations, such as randomized JIT pages, constant variable modifications, and random NOP insertion. As our approach is unrelated to JIT-spraying attacks [24], these mitigations provide no additional protection against our code-reuse attack.
- **Base address randomization:** We assume that the target platform deploys base address randomization by means of ASLR, and that all useful, predictable, mappings have been eliminated.
- **Fine-grained ASLR:** We assume that the target platform enforces fine-grained code randomization on executables and libraries. In particular, we assume a strong fine-grained randomization scheme, which (i) permutes the order of functions [22, 114] and basic blocks [191, 63], (ii) swaps registers and replaces instructions [149], and (iii) performs randomization upon each run of an application [191, 63].

Nevertheless, even given all these fortified defenses, we show that our framework for code-reuse attacks can readily undermine the security provided by these techniques. We only assume that the adversary can provide a single leaked runtime address (*e. g.* a function pointer) to our framework.

5.3.2 *Basic Attack Principle*

JIT-ROP circumvents fine-grained ASLR by finding gadgets and generating the return-oriented payload on-the-fly at runtime. As for any other real-world code-reuse attack, it

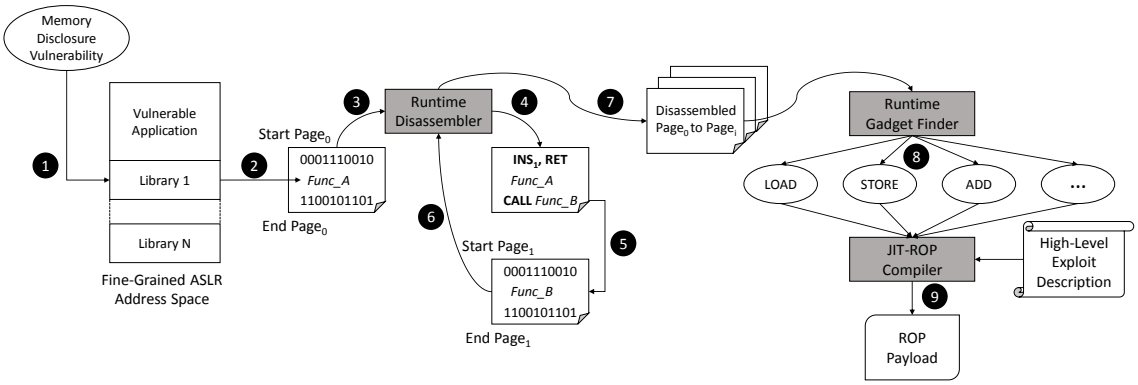


Figure 49: Workflow of a JIT-ROP attack

only requires a memory disclosure of a *single* runtime address. However, in contrast to standard code-reuse attacks against ASLR, JIT-ROP only requires the runtime address of a valid code pointer, without knowing to which precise code part or function it points to. Hence, JIT-ROP can use any code pointer such as return addresses on the stack to instantiate the attack. Based on that leaked address, JIT-ROP discloses the content of multiple memory pages and generates the return-oriented payload at runtime. The detailed workflow of a JIT-ROP attack is shown in Figure 49, where we assume that fine-grained ASLR has been applied at application load-time to each executable module in the address space of the vulnerable application.

First, the adversary exploits a memory disclosure vulnerability to retrieve the runtime address of a code pointer (Step ❶). One of the main observations of JIT-ROP is that the disclosed address will reside on a 4KB-aligned memory page ($Page_0$ in Figure 49). Hence, at runtime, we can identify the start and end of $Page_0$ (Step ❷). Afterwards, JIT-ROP deploys a runtime disassembler, whose task is to disassemble $Page_0$ on-the-fly (Step ❸). The disassembled page (Step ❹) provides 4KB of gadget space, but more importantly, it will likely contain direct branch instructions to other pages, e.g., a call to Func_B (Step ❺). Since Func_B resides on another memory page (namely $Page_1$), JIT-ROP can again determine the page start and end, and disassemble $Page_1$ (Step ❻). This procedure can be repeated as long as JIT-ROP identifies new direct branches pointing to yet undiscovered memory pages (Step ❼). Based on the disassembled pages, JIT-ROP deploys a runtime gadget finder to identify useful gadgets such as LOAD, STORE, or an ADD (Step ❽). Finally, JIT-ROP generates the return-oriented payload based on the discovered gadgets and a high-level description provided by the adversary (Step ❾). All the different components involved in a JIT-ROP attack are embedded into a single exploit file (such as a JavaScript file for browser-based attacks).

5.3.3 *Implications*

Due to the fact that JIT-ROP entirely performs at runtime and does not rely on any offline static analysis, it can circumvent conventional as well as fine-grained ASLR solutions. Since it also requires only a single code pointer to instantiate the attack (without

precisely knowing to which specific code or function the code pointer points to), we believe that JIT-ROP will soon be leveraged in real-world return-oriented exploits.

In our evaluation, we demonstrate the power of JIT-ROP by applying it to exploit Internet Explorer (IE) 8 running on Windows 7 using CVE-2012-1876. As a proof-of-concept, the vulnerability is used to automatically load the Windows Calculator application upon browsing a HTML page. The memory disclosure vulnerability we exploited to instantiate our attack allowed us to harvest 301 code pages from the Internet Explorer process (including those pages harvested from library modules). That is, the adversary can disclose the content of $\approx$ 1.2MB randomized code. The leaked code pages provided us with a large code base (including a large amount of gadgets) to perform code-reuse attacks. In particular, we discovered many useful call sites to dangerous functions such as *LoadLibrary()* and *GetProcAddress()* which allow us to invoke any function we desire. The number of code pages leaked naturally depends on the starting pointer. For the interested reader, we performed an extensive evaluation on how many pages and gadgets can be discovered starting code page harvesting from different pointers [172].

A knee-jerk reaction to mitigate JIT-ROP attacks is to simply re-randomize code pages at a high rate; doing so would render our attack ineffective as the disclosed pages might be re-randomized before the just-in-time payload executes. While this may indeed be one way forward, we expect that the re-randomization costs [191, 63] would make such a solution impractical. In fact, re-randomization is yet to be shown as an effective mechanism for user applications.

Another potential mitigation technique is instruction set randomization (ISR) (*e. g.* [20, 112]), which mitigates code injection attacks by encrypting the binary's code pages with a random key and decrypting them on-the-fly. Although ISR is a defense against code injection, it complicates code-reuse attacks when it is combined with fine-grained ASLR. In particular, it can complicate the gadget discovery process because the entire memory content is encrypted. On the other hand, ISR has been shown to be vulnerable to key guessing attacks [176, 193]—that become more powerful in the face of memory disclosure attacks like ours,—suffers from high performance penalties [20], or requires hardware assistance that is not yet present in commodity systems [112].

5.4 ISOMERON: EXECUTION-PATH RANDOMIZATION

Unfortunately, just-in-time return-oriented programming (JIT-ROP) effectively undermines code randomization. To counter that threat, we investigated a hybrid approach that leverages code and execution-path randomization to address the shortcomings of fine-grained code randomization techniques proposed to date. In particular, our approach dynamically randomizes the execution-path between the original application and a diversified – but semantically equivalent – application. Our prototype implementation, dubbed ISOMERON [66], can be readily applied to existing applications since it neither requires access to source code nor an offline static analysis phase. In general, our new defense framework performs as follows:

1. It takes the original executable code (C) and creates a copy C′. In addition, it deploys a code diversifier to apply fine-grained ASLR to C′.
2. We load the original and diversified executable code (C and C′) in the address space of the application.
3. A coin is flipped to continuously randomize the execution-path between C and C′. Specifically, each time a function is called, a coin is flipped and the program continues execution either in the original C or C′.

In contrast to previous work on fine-grained ASLR, our solution prevents both traditional return-oriented programming attacks and JIT-ROP attacks. Due to the continuous random choice between C and C′, the adversary's gadget chain will break. In particular, the adversary can no longer predict whether C or C′ will be executed next.

5.4.1 *Assumptions*

In this section, we describe our assumptions with regards to defense mechanisms in-place on the target system, and the capabilities of the adversary.

5.4.1.1 *Deployed Defense Mechanisms*

On the target platform, we assume the enforcement of the non-executable memory security model and fine-grained address space layout randomization (ASLR).

- **Data execution prevention**: We assume that the target platform deploys the data execution prevention (cf. Section 2.1.4).
- **Fine-grained ASLR**: We also assume the presence of fine-grained ASLR (cf. Section 5.1). In particular, we require that fine-grained ASLR eliminates and breaks the return-oriented gadgets inside an executable module. Note that breaking and eliminating in this context does not mean that the new executable module produced by fine-grained ASLR will not contain any useful return-oriented gadgets. Instead, it means that the original return-oriented gadgets either (i) do reside at a different offset in the randomized executable, (ii) are eliminated in the randomized executable by replacing instructions with an equivalent instruction[3], or (iii) are broken because of instruction reordering or register replacement. This requirement is fulfilled by *all* the proposed fine-grained ASLR solutions [149, 97, 191, 63].

3 Recall that, on x86, return-oriented programming attacks can leverage unintended instruction sequences. Since fine-grained ASLR replaces instructions with equivalent instructions the bytestream changes. Hence, the unintended instruction will have a different semantic than before.

5.4.1.2 *Capabilities of Adversary*

The capabilities of the adversary are as follows.

- **Exploiting memory vulnerabilities**: Memory vulnerabilities such as buffer overflow errors are one of the most common types of vulnerabilities we found in today's software programs. Hence, we assume that the software running on the target platform suffers from a vulnerability allowing an adversary to instantiate a runtime exploit.
- **Exploiting memory disclosure**: We also assume that an adversary can gain access to all code pages of the main application and its shared libraries. Such memory disclosure attacks are indeed realistic as recent attacks like JIT-ROP have shown [172]. This also implies that we target an adversary who can circumvent fine-grained code randomization schemes, which we assume to be deployed on the target system. At first glance, it seems contradictory that our adversary can bypass one of our defense mechanisms we assume. However, as we will show in the remainder of this section, our defense mechanism deploys some techniques of fine-grained ASLR *without* being vulnerable against memory disclosure attacks.

On the other hand, we assume that a very small amount of data memory, maintained by ISOMERON to correctly enforce execution-path randomization, is not accessible to the adversary. This can be either achieved through hardware-based isolation techniques such as segmentation or by deploying software fault isolation techniques [187]. That said, we assume the adversary to gain access to all code and data memory except the data area of ISOMERON.

5.4.2 *High-Level Idea*

In this section we present the high-level idea of our defense technique to protect against traditional and just-in-time return-oriented programming attacks.

Our idea is to enforce code and execution-path randomization simultaneously. Figure 50 shows a simplified view of the address space of such a protected application: it contains the original application as well as linked libraries. Further, for each of these executable segments, we create a diversified copy, and maintain them in the same address space of the application. Hence, in the same address space resides the original and a corresponding diversified program image. To create the diversified program, we deploy the principles of fine-grained ASLR, where we pose the requirement that fine-grained ASLR eliminates or breaks the gadgets in the diversified image (cf. Section 5.4.1.1).

The novelty of our approach is to combine fine-grained code randomization with *execution-path randomization* at runtime. At the granularity of function calls, we randomly decide whether to continue execution on the diversified or the original program image. This random decision is repeated whenever a function call occurs, and our execution diversifier ensures that a function is completely executed either from the original or diversified program image.

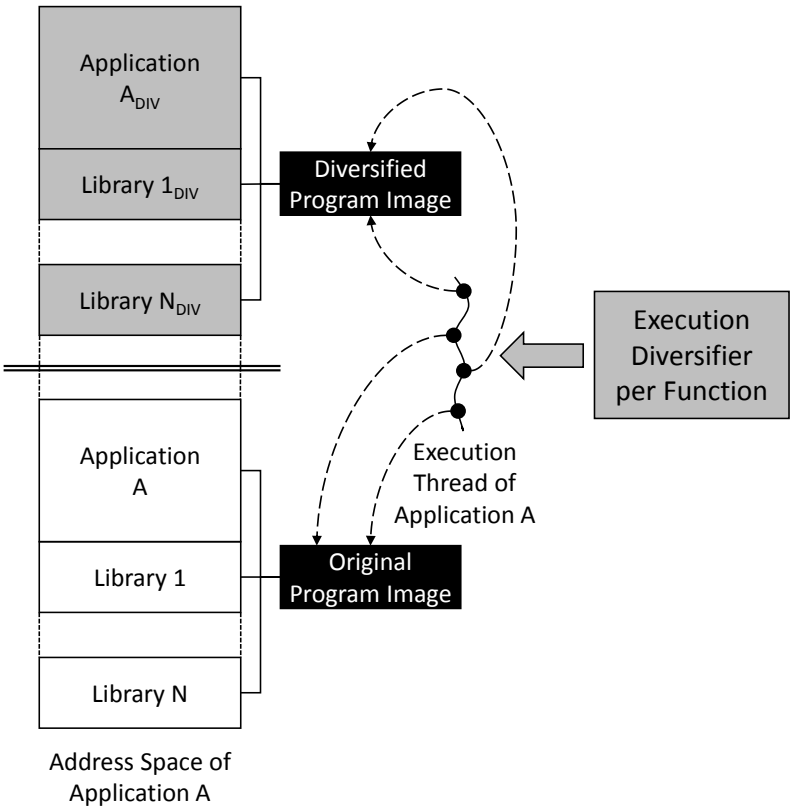


Figure 50: High-level approach to defeat traditional and just-in-time return-oriented programming attacks

5.4.2.1 *Step One: Program Twinning*

We first tackle the problem of cloning a program image within a single virtual address space. Our solution applies for this principles of dynamic instrumentation. Similar to common instrumentation frameworks, we instrument code on the granularity of basic blocks (BBLs). However, instead of emitting a single BBL into one instrumentation cache, our framework can emit multiple (diversified) copies into multiple instrumentation caches. This specific feature of program twinning required us to develop our own instrumentation framework, since currently available solutions do not per-se support code duplication.

5.4.2.2 *Step Two: Twin Diversification*

At runtime, we apply fine-grained ASLR on the executable code of an application. The level of fine-grained ASLR is configurable, but we require and ensure that each possible instruction sequence and gadget is placed at a different address. In other words, it should not happen that an instruction sequence resides at the same offset in the original and diversified version [114, 149, 97, 191, 63].

5.4.2.3 *Step Three: Coin-Flip Instrumentation*

At the granularity of function calls, we randomly decide whether to continue execution on the diversified or the original program image. This random decision is repeated whenever a function call occurs, and our execution diversifier ensures that a function is completely executed either from the original or diversified program image to preserve the program's original semantics. The rationale behind performing the random decision on a per-function level stems from the fact that we can only preserve the original semantics of the application when a function is entirely executed from either the original or the diversified address space.

Instrumentation of Direct Function Calls. In order to perform the random decision, we need to ensure that we gain control when a function call occurs. Figure 51 shows in detail the instrumentation of function calls. As a running example, we use an application that consists of only two functions: *Func_A()* and *Func_B()*. The latter function only contains the x86 return instruction (`RET`), while the former one contains two instructions (`INS1` and `INS2`), and a function call to *Func_B()*. The only code diversification we apply in this example is that `INS1` has been exchanged with `INS2`.

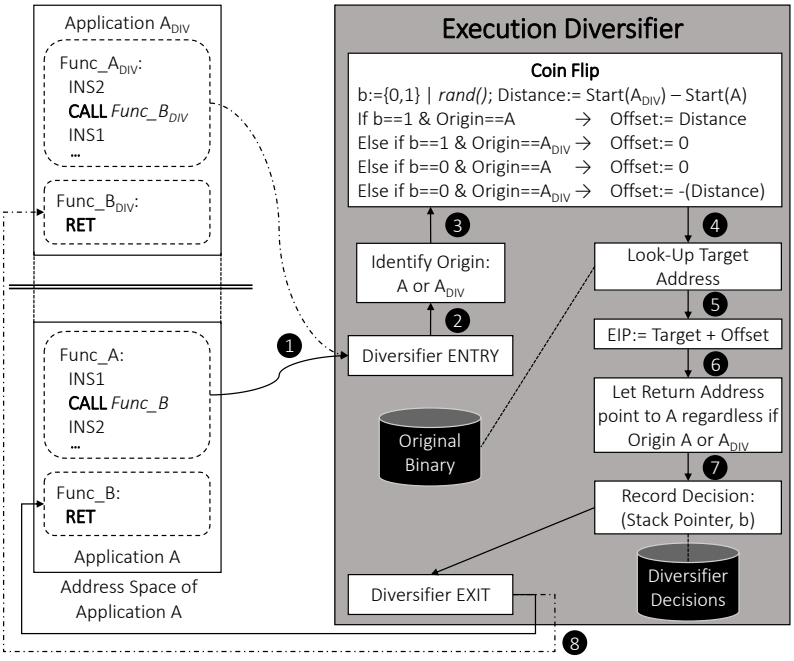


Figure 51: Instrumentation of function calls in Isomeron

For our running example, the function call to *Func_B()* will either be triggered on the original or diversified program image. This depends on where the program has started executing, which is randomized in our approach as well. Since the call has been instrumented, the control-flow will be dispatched to our execution diversifier (Step ❶).

Since the (instrumented) function call will automatically push the return address onto the stack, our execution diversifier can easily identify from where it has been invoked

(Step ❷). Note that the memory layout of the running application is known to our execution diversifier.

Next, we perform a random coin flip (Step ❸) that outputs a random bit $b \in (0, 1)$. Further, our execution diversifier calculates the memory distance between the diversified and original image. Note that this calculation only needs to be performed once, and can be looked-up in future coin-flip rounds. Based on the origin and the value of b, we calculate the *offset* to be added on the instruction pointer. In general, the offset is zero, when execution should be continued on the image from where the function call originates. If a program image transition is needed, the offset will simply be the distance between the two program images.

In Step ❹ and ❺, we look-up the original target address of the call by using the original binary, and add the offset (calculated in Step ❸) to determine the new value of the instruction pointer (on x86: `EIP`).

In addition, we ensure that the return address on the stack always points to the original program modules (Step ❻). Otherwise, an adversary could determine which functions are currently executing in the original and the diversified version by inspecting the stack. Hence, to ensure that function returns are correctly returning to their caller in the correct program version, we also record each diversifier decision (Step ❼). Recall that we assume that the adversary cannot access this memory area (cf. Section 5.4.1.2). Lastly, our execution diversifier redirects the control-flow to *Func_B()* (Step ❽).

Instrumentation of Function Returns. Figure 52 depicts the instrumentation for function returns. We instrument function returns similar to function calls. That is, we gain control in our execution diversifier whenever the program issues a return instruction (Step ❶). To identify the origin, we use a slightly different approach from the one used for function calls. Specifically, we read the current value of the stack pointer because it points to the return address the program attempts to use. Recall that the return address will always point to the original program image, but we stored the corresponding decision in our execution diversifier. Hence, we make a request to our decision database for the current stack pointer (Step ❸). In case execution needs to be redirected to the diversified program version, we add the distance between the two program images. Otherwise, we take the return address from the program stack. Since the adversary does not know the previous diversifier decisions, she can only guess where execution will continue. However, this information is crucial because either `INS1` or `INS2` will be executed next.

Instrumentation of Indirect Jumps and Calls. The case of indirect jump and calls is handled similar to their direct counterparts, with the exception that the target address is calculated at runtime. To mitigate the misuse of indirect branches, we limit the possible destination addresses to those, which are included in the relocation information. Using the relocation information, we can reliable identify jump tables and limit the indirect jumps to targets within the jump table. Due to the fine-grained randomization, we also have to ensure that jumps take place within the same copy. For indirect calls, we determine the target address, make sure that it is a valid target address according to our policies, and proceed then in the same way as we are handling direct calls.

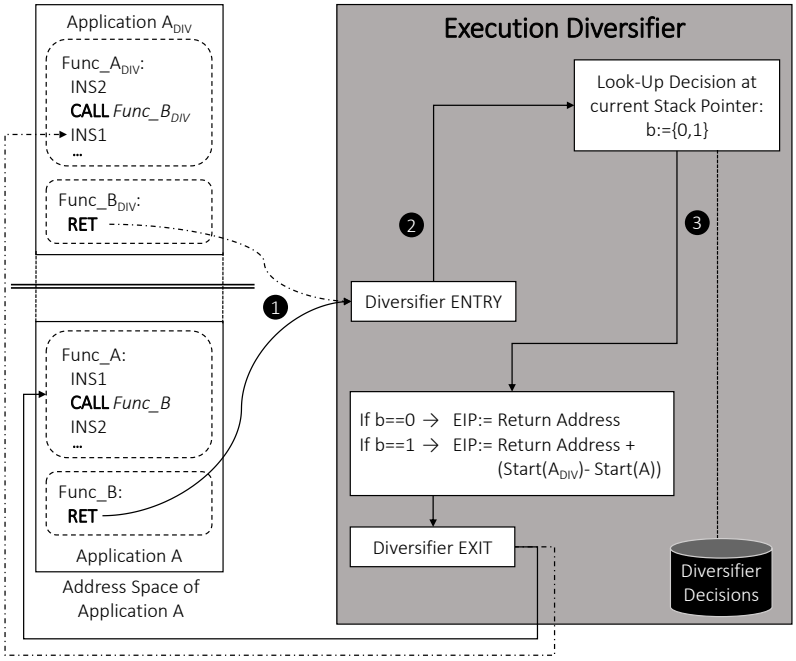


Figure 52: Instrumentation of function returns in ISOMERON

ISOMERON prevents conventional and JIT-ROP attacks because the adversary cannot reliably build a return-oriented gadget chain. Each image will contain a different set of gadgets since fine-grained ASLR eliminates or breaks the original gadgets in the diversified program image. Due to execution-path randomization, the adversary has a probabilistic chance of 50% for *each* pointer in the return-oriented payload that the execution will transfer to the attacker's intended gadget.

5.4.3 *Implementation and Evaluation*

In this subsection, we briefly summarize the implementation details and evaluation results of ISOMERON. For a more detailed description, we refer the reader to [66].

In general, ISOMERON can be either implemented as a compiler extension or a binary instrumentation-based solution. We opted for the binary instrumentation-based approach as it allows us to instrument applications where source code is not readily available. To this end, we developed a novel dynamic binary instrumentation (DBI) framework which performs instrumentation at application runtime. Note that we could not re-use existing DBI frameworks such as Pin [127] or DynamoRIO [29] since these frameworks do not allow us to emit differently instrumented copies of the same code.

Our new DBI framework is realized as a Windows dynamically linked library (DLL). We intercept basic blocks, analyze, translate, and emit them into a code cache. One major difference to existing instrumentation frameworks is that our translator emits an additional instrumented, diversified basic block into a second code cache. Our execution-path diversifier switches the execution between both code caches.

Performance. DBI frameworks generate a significant performance overhead by design. This is due to the fact that code is instrumented at runtime. While DBI supports dynamically generated code, and allows us to avoid any static analysis, it adds additional performance overhead. To measure the performance overhead, we applied ISOMERON to SPEC CPU2006 benchmark tools. As a baseline, we used Intel's DBI PIN [127] without any instrumentation. That is, we just let the benchmarks execute under PIN. We observed an average overhead of 19% which indicates that ISOMERON add modest performance overhead compared to a state-of-the-art DBI framework running without any instrumentation.

5.5 RELATED WORK

In this section, we elaborate on related work on code randomization. First, in Section 5.5.1, we investigate classic fine-grained code randomization schemes that do not resist just-in-time code-reuse attacks. Afterwards, in Section 5.5.2, we elaborate on recently proposed code randomization schemes that aim at preventing just-in-time code-reuse attacks. We also refer the reader to a recently published systematization of knowledge paper by Larsen et al. [120] that provides a detailed comparison of fine-grained code randomization schemes.

Note that we will not elaborate on randomization schemes that target the data area of an application. Such schemes typically tackle the first step of a runtime exploit, *i.e.* the initial pointer overwrite. As mentioned before, we assume the target program to suffer from a memory corruption error that an attacker can exploit to instantiate a runtime exploit.

5.5.1 *Classic Code-Randomization Schemes*

We distinguish between compiler-based and binary-instrumentation based approaches to classic fine-grained code randomization.

5.5.1.1 *Source Code and Compiler-Based Solutions*

The original software diversity approach targets and proposes a compiler-based solution [50]. In particular, Cohen [50] elaborates on a number of randomization techniques such as instruction equivalence, garbage insertion, instruction reordering, as well as adding/removing jump and call instructions. Franz [82] and Jackson et al. [106] have explored the feasibility of a compiler-based approach for large-scale software diversity in the mobile market. The authors suggest that app store providers integrate a multi-compiler (diversifier) in the code production process. However, this approach has some shortcomings. Typically, app store providers have no access to the applications' source code. Hence, they cannot generate thousand of diversified app copies. Further, once an app instance is installed on the customer's device, it remains unchanged until an update is provided, which increases the chance of an adversary compromising this particular instance.

Whereas the aforementioned schemes do not provide any implementation details and evaluation, Jackson et al. [107] implement and evaluate a diversity compiler for LLVM and GCC. Their randomization technique is solely based on inserting non-alignment NOP instructions allowing protection of the entire system including the kernel. To improve performance, Homescu et al. [100] apply profiling techniques to identify frequently executed code parts. For these hot code parts, Homescu et al. [100] sparsely insert NOP instructions, which yields significantly better performance. Moreover, Homescu et al. [99] explore NOP diversification for dynamically-generated code. However, emitting NOP instructions does not yield a high randomization entropy, because basic block and function order remain unchanged.

As demonstrated by Shacham et al. [170], randomizing an application only once allows an attacker to launch brute-force attacks. In particular, code-reuse attacks are easily possible if an attacker gains access to the applications' binary files, because code randomization is only applied at compile-time. To tackle this shortcoming, Bhatkar et al. [22] presented an automatic source code transformer and its implementation for x86/Linux. The main idea is to transform the source code of an application to a source code that supports re-diversification for each run. In particular, the authors perform function reordering to mitigate code-reuse attacks with an average overhead of 11%.

Most fine-grained randomization schemes focus on applying code randomization to user-level applications. Instead, Giuffrida et al. [88] introduce a fine-grained code randomization scheme that is specifically tailored to randomize operating system kernels. The presented solution operates on the LLVM intermediate representation, and applies a number of randomization techniques. However, the approach of Giuffrida et al. [88] is best suited to microkernels, while most modern operating systems (Windows, Linux, Mac OSX) still follow a monolithic design.

In general, compiler-based solutions have the potential to provide among all software diversity approaches the highest degree of entropy due to the access to source code. However, source code is rarely available in practice, and current app store models are not compatible to a multicompiler approach. In contrast, our fine-grained ASLR tool XIFER can directly be applied to third-party apps, and keeps compliance to code signing by diversifying the application in memory.

5.5.1.2 *Binary-Instrumentation Based Solutions*

These techniques directly operate on the application binary to perform software diversity. Bhatkar et al. [21] analyze and propose enhanced randomization techniques such as code transformation and insertion of random gaps between adjacent objects. However, the authors only implemented a tool that performs base address randomization and insertion of random gaps, while code transformation techniques such as function permutation have not been realized.

Kil et al. [114] address this shortcoming, and introduce address space layout permutation (ASLP), a fine-grained randomization scheme performing function permutation. The proposed scheme statically rewrites ELF executables to permute all functions and data objects of an application. Moreover, the underlying Linux kernel has been instrumented to increase the randomization entropy for the base address randomization of shared libraries.

Whereas Kil et al. [114] target Linux and require accurate disassembly, Pappas et al. [149] explore fine-grained code randomization for Windows executables where no side information (*e. g.* relocation information) is available. Their tool, called ORP, randomizes instructions and registers within a basic block to mitigate return-oriented programming attacks, but leaves all functions at their original position. Hence, it cannot prevent pure return-into-libc attacks. A more fine-grained randomization approach is taken by Hiser et al. [97]: ILR (instruction location randomization) randomizes the location of each single instruction in the virtual address space, while the execution is guided by a so-called

fall-through map. For this, a program needs to be analyzed and re-assembled during a static analysis phase. In particular, the static analysis phase creates the fall-through map and applies binary rewriting to diversify the location of each instruction. However, ILR induces significant performance overhead, is not compatible to memory models that deploy a cache, and suffers from a high space overhead. In fact, the rewriting rules reserve on average 104 MB for only one benchmark of the SPEC CPU benchmark suite. Furthermore, ILR suffers from code coverage deficiencies because it does not randomize many instructions of the program due to imprecision of the static analysis phase. This concerns in particular the destination addresses of indirect jumps, indirect calls, and function returns. Still, ILR randomizes the location of these instructions in memory, but deploys a translation mechanism that maps the non-randomized destination addresses to randomized addresses at runtime.

In contrast to ILR and ORP, XIFER efficiently mitigates return-into-libc attacks by randomizing function entry points for each run, without deploying a translation mechanism. Furthermore, ASLP, ORP, and ILR rewrite the binary in an (offline) static analysis phase, and thereby are not compatible to application signatures. Static rewriting has also another obvious drawback: a program will be only re-diversified if the static analysis phase is repeated. Static analysis requires relatively long time. For instance, in [97] it takes already 36s for only randomizing the binary; *not* including the time for generating the rewriting rules. Hence, it is very likely that most program binaries will repeatedly execute with the same diversification layout, increasing the chance for an adversary to learn the program layout.

Wartell et al. [191] presented binary stirring (STIR). Similar to our approach, STIR performs basic block permutation for each application. For this, STIR leverages static analysis and load-time randomization. In the static analysis phase the binary is transformed to facilitate load-time randomization. In particular, STIR creates a copy of the code section. The original code section is treated as a data-only section to preserve static data interleaved with code at its original address. In contrast, the copy of the code section is disassembled to allow load-time basic block randomization. At runtime, a trusted library takes control first and performs basic block permutation before the program starts executing. Similar to the aforementioned approaches, STIR is not compatible to code signatures since it statically transforms the binary. In addition, it significantly increases the file size; on average by 73%. On the other hand, it performs highly efficient incurring only 1.6% performance overhead. In addition, it requires no relocation information. An open security issue concerns the address translation from old target addresses to new randomized addresses. According to the description given by Wartell et al. [191], STIR allows translation for all function start addresses. As a consequence, in contrast to XIFER, it cannot prevent return-into-libc attacks.

Whereas our tool XIFER and STIR [191] focus on randomizing basic blocks, Gupta et al. [94] focus on function permutation for Linux ELF binaries. Their fine-grained randomization scheme, called Marlin, identifies function blocks by using the public tool `Unstrip`, a tool that restores symbol information. Marlin is based on randomizing the functions' symbols to enable function permutation.

5.5.2 *Advanced Code-Randomization Schemes*

Unfortunately, conventional approaches to fine-grained code randomization can be circumvented with our just-in-time return-oriented programming (JIT-ROP) attack (cf. Section 5.3). This is mainly due to the fact that randomization in these schemes only occurs once at compile-time or load-time. Hence, disclosure of a large amount of code pages allows an adversary to read randomized code and build a return-oriented exploit thereof.

To tackle the limitations of conventional code randomization schemes one could continuously re-randomize the program layout. That is, an adversary would still be able to read randomized code, but the randomized code is replaced before the adversary launches the return-oriented exploit. In fact, one of the schemes we have discussed above enables re-randomization: Giuffrida et al. [88] allow program modules of a microkernel to be re-randomized after a specified time period. Unfortunately, re-randomization induces significant runtime overhead, *e. g.* nearly 50% overhead when applied every second. In addition, it remains questionable whether re-randomization can be applied to user-level applications.

As a matter of fact, the aforementioned instruction location randomization (ILR) proposal suggested by Hiser et al. [97] cannot be directly bypassed by JIT-ROP attacks. This is because execution is guided based on a fall-through map that defines the successor instruction of each single instruction. On the other hand, as we discussed above, ILR suffers from several practical problems. As a consequence, a number of new and advanced code randomization schemes have been suggested to efficiently and effectively prevent JIT-ROP attacks.

Backes and Nürnberger [18] propose Oxymoron, a fine-grained code randomization scheme that enables code sharing for randomized shared libraries. In particular, Oxymoron hides direct code references embedded in direct call and jump instructions to defend against JIT-ROP attacks. Recall that JIT-ROP attacks follow these references to identify and disassemble valid code pages. As such, the adversary is restricted to a single code page, *i. e.* the page that is referenced by the initial leaked pointer (Page_0 in Figure 49). The hiding of these crucial references is achieved by exploiting memory segmentation on Intel x86, and adding an extra layer of indirection. That is, direct calls and jumps are replaced with indirect calls and jumps that leverage a segment register and an index as their base register to redirect control-flow to a secret table that contains the original target addresses. Unfortunately, Oxymoron can be bypassed with an improved version of JIT-ROP attacks [66]: an attacker can harvest multiple function pointers and return addresses residing on the application's stack and heap. These pointers allow the attacker to reliably locate and disclose many code pages, and launch a return-oriented exploit.

The improved JIT-ROP attack is possible because code pages still remain readable in Oxymoron. Hence, any valid code pointer leads to a 4 KB code page that can be leveraged for a code-reuse attack. As a consequence, Backes et al. [19] set code pages to non-readable in a follow-up work. Their approach, denoted as XnR (eXecute-no-Read), sets all code pages except the page on which execution currently takes place to non-present. Hence, an adversary can only read from the currently executing code page, but

not from any other page. A dedicated page fault handler is triggered whenever the program leaves the current page, and aims to execute code from a non-present page. In such cases, the non-present page is set to executable and readable, while the previous page is set to non-present. To improve performance, XnR keeps a window of multiple memory pages that are set to executable and readable. XnR is only effective against conventional code-reuse attacks if it is deployed in conjunction with a fine-grained randomization scheme. However, it remains unclear which randomization scheme is best suited to work in conjunction with XnR. Further, code pages, where execution currently takes place, still need to be mapped as readable. This allows an attacker to read the currently executing page. Hence, if an attacker continuously reads code pages during program execution, she might be able to collect a sufficiently large code base to launch a return-oriented exploit. In particular, this is a security problem when code is not re-randomized per application run. Gionta et al. [87] tackle this shortcoming by leveraging a split Translation Lookaside Buffer (TLB) architecture which allows code pages to be only executable and not readable. However, split-TLB based architectures do no longer exist in modern processor architectures.

Recently, we investigated a hardware-based approach to enable execute-only memory [55]. Our framework Readactor exploits Intel's extended page tables (EPT) to conveniently mark memory pages as non-executable. In addition, we developed a LLVM-based compiler that i) performs function permutation, ii) strictly separates code from data, and iii) hides code pointers. We require the latter to tackle entropy problems of several fine-grained randomization schemes. Consider a fine-grained randomization scheme that only performs randomization at the granularity of pages [18] or at the granularity of functions [114]. Although these schemes are more efficient than instruction-level randomization, they allow an attacker to determine a reasonable code base: given a single leaked function pointer, the attacker can statically determine all gadgets on the corresponding memory page (for page-level randomization), and all gadgets inside the corresponding function (for function-level randomization). In Readactor, we limit such memory disclosure threats by introducing a layer of indirection. Specifically, we dispatch all indirect branches through dedicated non-readable trampolines. The attacker has only access to the trampoline addresses, but not to the actual runtime addresses of functions and call sites. However, the trampoline addresses provide a small code base which may be exploited for a code-reuse attack.

5.6 CONCLUSION AND SUMMARY

The principle of software diversity has recently facilitated the design and implementation of many new defenses against code-reuse attacks. In particular, fine-grained code randomization aims at significantly complicating code-reuse attacks which rely on statically determined code sequences (gadgets). Whereas most schemes proposed so far only perform fine-grained code randomization once, we introduce a tool, called XIFER, which efficiently enforces basic block permutation for each application run without requiring the source code of the application.

We also examined the underlying assumptions of fine-grained code randomization schemes (including our own approach), and noticed that they only defend against code-reuse attacks where the attacker determines gadgets within an offline static analysis phase. As a consequence, we investigated the feasibility of dynamic code-reuse attacks which determine gadgets on-the-fly without any static analysis. Our analysis reveals that just-in-time return-oriented programming (JIT-ROP) attacks can be generated and applied to modern applications such as web browsers. In fact, our attack demonstrates that memory paging in modern processor architectures can be exploited to disclose the randomized code of many memory pages based on a single leaked runtime address. To tackle these novel code-reuse attacks, we explored a combination of code randomization with execution-path randomization in our binary instrumentation tool ISOMERON. Although our new approach to fine-grained randomization resists JIT-ROP attacks that acquire knowledge of all randomized code pages, it currently suffers from performance overhead which is due to dynamic binary instrumentation. Hence, we will investigate compiler-based and approaches based on static rewriting to improve the performance of ISOMERON.

6 DISCUSSION AND CONCLUSION

Modern runtime exploits perform malicious program actions based on the principle of code reuse. These attacks require no code injection, bypass widely-deployed defense mechanisms, allow Turing-complete computation, can be applied to many processor architectures, and are highly challenging to prevent. Large-scale cyber attacks that have been recently discovered such as the well-known Stuxnet virus include code-reuse attack techniques [129].

As we have shown in this dissertation, a large number of defenses and improved code-reuse attacks have appeared in the academic literature. Moreover, Microsoft recently released a new exploitation defense tool, called EMET, which includes a number of mitigation techniques to prevent runtime exploits [131]. Furthermore, the upcoming version of Windows 10 will include a new feature, called control-flow guard, which aims at enforcing control-flow checks for indirect calls to limit code-reuse attacks [133]. Google recently explored defenses against code-reuse attacks and introduced an instrumented compiler toolchain that enables control-flow checks for indirect calls [182].

The main objective of this dissertation is to explore the limitations of existing defenses against code-reuse attacks, and introduce the design and implementation of novel defense schemes that mitigate these attacks. We briefly summarize the main results of this dissertation in Section 6.1. Next, in Section 6.2, we provide a discussion on control-flow integrity (CFI) and fine-grained code randomization schemes. Lastly, we elaborate on future research directions in Section 6.3.

6.1 DISSERTATION SUMMARY

In Chapter 3, we presented two advanced code-reuse attacks. Our first attack demonstrates that defense mechanisms that are solely enforcing control-flow checks for return instructions can be bypassed. Our attack technique introduces Turing-complete code-reuse attacks based on the exploitation of indirect jump and call instructions targeting mobile devices with an underlying ARM processor [41]. The second attack demonstrates limitations of recently proposed control-flow integrity (CFI) schemes. In particular, we are able to stitch gadgets from call-preceded sequences thereby undermining behavioral-based heuristics [64].

In Chapter 4, we introduce new CFI-based defenses against code-reuse attacks. CFI inserts control-flow checks for indirect branch instructions that validate at runtime whether an application follows a legitimate control-flow path [4]. Our CFI framework MoCFI enables for the first time CFI for mobile devices [62]. To this end, we developed a binary rewriter that instruments mobile applications at application load-time to preserve static code signatures. Our reference implementation of MoCFI targets the closed-source iOS operating system. We also demonstrated that MoCFI can be leveraged as an inlined

reference monitor to enable fine-grained application sandboxing for iOS-based devices. However, we also observed that software-based CFI schemes (including MoCFI) suffer from performance overhead. As a consequence, we explored hardware-assisted CFI for embedded systems, and introduced a CFI hardware implementation targeting Intel Siskiyou Peak. Our approach, called HAFIX, features dedicated CFI instructions and CFI memory, and efficiently protects function returns while only incurring 2% performance overhead.

In Chapter 5, we investigate fine-grained code randomization as an alternative defense approach to defend against code-reuse attacks. Our tool XIFER performs basic block permutation for each application run [63]. Hence, the adversary can no longer statically determine the memory addresses of useful code sequences for a code-reuse attack. However, most fine-grained code randomization schemes can be bypassed by means of just-in-time code-reuse attack technique. These attacks exploit a memory disclosure vulnerability to disclose a large amount of code pages, and generate gadgets thereof [172]. To tackle this attack, we present ISOMERON, a dynamic binary instrumentation tool that combines fine-grained code randomization with execution-path randomization [66].

6.2 COMPARING CONTROL-FLOW INTEGRITY AND CODE RANDOMIZATION

In this dissertation, we explored the most prominent defense principles against code-reuse attacks: control-flow integrity (CFI) [4] and software diversity [50] such as fine-grained code randomization. CFI is based on explicit checks, while software diversity relies on a secret, *i.e.* the randomization offset for base address randomization, making code-reuse attacks highly cumbersome. Both approaches have their advantages and disadvantages. In the remainder of this section, we discuss and compare both defense approaches. We focus our discussion on security aspects since performance highly varies among different instantiations and deployed instrumentation techniques.

CFI provides provable security [2]. That is, one can formally verify that CFI enforcement is sound. In particular, the explicit control-flow checks inserted by CFI into an application provide strong assurance that a program's control-flow cannot be arbitrarily hijacked by an adversary. In contrast, code randomization does not put any restriction on the program's control-flow. In fact, the attacker can provide any valid memory address as an indirect branch target.

Both schemes assume that code is not writable. Hence, an attacker cannot simply replace the original code with malicious code. In addition, classic fine-grained code randomization schemes (cf. Section 5.5.1) assume that the adversary does not possess full knowledge of the program's memory layout. Otherwise, an attacker could leverage this information to perform just-in-time code-reuse attacks [172]. Due to explicit control-flow checks full knowledge of the program's memory layout still does not allow an attacker to undermine CFI protection. Another related problem of protection schemes based on code randomization are side-channel attacks [103, 167]. These attacks exploit timing and fault analysis side channels to infer randomization information.

On the other hand, CFI relies on the precision of the application's control-flow graph (CFG). If valid branch addresses cannot be identified during static analysis, CFI needs

to make security compromises, and expands the set of valid addresses to preserve the program's functionality. In the worst-case, this taken compromise introduces malicious control-flow paths that an attacker can exploit. In particular, coarse-grained CFI solutions that intentionally allow many branch addresses introduce a large number of new control-flow paths that an attacker can exploit. For instance, many CFI schemes allow return instructions to target any call-preceded instruction [150, 204]. Such coarse-grained policies allow an attacker to construct return-oriented exploits from legitimate control-flow paths as we have shown in Section 3.2.

CFG imprecision allows an attacker to easily generate generic attack vectors that can be mounted on all computer systems that deploy the CFI-protected software program. This is not directly possible if programs are protected with fine-grained code randomization, because the code layout will differ for each platform.

In conclusion, CFI provides stronger security guarantees than code randomization schemes. However, CFI relies on an accurate and precise CFG. If the CFG is incomplete or includes too many valid control-flow paths, fine-grained code randomization can provide better security.

Recently, several defense mechanisms started to combine CFI with code randomization. For instance, Zhang et al. [202] allocate for each possible branch target a code stub on a Springboard section. All code stubs on the Springboard section are randomized to prevent an adversary from exploiting code stubs to construct return-oriented exploits. In particular, Mohan et al. [136] present the design and implementation of opaque CFI (O-CFI). This binary instrumentation-based solution leverages coarse-grained CFI checks and code randomization to prevent return-oriented exploits. For this, O-CFI identifies a unique set of possible target addresses for each indirect branch instruction. Afterwards, it uses the per-indirect branch set to restrict the target address of the indirect branch to only its minimal and maximal members. To further reduce the set of possible addresses, it arranges basic blocks belonging to an indirect branch set into clusters (so that they are located nearby to each other), and also randomizes their location. However, O-CFI relies on precise static analysis. In particular, it statically determines valid branch addresses for return instructions which typically leads to coarse-grained policies. Nevertheless, Mohan et al. [136] demonstrate that combining CFI with code randomization is a promising research direction.

6.3 FUTURE RESEARCH DIRECTIONS

Preventing return-into-libc Attacks. Most proposed control-flow integrity (CFI) and fine-grained code randomization defenses focus on the detection and prevention of return-oriented programming attacks, but do not provide full protection against return-into-libc attacks. This is only natural, given the fact that the majority of code-reuse attacks require a few return-oriented gadgets to initialize registers and prepare memory before calling a system call or critical function. However, Schuster et al. [163] have recently demonstrated that code-reuse attacks based on only calling a chain of virtual methods allow arbitrary malicious program actions. In addition, Tran et al. [183] have demonstrated that pure return-into-libc attacks can achieve Turing-completeness. De-

tecting such attacks is highly challenging: modern programs link to a large number of libraries, and require dangerous API and system calls to operate correctly [89]. Hence, for these programs, dangerous API and system calls are legitimate control-flow targets for indirect and direct call instructions; even if fine-grained CFI policies are enforced. In order to detect code-reuse attacks that exploit these functions, CFI needs to be combined with additional security checks, *e.g.* with dynamic taint analysis or techniques to perform argument validation. We believe that developing such CFI extensions is an important future research direction.

Memory Safety. In this dissertation, we have focused on the second stage of runtime exploits, *i.e.* the malicious computation after the control-flow of a program has been hijacked. Until recently, defenses that aim at preventing the first stage of a runtime exploit, *i.e.* the initial code pointer overwrite, were either incomplete or incurred too high performance overhead. Recently, Szekeres et al. [180] presented the idea of code pointer integrity (CPI), and its implementation on x86 and x86-64 [119]. CPI separates code pointers as well as pointers to code pointers in a safe memory region that can only be accessed by instructions that are proven to be safe at compile-time. CPI operates very efficient on C code, but may incur performance overhead of more than 40% for C++-compiled code. With respect to security, CPI relies on the protection of the safe memory region which is efficiently possible on x86 leveraging segmentation. However, on x86-64 where segmentation is not fully available, CPI relies on hiding the safe memory region. Unfortunately, Evans et al. [76] have recently demonstrated that side-channel attacks can be leveraged to locate and alter the safe memory region on x86-64. Given the new approaches and attacks in this area, we believe that defenses that aim at memory safety for type-unsafe languages are a promising future research direction.

7

ABOUT THE AUTHOR

Lucas Davi is a research assistant at the Technische Universität Darmstadt and the Intel Collaborative Research Institute for Secure Computing (Intel CRI-SC), Germany. In 2010, he received his M.Sc. in IT-Security from Ruhr-University Bochum, Germany. His research has mainly focused on modern memory corruption attacks such as return-oriented programming for different processor architectures. To tackle these attacks, he developed countermeasures that are based on control-flow integrity and fine-grained code randomization.

AWARDS

- Best Paper at ASIACCS 2015
- Distinguished Paper at ASIACCS 2013
- Best Student Paper at IEEE Security & Privacy 2013
- Best PhD Presentation at ACM S3 2012
- 2nd Place German IT Security Competition Award 2010
- 1st Place CAST IT-Security Award 2010 (Best Master Thesis)

ACADEMIC ACTIVITIES

- Program Committee, USENIX Workshop on Offensive Technologies, WOOT 2015
- Program Committee, International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2015
- Program Committee, ACM CCS Workshop on Trustworthy Embedded Devices, TrustED 2015
- Publications Chair, ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2013
- Organization Committee, ACM CCS 2013
- Program Committee, International Conference on Availability, Reliability and Security, ARES 2012-2014

PEER-REVIEWED PUBLICATIONS

Orlando Arias, Lucas Davi, Matthias Hanreich, Yier Jin, Patrick Koeberl, Debayan Paul, Ahmad-Reza Sadeghi, Dean Sullivan. HAFIX: Hardware-Assisted Flow Integrity Extension. In *Proceedings of the 52nd Design Automation Conference*, DAC'15, 2015.

Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (SP'15), 2015.

Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten 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* (SP'15), 2015.

Mihai Bucicoiu, Lucas Davi, Razvan Deaconescu, Ahmad-Reza Sadeghi. XiOS: Extended Application Sandboxing on iOS (Best Paper Award). In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (ASIACCS'15), 2015.

Ahmad-Reza Sadeghi, Lucas Davi, Per Larsen. Securing Legacy Software against Real-World Code-Reuse Exploits: Utopia, Alchemy, or Possible Future? –Keynote– In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (ASIACCS'15), 2015.

Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium* (NDSS'15), 2015.

Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi. The Beast is in Your Memory: Return-Oriented Programming Attacks Against Modern Control-Flow Integrity Protection Techniques. In *BlackHat USA*, 2014.

Lucas Davi, Patrick Koeberl, Ahmad-Reza Sadeghi. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Proceedings of the 51st Design Automation Conference - Special Session: Trusted Mobile Embedded Computing* (DAC'14), 2014.

Blaine Stancill, Kevin Z. Snow, Nathan Otterness, Fabian Monrose, Lucas Davi, Ahmad-Reza Sadeghi. Check My Profile: Leveraging Static Analysis for Fast and Accurate Detection of ROP Gadgets. In *Proceedings of the 16th Research in Attacks, Intrusions and Defenses Symposium* (RAID'13), 2013.

Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: the More Things Change, the More They Stay the Same. In *BlackHat USA*, 2013.

Tim Werthmann, Ralf Hund, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz. PSiOS: Bring Your Own Privacy & Security to iOS Devices (Distinguished Paper Award). In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security* (ASIACCS'13), 2013.

Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization (Best Student Paper Award). In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (SP'13), 2013.

Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, Ahmad-Reza Sadeghi. Gadge Me If You Can - Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security* (ASIACCS'13), 2013.

Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Ahmad-Reza Sadeghi. Over-the-air Cross-Platform Infection for Breaking mTAN-based Online Banking Authentication. In *BlackHat Abu Dhabi*, 2012.

Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, Ahmad-Reza Sadeghi. XIFER: A Software Diversity Tool Against Code-Reuse Attacks. In *Proceedings of the 4th ACM International Workshop on Wireless of the Students, by the Students, for the Students* (S3'12), 2012.

Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, Ahmad-Reza Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (NDSS'12), 2012.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (NDSS'12), 2012.

Sven Bugiel, Lucas Davi, Steffen Schulz. Scalable Trust Establishment with Software Reputation. In *Proceedings of the 6th ACM Workshop on Scalable Trusted Computing* (STC'11), 2011.

Lucas Davi, Alexandra Dmitrienko, Christoph Kowalski, Marcel Winandy. Trusted Virtual Domains on OKL4: Secure Information Sharing on Smartphones. In *Proceedings of the 6th ACM Workshop on Scalable Trusted Computing* (STC'11), 2011.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, Bhargava Shastry. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the 1st ACM CCS Workshop on Security and Privacy in Mobile Devices* (SPSM'11), 2011.

Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, Ahmad-Reza Sadeghi. CFI Goes Mobile: Control-Flow Integrity for Smartphones. In *Proceedings of the 1st International Workshop on Trustworthy Embedded Devices* (TrustED'11), 2011.

Lucas Davi, Ahmad-Reza Sadeghi, Marcel Winandy. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (ASIACCS'11), 2011.

Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, Marcel Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security* (CCS'10), 2010.

Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Marcel Winandy. Privilege Escalation Attacks on Android. In: *Proceedings of the 13th Information Security Conference* (ISC'10), 2010.

Lucas Davi, Ahmad-Reza Sadeghi, Marcel Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks. In *Proceedings of the 4th ACM Workshop on Scalable Trusted Computing*, (STC'09), 2009.

BOOKS

N. Asokan, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Kari Kostiainen, Elena Reshetova, Ahmad-Reza Sadeghi. Mobile Platform Security. In *Synthesis Lectures on Information Security, Privacy, and Trust*, vol. 4, no. 3, Morgan & Claypool, 2013.

TECHNICAL REPORTS

Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. *Technical Report TUD-CS-2014-0097*, 2014.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. *Technical Report TR-2011-04*, 2011.

Lucas Davi, Alexandra Dmitrienko Ahmad-Reza Sadeghi, Marcel Winandy. Return-Oriented Programming without Returns on ARM. *Technical Report no. HGI-TR-2010-002*, 2010.

Lucas Davi, Ahmad-Reza Sadeghi, Marcel Winandy. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. *Technical Report no. HGI-TR-2010-001*, 2010.

POSTERS

Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, Ahmad-Reza Sadeghi. POSTER: Control-Flow Integrity for Smartphones. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (CCS'11), 2011.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, Bhargava Shastry. POSTER: The Quest for Security against Privilege Escalation Attacks

on Android In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (CCS'11), 2011.

BIBLIOGRAPHY

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS'05, 2005. URL http://doi.acm.org/10.1145/1102120.1102165.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control-flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering*, ICFEM'05, 2005. URL http://dx.doi.org/10.1007/11576280_9.

[3] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, George C. Necula, and Michael Vrable. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, 2006. URL http://dl.acm.org/citation.cfm?id=1298455.1298463.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009. URL http://doi.acm.org/10.1145/1609956.1609960.

[5] Jonathan Afek and Adi Sharabani. Dangling pointer: Smashing the pointer for fun and profit, 2007. URL https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf.

[6] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, SP'08, 2008. URL http://dx.doi.org/10.1109/SP.2008.30.

[7] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 2000. URL http://phrack.org/issues/49/14.html.

[8] Apple Inc. App review. URL https://developer.apple.com/appstore/guidelines.html.

[9] Apple Inc. Manual page of dyld-the dynamic link editor, 2011. URL http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/dyld.1.html.

[10] Apple Inc. Manual page of mmap, 2011. URL http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/mmap.2.html.

[11] Apple Inc. Entitlement key reference, 2011. URL http://developer.apple.com/library/mac/#documentation/Miscellaneous/Reference/

EntitlementKeyReference/Chapters/EnablingAppSandbox.html#//apple_ref/doc/uid/TP40011195-CH4-SW1.

[12] Apple Inc. iOS SDK release notes for iOS 6, 2013. URL http://developer.apple.com/library/ios/#releasenotes/General/RN-iOSSDK-6_0/index.html.

[13] Orlando Arias, Lucas Davi, Matthias Hanreich, Yier Jin, Patrick Koeberl, Debayan Paul, Ahmad-Reza Sadeghi, and Dean Sullivan. HAFIX: Hardware-assisted flow integrity extension. In *In Proceedings of the 52nd Design Automation Conference*, DAC'15, 2015. To appear.

[14] ARM Ltd. Procedure call standard for the ARM architecture, 2009. URL http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf.

[15] ARM Ltd. ARM launches Cortex-A50 series, the world's most energy-efficient 64-bit processors, 2012. URL http://www.arm.com/about/newsroom/arm-launches-cortex-a50-series-the-worlds-most-energy-efficient-64-bit-processors.php.

[16] Itzhak Avraham. Exploitation on ARM - technique and bypassing defense mechanisms, 2010. URL https://www.defcon.org/images/defcon-18/dc-18-presentations/Avraham/DEFCON-18-Avraham-Modern%20ARM-Exploitation-WP.pdf.

[17] Elias Bachaalany. Inside EMET 4.0. REcon Montreal, 2013. URL http://recon.cx/2013/slides/Recon2013-Elias%20Bachaalany-Inside%20EMET%204.pdf.

[18] Michael Backes and Stefan Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the 23rd USENIX Security Symposium*, 2014. URL http://dl.acm.org/citation.cfm?id=2671225.2671253.

[19] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik 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. URL http://doi.acm.org/10.1145/2660267.2660378.

[20] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS'03, 2003. URL http://doi.acm.org/10.1145/948109.948147.

[21] S. Bhatkar, D.C. 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*, 2003. URL http://dl.acm.org/citation.cfm?id=1251353.1251361.

[22] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, 2005. URL http://dl.acm.org/citation.cfm?id=1251398.1251415.

[23] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, SP'14, 2014. URL http://dx.doi.org/10.1109/SP.2014.22.

[24] Dionysus Blazakis. Interpreter exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, 2010. URL http://dl.acm.org/citation.cfm?id=1925004.1925011.

[25] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC'11, 2011. URL http://doi.acm.org/10.1145/2076732.2076783.

[26] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'11, 2011. URL http://doi.acm.org/10.1145/1966913.1966919.

[27] blexim. Basic integer overflows. *Phrack Magazine*, 60(10), 2002. URL http://www.phrack.org/issues.html?issue=60&id=10#article.

[28] Larry Boettger. The Morris worm: How it affected computer security and lessons learned by it, 2000. URL http://www.giac.org/paper/gsec/405/morris-worm-affected-computer-security-lessons-learned/100954.

[29] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004. URL http://groups.csail.mit.edu/cag/rio/derek-phd-thesis.pdf.

[30] Glenn Brunette. Solaris non-executable stack overview. https://blogs.oracle.com/gbrunett/entry/solaris_non_executable_stack_overview, 2007.

[31] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS'08, 2008. URL http://doi.acm.org/10.1145/1455770.1455776.

[32] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural support for software-based protection. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID'06, pages 42–51, 2006. URL http://doi.acm.org/10.1145/1181309.1181316.

[33] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS'12, 2012. URL http://www.internetsociety.org/towards-taming-privilege-escalation-attacks-android.

[34] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android. Technical Report TUD-CS-2012-0226, Technische Universität Darmstadt, 2012. URL https://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/mytunes_tr.pdf.

[35] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, 56(5), 1996. URL http://www.phrack.org/issues.html?issue=49&id=14.

[36] C4SS!o and h1ch4m. MPlayer Lite r33064 m3u Buffer Overflow Exploit (DEP Bypass), 2011. URL http://www.exploit-db.com/exploits/17565/.

[37] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, 2014. URL http://dl.acm.org/citation.cfm?id=2671225.2671250.

[38] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, 2006. URL http://dl.acm.org/citation.cfm?id=1298455.1298470.

[39] Stephen Checkoway and Hovav Shacham. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical Report CS2010-0954, UC San Diego, 2010. URL http://cseweb.ucsd.edu/~hovav/dist/noret.pdf.

[40] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav 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. URL http://dl.acm.org/citation.cfm?id=1855491.1855497.

[41] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS'10, 2010. URL http://doi.acm.org/10.1145/1866307.1866370.

[42] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. Springer Berlin Heidelberg, 2009. URL http://dx.doi.org/10.1007/978-3-642-10772-6_13.

[43] Ping Chen, Xiao Xing, Bing Mao, and Li Xie. Return-oriented rootkit without returns (on the x86). In *Information and Communications Security*, volume 6476 of *Lecture Notes in Computer Science*. 2010. URL http://link.springer.com/chapter/10.1007%2F978-3-642-17650-0_24.

[44] Ping Chen, Xiao Xing, Bing Mao, Li Xie, Xiaobin Shen, and Xinchun Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'11, 2011. URL http://doi.acm.org/10.1145/1966913.1966918.

[45] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005. URL http://dl.acm.org/citation.cfm?id=1251398.1251410.

[46] Xiaobo Chen, Dan Caselden, and Mike Scott. The dual use exploit: CVE-2013-3906 used in both targeted attacks and crimeware campaigns, 2013. URL https://www.fireeye.com/blog/threat-research/2013/11/the-dual-use-exploit-cve-2013-3906-used-in-both-targeted-attacks-and-crimeware-campaigns.html.

[47] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert Huijie Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS'14, 2014. URL http://www.internetsociety.org/doc/ropecker-generic-and-practical-approach-defending-against-rop-attacks.

[48] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, ICDCS'01, 2001. URL http://dl.acm.org/citation.cfm?id=876878.879316.

[49] Tzi-cker Chiueh and Manish Prasad. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the 2003 USENIX Annual Technical Conference*, ATC'03, 2003. URL https://www.usenix.org/legacy/event/usenix03/tech/full_papers/prasad/prasad_html/camera.html.

[50] Frederick B. Cohen. Operating system protection through program evolution. *Computer & Security*, 12(6), 1993. doi: 10.1016/0167-4048(93)90054-9.

[51] Matt Conover. woowoo on heap overflows, 1999. URL http://www.cgsecurity.org/exploit/heaptut.txt.

[52] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998. URL http://dl.acm.org/citation.cfm?id=1267549.1267554.

[53] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, 2003. URL http://dl.acm.org/citation.cfm?id=1251353.1251360.

[54] cplusplus.com. Polymorphism. URL http://www.cplusplus.com/doc/tutorial/polymorphism/.

[55] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, SP'15, 2015. To appear.

[56] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP'14, 2014. URL http://dx.doi.org/10.1109/SP.2014.26.

[57] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'15, 2015. URL http://doi.acm.org/10.1145/2714576.2714635.

[58] Roman Danyliw and Allen Householder. "Code Red" worm exploiting buffer overflow in IIS Indexing Service DLL, 2001. URL http://www.cert.org/historical/advisories/ca-2001-19.cfm?

[59] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, STC'09, 2009. URL http://doi.acm.org/10.1145/1655108.1655117.

[60] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, 2010. URL http://dl.acm.org/citation.cfm?id=1949317.1949356.

[61] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'11, 2011. URL http://doi.acm.org/10.1145/1966913.1966920.

[62] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS'12, 2012. URL http://www.internetsociety.org/mocfi-framework-mitigate-control-flow-attacks-smartphones.

[63] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can - secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'13, 2013. URL http://doi.acm.org/10.1145/2484313.2484351.

[64] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014. URL http://dl.acm.org/citation.cfm?id=2671225.2671251.

[65] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. Technical Report TUD-CS-2014-0097, Technische Universität Darmstadt, 2014. URL https://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/techreport-stitching-gadgets.pdf.

[66] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS'15, 2015. URL http://www.internetsociety.org/doc/isomeron-code-randomization-resilient-just-time-return-oriented-programming.

[67] Robert I. Davidson and Nathan Myhrvold. Method and system for generating and auditing a signature for a computer program, 1996. URL http://www.google.com/patents/US5559884. Patent US5559884.

[68] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. QUIRE: Lightweight provenance for smartphone operating systems. In *Proceedings of the 20th USENIX Security Symposium*, 2011. URL http://dl.acm.org/citation.cfm?id=2028067.2028090.

[69] Jose Duart. Objective-C helper script, 2010. URL https://github.com/zynamics/objc-helper-plugin-ida.

[70] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, 2010. URL http://dl.acm.org/citation.cfm?id=1925004.1925012.

[71] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001. URL http://research.microsoft.com/pubs/69850/tr-2001-50.pdf.

[72] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in ios applications. In *Proceedings of the 18th Annual Network and Distributed System*

Security Symposium, NDSS'11, 2011. URL http://www.internetsociety.org/doc/pios-detecting-privacy-leaks-ios-applications-papers.

[73] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS'09, 2009. URL http://doi.acm.org/10.1145/1653662.1653691.

[74] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010. URL http://dl.acm.org/citation.cfm?id=1924943.1924971.

[75] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004. URL http://www.ru.is/faculty/ulfar/thesis.pdf.

[76] Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, SP'15, 2015. To appear.

[77] Adrienne Porter Felt, Helen Wang, Alex Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, 2011. URL http://dl.acm.org/citation.cfm?id=2028067.2028089.

[78] Stephanie Forrest, Anil Somayaji, and David Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS'97, 1997. URL http://dl.acm.org/citation.cfm?id=822075.822408.

[79] Aurélien Francillon and Claude Castelluccia. Code injection attacks on Harvard-architecture devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS'08, 2008. URL http://doi.acm.org/10.1145/1455770.1455775.

[80] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, SecuCode'09, 2009. URL http://doi.acm.org/10.1145/1655077.1655083.

[81] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, 2001. URL http://dl.acm.org/citation.cfm?id=1251327.1251332.

[82] Michael Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW'10, 2010. URL http://doi.acm.org/10.1145/1900546.1900550.

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

[84] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC'09, 2009. URL http://dx.doi.org/10.1109/ACSAC.2009.16.

[85] Robert Gawlik and Thorsten 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. URL http://doi.acm.org/10.1145/2664243.2664249.

[86] gera. Advances in format string exploitation. *Phrack Magazine*, 59(12), 2002. URL http://www.phrack.com/issues.html?issue=59&id=7.

[87] Jason Gionta, William Enck, and Peng 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. URL http://doi.acm.org/10.1145/2699026.2699107.

[88] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Security Symposium*, 2012. URL http://dl.acm.org/citation.cfm?id=2362793.2362833.

[89] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, SP'14, 2014. URL http://dx.doi.org/10.1109/SP.2014.43.

[90] Enes Göktas, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium*, 2014. URL http://dl.acm.org/citation.cfm?id=2671225.2671252.

[91] Dan Goodin. Apple quicktime backdoor creates code-execution peril, 2010. URL http://www.theregister.co.uk/2010/08/30/apple_quicktime_critical_vuln/.

[92] Google. Security and permissions, 2012. URL http://developer.android.com/guide/topics/security/security.html.

[93] John Greenough. The Internet of Everything. URL http://uk.businessinsider.com/internet-of-everything-2015-bi-2014-12?op=1?r=US.

[94] Aditi Gupta, Sam Kerr, MichaelS. Kirkpatrick, and Elisa Bertino. Marlin: A fine grained randomization approach to defend against ROP attacks. In *Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*. 2013. URL http://dx.doi.org/10.1007/978-3-642-38631-2_22.

[95] Suhas Gupta, Pranay Pratap, Huzur Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA'06, pages 65–72, 2006. URL http://doi.acm.org/10.1145/1138912.1138926.

[96] Jin Han, Su Mon Kywe, Qiang Yan, Feng Bao, Robert Deng, Debin Gao, Yingjiu Li, and Jianying Zhou. Launching generic attacks on iOS with approved third-party applications. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, 2013. URL http://dx.doi.org/10.1007/978-3-642-38980-1_17.

[97] Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, SP'12, 2012. URL http://dx.doi.org/10.1109/SP.2012.39.

[98] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*, WOOT'12, 2012. URL http://dl.acm.org/citation.cfm?id=2372399.2372409.

[99] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Librando: Transparent code randomization for just-in-time compilers. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS'13, 2013. URL http://doi.acm.org/10.1145/2508859.2516675.

[100] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'13, 2013. URL http://dx.doi.org/10.1109/CGO.2013.6494997.

[101] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the Droids you're looking for: Retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS'11, 2011. URL http://doi.acm.org/10.1145/2046707.2046780.

[102] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the*

18th Conference on USENIX Security Symposium, 2009. URL http://dl.acm.org/citation.cfm?id=1855768.1855792.

[103] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, SP'13, 2013. URL http://dx.doi.org/10.1109/SP.2013.23.

[104] Vincenzo Iozzo and Charlie Miller. Fun and games with Mac OS X and iPhone payloads. In *Black Hat Europe*, 2009. URL http://www.blackhat.com/presentations/bh-europe-09/Miller_Iozzo/BlackHat-Europe-2009-Miller-Iozzo-OSX-IPhone-Payloads-whitepaper.pdf.

[105] Vincenzo Iozzo and Ralf-Philipp Weinmann. Pwn2Own contest. http://blog.zynamics.com/2010/03/24/ralf-philipp-weinmann-vincenzo-iozzo-own-the-iphone-at-pwn2own/, 2010.

[106] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. Compiler-generated software diversity. In *Moving Target Defense*, volume 54 of *Advances in Information Security*. 2011. URL http://dx.doi.org/10.1007/978-1-4614-0977-9_4.

[107] Todd Jackson, Andrei Homescu, Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Diversifying the software stack using randomized nop insertion. In *Moving Target Defense II*, volume 100 of *Advances in Information Security*. 2013. URL http://dx.doi.org/10.1007/978-1-4614-5416-8_8.

[108] Shahriyar Jalayeri. Bypassing EMET 3.5's ROP mitigations, 2012. URL https://repret.wordpress.com/2012/08/08/bypassing-emet-3-5s-rop-mitigations/.

[109] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS'14, 2014. URL http://www.internetsociety.org/doc/safedispatch-securing-c-virtual-calls-memory-corruption-attacks.

[110] jduck. The latest Adobe exploit and session upgrading, 2010. URL http://bugix-security.blogspot.de/2010/03/adobe-pdf-libtiff-working-exploitcve.html.

[111] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA'12, 2012. URL http://dl.acm.org/citation.cfm?id=2337159.2337171.

[112] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS'03, 2003. URL http://doi.acm.org/10.1145/948109.948146.

[113] MJ Keith. Android 2.0-2.1 reverse shell exploit, 2010. URL http://www.exploit-db.com/exploits/15423/.

[114] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC'06, 2006. URL http://dx.doi.org/10.1109/ACSAC.2006.9.

[115] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, 2002. URL http://dl.acm.org/citation.cfm?id=647253.720293.

[116] Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In *Proceedings of the 41st Annual Design Automation Conference*, DAC'04, 2004. URL http://doi.acm.org/10.1145/996566.996771.

[117] Tim Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University Bochum, 2009. URL http://static.googleusercontent.com/media/www.zynamics.com/en//downloads/kornau-tim--diplomarbeit--rop.pdf.

[118] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005. URL http://users.suse.com/~krahmer/no-nx.pdf.

[119] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 2014. URL http://dl.acm.org/citation.cfm?id=2685048.2685061.

[120] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, SP'14, 2014. URL http://dx.doi.org/10.1109/SP.2014.25.

[121] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys'10, 2010. URL http://doi.acm.org/10.1145/1755913.1755934.

[122] Wilson Lian, Hovav Shacham, and Stefan Savage. Too lejit to quit: Extending JIT spraying to ARM. In *22nd Annual Network and Distributed System Security Symposium*, NDSS'15, 2015. URL http://www.internetsociety.org/doc/too-lejit-quit-extending-jit-spraying-arm.

[123] Felix Lindner. Router exploitation, 2009. URL http://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-SLIDES.pdf.

[124] Limin Liu, Jin Han, Debin Gao, Jiwu Jing, and Daren Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *Proceedings of the 10th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM'11, 2011. URL http://dx.doi.org/10.1109/TrustCom.2011.9.

[125] Kangjie Lu, Dabi Zou, Weiping Wen, and Debin Gao. Packed, printable, and polymorphic return-oriented programming. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, 2011. URL http://dx.doi.org/10.1007/978-3-642-23644-0_6.

[126] Kangjie Lu, Siyang Xiong, and Debin Gao. Ropsteg: Program steganography with return oriented programming. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY'14, 2014. URL http://doi.acm.org/10.1145/2557547.2557572.

[127] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay J. Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 26th ACM SIGPLAN conference on Programming language design and implementation*, PLDI'05, 2005. URL http://doi.acm.org/10.1145/1065010.1065034.

[128] Marion Marschalek. Dig deeper into the ie vulnerability (cve-2014-1776) exploit, 2014. URL https://www.cyphort.com/dig-deeper-ie-vulnerability-cve-2014-1776-exploit/.

[129] Aleksandr Matrosov, Eugene Rodionov, David Harley, and Juraj Malcho. Stuxnet under the microscope, 2001. URL http://www.esetnod32.ru/company/viruslab/analytics/doc/Stuxnet_Under_the_Microscope.pdf.

[130] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, 2006. URL http://dl.acm.org/citation.cfm?id=1267336.1267351.

[131] Microsoft. Enhanced Mitigation Experience Toolkit. URL https://www.microsoft.com/emet.

[132] Microsoft. Data execution prevention (DEP), 2006. URL http://support.microsoft.com/kb/875352/EN-US/.

[133] Microsoft Corporation. Visual Studio 2015 preview: Work-in-progress security feature, 2014. URL http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx.

[134] Charles Miller and Dino Dai Zovi. *The Mac Hacker's Handbook*. Wiley Publishing, 2009. ISBN 0470395362, 9780470395363.

[135] Charlie Miller and Dion Blazakis. Pwn2Own contest, 2011. URL http://www.ditii.com/2011/03/10/pwn2own-iphone-4-running-ios-4-2-1-successfully-hacked/.

[136] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque control-flow integrity. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS'15, 2015. URL http://www.internetsociety.org/doc/opaque-control-flow-integrity.

[137] Collin Mulliner and Charlie Miller. Injecting SMS messages into smart phones for security analysis. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, 2009. URL http://dl.acm.org/citation.cfm?id=1855876.1855881.

[138] N. Seriot. SpyPhone, 2011. URL https://github.com/nst/SpyPhone.

[139] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'09, 2009. URL http://doi.acm.org/10.1145/1542476.1542504.

[140] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM'10, 2010. URL http://doi.acm.org/10.1145/1806651.1806657.

[141] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'10, 2010. URL http://doi.acm.org/10.1145/1755688.1755732.

[142] NC State University. What is the Slammer worm/SQL worm/Sapphire worm?, 2001. URL https://ethics.csc.ncsu.edu/abuse/wvt/Slammer/study.php.

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

[144] Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS'13, 2013. URL http://doi.acm.org/10.1145/2508859.2516649.

[145] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, 2014. URL http://doi.acm.org/10.1145/2594291.2594295.

[146] Ben Niu and Gang 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. URL http://doi.acm.org/10.1145/2660267.2660281.

[147] Gene Novark and Emery D. Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS'10, 2010. URL http://doi.acm.org/10.1145/1866307.1866371.

[148] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC'10, 2010. URL http://doi.acm.org/10.1145/1920261.1920269.

[149] Vasilis Pappas, Michalis Polychronakis, and Angelos 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*, SP'12, 2012. URL http://dx.doi.org/10.1109/SP.2012.41.

[150] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, 2013. URL http://dl.acm.org/citation.cfm?id=2534766.2534805.

[151] PaX Team. PaX address space layout randomization (ASLR). URL http://pax.grsecurity.net/docs/aslr.txt.

[152] Jannik Pewny and Thorsten Holz. Compiler-based CFI for iOS. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC'13, 2013. URL http://doi.acm.org/10.1145/2523649.2523674.

[153] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy Magazine*, 2(4):20–27, 2004. URL http://dx.doi.org/10.1109/MSP.2004.36.

[154] Aravind Prakash, Heng Yin, and Zhenkai Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'13, 2013. URL http://doi.acm.org/10.1145/2484313.2484352.

[155] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS'15, 2015. URL http://www.internetsociety.org/doc/vfguard-strict-protection-virtual-function-calls-cots-c-binaries.

[156] J. Rattner. *Extreme scale computing*. ISCA Keynote, 2012.

[157] Gaisler Research. LEON3 synthesizable processor. URL http://www.gaisler.com.

[158] rix. Smashing C++ VPTRS. *Phrack Magazine*, 56(8), 2000. URL http://phrack.org/issues/56/8.html.

[159] Tracy Robinson. Celebrating 50 billion shipped ARM-powered chips, 2014. URL http://community.arm.com/community/news/blog/2014/02/12/celebrating-50-billion-shipped-arm-powered-chips.

[160] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, 2012. URL http://doi.acm.org/10.1145/2133375.2133377.

[161] saurik. MobileSubstrate. URL http://iphonedevwiki.net/index.php/MobileSubstrate.

[162] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. In *Research in Attacks, Intrusions and Defenses*, volume 8688 of *Lecture Notes in Computer Science*. 2014. URL http://dx.doi.org/10.1007/978-3-319-11379-1_5.

[163] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten 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*, SP'15, 2015. To appear.

[164] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011. URL http://dl.acm.org/citation.cfm?id=2028067.2028092.

[165] Mathew J. Schwartz. iOS Social Apps Leak Contact Data, 2012. URL http://www.informationweek.com/news/security/privacy/232600490.

[166] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the 19th USENIX Security Symposium*, 2010. URL http://dl.acm.org/citation.cfm?id=1929820.1929822.

[167] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security*, CCS'14, 2014. URL http://doi.acm.org/10.1145/2660267.2660309.

[168] Fermín J. Serna. CVE-2012-0769, the case of the perfect info leak, 2012. URL http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[169] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS'07, 2007. URL http://doi.acm.org/10.1145/1315245.1315313.

[170] Hovav Shacham, Eu jin Goh, Nagendra Modadugu, Ben Pfaff, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS'04, 2004. URL http://doi.acm.org/10.1145/1030083.1030124.

[171] Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702s.

[172] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, SP'13, 2013. URL http://dx.doi.org/10.1109/SP.2013.45. Received the Best Student Paper Award.

[173] Solar Designer. Non-executable stack patch, 1997. URL http://lkml.iu.edu/hypermail/linux/kernel/9706.0/0341.html.

[174] Solar Designer. lpr LIBC RETURN exploit, 1997. URL http://insecure.org/sploits/linux.libc.return.lpr.sploit.html.

[175] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in Windows Vista, 2008. URL http://www.phreedom.org/research/bypassing-browser-memory-protections/. Presented at Black Hat 2008.

[176] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*, 2005. URL http://dl.acm.org/citation.cfm?id=1251398.1251408.

[177] Eugene H. Spafford. The internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1), 1989. URL http://doi.acm.org/10.1145/66093.66095.

[178] Blaine Stancill, KevinZ. Snow, Nathan Otterness, Fabian Monrose, Lucas Davi, and Ahmad-Reza Sadeghi. Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In *Research in Attacks, Intrusions, and Defenses*, volume 8145 of *Lecture Notes in Computer Science*. 2013. URL http://dx.doi.org/10.1007/978-3-642-41284-4_4.

[179] Statista Inc. Number of apps available in leading app stores as of july 2014, 2015. URL http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[180] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, SP'13, 2013. URL http://dx.doi.org/10.1109/SP.2013.13.

[181] Matt Thomlinson. Announcing the BlueHat prize winners, 2012. URL https://blogs.technet.com/b/msrc/archive/2012/07/26/announcing-the-bluehat-prize-winners.aspx?Redirected=true.

[182] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, 2014. URL http://dl.acm.org/citation.cfm?id=2671225.2671285.

[183] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, 2011. URL http://dx.doi.org/10.1007/978-3-642-23644-0_7.

[184] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, 2012. URL http://dx.doi.org/10.1007/978-3-642-33338-5_5.

[185] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. Persistent data-only malware: Function hooks without code. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS'14, 2014. URL http://www.internetsociety.org/doc/persistent-data-only-malware-function-hooks-without-code.

[186] Peter Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit, 2010. URL http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf.

[187] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, 1993. URL http://doi.acm.org/10.1145/173668.168635.

[188] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on iOS: When benign apps become evil. In *Proceedings of the 22nd USENIX Security Symposium*, 2013. URL http://dl.acm.org/citation.cfm?id=2534766.2534814.

[189] Zhi Wang and Xuxian Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, SP'10, 2010. URL http://dx.doi.org/10.1109/SP.2010.30.

[190] Zhi Wang, Renquan Cheng, and Debin Gao. Revisiting address space randomization. In *Proceedings of the 13th International Conference on Information Security and Cryptology*, ICISC'10, 2010. URL http://dl.acm.org/citation.cfm?id=2041036.2041054.

[191] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In

Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS'12, 2012. URL http://doi.acm.org/10.1145/2382196.2382216.

[192] Ralf-Philipp Weinmann. All your baseband are belong to us, 2010. URL http://2010.hack.lu/archive/2010/Weinmann-All-Your-Baseband-Are-Belong-To-Us-slides.pdf.

[193] Yoav Weiss and Elena Gabriela Barrantes. Known/chosen key attacks against software instruction set randomization. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC'06, 2006. URL http://dx.doi.org/10.1109/ACSAC.2006.33.

[194] Tim Werthmann, Ralf Hund, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. PSiOS: Bring your own privacy & security to iOS devices. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'13, 2013. URL http://doi.acm.org/10.1145/2484313.2484316. Received the Distinguished Paper Award.

[195] Ken Westin. GnuTLS crypto library vulnerability CVE-2014-3466, 2014. URL http://www.tripwire.com/state-of-security/latest-security-news/gnutls-crypto-library-vulnerability-cve-2014-3466/.

[196] Simon Winwood and Manuel Chakravarty. Secure untrusted binaries – provably! In *Proceedings of the 3rd International Conference on Formal Aspects in Security and Trust*, FAST'05, 2006. URL http://dx.doi.org/10.1007/11679219_13.

[197] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN'12, 2012. URL http://dl.acm.org/citation.cfm?id=2354410.2355130.

[198] Rubin Xu, Hassen Saidi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security Symposium*, 2012.

[199] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, SP'09, 2009. URL http://dx.doi.org/10.1109/SP.2009.25.

[200] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS'11, 2011. URL http://doi.acm.org/10.1145/2046707.2046713.

[201] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *Proceedings of the 22nd USENIX Security Symposium*, 2013. URL http://dl.acm.org/citation.cfm?id=2534766.2534798.

[202] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, SP'13, 2013. URL http://dx.doi.org/10.1109/SP.2013.44.

[203] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Defending virtual function tables' integrity. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS'15, 2015. URL http://www.internetsociety.org/doc/vtint-protecting-virtual-function-tables%E2%80%99-integrity.

[204] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, 2013. URL http://dl.acm.org/citation.cfm?id=2534766.2534796.

[205] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Anomalous path detection with hardware support. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES'05, 2005. URL http://doi.acm.org/10.1145/1086297.1086305.

[206] Dino Dai Zovi. Practical return-oriented programming. SOURCE Boston, 2010. URL http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf.

[207] Dino Dai Zovi. Apple iOS security evaluation: Vulnerability analysis and data encryption. In *Black Hat USA*, 2011. URL https://media.blackhat.com/bh-us-11/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf.

Erklärung gemäß §9 der Promotionsordnung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwednung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Germany, April 2015

Lucas Vincenzo Davi