Universidade Estadual de Campinas
Instituto de Computação

João Batista Corrêa Gomes Moreira

Protection Mechanisms Against Control-Flow Hijacking Attacks

Mecanismos de Proteção Contra Ataques de Sequestro de Controle de Fluxo

CAMPINAS
2016

João Batista Corrêa Gomes Moreira

Protection Mechanisms Against Control-Flow Hijacking Attacks

Mecanismos de Proteção Contra Ataques de Sequestro de Controle de Fluxo

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Sandro Rigo
Co-supervisor/Coorientador: Prof. Dr. Vasileios Kemerlis

Este exemplar corresponde à versão final da Tese defendida por João Batista Corrêa Gomes Moreira e orientada pelo Prof. Dr. Sandro Rigo.

CAMPINAS
2016

Agência(s) de fomento e nº(s) de processo(s): CNPq, 140914/2012-9; CNPq, 163446/2013-0; CAPES

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

M813p

Moreira, João Batista Corrêa Gomes, 1985-
Protection mechanisms against kernel control-flow hijacking attacks / João Batista Corrêa Gomes Moreira. – Campinas, SP : [s.n.], 2016.

Orientador: Sandro Rigo.
Coorientador: Vasileios Kemerlis.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas de segurança. 2. Compiladores (Programas de computador). 3. Sistemas operacionais (Computadores). I. Rigo, Sandro,1975-. II. Kemerlis, Vasileios. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Mecanismos de proteção contra ataques de sequestro de controle de fluxo
Palavras-chave em inglês:
Security systems
Operating systems (Computers)
Compilers (Computer programs)
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Sandro Rigo [Orientador]
Michalis Polychronakis
Fernando Magno Quintao Pereira
Paulo Lício de Geus
Diego de Freitas Aranha
Data de defesa: 16-12-2016
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

João Batista Corrêa Gomes Moreira

Protection Mechanisms Against Control-Flow Hijacking Attacks

Mecanismos de Proteção Contra Ataques de Sequestro de Controle de Fluxo

Banca Examinadora:

- Prof. Dr. Sandro Rigo
  IC - UNICAMP
- Prof. Dr. Michalis Polychronakis
  CS - Stony Brook University
- Prof. Dr. Fernando Magno Quintao Pereira
  DCC - UFMG
- Prof. Dr. Paulo Lício de Geus
  IC - UNICAMP
- Prof. Dr. Diego de Freitas Aranha
  IC - UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 16 de dezembro de 2016

Acknowledgements

I would like to thank my advisors, Profs. Sandro Rigo and Vasileios Kemerlis, who, above all work we have done together, I consider friends.

I thank Natasha Magno, my dearest love, for bringing some sense and sensibility into my life. Thank you for being by my side, even when we were in different hemispheres.

I thank my first family – Maria do Rosario, Joao Moreira, Tida, Eloina, Fred, Grandmothers Eloina and Lazara, Grandfathers Joao and Adherbal, Aunts Amelia, Conceicao and Dulce, Uncle Bazinho, for the unconditional support. You guys make me strong.

I would like to thank my Campinas family – Bruno Cardoso, Thiago Borges, Bruno Dilly, Rafael Antognolli, Gabriel Telles, Manoel Carlos, Jacqueline Magno, Nadjeda Magno and Pequi; my Seattle family – Rafael Auler, Gabor Simko and Nora Balint; and my NYC family – Amanda Brandao, Arnaldo Aires, Cyla Costa, Daniel Negri, Guilherme Bueno, Jason Apostolakis, Ludmilla Negri and Ricardo Perini; for being the best people around. I'm grateful for the chance of sharing pieces of my life with you.

I would like to thank Mariusz Jakubowski, for mentoring me during the internship at MSR; and Prof. Angelos Keromytis, for the visit opportunity at Columbia University. I also would like to thank the Columbia University crew – Prof. Michalis Polychronakis, Prof. Jason Polakis, Dimitris Mitropoulos, Marios Pomonis, Suphannee Sivakorn, Theofilos Petsios and Yorgos Argyros, for the great times we shared in NYC.

I thank the UNICAMP/LSC labmates, for all the fun and learning – Alexandro Baldassin, Daniel Nicacio, Divino Cesar, Eduardo Ferreira, Emilio Francesquini, Felipe Klein, Jorge Gonzalez, Lais Minchillo, Leonardo Ecco, Leonardo Piga, Maxiwell Garcia, Tiago Falcao and Raoni Fassina.

I thank CAPES, CNPq, Motorola and Fundacao Lemann, for the funding.

Finally, I thank the universe, and whatever reasons may exist behind it.

Resumo

Ataques de sequestro de controle de fluxo são uma ameaça aos sistemas computacionais conhecida desde os anos 80. Estes ataques tipicamente acontecem através de operações de memória erroneamente implementadas e que permitem a corrupção arbitrária de valores utilizados para apontar destinos em saltos indiretos. Ao modificar estes valores, atacantes podem redirecionar o fluxo de controle de um software, forçando a execução de rotinas maliciosas. Apesar de muitas propostas terem surgido propostas para solucionar este problema, os atacantes se mostraram capazes de desenvolver novas técnicas para disparar os ataques e comprometer os sistemas com sucesso, contornando as proteções. Dentre estas técnicas, ataques baseados em Return-Oriented Programming (ROP), aparecem como os mais relevantes, visto que estes possibilitam a execução de computação Turing-completa sem a necessidade de injetar código no espaço de memória do software atacado.

O kernel de sistemas operacionais também pode ser alvo de ataques ROP. Desde a introdução de políticas que impedem a execução da memória de dados W^X e a posterior mitigação de ataques return-to-user, ROP se tornou a forma mais eficiente para realizar a corrupção do controle de fluxo do kernel. Como o kernel executa com privilégios de sistema que permitem o comprometimento completo do sistema quando explorado, proteger este componente de software contra estas formas de ataque se tornou imprescindível. Nesta tese, nós propomos, analisamos e otimizamos soluções para validação do controle de fluxo no contexto do kernel.

Primeiro nós propomos kCFI, uma solução de Integridade de Controle de Fluxo baseada em compiladores, de fina granularidade. Nesta solução, computa-se um grafo de controle de fluxo que é utilizado para instrumentar o binário do kernel com verificações de controle de fluxo, garantindo que todos os saltos indiretos tenham um destino previamente marcado como válido. De acordo com nosso conhecimento, kCFI é a primeira implementação de fina granularidade capaz de suportar o kernel do Linux, apresentando custos de desempenho de 8% em micro-benchmarks e 2% em macro-benchmarks. Estes valores são os menores já observados para uma solução de Integridade de Controle de Fluxo para o kernel.

Sabe-se que soluções de Integridade de Controle de Fluxo podem se beneficiar de informações dinâmicas do contexto de execução de um software para criar políticas ainda mais restritivas. Por isso, terminamos nosso trabalho apresentando uma análise a respeito da implementação de uma pilha auxiliar, a ser utilizada na validação de retornos de funções, dentro do kernel. Neste estudo, propõe-se um projeto de arquitetura de uma pilha auxiliar que é compatível com os requerimentos do kernel e que pode ser acoplada junto com o kCFI. Exploram-se também duas diferentes extensões da arquitetura x86-64 para avaliar suas eficiências na proteção seletiva das regiões de memória utilizadas pela pilha auxiliar.

Abstract

Control-flow hijacking attacks have been a known threat to computer systems since the 80s. These attacks typically take place through wrongly implemented memory operations that allow arbitrary corruption of values used to point targets in indirect branches. By modifying these values, attackers redirect control-flow as desired, forcing the execution of malicious routines. Although many solutions have been proposed to disable these threats, attackers have been able to bypass these mechanisms, developing new techniques to launch exploits and compromise systems successfully. From these techniques, Return-Oriented Programming (ROP) attacks stand as the most relevant, as they manage to perform arbitrary Turing-complete computation without the need of code injection in the memory space of the attacked software.

Kernel software is also targetable by ROP attacks. Since the introduction of W^X policies and the later mitigation of return-to-user attacks, ROP became the most prominent form of kernel control-flow corruption. As kernel runs with higher privileges that allow full system compromise upon exploitation, hardening this software component against these forms of attack became a valuable asset. In this thesis, we propose, analyze and optimize solutions for control-flow assertion in kernel software.

First, we propose kCFI, a fine-grained compiler-based Control-Flow Integrity solution for operating system kernels. This protection computes a kernel control-flow graph and instruments its binary with control-flow assertions to ensure that all indirect branches happen through paths foreseen in the graph. To the best of our knowledge, kCFI is the first fine-grained implementation capable of supporting the Linux kernel, presenting an average performance cost of 8% on micro-benchmarks and 2% on macro-benchmarks, which are the smaller observed overheads for a kernel Control-Flow Integrity solution.

Given that Control-Flow Integrity solutions can benefit from dynamic context information to create even more restrictive policies, we finish our work presenting a feasibility analysis for a shadow stack implementation to be used on function return validation, in the kernel. In this study, we first propose a shadow stack architecture design that is compliant with kernel requirements and that can be built on top of kCFI. We also explore two different x86-64 architecture extensions to assess their efficiency on selectively protecting the memory regions used by the shadow stack.

List of Figures

List of Tables

Acronyms

AIA . . . . . Average Indirect targets Allowed
AIR . . . . . Average Indirect target Reduction
ASLR . . . . . Address Space Layout Randomization
BTS . . . . . Branch Trace Store
CET . . . . . Control-Flow Enforcement Technology
CFG . . . . . Control-Flow Graph
CFI . . . . . Control-Flow Integrity
CGD . . . . . Call Graph Detach
COP . . . . . Call-Oriented Programming
CVE . . . . . Common Vulnerabilities and Exposures
DBI . . . . . Dynamic Binary Instrumentation
DBM . . . . . Dynamic Binary Modification
DBT . . . . . Dynamic Binary Translation
DFA . . . . . Data Flow Analysis
DoS . . . . . Denial of Service
DSA . . . . . Data-Structure Analysis
GCC . . . . . GNU Compiler Collection
GOT . . . . . Global Offset Table
IDS . . . . . Intrusion Detection System
IPC . . . . . Inter-Process Communication
IR . . . . . Intermediate Representation
IRQ . . . . . Interrupt Request
IST . . . . . Interrupt Stack Table
JOP . . . . . Jump-Oriented Programming
KASLR . . . . . Kernel Address Space Layout Randomization
LKM . . . . . Loadable Kernel Module
LLVM . . . . . Low-Level Virtual Machine
MCE . . . . . Machine Check Exceptions
MPK . . . . . Memory Protection Keys
MPX . . . . . Memory Protection Extensions
MAC . . . . . Message Authentication Code
NMI . . . . . Non-Maskable Interrupt
NX . . . . . Non-eXecutable
OS . . . . . Operating System
PAN . . . . . Privileged-Access Never
PXN . . . . . Privileged-Execute Never
RAP . . . . . Return Address Protection

ret2dir . . . . . return-to-direct-mapped-pages
ret2usr . . . . . return-to-user
ROP . . . . . Return-oriented Programming
SMAP . . . . . Supervisor Mode Access Protection
SMEP . . . . . Supervisor Mode Execution Prevention
SSA . . . . . Single Static Assignment
SVA . . . . . Secure Virtual Architecture
TLB . . . . . Translation Lookaside Buffer
TSS . . . . . Task State Segment
vDSO . . . . . virtual Dynamic Shared Object
VM . . . . . Virtual Machine
VMI . . . . . Virtual Machine Introspection
VMI-IDS . . . . . Virtual Machine Introspection-based Intrusion Detection System
W^X . . . . . Write XOR Execute
WP . . . . . Write Protect
XPFO . . . . . eXclusive Page-Frame Ownership

Contents

Chapter 1

Introduction

Everybody's talking about the stormy weather
And what's a man to do but work out whether it's true?
Looking for a man with a focus and a temper
Who can open up a map and see between one and two

Sonic Youth

Your money became a number stored in a 64 bit variable sinked in a data-center computer somewhere below ground. Missiles are now launched from computerized flying non-tripulated slingshots. Even your heart now talks to your watch and tells your robot doctor that you did not go for a run last night. The world is no longer a fully analog place and, as a significant part of our lives became resident inside computer realms, the requirement for security in these systems became critical.

Computer systems need to be safe, as a lot of our modern life depends on them. With so much at stake, a race between hackers drove the computer security field wildly in the last decades. Attackers, those trying to exploit the systems, and defenders, those trying to protect them, have been working fiercely to develop new techniques. This race pushed the research on security towards a plethora of new methods and strategies that covers all areas of computer science and engineering.

Due to their complexity, modern computer systems needed to be divided into different components, which are normally layered to provide functionality abstractions. These abstractions are crucial to the evolution of the computer systems, as they allow scientists and engineers to improve specific computational components without the requirement of fully understanding or reimplementing the entire system. Amongst the system's components, the operating system (OS) kernel stands as a corner stone responsible for controlling hardware and software, providing resources and services to other components.

From a security perspective, the OS kernel is a high-value asset, as it operates under a privileged mode that allows ambient access to every system resource. The exploitation of kernel-level software vulnerabilities has thus been a major goal of attackers for achieving privileged full system access. In the past few years, kernel exploits have seen renowned interest, as the exploitation of user space client and server applications is becoming more challenging due to the deployment of sandboxing and container technologies, along with numerous other exploit mitigations. Once confined within a sandbox, it is often easier for an attacker to exploit a kernel vulnerability to gain full system access, instead of finding

a sandbox-specific vulnerability. The former is typically easier due to the abundance of exploitable kernel bugs, and the relative lack of kernel-level exploit mitigations compared to user space programs. Indicatively, already before the end of 2016 there have been 427 reported kernel vulnerabilities according to the National Vulnerability Database, 172 more than in 2015 [115].

Due to the complexity and unique characteristics of the kernel, the deployment of exploit mitigations is often lacking or is not existent at all compared to user space. For instance, although address space layout randomization (ASLR) [172] has started being employed by major OS kernels [51], it still suffers from limited entropy issues compared to user space implementations. In addition, even when there are no entropy issues, kernel memory leakage or side channel vulnerabilities can be leveraged by attackers to pinpoint module addresses and bypass ASLR [165,167]. Given that *Return-Oriented Programming* (ROP) [15] is becoming the most prevalent exploitation technique at the kernel setting, this is a crucial issue, as ASLR is currently the main deployed mitigation against kernel-level code reuse attacks in modern OS kernels.

Control-Flow Integrity [4] is an additional, orthogonal defense against ROP attacks that, after many refinements in academic research [10, 22, 44, 110, 123, 124, 131, 135, 136, 176, 235, 236], is finally getting traction in user space with compiler, OS, and hardware support [1,2,38,177]. By confining program execution within the bounds of a precomputed profile of allowed control flow paths, CFI can prevent most of the irregular control flow transfers that take place during the execution of ROP code.

In contrast to the large body of works on CFI for the protection of user space programs, only a few efforts have focused on the application of CFI at the kernel setting [41, 65, 101]. Due to the complexity of kernel code and its unpredictable execution behavior caused by interrupts and exceptions, existing implementations either apply an overly permissible coarse-grained policy, to avoid the complexity of extracting a complete control flow graph [41], or are not compatible with dynamically loadable kernel modules, to facilitate the extraction of the control flow graph (CFG) needed for deriving a more fine-grained policy [65]. Coarse-grained CFI offers only limited protection, as it still permits plenty of valid code paths for the construction of functional ROP exploits [18, 45, 67, 68, 161], while lack of support for dynamically loadable modules limits the practical applicability of the protection, especially for Linux, which heavily relies on them. It is indicative that previous proposals focus exclusively on FreeBSD and MINIX, which pose fewer complexities compared to Linux.

1.1 Hypothesis

Fine-grained Control-Flow Integrity (CFI) stands as a prominent defense against modern control-flow hijacking attacks. The effectiveness, however, of such CFI schemes depends on how precisely they can approximate the CFG of the protected application. Relaxed CFGs result in weaker protection, while over-approximated ones may break functionality. Despite the need for extracting source code semantics from the application to be protected, previous CFI schemes, and particularly those used on OS kernel settings, require

hazardous policy (enforcement) relaxations or impose limitations to the resulting scheme.

Considering the above, we hypothesize that by leveraging multi-level program analyses it is possible to construct a performant fine-grained CFI scheme, for the kernel setting, which is not limited in terms of (protection) coverage and does not preclude functionalities found in commodity OSes.

1.2 Thesis Statement

This thesis argues that by coupling novel binary- and source-level static analyses with runtime information it is possible to construct a *fine-grained* CFI scheme for the kernel setting that (a) provides *full* coverage to low-level OS code (including assembler code), (b) is compatible with indispensable functionalities of commodity kernels (e.g., support for dynamically loadable modules), and (c) incurs a moderate performance overhead.

1.3 Contributions

- We present the design and implementation of kCFI, a kernel-level fine-grained CFI mechanism that fully supports dynamically loadable modules and hand-written assembly code. kCFI does not depend on incomplete pointer analysis, nor restricts language features, as done in previous works. kCFI is the first solution of its kind to support the Linux kernel, and it is orthogonal to other widely deployed exploit mitigations.

- We present a novel technique called *Call Graph Detaching*, which enhances the offered protection by enabling the construction of more precise CFGs and, consequently, enforcing a more restrictive CFI policy, with negligible additional performance cost.

- We assess the use of kCFI on the Linux kernel, leveraging features of the x86-64 architecture to optimize performance.

- We have experimentally evaluated kCFI in terms of performance and security using standard benchmarks and state-of-the-art attack techniques. Our results demonstrate that kCFI offers effective protection while incurring a low overhead comparing to previous proposals.

- We propose the design for a kernel shadow stack implementation which leverages the use of hardware features and memory management capabilities to efficiently protect this data structure against corruption attacks.

- We present a first study on the performance of different hardware features employed for memory protection on the kernel shadow stack context.

1.4 Thesis Roadmap

Chapter 2 provides background information regarding computational security, especially on what concerns control-flow hijacking attacks. Chapter 3 presents the design and implementation of kCFI, our proposed CFI solution. Chapter 4 presents the experimental evaluation of kCFI regarding the introduced performance and code size overheads, plus a security efficiency analysis and a discussion regarding limitations of the approach. Chapter 5 brings the design of a shadow stack implementation for the kernel context, providing a first study about different strategies for the scheme's self-protection and the overheads introduced by them. Chapter 6 compares our work with relevant state-of-the-art kernel protections. Finally, Chapter 7 draws the conclusions of our work.

Chapter 2

Background

> Empty your mind. Be formless, shapeless, like water. Put water into a cup, it becomes the cup. Put water into a teapot, it becomes the teapot. Water can flow or creep or drip or crash. Be water, my friend.
>
> *Bruce Lee*

In this Chapter, we present the key concepts for understanding this thesis. First, we introduce the components relevant to the system platform employed in our implementations, giving a general overview about each one of them. We also consider relevant those components not directly related, but employed in implementations that are tangent to our work, providing means for later comparison. Afterward, we present a chronologically organized explanation on the evolution of the control-flow hijacking attacks, leveraging threats and mitigations in a way to make clear the motivation behind each idea that influenced our work.

2.1 System Platform

Linux [60, 128] is an open-source commodity OS that supports different hardware architectures. Its kernel source code is mostly written in C with architecture-dependent parts written in the respective Assembly language. Linux separates running software by memory areas, most notably known as kernel space, that comprises data and code respective to the Linux kernel, and user space, where ordinary programs reside. As Linux groups different functionalities such as device drivers, file system services, process scheduling and memory allocation in the kernel space, it is considered a monolithic [171] system. The Linux kernel supports dynamically loadable kernel modules (LKM), which can be compiled individually and loaded in a different time than the system core.

The Low-Level Virtual Machine (LLVM) [99] is a modular open-source compiler infrastructure that supports different programming languages and target architectures. LLVM first translates high-level source code into an intermediate representation (IR) that follows the Static Single Assignment (SSA) [24] form. For code written in C or C++, this work is done through a front-end component called Clang [144], which has an extensive set of built-in analysis tools. The IR code, illustrated in Figure 2.1, is then optimized and

submitted to the back-end portion of the compiler, that translates it into assembly code for the targeted architecture. The architecture behind LLVM and its IR was designed to favor plugging new optimizations and analysis to the compilation pipeline. LLVM also implements different offline tools, such as the Clang Static Analyzer [145], which is capable of highlighting bugs in source code files.

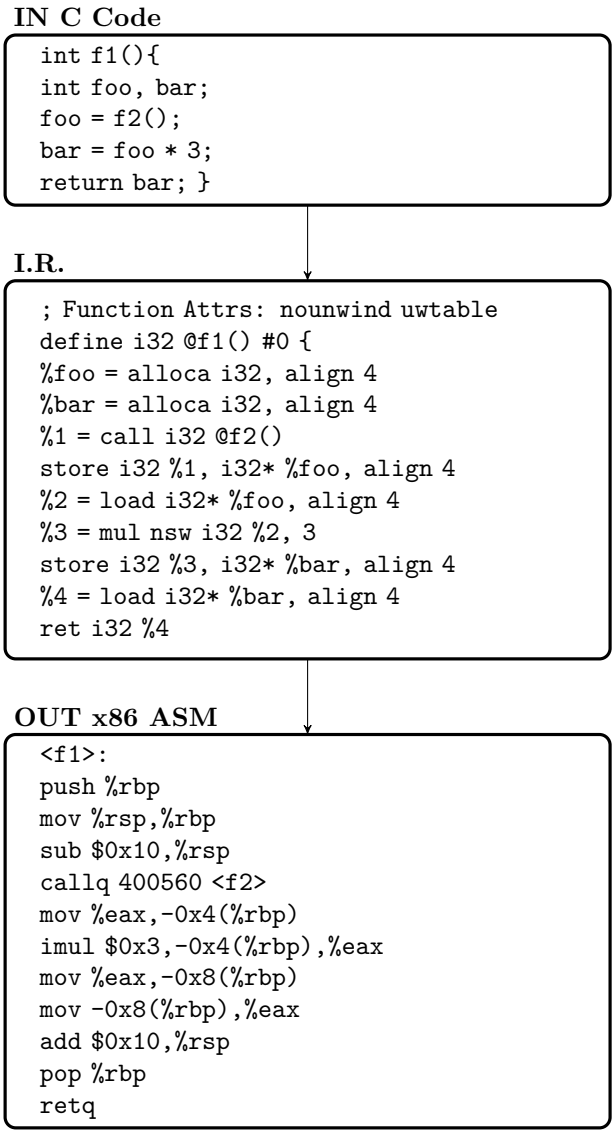


Figure 2.1: LLVM compilation stages

The GNU Compiler Collection (GCC) [59] is the official Linux OS compiler. Although GCC also has its perks, its internal IR management differs from LLVM's, especially on what concerns offline accessibility: LLVM supports optimizing binary IR representations through a tool called *opt* [146]. Besides, the documentation existent for both compilers is wider for LLVM's internals. The quality of the code generated by both compilers is comparable.

LLVMLinux [103] is a project that provides support for building the Linux kernel with LLVM. The project consists of a patch set that must be applied to the kernel source code to make it compatible with language standards accepted by Clang. The project also brings

`Makefile` rules and tools for testing the compiled kernel under virtualized environments.

2.1.1 Virtual Machines

Virtual Machines (VMs) are systems that emulate real computers. It is possible to define a VM as a combination between hardware resources and the software responsible for performing its virtualization [166]. The virtualization process consists in mapping the real resources into virtual resources, creating emulated devices to the virtualized machine, and in the use of instructions on the actual machine to execute tasks of the virtualized machine. VMs enable more portability and flexibility of the virtualized software, allowing the execution of software in non-native system architectures. VMs can be classified as process VMs, which run a single virtualized process, and system VMs, that emulate full execution environments and enable the virtualization of entire OSs. VM capabilities are also supported by hardware extensions [47, 179], what avoids requirements of para-virtualization support [227]. Some of the most prominent VM implementations are VMware [183–185], QEMU [8, 150], KVM [94], and Xen [7, 48].

Dynamic Binary Modification (DBM) tools enable access and control over user-level instructions during runtime [74]. These tools are mostly used for runtime analysis and modification of a particular program, providing ways to perform context-driven instrumentation. DBMs are implemented as a software layer between the OS and the running program, allowing the management of its execution and ensuring control over the execution flow. A widely used class of DBMs are the Dynamic Binary Instrumentation (DBI) platforms, which are specialized in inserting new instructions in the executed binary. Amongst the most well known DBI tools [74] are *DynamoRIO* [6, 13, 14, 50], Pin [106] and Valgrind [117–119].

2.2 Early Control-Flow Hijacking Attack and Defense

Control-Flow Hijacking attacks are a class of software attacks in which the program's legitimate control-flow is forcibly subverted in a way to run unexpected (and possibly malicious) computations. These attacks are typically launched through bugs in memory handling functions that end up allowing users to overwrite memory, permitting the corruption of code pointers that are later used to diverge the application's control-flow.

Since the discovery of these attacks, different mechanisms to prevent them were developed, requiring attackers to develop new techniques to bypass these defenses and successfully subvert the control of applications. The evolution of these attacks and defenses, driven by both hardware and software strategies, create a myriad of concepts that can be better understood together with each motivation. For this reason, we organized the following sections as a general timeline that shows how and why different techniques were developed until culminating in the state-of-the-art Control-Flow Hijacking attacks and prevention mechanisms.

2.2.1 First Methods for Control-Flow Hijacking

A buffer overflow [58, 76] consists in a program error in which a memory operation over a buffer exceeds its limits, possibly overwriting adjacent data. Normally, a buffer overflow occurs due to the use of functions that perform memory operations without verifying the correctness of its operated addresses regarding the target buffer limits. These bugs are usually seen in programs implemented with languages that do not have sophisticated memory management mechanisms, such as C and C++. Despite that, as unsafe languages are used to implement their underlying sandboxes, safe languages, like Java or Ruby, end up being indirectly affected [193, 204, 209, 213]. When these bugs are present, users may be able to craft malicious inputs, larger than the buffers meant to hold them, and, by overwriting adjacent memory, corrupt code pointers.

In 1988, a student called Robert Morris wrote what would become the first widely known attack based on buffer overflows [52]. This attack was incorporated into a computer worm later referred to as the *Morris Worm*. In the same year, the malicious program was spread throughout the Internet, contaminating more than six thousand UNIX servers. In 1996, a description of how to use buffer overflows as a vector for system corruption was published on the virtual magazine *Phrack* [139]. This paper, entitled *Smashing the Stack for Fun and Profit* [127], shows that unbounded memory operations on stack variables allow an attacker to overwrite the return address stored in the respective function stack-frame, enabling control-flow hijacking towards different parts of the program. The exposed technique also describes how code can be written in a binary form known as *shellcode* and injected on the stack to be later reused in the launched attack.

Attacks on buffers allocated in the heap region are also possible [82]. Similarly to the stack case, a buffer overflow while managing heap memory allows overwriting adjacent data. Heap management uses connected lists to handle its non-contiguous allocated blocks. While deallocating elements on the list, information present in its header is used to reorganize the pointers in the data structure. This way, overflowing a heap buffer allows overwriting headers of the next node on the list and consequently corrupting the data which will be used by a deallocation (`free()`) operation on that node. By corrupting this data, specifically addresses that must be updated to keep the blocks list consistent, it is possible to turn `free()` operations into arbitrary memory writes that can be used to overwrite function pointers or return addresses [5, 9, 81].

Control-flow hijacking is also possible through attacks which are not based on buffer overflows, as it happens on *format strings* [163] attacks. These attacks focus on C functions, such as `printf()` or `fprintf()`, that receive a parameter which is a string used to convert variable values into human readable messages. If a user is capable of manipulating the contents of the string to be printed making the program understand it as a format string, he can force arbitrary reads and writes in the program memory, again enabling function pointer or return address corruption. *Format Strings* were also seen in Kernel code [205–207].

2.2.2 Protections Against Control-Flow Hijacking

Although the attacks described in Section 2.2.1 remain a threat, these were mitigated in many different ways by security researchers and software developers. The solutions described in this subsection are a important for understanding the motivations and concepts behind the evasion techniques used in modern exploitation strategies.

StackGuard [40, 225] is a solution developed with the goal of protecting return addresses stored on the stack. This solution consists of placing particular values, namely canaries, between memory buffers and stack variables in a way that memory operations across buffer boundaries will cause canary destruction. By checking the canary before dereferencing values on the stack, it is possible to identify a pointer corruption and prevent a possible exploitation. Similar approaches were also developed to protect heap variables [121].

StackShield [182] is a solution that separates the stack used to store return addresses from the stack used to hold data, avoiding that overflowed data buffers overwrite adjacently saved addresses. StackShield also implements checks for valid ranges whenever dereferencing return addresses and function pointers, allowing the identification of targets out of code memory regions.

Address Space Layout Randomization (ASLR) [172, 229] consists in randomizing the position of objects in memory. This technique raises the bar on system exploitation by preventing memory objects from being placed on the same memory address in different execution instances, thus, making it much harder for an attacker to craft an exploit since she does not have consistent knowledge regarding the position of the data structures or code. ASLR also works on Kernel space [51].

Write XOR Execute (W^X) policies enforce every memory page to be exclusively executable or writable [98, 102, 112, 129]. By forbidding the contents of writable memory regions to be interpreted as instructions, this feature prevents attackers from injecting malicious functions as data inputs and later diverting control-flow to execute them. To enable this policy, some processor architectures offer a Non-eXecutable (NX) bit that allows efficient flagging of memory page permissions.

Shadow stacks [23, 43, 61, 116, 143] are data structures used to replicate return addresses stored in the program stack, allowing its assertion before its use. Shadow stacks present noticeable computational costs, being expensive even in its light-weight implementations [43]. Ideally, as a shadow stack is also a memory region, it needs to be protected against all writes except its legitimate updates. Switching the memory write permissions repeatedly for every shadow stack operation introduces more overhead, and this motivated developers to trade off security for efficiency. As an example, RAD [23] applies fixed write protection to the shadow stack boundaries, deflecting attacks through contiguous memory writes meant to reach the data structure. Although better regarding performance, this solution does not protect against arbitrary memory writes [5, 9, 81, 82].

Shadow stack implementations also have to deal with binary constraints and language features that break synchronization between the application's stack and the shadow stack. For example, to fix parity issues caused by `setjmp/longjmp` functionalities, some implementations relax the shadow stack policy allowing returns to any stacked address, irre-

spectively to its order. This approach opens a window for techniques like *loop injection*, used in Control-Flow Bending attacks [17].

2.2.3 Defense Evasion

From the defenses described in Section 2.2.2, StackGuard, ASLR and W^X were widely adopted into distributed software. As these impose hard limitations to the attacks exposed on Section 2.2.1, attackers developed defense evasion techniques, allowing successful attacks even when under protected environments. These techniques, later, culminated into the state-of-the-art attack techniques.

Systems protected with StackGuard and StackShield are still vulnerable to attack methodologies that can overwrite return addresses without corrupting canaries, as it happens while exploiting bugs that allow arbitrary memory writes. Both protections also remain vulnerable to attacks focused on function pointer corruption, including entries in the *Global Offset Table* (GOT) [16, 153]. Brute-force techniques to enable the construction of exploits capable of overwriting memory without destroying the canary were also developed [107].

A class of bugs known as memory disclosure bugs allows attackers to retrieve information about the memory layout that may be used to derive addresses needed for exploitation. These attacks happen either through wrongly implemented memory read operations, that disclose the position of memory objects [49, 170], or through brute-force attacks [165], exploiting child processes that are recreated after crashes caused due to invalid memory accesses, and that always inherit the parent's randomization characteristics. Even very fine-grained ASLR implementations are susceptible to attacks through memory disclosure [167] since an attacker can repeatedly exploit memory disclosure bugs and use just-in-time compilation methods to generate attack payloads dynamically.

As modern OSes adopted W^X policies [98, 102, 112, 129], attackers had to evolve their exploitation strategies to reuse instructions present in code memory pages, which are inherently flagged as executable. *Return-into-libc* [46] attacks, the first known form of code-reuse attack, consisted in overwriting code pointers to divert control-flow towards functions contained in libraries mapped into the process memory space. Although powerful, these attacks are limited to the functionalities already existent in the program.

To expand the computational capacity of code-reuse attacks, techniques based on ROP [15, 142] were developed, initiating a plethora of new exploitation methods. Due to its capacity of overcoming the limitations imposed by widely adopted defenses, employing ROP-based attacks became a prominent offensive strategy both in user and kernel spaces. Given its relevance, and the broad set of derivative attack and defense approaches, we reserve Section 2.3 to do an in-depth analysis of ROP-based and subsequent methodologies.

2.3 Control-Flow Hijacking State-of-the-Art

The previously described defenses pushed attackers to develop new methodologies for exploiting computer systems, leading to control-flow hijacking attacks that efficiently employ code-reuse. Although these attacks remain not fully addressed, relevant efforts

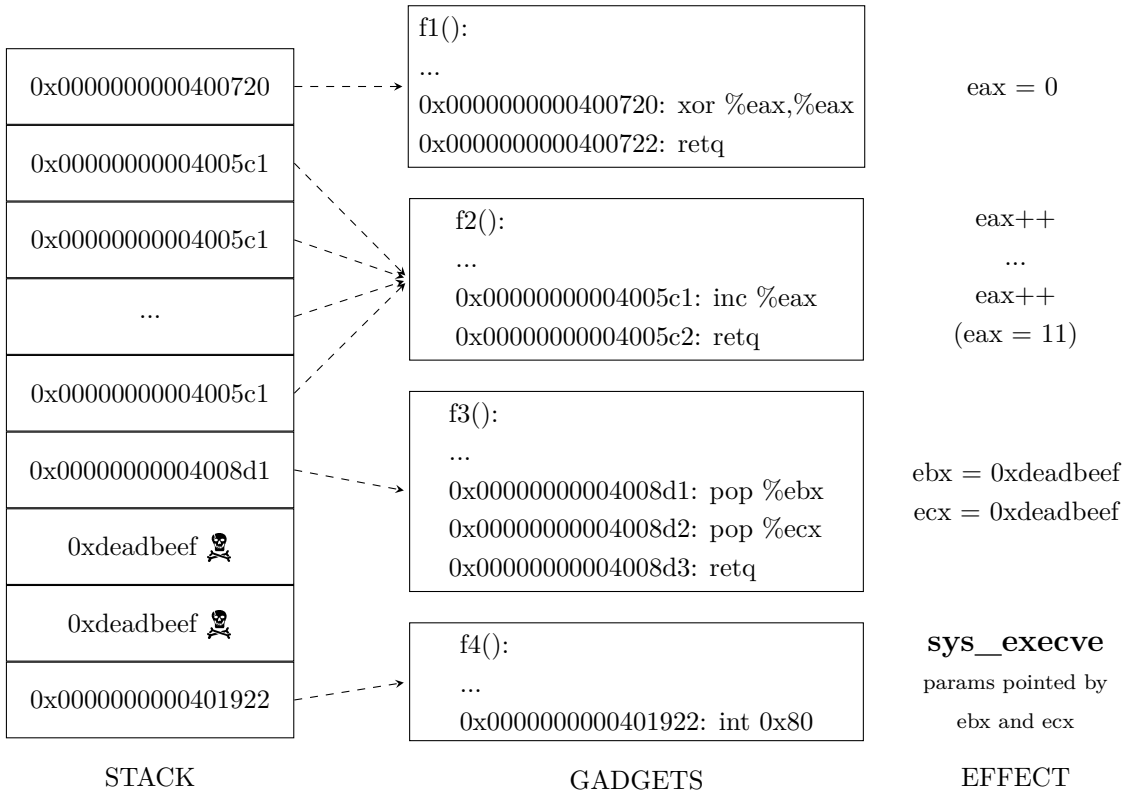


Figure 2.2: ROP example: sys_execve

were done to prevent their feasibility, accounting both software and hardware strategies. In this Section, a more detailed explanation regarding these attacks is provided, followed by the mechanisms meant to protect against them and a description on how all these pieces come together in the kernel context.

2.3.1 Return-Oriented Programming

ROP is a technique that allows an attacker to overcome restrictions that prevent the execution of injected code by reusing snippets from legitimate functions, which are allowed to be executed. These code snippets, called gadgets, are sequences of instructions ended by a `ret`[1]. When executed, gadgets run a small piece of the intended computation and then redirect control-flow through the closing return instruction. To properly chain gadgets in a ROP attack, an attacker must control the process stack and populate it with return addresses that will redirect the code throughout the desired snippets. Hence, as each small operation executes and returns, the address of the subsequent gadget is popped out of the stack. By combining multiple gadgets, an attacker can perform arbitrary Turing-complete computation.

Figure 2.2 shows an ROP attack example where an attacker was capable of exploiting a memory related bug to write arbitrarily on the program's stack. The exemplified attack targets an x86 machine running Linux, from which the system call `sys_execve` is invoked.

[1] `ret`-ended gadgets are the most basic kind of gadget, but there exist gadgets which are ended with different kinds of indirect branches.

On the left side of the figure, it is possible to see the corrupted stack containing the addresses of the gadgets. In the center, the figure displays four different gadgets that will be used during the attack. On the right side, it is possible to see a pseudo-code that represents the effective actions executed during the attack.

In Figure 2.2 when the exploited function returns, it pops the first address out of the stack and redirects the flow towards the gadget in `f1()` that will `xor` the register `eax` with itself, resulting in a zero value assignment. The next 11 stacked addresses will redirect the flow towards a gadget that increments the register `eax` by one, leaving the register with the value 11 after all of them are popped. The next address redirects the flow towards two `pop` instructions that will copy values from the stack into the registers `ebx` and `ecx`. As the attacker controls the stack, she is capable of controlling whatever value may be loaded into these registers. Finally, the last stacked address redirects the flow towards an `int` instruction, which will invoke the syscall `sys_execve` that will use `ebx` and `ecx`, which are controlled by the attacker, to dereference its parameters, resulting in an arbitrary execution of code.

By exploiting memory disclosure bugs, it is possible for attackers to leak source code of target programs, including information that may neutralize randomization based protections. By using ROP compilers [130,158], attackers can then build their attack payload dynamically, fitting specific attack requirements.

A commonly used strategy to employ ROP is pivoting the stack [238] to a different memory region. When applying this methodology, the attacker creates the chain of return addresses in a different memory area, such as the heap, and then manages to overwrite a stack pointer saved on the stack frame with the address of her return address chain, moving the stack frame of the process entirely to a different place. This technique is useful for bypassing stack protections or when the system exploitation happens through vulnerabilities apart from stack buffer overflows.

Although ROP became notorious for exploiting the branch respective to return instructions, derivations based on other kinds of branches exist [19]. *Jump-Oriented Programming* (JOP) [11] exploits indirect jumps to chain gadgets. These attacks do not rely on the stack to control flow. Instead, they make use of a particular kind of gadget, called *dispatcher gadget* which is responsible for chaining the functional gadgets through using a dispatch table. *Call-Oriented Programming* (COP) [18] is also a variant which is based on indirect calls. As these instructions frequently have memory operands, *dispatcher gadgets* are not required and the attack can be launched through corruptible pointers in memory.

2.3.2 Control-Flow Integrity

Control-Flow Integrity (CFI) [4] is a technique meant to ensure the correctness of control-transfers during software execution. The general idea behind CFI consists in instrumenting the program with assertions that will validate the targets of indirect control flow transfers right before its execution takes place. By doing that, these mechanisms are capable of identifying and deflecting attacks such as ROP and variants [11, 142, 164]. Different techniques for verifying the validity of a target exist, but the most generic one consists in dereferencing the pointer used in the indirect control transfer and checking if it

is pointing towards pre-defined values (tags). In this sense, enhancing software with CFI consists in correctly calculating the set of valid targets for every indirect control-transfer instruction, instrumenting these instructions with assertions and the destinations with the respective tags.

A CFI mechanism is said to be coarse-grained [110, 236] if its control-flow target validation is loose, allowing returns to any instruction after a *call* instruction and indirect calls to the first instruction of any function. These mechanisms were proven vulnerable as malicious flows could be traced inside the protected applications even though the CFI restrictions were applied [18,45,67]. Motivated by that, fine-grained CFI mechanisms were developed [108,111,177]. On what concerns protecting return addresses, these techniques enforce that returns must only be allowed to instructions after a `call` to the respective returning function. For protecting indirect calls, they enforce that only the first instruction of a subset of functions, built through a pre-defined rule, are considered a valid target.

One of the biggest challenges of implementing fine-grained approaches for CFI is building its CFG with valid destinations for branches. As known, *complete* and *sound* pointer analysis is an undecidable problem [152], which prevents the creation of precise target sets for indirect branches and creates the need for heuristics to solve the problem. Besides, some language features and optimizations, such as `setjmp/longjmp`, frequently break assumptions on regular control-flow of programs, requiring these schemes to find ways of being compliant or limiting its use [43, 57].

Conti *et al.* [32] showed that the approach used in the implementation of CFI schemes is very relevant. In this paper, the authors exploit the CFI mechanism known as IFCC [176], which is based on restricted pointer indexing, a different approach for validating indirect call targets. This technique consists in converting indirect branches into jump tables, restricting execution to only branch through its compromised addresses. Conti *et al.* exploit this mechanism by overwriting registers at the moment that they are saved on the stack, in between function invocations. As these registers are later used to dereference the jump tables, the author is capable of subverting the mechanism by replacing the used tables with malicious ones. The paper also shows a methodology for attacking user space CFI-instrumented software by corrupting the return address which is later used by the OS kernel for returning from interrupts. As the OS is not instrumented with CFI checks, returning to any address is considered valid.

Attacks on Fine-grained CFI

Two attacks on user space fine-grained CFI [17, 53] demonstrated that these mechanisms may also be vulnerable under very particular circumstances. Both attacks are specializations of non-control-data attacks [3] that exploit function arguments and employ control-flow hijacking accordingly to the CFI enforced CFG, thus, not triggering violations.

Carlini *et al.* [17] shown that it is possible to achieve arbitrary computation through the corruption of function arguments. The attack, named *Control-Flow Bending*, is demonstrated through a technique called *printf-oriented programming*[2] which is based on the

[2]Despite the use-case, the concepts exposed are not limited to `printf()`.

corruption of `printf()` arguments similarly as done in format-strings attacks[3]. Yet, Carlini *et al.* also show that shadow stacks stand as meaningful defenses against control-flow hijacking attacks, effectively preventing certain attacks.

Evans *et al.* [53] describe Control Jujutsu, an attack over a hypothetical fine-grained CFI mechanism whose indirect-call targets were gathered through the Data-Structure Analysis (DSA) [100] algorithm. The attack consists in corrupting the arguments of a specific function in a way to achieve the arbitrary computation and later forcing its invocation through retargeting an indirect branch without diverting the enforced CFG. As these attacks focus indirect forward-edges, it can be limited by more restrict CFG computation algorithms, but can't be tackled through shadow stacks.

2.3.3 Branch-ratio Based Defenses

Willing to prevent ROP attacks, researchers proposed defense mechanisms based on the analysis of branch ratios. The concept behind these mechanisms consists in dynamically checking the length of instruction sequences in between indirect branches, searching patterns which are composed by chains of small instruction sequences and are intrinsic to gadget chaining in ROP attacks. An advantage of using such scheme is that source code and debugging symbols are not required.

kBouncer [131] is a system that implements branch-ratio analysis to prevent ROP. kBouncer exploits a branch-tracing mechanism called *Last Branch Recording* (LBR) [36] which is supported by recent Intel CPUs to monitor the frequencies of branch instructions. The analysis is made whenever an invocation to the system's API happens, ensuring that privileged code only runs after the system integrity is validated. kBouncer was implemented on the Windows OS platform.

ROPecker [22] implements a similar mechanism that also detects ROP by using LBR. Instead of trapping system's API entries, ROPecker uses a sliding window that ensures verifications by setting most recent visited code regions of the protected application as executable, leaving the rest of the application outside this window. Attempts to execute code which is out of the window triggers an exception that invokes the ROP checker.

Researchers demonstrated attacks to defenses based on branch-ratio analysis [18, 45, 67, 68]. As both rely on a pattern which is matched when a threshold of small instruction sequences between branches are chained, attackers decoyed the pattern by placing long but innocuous sequences of instructions amongst their gadgets. By ensuring that these long sequences are executed before the invocation of the ROP checker, *i.e.,* before invoking the system API or before invoking code outside ROPecker's sliding window, the pattern is broken and the attack is missed.

Trying to improve these systems, Tymburibá *et al.* [155, 178] proposed the use of gadget length thresholds which are application specific. By doing that, the size of the gadgets considered harmful is increased, and it becomes harder for attackers to decoy their exploits. Finally, the work by Botacin *et al.* [12] shows the use of a different hardware resource called *Branch Trace Store* (BTS) to store branches on memory pages instead of

[3]The attack does not depend on the existence of format-strings bugs, as by controlling the arguments the attacker is capable of modifying the referenced strings in a way to create the needed conditions.

registers, as done in LBR implementations. By doing such, this implementation is capable of analyzing longer traces, being more resilient to evasion techniques.

2.3.4 Hypervisor-based Defenses

Different solutions have been proposed for cases where it is not possible to recompile the program. Most of these solutions are based on the insertion of an extra layer between the application and the system, what enables monitoring of the binary activities and detection of possible behavior corruptions.

The use of VMs as a security enabler came together with the concept of *Virtual Machine Introspection* (VMI). This idea consists in the ability to observe and analyze the internal state of a VM [63, 114, 137, 138]. Through VMI, a system developed to detect and prevent attacks is capable of observing the protected target from an external perspective that is protected against attacks through the extra isolation provided by VM abstraction. *Intrusion Detection Systems* (IDS) [159] based on VMI (VMI-IDS) were proposed by researchers [63,80,93,95,186] as a more robust solution for monitoring systems in opposition to previously proposed network-based or host-based IDSs [160]. Also, VMs were used [62] as enablers for Trusted Computing [69, 157].

Native Client [230] tries to enable high-performance native execution of applications running inside a web browser. NaCl works as a sandbox, fencing and monitoring non-trustable applications that may have been downloaded from the Internet and ensuring its execution without causing system compromise.

DBM has been used as the enabling platforms for techniques focused on the prevention of software attacks, such as Data Flow Analysis (DFA) [21, 79, 120, 231], program-flow verification [91, 92, 143], instruction set randomization [83, 140] and system call filtering [162]. This is a prominent set of new techniques to defeat software attacks but, since they rely on the use of a DBM tool, they inherently introduce high overheads.

Due to the high overheads generated by the use of DBM tools, techniques to reduce its adverse effects have been developed, especially through the exploitation of the inherent parallelism in executing an application and simultaneously analyzing it [71, 237]. This approach, with different perks and drawbacks on each implementation, was also applied while using DBM tools as a security enhancer [79].

ShadowReplica [79] is a DBM-based mechanism developed on top of Pin that tries to accelerate memory analysis of programs through the use of parallel execution. This solution relies on offline static and dynamic analysis to generate optimized code that will run in parallel and collect information from the application with low communication overhead, decoupling execution and verification in different threads. ShadowReplica was mainly used for dynamic correctness verification of programs. Even though it also implements a CFI-based security feature, its primary goal is accelerating a DFA based analyses, proposing new optimizations to decrease communication costs imposed by the parallel architecture of the tool.

It is worth mentioning that extra layers have also been used as an enhancement for attacks. BluePill [156] is a hypervisor-based *rootkit* that hides itself by not changing the system it infects. Instead of that, BluePill virtualizes the target and acts as an extra layer

between it and the hardware. Although it is possible to detect the rootkit by verifying side-effects due to the virtualization process, there is no direct evidence of infection on the attacked OS, what makes its detection harder.

2.3.5 Hardware Extensions

As security became a critical issue, hardware developers started to incorporate new extensions to its CPUs in order to assist developers in the task of implementing system security with low overhead.

Supervisor Mode Execute Protection (SMEP) and *Supervisor Mode Access Prevention* (SMAP) [34,36,66,232] are features implemented by Intel processors[4] to improve user and kernel address space isolation. These extensions prevent direct access to addresses whose page table entry is marked with a supervisor bit, raising a page fault. By doing this, SMEP and SMAP prevent control transfers and pointer dereferences from kernel to user space in a more efficient way than PaX. The ARM architecture has similar features under the names *Privileged Execute-Never* (PXN) and *Privileged Access-Never* (PAN) [113,168].

Memory Protection Extensions (MPX) [73,151] is another hardware feature developed by Intel[5]. These extensions include new registers and instructions that allow defining and verifying access boundaries for a pointer dereference prior to its use. Therefore, whenever a pointer is meant to operate in a memory array, its boundaries can be set to the array's first and last positions and the pointer dereference can be checked to be targeting an address in between these boundaries. By introducing this new technology, Intel provides means for developers to protect systems against attacks that happen through bugs such as buffer overflows .

A hardware extension which is currently under development by Intel is *Memory Protection Keys* (MPK) [35]. This extension uses 4 available bits on the page tables entries to assign the page one of the sixteen possible key values. Another 32 bit register, which is thread individual, is used to map access and write permissions on each group of pages that has a specific key. By doing that, MPK allows partitioning the whole system memory into 16 regions, which can have its access permissions selectively set.

Recently Intel released an extension specification for *Control-Flow Enforcement Technology* (CET) [38]. CET is meant to provide means for developers to defeat ROP attacks by enabling hardware supported CFI. CET will implement a hardware shadow stack to validate parity between return and call instructions, assuring the correctness of back-edges in the control-flow. For forward-edge control-flow, CET incorporates a new instruction, which is used to mark valid destinations for indirect calls. The verification feature is incorporated into the functionalities of `call` and `ret` instruction, while marking the code requires instrumentation with the new provided instruction. Although the shadow stack feature might be a game-changer on what concerns defending against ROP attacks, the coarse-grained forward-edge policy proposed was already proven insufficient to defeat such threat [18, 45, 67, 175].

[4]SMEP and SMAP were introduced respectively in the *Ivy-bridge* and *Broadwell* architectures
[5]MPX was introduced in the *Skylake* architecture

2.3.6 Kernel Control-Flow Hijacking

Kernel software is subject to exploitable programming bugs as much as regular user-land programs. Kernel's inherent privileges give attackers advantages while exploiting it, as it provides more accesses and visibility over the system. Indicatively, before the end of 2016 there have been 427 reported kernel vulnerabilities according to the National Vulnerability Database, 172 more than in 2015 [115]. A study on OS vulnerabilities [20] shown that, from January 2010 to March 2011, 141 Linux kernel vulnerabilities were published in the Common Vulnerabilities and Exposures (CVE) [39] list. These vulnerabilities, catalogued into 10 different categories of programming bugs, ultimately lead to different classes of attacks that can be classified as *memory corruption* [187, 192, 194, 195, 197, 198, 210–212, 214,216,217], *policy violation* [188–191,201], *Denial of Service (DoS)* [221–224] and *Information Disclosure* [196, 199, 200, 202, 203, 208, 215, 218–220], illustrating that kernel code is not only prone to attacks, but also that it can be exploited in many different ways.

If no security measures for address space isolation are applied, pointers present in the kernel can dereference addresses in user space. Attackers can exploit this characteristic by employing a technique called *return-to-user* (*ret2usr*) [85], which consists in injecting a malicious payload into user space through a regular process, and then using memory corruption bugs to overwrite control-flow pointers and divert execution to the payload's address. By applying this technique, attackers don't have to defeat protections such as kernel address space layout randomization (KASLR) [51] and W^X policies [98, 102, 112, 129].

PaX [173] is a Linux hardening patch set that protects the system against many different kinds of attack, including some variants of control-flow hijacks. This patch incorporates stack randomization features for both kernel and user space, W^X policies for memory pages and memory isolation mechanisms that restricts dereference of user space memory from the kernel [132–134], protecting against *ret2usr* attacks.

More recently, PaX integrated a feature called *Return Address Protection* (RAP) [174] to its list of enhancements. RAP is a protection conceptually based on the XOR random canary approach introduced in a later version of StackGuard [40]. RAP encrypts return addresses and keeps the decryption key, which is an XOR cookie, in a reserved general-use register. Return address encryption based mechanisms have been proposed before by the academia [126] and, while they certainly impose an obstacle to attackers, the whole scheme relies on the secrecy of the cookie, which compromises the protection if leaked.

PaX has a significant impact as a protection scheme, but the ideas behind it are not compatible with all kinds of architectures. Address space isolation on PaX uses memory segmentation features, which are not supported by x86-64 CPUs, creating high overheads on such architectures. Although applied by default in many Linux distributions, PaX was never incorporated into the Kernel upstream source code.

kGuard [89] is a multi-platform compiler-based solution to enforce isolation between kernel and user space. kGuard works by instrumenting indirect branches with assertions that verify if its destination is compromised inside the kernel address space range. If this assertion fails, it invokes a violation handler function, as this may mean that the system is under attack. The paper also describes the insertion of `nop` instruction sleds

to prevent attacks through trampolines, in which a first branch is used to jump to the address between the assertion and a second branch, also inside kernel space, which is then used to jump to user space.

Kemerlis *et al.* [87] showed how to bypass memory isolation mechanisms and indirectly manipulate kernel memory from processes resident in user space by using a technique called *Return-to-direct-mapped-pages* (ret2dir). In his attack, Kemerlis *et al.* exploit the kernel's scheme for dynamic memory allocation to create synonyms, which are kernel mappings of memory pages also mapped is user space. This technique can be used to launch attacks without dereferencing addresses in user space and, showing that, if an attacker is capable of manipulating kernel memory, exploitation is feasible.

Kemerlis *et al.* also introduce a mechanism called *eXclusive Page Frame Ownership* (XPFO) [87], ensuring that the same memory page frame cannot be accessible simultaneously in kernel and user space. XPFO prevents *ret2dir* attacks and, together with the previously mentioned techniques, improves memory space isolation significantly.

Despite their clear enhancements, previously mentioned methods do not fully close the problem on kernel isolation, as an attacker may still be able to inject attack payloads in kernel memory through general forms of data sharing between the two spaces, like user-controlled content being pushed to kernel through I/O buffers, pipes and message queues.

It has been shown that an attacker can craft ROP attacks only using code present in kernel space [77]. In this paper, the author describes how to build a rootkit using only gadgets existent in kernel code, either from its core or its modules, through ROP. Although the attack applies *ret2usr* strategies by pivoting the stack to a user space memory region, the methodology behind the attack is not hardly dependent on such techniques as it can be replicated fully in kernel space if the attacker has a way of overwriting stack values or writing on any of its memory regions.

Some implementations of CFI focus on protecting the OS kernel, hardening it against ROP attacks. HyperSafe [226] implements CFI for indirect branches on hypervisors by using restricted pointer indexing. HyperSafe also employs a technique named memory lockdown, which ensures that part of the hypervisor memory, including the page tables, is permanently write-protected, ensuring that data and code in these pages are protected even in an occurrence of a memory corruption bug. The jump tables used for the restricted pointer indexing approach are amongst the lockdown protected structures.

KCoFI [41] is a coarse-grained kernel CFI implementation for OSes, with support to FreeBSD. Instead of computing the system's call graph, this system employs a single tag for validating every executed indirect branch. KCoFI was built on top of a secure execution layer called *Secure Virtual Architecture* (SVA) [42] and presents prohibitive overheads that range from 2x to 3.5x while running microbenchmark operations. KCoFI's biggest contribution is on being the first CFI system for OSes but, for using a coarse-grained policy, this approach has been proved flawed [18, 45, 67].

The first fine-grained CFI mechanism for OSes was proposed by Ge *et al.* [65], with support to FreeBSD and MINIX. Similarly to HyperSafe [226], this implementation also employs restricted pointer indexing for achieving CFI. The enforced CFG is built through a pointer analysis which is dependent on code constraints and restricts language features

such as arithmetic on pointer values. Also, to achieve performance, the implementation depends on finding indirect calls with a single possible target and transforming them into direct branches. Although an interesting solution to decrease the introduced overhead, depending on these transformations is a problem since complex code bases may limit optimization opportunities. This approach inherently breaks the support for LKMs due to the fixed jump tables employed in the restricted pointer indexing implementation.

Finally it is important to notice that both address space isolation and CFI mechanisms are orthogonal and complimentary means for OS protection. While address space isolation precludes *ret2usr* attacks, it forbids a user from manipulating user space memory in a way to create malicious targets that are valid under the CFI policy enforcement. Even though, through launching ROP attacks confined in kernel space it is possible for an attacker to flip the proper bits in a way to disable memory isolation features and allow attack continuity through user memory space [122]. In this sense, CFI is an important security component as it prevents the execution of such attacks in kernel memory, avoiding any sort of malicious arbitrary execution through kernel code reuse and ensuring the proper operation of other security features.

Chapter 3

kCFI: Design and Implementation

Never send a human to do a machine's job

Agent Smith

As a step towards practical and effective kernel-level CFI for commodity operating systems, in this chapter we present the concept, design, and implementation of kCFI, a fine-grained CFI protection for the Linux kernel. The proposed approach combines the benefits of a tag-based, fine-grained CFI policy enforcement for both forward and backward-edges—the first of its kind to fully support the Linux kernel—which offers increased protection compared to coarse-grained CFI, with full support of dynamically loadable kernel modules, a crucial feature for supporting the Linux kernel that is missing from previous kernel-level CFI solutions. kCFI is a purely compiler-based approach, and its CFI enforcement mechanism does not rely on any runtime supervisor module or routine, avoiding any associated overheads.

3.1 Overview

kCFI extracts the kernel's CFG by statically analyzing its code at both the source and binary level. This approach captures a detailed view of the CFG that implicitly deals with challenges such as aliasing or divergence between final machine code and its high-level source code representation due to compiler optimizations and hand-written assembly code. Instead of permitting control flow transfers to any function or call site, the enforced fine-grained policy is based on confining indirect edges to paths that may lead only to functions of the same prototype. Only functions with a prototype that matches the indirect call pointer type are considered valid targets, and this is applied to both forward and backward-edges.

kCFI introduces *Call Graph Detaching* (CGD), a novel technique that detaches the direct call graph from the indirect call graph in the protected code to reduce the over-approximation of the permitted control flow transfers even further. CGD prevents the issue of transitively extending the set of return targets of direct function invocations to the sets of valid return targets of indirectly invoked functions, thus making the enforced CFG more restrictive and less prone to Control-Flow Bending [17] and other similar CFI bypass attacks.

Performance overhead is a crucial factor that affects the practical applicability of exploit mitigation techniques, especially for the kernel setting. To minimize the runtime overhead introduced by the extra control-flow checks, kCFI leverages architectural traits, such as cache locality and no-operation instructions, to achieve better performance. This is demonstrated by the results of our experimental evaluation, discussed in Chapter 4.

3.1.1 Threat Model

Adversarial Capabilities. We assume attackers with the ability to execute (or control the execution of) user programs on the OS, seeking to elevate privilege by (ab)using memory corruption vulnerabilities in kernel code [28, 29]. Our model allows overwriting kernel code pointers (function pointers, return addresses, dispatch tables) with *arbitrary* values [54, 169], typically through the interaction with the OS via buggy interfaces—*e.g.,* generic pseudo-filesystems (`procfs` [90], `debugfs` [33]), virtual device files (`devfs` [96]), the system call layer. Code pointers may be hijacked directly [54] or controlled indirectly (*e.g.,* by first corrupting a pointer to a data structure that contains control data and subsequently tampering with its contents [55], similarly to `vtable` pointer hijacking [64, 72,78,141,177,234]). Lastly, attackers can control *any* number of code pointers and trigger the kernel to dereference them on demand; note that *memory disclosure* bugs [30, 31] are extraneous to our proposed scheme(s). Our adversarial model is realistic and consistent with prior work in the field [41, 65].

Hardening Assumptions. We assume an OS that fully implements the W^X policy [98, 102, 180] preventing *direct* code injection in kernel space. In addition, we surmise an OS kernel hardened against *ret2usr* attacks; in modern platforms, we presume the existence of SMEP (Intel CPUs) [232], while for legacy systems we assume protection by kGuard [89] or KERNEXEC (PaX) [134,168]. Note that the kernel may also support KASLR [51], stack-smashing protection [181], pointer (symbol) hiding [154], SMAP/PAN/UDEREF [34,113, 132,133], or any other hardening feature. kCFI does not require or preclude such features, as they are all *orthogonal* to the proposed scheme(s) and can only increase the security of the kernel.

3.2 Design

Under our threat model the control flow of the kernel can be freely hijacked: *any* code pointer can (potentially) be *controlled* by the attacker. Our hardening assumptions, though, guarantee that kernel execution can no longer be redirected to code injected in kernel space or hosted in user space—W^X hinders direct code-injection in kernel space[1], whereas the deployed *ret2usr* protection(s) will prevent any attempt to execute user code in kernel mode. Hence, we anticipate that attackers will be composing their shellcode by stitching together gadgets from the executable (`.text`) sections of the kernel in a ROP/JOP fashion [19, 77, 164], or utilize state-of-the-art code reuse techniques [18, 45, 46, 67].

[1]Researchers recently uncovered that many OS kernels do not properly enforce the W^X policy in kernel space [87]; major OS vendors have since taken steps to eradicate the problem [98].

The main concept behind CFI consists in computing the CFG of a given program and confining all indirect control transfers to its edges. While CFI policy enforcement can effectively prevent control-flow hijacking attacks, its employment on kernel code demands specific challenges to be addressed. First, performance overheads must be minimal, preventing the use of intermediary layers between the kernel and the hardware, and requiring the use of lightweight approaches such as code instrumentation for policy enforcement. Second, the enforcement must be compatible with kernel intricacies such as self-modifying code and LKMs. Third, control transfers caused by events such as interrupts or exceptions may remain valid, even though their occurrence is not predictable in the CFG.

The scheme adopted by kCFI was designed to be compliant with the above challenges. By employing tag-based assertions, it supports self-modifying code and LKMs, as long as these portions of code are compiled in a compatible way. kCFI is fully enabled through compiler instrumentation, not requiring any supervisor module, virtualization support, or dynamic translation techniques. Although inspired by the original CFI proposal by Abadi *et al.* [4], kCFI differs from it as its instrumentation primitives were designed to take advantage of x86-64 architectural traits, generating close to zero memory contention. In fact, without harming any feature on the original system, kCFI enforces CFI with average overheads of 8%, while similar systems [41] incur costs that exceed 100%, or are only comparable after optimized with code transformations that break the above-mentioned requirements [65].

While kCFI enforces its policy on control flow transfers that reside in kernel address space, it relies on other well-known and widely adopted solutions to isolate memory address spaces [89, 134, 168, 232]. This complementary approach ensures that all control transfers from kernel to user space happen through clearly defined exit points that will drop system execution privileges. Besides simplifying the protection without leaving open windows for *ret2usr* attacks, the scheme is compatible with all kernel control transfers, including those that are unpredictable, like interrupt handlers.

In a coarse-grained CFI system, every branch target is valid for all branches, without any distinction. Fine-grained CFI schemes offer stronger protection, as they reduce the number of valid targets for each branch by applying rules to build these sets in a more restrictive way.

kCFI implements fine-grained CFI by ensuring that all returns target instructions following a `call` to the returning function while indirect calls target functions meant to be reached through that invocation. As complete and sound pointer analysis is impossible [152], kCFI over-approximates the CFG by considering valid targets for an indirect call all those functions that have a matching prototype with the pointer used in the indirect `call`. In the code base we used for experimentation, the most common function prototype was `void()`. When enforcing CFI on indirect calls whose pointers had this prototype, the approach reduced the set of valid targets to approximately 0.3% of the set that would have been allowed by a coarse-grained CFI policy.

(a) Return site(s) instrumentation (tag).

```
1 ...
2 callq   0xffffff810001eb <func>
3 nopl    0x138395f
4 ...
```

(b) Epilogue(s) instrumentation (guard).

```
1 ...
2 mov     (%rsp),%rdx
3 cmpl    $0x138395f,0x4(%rdx)
4 je      <8>
5 push    %rdx
6 callq   <kcfi_vhndl>
7 pop     %rdx
8 retq
```

Figure 3.1: Examples of a kCFI return guard and tag pair.

3.2.1 Code Instrumentation

kCFI protects the OS kernel from code reuse attacks by ensuring that computed control transfers adhere to the CFG of the kernel using a label-based control-flow enforcement approach [4, 41]. To this end, kCFI instruments *indirect* branch instructions (*e.g.,* `callq` and `retq` in x86-64) with control-flow assertions, in a manner similar to kGuard [89]; such control-flow assertions verify (at runtime) the target of the respective branch instructions, and authorize the control transfer(s) only if prescribed by the CFG. We refer to the code sequences used for implementing the control-flow assertions as *guards*, and to the (inlined) code labels that are checked by the guards, to validate branch targets, as *tags*.

Figure 3.1 illustrates a return guard and tag pair for the x86-64 architecture. In Figure 3.1(a), the routine `func` is invoked by a direct `callq` instruction (line 2), which is followed by a *return tag*, implemented as a `nopl` instruction that encodes `func`'s return ID (`0x138395f`; line 3). Representing the tag as a `NOP` instruction is important, as it *transparently* marks the return site(s) of `func` (*i.e.,* without affecting the semantics of the code). Figure 3.1(b) shows the corresponding *return guard* that confines a `retq` instruction of `func`. This snippet loads the intended return address from the stack into the `%rdx` register (line 2), dereferences it, and compares the result with the expected ID (line 3); the 4-byte offset in the dereference skips the `nopl` opcode, as only the encoded value (`0x138395f`) must be compared. If the two IDs match, the control jumps to the `retq` instruction and the branch is taken (lines 4 and 8); else, the phony branch address is pushed onto the stack, and a violation handler (`kcfi_vhndl`) is invoked (lines 5–7).

Along the same line, Figure 3.2 depicts an entry-point guard and tag pair for the x86-64 architecture. The prologues of routines that can be indirectly invoked are marked with an *entry-point tag*, similarly to return sites; Figure 3.2(a) shows the entry-point tag of routine `func`, also implemented as a `nopl` instruction that (transparently) encodes the routine's entry-point ID (`0xbcbee9`). Figure 3.2(b) illustrates the corresponding *entry-point guard* that confines an indirect `callq` instruction to `func`. Assuming that the address of `func` is loaded in register `%rax`, this snippet dereferences `0x4(%rax)` and compares the result with the expected ID (`0xbcbee9`; line 2). Again, if the two IDs match, the control jumps to

(a) Prologue(s) instrumentation (tag).

```
1 ...
2 <func>:
3 nopl    0xbcbee9
4 ...
```

(b) Indirect call site(s) instrumentation (guard).

```
1 ...
2 cmpl    $0xbcbee9,0x4(%rax)
3 je      <7>
4 push    %rax
5 callq   <kcfi_vhndl>
6 pop     %rax
7 callq   *%rax
8 nopl    0x138395f
9 ...
```

Figure 3.2: Example of a kCFI entry-point guard and tag pair.

the `callq` instruction and the indirect invocation of `func` takes place (lines 3 and 7); else, the bogus branch address is pushed onto the stack and `kcfi_vhndl` (violation handler) is invoked (lines 4–6). Note that `callq *%rax` is followed by a return guard (*i.e.,* `func`'s return guard; return site, line 8).

The end result of the above (control-flow) confinement scheme is the following: (a) `retq` instructions can only transfer control to the return site(s) of the routine they belong to (*e.g.,* the `retq` instruction of `func` can only transfer control to the return sites of `func`; Figure 3.1); and (b) indirect `callq` instructions, which correspond to function pointers, are paired with the beginning of the routines that can be indeed invoked through the respective function pointer (*e.g.,* the `callq` instruction of Figure 3.2(b) can only transfer control to the prologue of `func` or functions with a similar prototype).

kCFI's guards are designed so that every confined branch instruction is paired with a call to the violation handler. By using this approach, instead of having a single (global) call to `kcfi_vhndl` (to which every guard transfers control upon an assertion failure), it is possible to precisely trace the violations by combining the pushed parameter and the violation handler's own return address. We found that this configuration increases the overall overhead by ∼2%, making its benefits more appealing than its costs.

Performance Requirements. The proposed tag-based scheme employed by kCFI conforms to the hard performance requirements imposed by OS kernels. Specifically, in the x86-64 architecture, kCFI is significantly more efficient than previous approaches, as demonstrated by the results presented in Chapter 4. Given the multi-level and inclusive nature of the cache hierarchy of Intel CPUs, the proposed guards do not generate compulsory cache misses; if a cache miss happens while dereferencing the branch target for validation, the only consequence is the anticipation of a load that would otherwise occur while branching.

kCFI also implements tags in a more efficient way than previous tag-based schemes. The original CFI proposal by Abadi *et al.* [4] uses `prefetch` instructions to mark valid branch targets (*e.g.,* encode return IDs). Although such instructions do not change the

semantics of a program, they do leave a footprint on cache organization and affect memory contention [233]. By employing `nopl` instructions, kCFI is capable of marking valid targets with close to zero side-effects (see Chapter 4). To the best of our knowledge, kCFI is the most efficient tag-based CFI scheme for the kernel setting.

Compatibility Requirements. Many of the previously-proposed CFI schemes for low-level software, like OS kernels, implement CFG validation techniques that rely on converting indirect branches into jump tables based on restricted pointer indexing [65, 226]. The kernel-level CFI scheme proposed by Ge *et al.* [65] uses this approach, and, to achieve good performance, has to rely on code optimizations (*e.g.,* function pointer constification) that heavily depend on the intricacies of certain code bases (*i.e.,* FreeBSD and MINIX). Restricted pointer indexing does not support dynamically loadable modules, while the extra indirection introduces additional overheads. In antithesis, kCFI is capable of protecting OS kernels despite any optimization opportunities, achieving better performance than the aforementioned implementation and not restricting any OS features.

3.2.2 Fine-grained CFI Policies

To enforce its fine-grained policy, kCFI relies on a CFG used as a reference for valid branches on the protected program. This CFG is built using two types of analysis. First, source code analysis provides all high-level information regarding functions, function pointers, and prototypes. Second, a binary analysis performed on a compiled version of the program allows fitting the CFG accordingly to back-end optimizations and transformations that occur during linking. Building the CFG through combining both analyses provides a precise and reliable CFG, as required for the development of a fine-grained CFI.

For every function represented on the CFG, kCFI creates an exclusive return tag. For every unique prototype respective to an indirect invocation pointer, kCFI creates one return tag and one entry-point tag. Whenever marking the code with return tags, kCFI gives precedence to those respective to the indirect invocation prototype. Entry-point tags are used to mark functions as valid targets for indirect calls. This approach guarantees a fit restrictiveness while ensuring that a function return may be allowed to all its valid return targets, irrespectively of whether it was invoked directly or indirectly. During instrumentation, kCFI considers indirectly invocable all those functions whose prototype has a corresponding function pointer prototype.

kCFI parses all `call` instructions inside each function while placing tags. If the `call` is indirect, then the return tag respective to the indirect invocation pointer prototype is placed right after it. If the `call` is direct, then kCFI first verifies if the function being called is also indirectly invocable. If it is, then the return tag placed is the one correspondent to the indirect invocation pointer prototype. Otherwise, the return tag for the function is used. kCFI also checks if each parsed function is indirectly invocable. If it is, then the entry-tag that corresponds to its prototype is placed in the function's prologue.

Next, kCFI places guards in each function. The tags checked by return guards are picked through a logic similar to the one described above: whenever a return guard for

an indirectly invocable function is generated, the tag used is the one corresponding to an indirect invocation pointer's prototype that matches the function's prototype. If the function is not indirectly invocable, the tag used is the one respective to the function. For generating entry-point guards, kCFI checks the prototype of the indirect invocation pointer used in the `call` and picks the entry-tag relative to it. This scheme ensures the proper bonding of all indirect branches and their relative targets.

As proposed by Abadi *et al.* [4], CFI can be enhanced by the use of a shadow stack to refine returns. It is known that such structures have high performance costs [43], which can be prohibitive to the kernel context. For this reason, the proposed kCFI design does not include a prompt shadow stack. Instead, In Chapter 5, we propose a kernel shadow stack as a security enhancement which is orthogonal to kCFI.

Call Graph Detaching

If a function is both directly and indirectly invocable, its return guards will be generated with the return tag respective to its prototype, irrespectively of whether it was directly or indirectly invoked. This creates a situation where transitively all instructions after a direct call to a function become valid return points to other functions with a similar prototype. This problem stretches the CFI policy and makes it more prone to bending attacks [17].

This problem is illustrated in Figure 3.3. The code snippet in Figure 3.3(a) invokes `foo()` both directly and indirectly. The code snippet in Figure 3.3(b) presents the return guard for the function `foo()` which checks for the tag placed after both direct and indirect calls to `foo()`. This not only allows `foo()` to return to both call sites, irrespectively of how it was invoked, but it may also allow any *different* function with the same prototype as `foo()`, as exemplified by function `bar()` in Figure 3.3(b), to return to the call site of the direct invocation of `foo()`, in a clear violation of the valid control flow.

To mitigate this problem, kCFI follows a novel approach by cloning functions instead of merging all valid return targets. In this way, a function named `foo()` is cloned into a new function called `foo_direct()`, which has the same semantics but checks for a different tag before returning. All direct calls to `foo()` are then replaced by calls to `foo_direct()`, and the tag placed after the call site is the one that corresponds to `foo_direct()`. We call this optimization *Call Graph Detaching* (CGD), as it detaches the kernel's direct call graph from the indirect call graph, preventing the need for merging and the consequent over-approximation caused by tag transitiveness. When applied to the code of Figure 3.3, CGD results in the code presented in Figure 3.4. The optimized version brings the replaced `call` instruction and the respective tag (3.4(a)) and both original `foo()` and cloned `foo_direct()` functions (3.4(b,c)).

Call graph detaching is applied only on functions that can be invoked both directly and indirectly. For that, the algorithm checks the existence of pointers with a matching prototype and direct invocations before cloning a function. As clones are used to replace the targets of direct calls, pointer operations that may end up targeting the function are not harmed by the scheme. An analysis of how CGD refines the granularity of the applied CFI policy is provided in Chapter 4.

(a) function *foo()* invocation

```
...
callq   0xffffff810001eb <foo>
nopl    0x1383ddd
...
movl    rcx, 0xffffff810001eb <foo>
callq   *rcx
nopl    0x1383ddd
...
```

(b) return guard on *foo()*

```
...
mov     (%rsp),%rdx
cmpl    $0x1383ddd,0x4(%rdx)
je      <8>
push    %rdx
callq   <kcfi_vhndl>
pop     %rdx
retq
```

(c) return guard on *bar()*

```
...
mov     (%rsp),%rdx
cmpl    $0x1383ddd,0x4(%rdx)
je      <8>
push    %rdx
callq   <kcfi_vhndl>
pop     %rdx
retq
```

Figure 3.3: Example of valid return targets merging.

3.2.3 CFI Map

To create its fine-grained CFI policy, kCFI uses a data structure called *CFI Map*, which is an augmented CFG built using source code and binary analysis. Besides function invocation relationships, this structure also holds function prototype information that enables mapping which functions may be called indirectly, and symbol information that permits correctly mapping functions hidden behind aliases.

To construct a complete CFI Map, the respective binary to be protected needs to be analyzed, and this is done through an early compilation. The construction process compiles the entire kernel while performing source code analysis. This compilation also instruments the generated code with identification marks that enable disambiguation of functions when their names collide in the final binary. When the kernel binary is ready, it is analyzed to fill any information gaps left during the source code analysis phase.

3.3 Implementation

kCFI was implemented in the form of a compiler infrastructure composed of a set of complementary tools that perform code analysis and CFI instrumentation. The whole set can be classified into four different groups of tools: (i) source code analysis, (ii) binary

(a) function *foo()* invocation

```
1 ...
2 callq   0xffffff8100033a <foo_direct>
3 nopl    0xaff883
4 ...
5 movl    rcx, 0xffffff810001eb <foo>
6 callq   *rcx
7 nopl    0x1383ddd
8 ...
```

(b) return guard on *foo()*

```
1 ...
2 mov     (%rsp),%rdx
3 cmpl    $0x1383ddd,0x4(%rdx)
4 je      <8>
5 push    %rdx
6 callq   <kcfi_vhndl>
7 pop     %rdx
8 retq
```

(c) return guard on *foo_direct()*

```
1 ...
2 mov     (%rsp),%rdx
3 cmpl    $0xaff883,0x4(%rdx)
4 je      <8>
5 push    %rdx
6 callq   <kcfi_vhndl>
7 pop     %rdx
8 retq
```

Figure 3.4: Example of call graph detaching (CGD).

analysis and CFI Map construction, (iii) Assembly patchers, and (iv) CFI instrumentation. From these groups, (i) and (iv) were implemented as LLVM compilation passes, and each implies a full compilation of the source code, while (ii) and (iii) are a group of tools written in C++ and Lua [148]. kCFI can be understood as a pipeline in which each group of tools is a stage that follows the above order. A detailed illustration of the different stages is presented in Figure 3.6, and is described in detail in the following. Overall, the goal of stages (i) and (ii) is to build a CFI Map for the whole kernel code, while (iii) and (iv) use the resulting CFI Map to instrument the kernel code with tags and guards.

Assembly Language Support

One of the drawbacks of using LLVM-based instrumentation is that assembly sources are not touched, as this kind of code is directly translated into binaries without having an IR form. The kernel has a significant part of its code written in assembly, which includes many indirect branches. While applying CFI, if such code is left unprocessed, two major problems arise: (i) indirect branches in assembly sources are left unprotected, and (ii) tags are not placed, breaking compatibility with C functions returning to assembly, or with assembly functions being called indirectly from C code. kCFI tackles this problem through the automatic rewriting of the assembly sources assisted by information extracted

during code and binary analysis, as explained in the following subsections.

CFI Map Data Structure

The CFI Map describes the kernel's CFG using four entities: (i) Nodes, which represent single functions; (ii) Clusters, which represent sets of functions with the same prototype; (iii) Edges, which represent a branch from a node to another node or to a cluster; and (iv) Aliases, which map symbols that may have different names but are equivalent in the final binary.

Nodes. These entities represent functions and hold as attributes their identifier, function name, prototype, and module. Each node also contains a tag which will be later used to validate allowed return points to the function.

Clusters. These entities represent a group of functions that have the same prototype. Each cluster has its identifier and information on the prototype it represents. Clusters also hold two tags, one used to validate allowed return points for the functions in the cluster, and another to mark the entry-points of these functions as valid targets for indirect calls.

Edges. Edges represent the invocation relationship between nodes and clusters. An edge has an identifier for itself and holds the identifiers for the node that corresponds to the edge's origin and for the node or cluster of the edge's target. Edges also have a type attribute that defines them as a direct or indirect call.

Aliases. Aliases represent symbols that can hide another symbol during compilation time.

Figure 3.5 shows an example of a CFI Map construction. In this figure, the three functions in the source code (a) are represented in the graph (b) as nodes (circles). The gray rectangle in the graph represents a cluster, which includes the nodes `i32 A(i32)` and `i32 B(i32)`. The solid edge represents the direct call to the function `void C(i32)` (See line 9 of Figure 3.5(a)), and the dashed one represents the indirect call to functions inside the cluster, which has the prototype `i32 (i32)` (See line 10 of Figure 3.5(a)).

By observing Figure 3.5 it is possible to infer important information for CFG enforcement: (i) there is a direct call from node `B` to `C`, implying a return from `C` to `B` that must be allowed; (ii) there is an indirect call from `B` to the cluster, so the returns of all functions in the cluster (`A` and `B`) to `B` must be allowed; and (iii) the indirect call from `B` must also be allowed to all functions in the cluster. Figure 3.5(c) shows the corresponding CFI Map file data structure.

kCFI Pipeline Overview

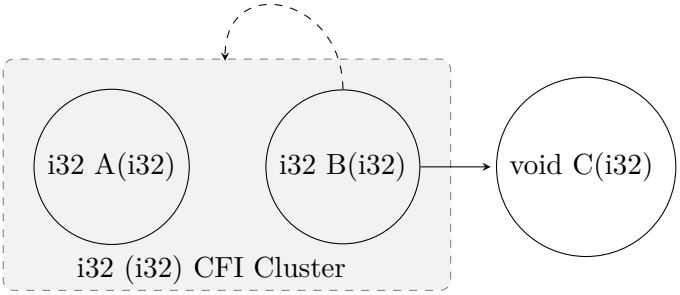
Figure 3.6 shows a diagram that describes how the different stages of kCFI are connected. In this diagram, squares represent files used or generated by the pipeline stages, dashed arrow shapes represent a full compilation through LLVM, dashed circles represent an offline step, and solid arrow edges describe which files are used or generated by each step.

Initially (1), the kernel source code is compiled with LLVM, generating a *vmlinux* file and multiple CFI Maps, one for each compiled module. The compiler also performs two tasks: instruments *vmlinux* with a unique identifier on the first instruction of every C function, and stores a catalog of all function declarations seen during compilation. All

(a) Example source code.

```
1  #pragma weak A = A_Alias
2
3  int A(int x){
4     return x*x;
5  }
6  int B(int y){
7     int(*f)(int);
8     f = &A;
9     C(30);
10    return 7 * f(y);
11 }
12 void C(int z){
13    while(1){ };
14 }
15 int A_Alias(int x){
16 }
```

(b) Resulting CFI Map.



(c) Resulting CFI Map data structure.

Nodes				
Identifier	Name	Prototype	Module	Return tag
290f2fd5	A	i32 (i32)	ex.c	1dc2aaf0
7d63f629	B	i32 (i32)	ex.c	6e28b9d1
6ba8458b	C	void (i32)	ex.c	164e44a8

Clusters			
Identifier	Prototype	Entry-point tag	Return tag
6a8597ea	i32 (i32)	69e1b040	46068a5c

Edges			
Identifier	From	To	Type
7dcdc019	7d63f629	6a8597ea	indirect
7728cc01	7d63f629	6ba8458b	direct

Aliases	
Identifier	Alias
290f2fd5	A_alias

Figure 3.5: Example of CFI Map construction.

CFI Maps are merged (2) into one single CFI Map, which is used together with the kernel binary to uncover all assembly functions (3). These are the ones not instrumented with a unique identifier during the previous compilation. The outcome of this stage (4) is then analyzed by a tool that retrieves all direct `call` targets and, by checking their unique identifiers, maps every (5) direct edge. Finally, some fixes to the CFI Map are applied (6) to support certain kernel corner cases, such as *syscalls* functions which are called through a pointer that does not hold a matching prototype.

Once the CFI Map is complete, it is first used as a reference to patch the assembly files. By combining its information with the catalog of declarations (7) built during the first compilation (1), the assembly files present in the original kernel source are rewritten with proper tags and guards. The patched source (8) is then ready to be compiled by LLVM, a process that instruments C functions using information from the CFI Map and generates the final protected *vmlinux* binary.

3.3.1 Source Code Analysis

Source code analysis is the first stage in kCFI's pipeline and it is implemented as a compilation pass in LLVM. It is represented in Figure 3.6 by the arrow shape (1). The

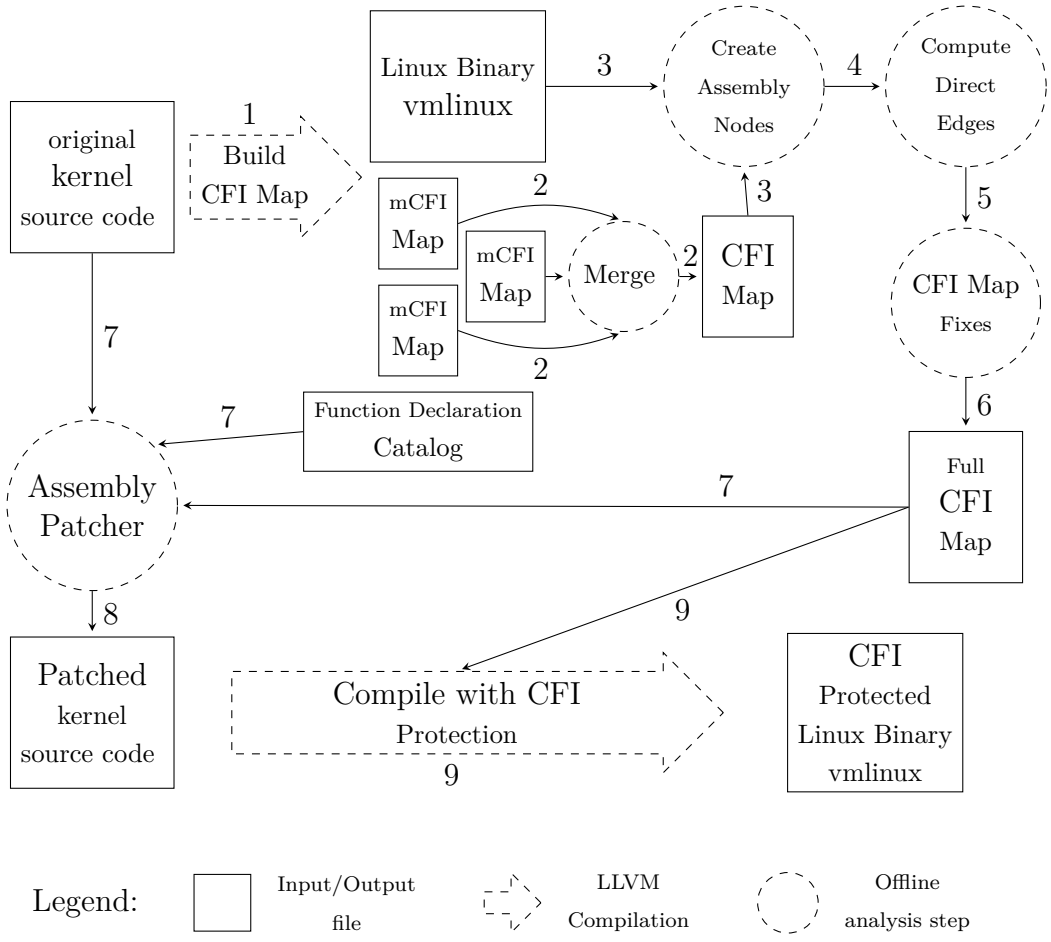


Figure 3.6: kCFI Pipeline

goal of this stage is to begin the construction of the CFI Map by adding source-level information retrieved from all functions in each compiled module. The kernel binary that results from this compilation is used by the following binary analysis stage.

During this process, each compiled function is stashed in the CFI Map as a node. From each function, all indirect calls are parsed and used to create indirect edge entries. Each edge's origin is the node that represents the function being parsed and its target is a cluster that represents the functions with the same prototype as the pointer. A new cluster is created if such a cluster does not already exist in the CFI Map. This process also stores all aliases and the respective symbols overridden by them. This is important to allow further CFI instrumentation to resolve symbols and match a branch's target precisely, as it appears in the final binary.

Since every direct call also represents an indirect branch, i.e., the return from the callee towards the caller, it is also important to map those edges. This pipeline stage is not appropriate for performing this mapping as it may result in incomplete edges, because (i) by the time the module is analyzed, some callee functions may have not yet been compiled, and thus their nodes will not exist in the CFI Map; and (ii) as some symbols have weak linking properties[2], if this is the case for the callee in the analyzed edge, it cannot be correctly identified prior to the linking process.

To ensure that direct edges are precisely mapped during binary analysis, the compilation process in this stage also instruments the prologue of each function with its respective node identifier, encoded similarly as done with tags, as described in Section 3.2. This instrumentation does not concern CFI enforcement, but allows the binary analyzer to directly correlate functions in the binary with node entries in the CFI Map, independently of aliases, optimizations, and linking properties.

By the end of the source code analysis, the data structure being built already holds node entries for all functions written in C, edge entries for all indirect calls present in C functions, cluster entries for all prototypes used to declare function pointers, and a map of all aliases with their respective masked symbols. This stage also generates a fully compiled kernel binary in which every C function is instrumented with a unique identifier.

3.3.2 Binary Analysis and CFI Map Fixes

The binary analysis phase complements the constructed CFI Map with information extracted from the compiled kernel. This stage comprises three main steps, which are represented in Figure 3.6 by the edges *4*, *5* and *6*.

The first step performed during binary analysis is a search for functions which were not instrumented with an identifier in the previous stage. A function left not instrumented means that it was not touched by LLVM in kCFI's first stage, because it was originally written in assembly, and not in C. Node entries for all these functions are added to the CFI Map, as part of the assembly nodes list. The address of the first instruction in each assembly function is also kept as the function's identifier.

The second step involves parsing all direct call instructions in the binary. By following

[2]Weak is a symbol property used to inform the linker that this symbol must be discarded if it collides with another one during the linking process.

the instruction's target address, it is possible to reach the call's destination, retrieve its identifier, and create a direct edge entry in the CFI Map with both the origin and target fields filled. If the target does not have an identifier, kCFI assumes that it is an assembly function and retrieves the identifier from the list of assembly nodes using the address of its first instruction.

CFI Map Fixes

The third step of the binary analysis stage applies fixes to the CFI Map, making it compliant with various kernel corner cases.

Alternative Calls. The Linux kernel dynamically replaces the targets of certain direct `call` instructions with more efficient implementations of the respective functions, depending on the presence of specific CPU features. The whole set of functions that may potentially be a target for a given `call` must thus be allowed to return to the same points, similarly to how it is performed for clusters; consequently, their tags must be merged. This is achieved by setting a unique tag value in the node entries that represent these functions in the CFI Map.

Syscalls. The functions that integrate the set of *syscalls* have different prototypes. Besides, *syscalls* can be invoked through both regular direct calls or indirectly, through the *syscall dispatcher*). As only one tag can be placed after the *syscall dispatcher*, fixing its verification requires merging all clusters that hold a prototype that matches a *syscall*'s prototype. This solution causes a broad loosening of the call graph, as it results in a broad merging of different clusters, and also creates clusters for functions which are never indirectly called from places other than the *syscall dispatcher*.

Instead of merging these clusters, kCFI handles *syscalls* in a special way. First, the *syscall table* file is parsed, and a catalog of *syscalls* is built. A generic tag for the *syscall dispatcher* is also created. Second, while compiling the kernel's source code, *syscalls* are compiled with a different kind of guard that allows returning upon validation of one among two different tags. The first tag that is checked is the one that corresponds to the function (either its node or cluster return tag, and the second is the generic tag created for the *syscall dispatcher*. An example of such a "specialized guard" with secondary tag support is shown in Figure 3.7.

```
1 mov   (%rsp),%rdx
2 cmpl   $0x138395,0x4(%rdx)
3 je     9
4 cmpl   $0x11deadca,0x4(%rdx)
5 je     9
6 push   %rdx
7 callq  <kcfi_vhndl>
8 pop    %rdx
9 retq
```

Figure 3.7: An example of a guard with secondary tag support.

By the end of this stage, the CFI Map holds node entries for all functions in the binary, with prototype information for those written in C; cluster entries for all prototypes used to

declare function pointers; edge entries for all the indirect calls in C code; and edge entries for all direct calls present in the final binary, with no ambiguity due to weak linking or aliasing.

3.3.3 Assembly Code Patching

Once the CFI Map is complete, an *Assembly Patcher* rewrites the assembly files that exist in the kernel source tree. Before changing the code, the tool needs to retrieve the tags that will be used during code instrumentation in three different circumstances.

Direct calls from Assembly code. Assembly code may call functions that check for a tag before returning. Consequently, calls from assembly code must also be followed by a tag respective to the called function. The Assembly Patcher collects these tags by parsing the target of every `call` instruction in the source code and retrieving its node from the CFI Map.

Indirect calls to Assembly functions. Assembly functions may be called indirectly. Consequently, they need to have a tag in their prologue to allow them as indirect call targets. As no prototype information is available in assembly code, retrieving this tag requires parsing all function names from the source code and searching for them in the declarations catalog created during the source code analysis to identify their prototypes. If a matching declaration with a respective cluster is found, then its cluster entry from the CFI Map is used. The system has been designed to prompt the user in case the search returns more than one match, but such cases were not seen during our evaluation.

Return instructions in Assembly code. Assembly functions also must be protected against control flow hijacking; it is thus important to instrument their indirect branches with guards. For these cases, all `ret` instructions are parsed and the name of the function to which they belong is searched in the declarations catalog. If a match with a respective cluster exists, the cluster is retrieved from the CFI Map. Otherwise, the function's node is retrieved.

After collecting all the tags, the Assembly Patcher is capable of rewriting the source files by correctly placing the tags after every direct `call`. Placing tags in the prologue of assembly functions is achieved by replacing the macro `ENTRY` with a specially crafted macro `ENTRYcfi`. While the first is regularly used in the Linux kernel source code to mark the beginning of functions and appropriately create their symbols, the later was crafted to extend `ENTRY` in taking one extra argument to be placed as a tag when the macro is expanded. Hence, the Assembly Patcher rewrites the code replacing `ENTRY` for `ENTRYcfi` macros with the corresponding tag, whenever it is required. Assembly return guards are placed in a similar way. All functions are rewritten with the instructions of the corresponding guard preceding their `ret` instructions.

Some assembly functions are generated through the expansion of macros. In these cases, there exist `call` instructions whose targets are passed as a macro parameter, making it impossible to add the tag directly in the macro source, as previously done. For such cases, kCFI has a crafted macro which is a variation of the original, but with one extra parameter corresponding to the tag to be placed. The process of rewriting consists of parsing the original macro invocation, retrieving the symbol which will be expanded as

a `call` target, searching the CFI Map for its corresponding tag, and then replacing the original macro in the source code with the crafted macro, also adding the tag among its parameters. Placing guards on macro generated functions again follows the same logic.

Assembly code inlined into C functions is not general enough to be tackled in an entirely automated way, and is handled by directly patching the source code. For these cases, we prepared a set of patches that already have placed tags and guards accordingly, but with a generic value. Before applying the patches, kCFI searches the CFI Map for the particular tags of each case and uses them to replace the general value. Finally, the file that holds the *syscall dispatcher* is also rewritten with its tag correctly placed.

As assembly code does not provide information about pointers' function prototypes, the methods proposed so far are insufficient to automate the generation of indirect call guards in such files. For the code base we used during our tests, kCFI protected 139 indirect branches in assembly modules. kCFI missed 6 instructions which involve no hazard, being part of initialization routines only invoked during boot time, 5 indirect calls that are inherently protected by being implemented in the form of verified target tables, and 5 indirect calls whose pointer prototypes cannot be inferred statically, but were feasible to be patched and moved to read-only data sections. Instructions belonging to code which is compiled along with the kernel, but which are not linked into the final binary, such as user space vDSO[3], were intentionally skipped as they cannot be hazardous.

3.3.4 CFI Instrumentation

The last stage of the kCFI pipeline compiles the kernel code with CFI protection. As assembly code was protected in the previous stage, C code is now instrumented with tags and guards through an LLVM back-end pass. For this process, kCFI relies on the CFI Map built during the previous steps. As no information is written to the CFI Map, this stage can be run concurrently with no harm, allowing parallel compilation. Applying or not the CGD optimization is a configuration option through the use of a special compilation flag.

To enforce CFI, kCFI parses all instructions in the code being compiled while they are still in LLVM Machine IR form. This abstraction is much closer to the final binary than high-level languages, allowing instrumentation to interpose instructions more precisely. Also, still being an LLVM representation, this allows the use of the compiler's API to retrieve high-level information, such as prototypes of pointers used in indirect calls, which are crucial for CFI enforcement.

The default violation handler function, which is invoked when a CFI assertion fails, was implemented to display meaningful debug information and then halt the system. Different violation handlers can be implemented in custom forms, e.g., for debugging purposes or even to allow fail-safe routines. These functions are written in C, as a regular kernel module, and are easily replaceable.

If CGD is in use, an extra compilation pass is needed before the CFI instrumentation. This pass checks all functions to identify the ones that are callable both directly and

[3]Kernel functions exported to be executed by processes in user space, avoiding the need for context switch

indirectly. This is done by checking the CFI Map for the existence of both a cluster with the matching prototype and at least one direct edge towards the function. If a function meets both requirements, it is cloned into a new function that is set as never indirectly invocable. While defining cloning targets, functions that are declared but not implemented are also considered, ensuring CGD applicability throughout all modules. Before proceeding, calls whose targets were cloned are replaced by calls to the cloned function, ensuring that one function is only callable either directly or indirectly, never both ways. The generated code is then delivered for CFI instrumentation, and this will place tags considering the branch's final target.

During instrumentation, indirect branches are preceded with guards and indirect branch targets are marked as valid with tags, as described in Section 3.2. All values used to generate tags and guards are retrieved from the CFI Map. Whenever consulting it, kCFI verifies the alias entries to avoid ambiguities that could lead to wrong instrumentation. No link-time optimization or binary modification is required after the kCFI pass processes the LLVM IR.

Chapter 4

kCFI: Evaluation

42

Deep Thought

kCFI was evaluated across three different perspectives: performance impact, security protection, and size overhead. To that end, we used as code base the Linux kernel version 3.19.0, with the LLVMLinux [103] patches applied. The whole kernel was compiled with a large number of built-in drivers and functionalities, resulting in a realistic code base for evaluation purposes — the final binary contained 132,859 functions linked in a 705MB (uncompressed) file. Besides the original files, a CFI-specific module that holds the violation handler function was also included in the source set and was linked in the final binary. All tests were performed on a system equipped with a quad-core 3.40GHz Intel(R) Core(TM) i7-6700 processor, 32GB of RAM, and a 500GB SSD hard drive. The Debian GNU/Linux 8 (Jessie) was running on top of the tested kernel.

To verify the performance degradation caused by kCFI's instrumentation, we tested three different versions of the Linux kernel: (i) *Vanilla*, the original kernel compiled with LLVM; (ii) *kCFI*, the kernel compiled with CFI protection; and (iii) *kCFI+CGD*, the kernel compiled with CFI protection using the call graph detaching optimization. Every program unit inside each benchmark used during the performance evaluation was executed 10 times, and the presented results are averages of the observed numbers.

To assess the security benefits introduced by kCFI, we use the *average indirect target reduction* (AIR) metric [110], which measures the program's restrictiveness according to the applied CFI policy in terms of call graph pruning. We also use the *average indirect targets allowed* (AIA) metric [65], which captures the number of permissible targets for every indirect target.

4.1 Performance Evaluation

Micro-benchmarks

The micro-benchmark LMbench [109] was used to verify kCFI's impact on kernel operations. From the whole benchmark, a subset of applications focusing on OS capabilities was selected, allowing the measurement of latencies through the execution of null syscall; I/O

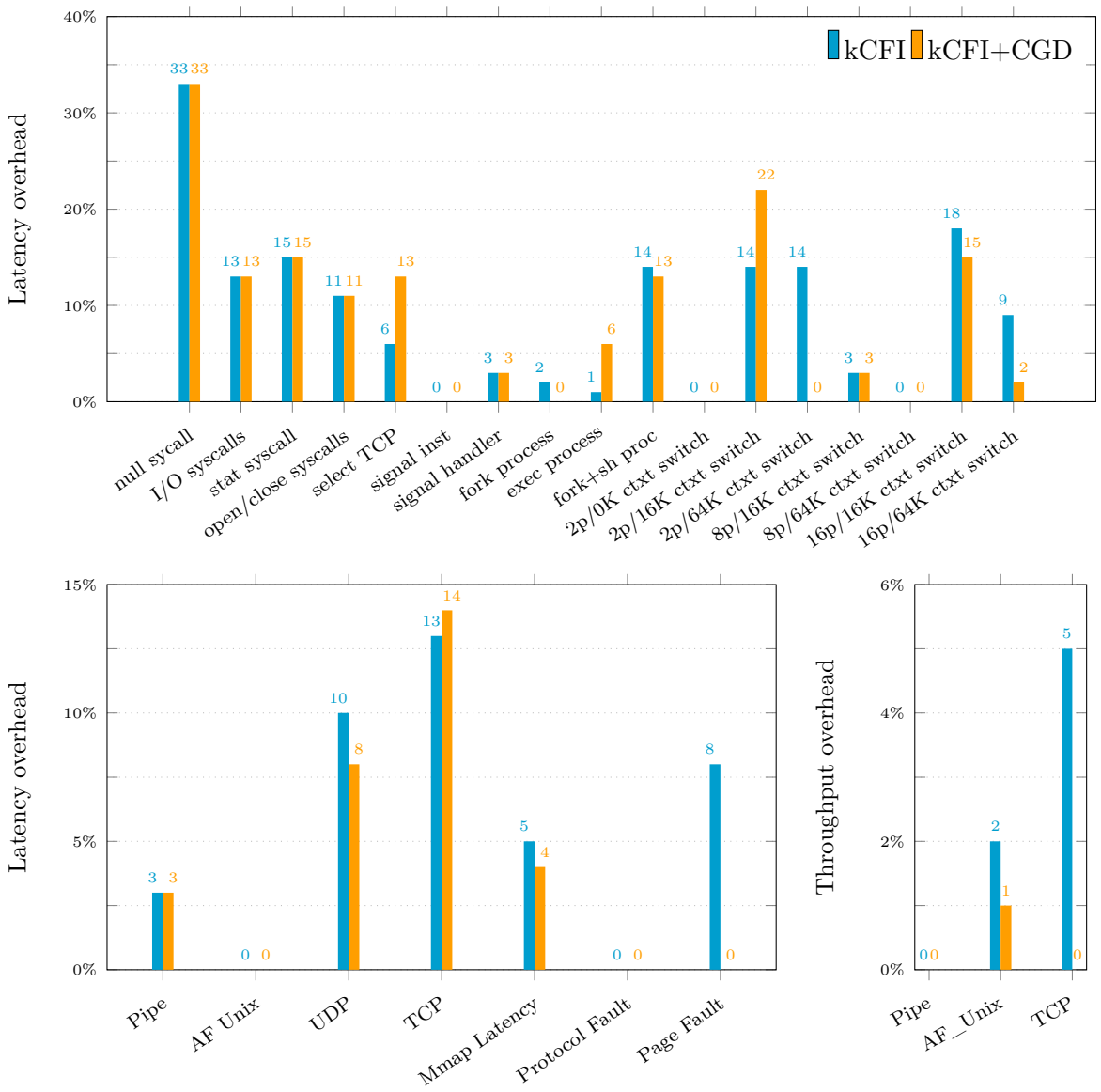


Figure 4.1: Performance overhead of kCFI on LMbench.

critical syscalls (`read/write`, `fstat` and `select`); `open/close` syscalls; signal handler installation, process creation followed by `exit`, `execve` and `/bin/sh`; context switching between processes; select syscall on 100 file descriptors; and page fault handling and inter-process communication with socket and pipe. Communication throughputs through pipes, unix sockets (AF_UNIX), and TCP sockets were also measured.

Figure 4.1 shows the performance overhead of kCFI and kCFI+CGD over Vanilla. In the chart, missing bars represent negligible overheads. The micro-benchmark tests are classified in latency and communication throughput overhead. For latency, kCFI incurs an average overhead of 8%, while kCFI+CGD 7%. The maximum overhead of both configurations reaches 33%[1]. For communication throughput, kCFI incurs an average overhead of 2%, while kCFI+CGD did not exhibit any discernible overhead.

[1]Overhead for the test *null syscall*, which presented the smaller latencies in the whole test-set and, for that, is more sensitive to overhead introduction.

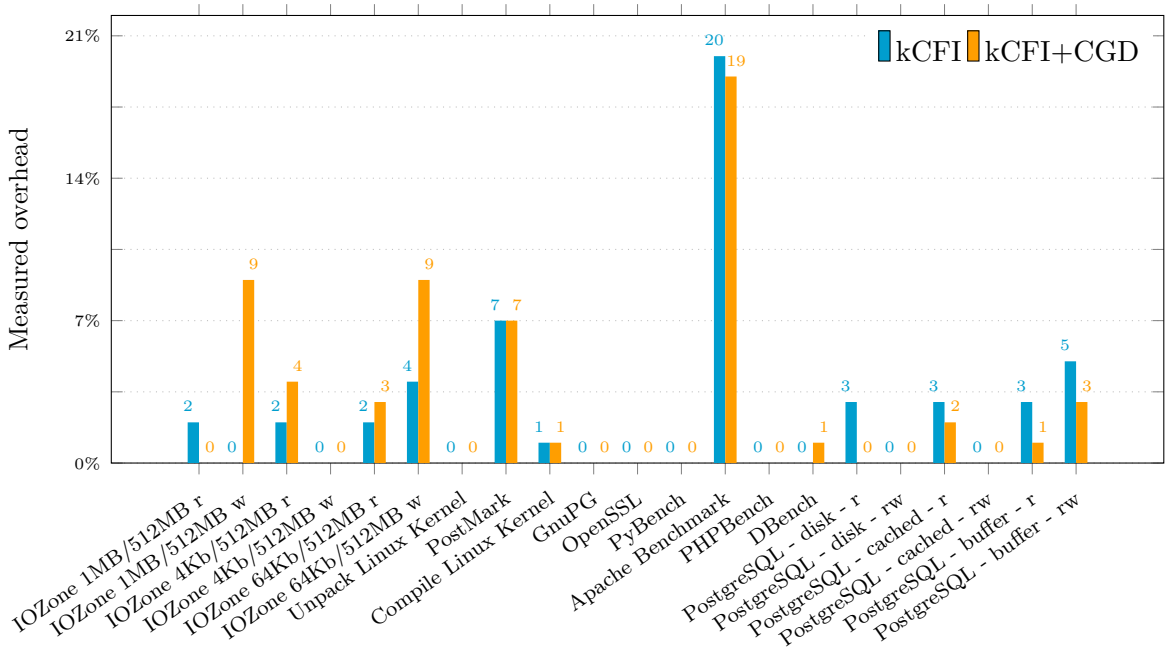


Figure 4.2: Performance overhead of kCFI on Phoronix.

Macro-benchmarks

To measure the effects of kCFI on user space applications running on top of an instrumented kernel, we used tests from the *Phoronix Test Suite* [97], which includes many applications commonly seen on server environments. The set was composed by the benchmarks *IOZone*, running 1MB, 4Kb and 64Kb read/write operations on a 512MB File, *Linux Kernel Unpacking*, *PostMark*, *Timed Linux Kernel Compilation*, *GnuPG* encrypting a 1GB file, *OpenSSL* running 4096-bit RSA, *PyBench*, *Apache Benchmark*, *PHPBench*, *Dbench* and *PostgreSQL* running read-only/read-write operations under heavy contention on disk, cached, and in buffer. Additionally to these tests, we also compared the execution of SPEC [75] benchmark on Vanilla and kCFI.

Figure 4.2 shows the overhead for each test. In this Figure, missing bars represent negligible overheads. The average overhead observed for the whole test suite was 2% for kCFI and 3% for kCFI+CGD, with a maximum of 20% and 19%, respectively, both for the Apache benchmark. Apart from Apache, all other applications exhibited overheads below 10%. To assess the reasons for the outlier, we also run the test *NGINX* (also part of Phoronix), which is another web server application. This test exhibited an average overhead of 20%, confirming that the higher observed values are due to their frequent interactions with kernel capabilities, turning them into applications more sensitive to such instrumentation. In a final test, we modified the Apache benchmark in a way to split client and server in two different computers. This test made it possible to identify which part of Apache benchmark was responsible for generating the large overhead. When the requests were made from a different machine, the overheads in the server became negligible, highlighting that the most significant part of the cost came from creating multiple requests, and not from responding to them.

A performance comparison between the Vanilla kernel and kCFI while running SPEC

benchmark presented an average overhead below 1%. As the applications in this benchmark are very CPU-intensive, they spend most of their execution cycles in user space, rarely stressing protected kernel code. For this reason, we consider the numbers obtained with Phoronix more representative.

4.2 Security Evaluation

The threat model considered assumes that both W^X and *ret2usr* hardening features are in use, leaving an attacker with no opportunities for code injection neither in kernel or user space. Successful exploits are then limited to code-reuse strategies over kernel instructions, commonly executed through ROP-based attacks [142]. These attacks are built upon chaining small instruction sequences terminated with an indirect branch. These instruction sequences are called gadgets, and each performs a small operation. Through chaining many gadgets, attackers achieve Turing-complete computation [15].

The goal of kCFI is to protect the system against control-flow hijacking by preventing code-reuse attacks through limiting valid targets on each indirect control flow transfer. Succesfully deploying a ROP attack on a system protected with kCFI requires its gadgets to be chained compliantly to the enforced fine-grained policies. Such enforcement turns the attack unfeasible as the gadgets cannot be arbitrarely reached through corrupted indirect transfers, severely limiting possibilities of recombination to achieve the desired computation.

Return-Oriented Programming-based Attacks. Gadgets used during return-oriented programming attacks are small instruction sequences terminated with a `ret` instruction, which will redirect the execution towards the next gadget. As kCFI only allows indirect branches to matching tags, most of these gadgets become unreachable and unusable. Gadgets that remain useful for fortuitously being preceded by a tag cannot freely chain others, as its closing `ret` will only be allowed to redirect control to a reduced set of valid targets.

Unintended Gadgets. As the x86 architecture does not require executed instructions to be aligned, it is possible to retarget an indirect branch to unaligned addresses that may contain unintended gadgets across original kernel instructions [164]. As kCFI enforces all indirect transfers to target an aligned tag, it requires unintended gadgets to also be preceded with unintended tags, restriction that significantly reduces available unintended gadgets. As the tags are randomly generated, remaining gadgets differ between compilation instances and can be removed through another recompilation that prevents this specific tag from being used.

Coarse-grained CFI Attacks. Coarse-grained CFI restricts control-flow based on loose approximations of the applications CFG [41, 44, 110, 236]. It has been shown that these policies are yet permissive, and systems protected through such weak CFI schemes remain exploitable through the use of special gadgets which are compliant with the restrictions imposed [18,45,67]. kCFI is not vulnerable to these attacks as it applies a more restrictive CFG while building its policies, reducing available gadgets and precluding the remaining from being freely chained.

Evasion Gadgets. Some CFI systems are based on branch monitoring, detecting attacks by matching ROP-characteristic execution patterns [22, 131]. Such protections were shown to be flawed, as attackers can use evasion gadgets that will diverge the exploit's control-flow path from what is considered an anomaly, turning the attack indistinguishable [18, 45, 67, 68, 161]. Such attacks are indifferent to kCFI as it detects attacks through stateless statical instrumentation instead of dynamic observation of execution patterns.

Call-oriented Programming-based Attacks. Call-oriented programming-based attacks employ gadgets which are terminated with an indirect `call` instruction instead of a `ret`, corrupting its function pointers to chain other gadgets consecutively [18]. As kCFI requires all indirect calls to target the first instruction in a limited set of functions, it diminishes the number of available gadgets useful for these attacks.

Jump-oriented Programming-based Attacks. Jump-oriented programming-based attacks use a particular gadget called *dispatcher gadget*, that employs an indirect jump to redirect control-flow through a maliciously crafted list of functional gadgets, chaining them to achieve Turing-complete computation [11]. Although kCFI does not instrument jump instructions, we ensured that all indirect jumps present in the kernel are unusable for Jump-oriented programming purposes. The largest part of these instructions use verified index registers only allowed to target addresses confined to an unwritable memory region. A minor portion has its target set through a hard-coded immediate just a few instructions before being used. Both situations leave no opportunity for indirect jump target corruption.

Control-Flow Bending and Control Jujutsu. Exploits that employ code-reuse attacks whose flows are confined to the valid CFG paths were shown to be deployable against systems protected with user-space fine-grained CFI. Both Control-Flow Bending [17] and Control Jujutsu [53] are specializations of non-control-data attack that corrupts the arguments of a specific function for performing malicious computation and then manages to divert control-flow through a valid path to invoke it. kCFI does not fully close the matter against these attacks, but it raises the bar by enforcing more restrictive CFGs through applying the CGD optimization. This optimization creates obstacles for the attacks, especially against Control-Flow Bending, as it depends on corrupting returns to inject loops in the CFG.

Although possible in theory, a better understanding of how these techniques affect kernel code is required prior to assuming their efficiency in this context. Control-Flow Bending, for example, is demonstrated through (but not limited to) the employment of a technique called *printf-oriented programming*, that exploits a specific format string character to create memory write operations. As such character and its respective functionality are not available in `printk`, the kernel version of `printf`, the attack needs to be deployed through different and less common functions. Similarly, Control Jujutsu depends on the possibility of indirectly invoking functions whose parameters can be corrupted to cause arbitrary computation. Both invocation possibility and function existence are uncertain in the kernel context and require a deeper analysis.

Real-world Exploits. To assess the effectiveness of kCFI against real-world attacks, we used the ROP exploits for CVE-2010-3301 [26] and CVE-2010-3904 [27] by Kemerlis *et al.* [87], targeting Linux v2.6.33.6, as well as their custom exploit for v3.12. We first

verified that the exploits were successful on the appropriate kernels, and then tested them on the same kernels armed with kCFI. In all cases, the respective exploits failed, as the ROP payloads relied on pre-computed gadget addresses, none of which remained a valid (control-flow) target under kCFI.

CFI Metrics

AIR. As proposed by Zhang and Sekar [110], the AIR metric provides an understanding on how more restrictive a program becomes regarding allowed indirect branch targets after the introduction of a CFI policy. Through this metric, it is possible to compare and estimate the precision of different CFI implementations. The AIR computation is made using Equation 4.1, in which n corresponds to the number of indirect branches that exist in the program, S is the total number of valid targets allowed for an unprotected indirect branch, and $|T_i|$ is the total number of valid targets allowed for the protected indirect branch i.

$$AIR = \frac{1}{n}\sum_{j=1}^{n} 1 - \frac{|T_j|}{S} \tag{4.1}$$

On the kernel code base we used during this work, the use of a coarse-grained CFI mechanism similar to KCoFI [41], which allows indirect branches to target the beginning of any function or any instruction after a `call`, achieves an AIR of 98.64%. Although such a high value may give the impression of decent protection, it has been demonstrated that this level of permissiveness is still not enough to protect against ROP-based attacks [18, 45, 67, 68, 161]

By applying kCFI or kCFI+CGD on the same code base, an AIR of 99.99% is achieved when comparing it to the unprotected kernel. When comparing kCFI with coarse-grained CFI, the achieved AIR value is 99.93%, which is a significant improvement on the restrictiveness of indirect code paths. kCFI+CGD achieves a slightly better AIR value of 99.94% over the coarse-grained CFI AIR.

AIA. Instead of using an implicit comparison metric, Ge *et al.* [65] proposed the use of the average indirect targets allowed (AIA) metric, which captures the overall average of allowed indirect branch targets. After computing the AIA values for a program with different CFI policies, it is possible to understand their effectiveness by comparing the computed values. Equation 4.2, in which n is the number of indirect branches and $|T_i|$ is the number of valid targets allowed for the protected indirect branch i, is used to calculate AIA values.

$$AIA = \frac{1}{n}\sum_{j=1}^{n} |T_j| \tag{4.2}$$

The AIA values for the unprotected, coarse-grained CFI, kCFI, and kCFI+CGD kernel versions are presented in Table 4.1. Besides the benefits of fine-grained protection over

Kernel Version	Unprotected	Coarse-grained	kCFI	kCFI+CGD
AIA (all branches)	69086149	941957	680.5	545.3
AIA (only calls)	69086149	941957	60.7	62.9
AIA (only rets)	69086149	941957	952.9	769.9

Table 4.1: Average Indirect Targets Allowed (AIA) metric comparison.

the less restrictive policies, denoted by the three-orders-of-magnitude lower AIR value, the benefits of kCFI+CGD over kCFI also become more clear. These benefits, when analyzed through AIR, end up being hidden by the much larger magnitude of the valid branch target sets in less restrictive versions.

In kCFI+CGD, all function clones are never indirectly invocable, which positively affects permissiveness for return edges. The slight increase for calls unveils the limitation of static methods for computing the level of permissiveness of a CFI implementation. Because kCFI+CGD selectively clones functions, some indirect call instructions end up being cloned, while others do not. Cloned calls are not more permissive than those of the original functions—the set of allowed targets for both is the same. Nevertheless, due to the uneven duplication of indirect calls caused by cloning, the optimization slightly increases the AIA value observed for calls. Despite this fact, as the execution of a cloned indirect call always replaces the execution of its respective original instruction, equivalent traces of kCFI and kCFI+CGD will result in the same number of executed indirect calls. As both the original and cloned calls have the same level of permissiveness, there is no harm to security, even though the metric suggests differently.

Permissiveness Comparison Through CFI Metrics

The notions expressed by AIR and AIA are heavily bound to the code base being protected. As different programs will be inherently different regarding their indirect call graphs, using these metrics to compare the effectiveness of protection mechanisms requires confining the evaluation to the same code bases.

The fine-grained CFI implementation by Ge *et al.* [65] targets mainly FreeBSD, while kCFI is focused on Linux. Based on the data presented in Ge et al.'s paper, we can observe that the Linux version we used is 37% larger in terms of source lines of code, and has 3.3x more functions (132,859) and 4.6x more `call` instructions (809,098) than the FreeBSD version they used. While they attest that the function `printf` has approximately 5,000 possible return points in FreeBSD, `printk`, which is the corresponding one in our code base, has around 64,000. These differences imply an indirect call graph which is inherently more permissive and harder to protect, invalidating AIA and AIR comparisons between both implementations.

Ge *et al.*'s implementation incurs performance overheads that are comparable to kCFI's. To achieve this performance, it amortizes the costs introduced by instrumentation through analysis and optimizations which are also bound to the code base, such as converting indirect branches that have only one possible target into direct branches. On larger code bases, where more functions may have to be indirectly invoked, these

optimization opportunities become more scarce.

4.3 Code Size Overhead

Due to the extra instrumentation instructions added in the original code, the protected binary exhibits an overhead in terms of code size. When compared to the unprotected binary, kCFI incurs a size overhead of 2%. kCFI+CGD incurs a slightly larger overhead than kCFI due to the introduction of cloned functions in the final binary. In total, 17779 functions were cloned while applying kCFI+CGD to our code base, what, in addition to the CFI instrumentation, caused an increment of 4% in code size. The observed absolute binary sizes are 705MB for the Vanilla kernel, 718MB for kCFI and 732MB for kCFI+CGD.

4.4 Discussion

Void Function Pointer Arguments. C code allows indirect invocation of functions with mismatching prototypes through generalized arguments declared as `void` in the pointer - For example, the function `void foo(int a)` can be called through a pointer with prototype `void (void)`. Although not a good programming practice for breaking data abstractions and harming code legibility, these constructions are used to mimic polymorphism, which is not a default feature in the language. On kCFI, this issue has a second side-effect which is a deal breaker: pointers used to call functions which have mismatching prototypes will trigger a violation.

First, for identifying problematic invocations, we relied on a dynamic profiling method which was enabled by a custom violation handler capable of logging all the spots where the problems happened. After booting and stressing the kernel through the execution of LMbench and Phoronix [97, 109], we used the generated information to fix the kernel code. In total, 15 prototype mismatches were observed. All of them were either fixed by changing the function's prototype or creating a wrapper function. A more conservative approach would be merging the clusters for the mismatching prototypes, but we rejected this solution to avoid decreasing CFI restrictiveness.

As not all of the prototype mismatches existent in Linux code are intentional [104,105], this leverages a good side-capability of kCFI, which is unveiling prototype mismatches in the instrumented code.

Cache Performance. The guard instrumentation has an effect on cache schemes. This instrumentation may cause an additional L1 cache miss that certainly will be covered by an L2 cache hit if other cache levels are inclusive. On the targeted hardware platform, L1 caches are divided in code and data cache, but the same is not done in lower levels. In the worst case, when the guard dereferences a value for comparison, if the value is not on any cache level, the guard memory read is only anticipating a compulsory miss that would happen by the time the branch executes. Other possible situations are: already having the value on L1 data cache due to this have been used by a previous guard or already having it on lower level caches, both being less costly than the first described.

Tail Call Elimination. Tail call elimination is an optimization used to eliminate the execution of redundant `ret` instructions at the end of functions whenever it is preceded by a `call` instruction. This `call`, referred to as a *tail call*, can be replaced by a `jmp` instruction to the same target, promoting stack-frame reuse by the tail-called function. Because of this, when finished, this function executes a single return directly to the function underneath the caller, and not two (one to the caller followed by another to the function underneath it).

Applying tail call elimination to code protected with CFI is problematic because the tail-called function's back-edge guard will check for its tag in the return address, but this address will actually hold the caller's tag. Figure 4.3 shows how the optimization breaks chains of tag comparisons between protected functions. When the tail call elimination is not applied (Figure 4.3(a)), the verifications occur normally and no violation is triggered. When the optimization is applied (Figure 4.3(b)), `anubis_crypt`'s guard will check for its tag on the return address, but the assertion will fail as the verified tag is respective to the function `anubis_encrypt`.

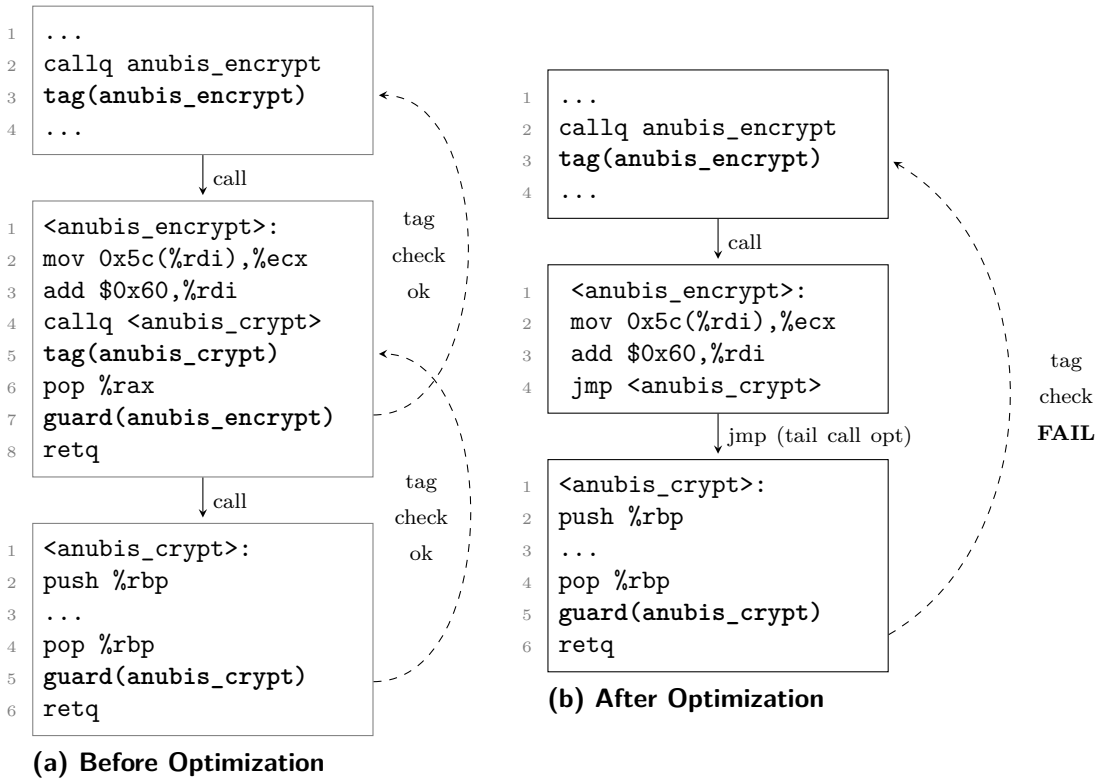


Figure 4.3: Assembly before/after tail call elimination

Trying to tackle tag inequities caused due to tail call elimination in the violation handler is not a good choice because of performance. This would require at least an extra pair of `push`/`pop` instructions, plus one `call` and one `ret` to be executed. These overheads, plus the computation executed inside the violation handler function to solve the false-positive, largely exceeds the benefits of the optimization.

Another approach for tackling the problem is through merging the CFI Map nodes respective to the caller and the tail called functions, so they would hold similar tags. Although a plausible solution, it would imply in a less fine-grained CFG, only being

reasonable if the benefits of using tail call eliminations on the Kernel were indispensable.

In order to evaluate if merging the nodes was a reasonable solution, we performed an experiment to check how much of the Kernel's performance is degraded by disabling tail call elimination. The Linux kernel in its Vanilla version was compiled with and without tail call elimination and the kernel micro-benchmark LMbench [109] was executed on top of these systems to assess the performance degradation. In this experiment, an average performance overhead of 5% was observed on the kernel without tail call elimination, what was considered as a low impact in a kernel micro-benchmark.

Considering the drawbacks on making CFI compatible with tail call elimination and the low overheads displayed when the optimization was disabled, we decided to follow kCFI's implementation with no support to it. As disabling tail call elimination became a requirement for kCFI, these overheads were incorporated to kCFI's own overhead. All previous performance comparisons considered that the protected version was compiled without tail call elimination, while the unprotected version had it enabled.

Dynamically Loadable Kernel Modules. Although kCFI supports the use of loadable kernel modules, the analyses done to build the CFI Map are bind to the source files being compiled. Because of that, compiling modules to be loaded in a system which was previously built may break assumptions used in CFI policies, such as which functions are indirectly invocable. In these cases, CFI Map compatibility must be recovered, and this can be done through full system recompilation.

Improving Granularity. As previously shown, kCFI introduces a significantly more restrictive policy than other coarse-grained CFI implementations. Yet, it is still possible to reduce its granularity even further through the use of context information which is dynamically verified during runtime, such as when a shadow stack is in place, as suggested by Abadi *et al.* [4]. Chapter 5 proposes different shadow stack architecures for the kernel, along with a discussion regarding its viability in terms of performance costs.

Chapter 5

A Study on Kernel Shadow Stacks

We see in order to move
We move in order to see

William Gibson

In this Chapter we present a first feasibility analysis on the implementation of a shadow stack mechanism in the Linux kernel, highlighting the implementation challenges and limitations. Through the results observed in these experiments, we propose an approach that takes advantage of Intel's SMAP [34] hardware extension to enable a lower-cost shadow stack protection strategy, alternative to using the regular page table access control mechanism.

5.1 Overview

A shadow stack mechanism consists of a data structure meant to store replicas of addresses which are saved in a regular process stack. Whenever a function is called, its respective return address is copied into the shadow stack and later used to ensure a valid return either by overwriting the address in the original stack with it or by comparing both. Through using a shadow stack, it is possible to detect and prevent attacks based on the corruption of stacked return addresses [23, 61, 116, 143].

Despite the benefits, shadow stacks are not largely employed on real-world systems due to the significant performance overheads they introduce. As these schemes assume a threat model where an attacker can exploit a vulnerability to write arbitrarily on unprotected memory, they must ensure that permission policies preserve values saved on the shadow stack. Nevertheless, the system itself must write on the data structure to update entries in the shadow stack, requiring consecutive reversal of memory permissions. Costs inherent to updates on access permissions are frequently too expensive, leading the system to infeasibility. Dang *et al.* [43] showed that even schemes without shadow stack protection present non-negligible overheads.

Features like `setjmp/longjmp` may cause many stack frames to be unwinded at once, causing assertions between values in the regular stack and the shadow stack to fail. Exceptions and other incongruences between `call` and `ret` are frequent on software due to language features or binary optimizations and they remain as a challenge for shadow

stack implementations. A standard approach to solving this problem is through allowing returns to a given address as long as this address is present in the stack, not necessarily at its top. This approach introduces more overhead while it is also not as restrictive as the previous one.

As explained by Abadi *et al.* [4], shadow stacks can be used to refine the granularity of CFI schemes. By using these mechanisms, backward-edges are restricted to target a single destination[1], instead of multiple call sites respective to the returning function. Such raise in the restrictiveness is interesting, particularly since it is known that the level of granularity provided by fine-grained CFI schemes can be exploited in a way to allow the invocation of dispatcher functions or the injection of execution loops in a program's control-flow without violating the enforced CFG. These resources are particularly interesting as they enable the achievement of Turing-completeness on techniques employed during Control-Flow Bending attacks [17]. We comment on how a shadow stack would pair with kCFI while discussing our threat model, in Section 5.1.2.

5.1.1 The need for a Shadow Stack

Some functions provide computational capabilities to attackers just by having their arguments corrupted [3, 17, 53]. For being *non-control-data attacks*, these inherently do not influence control-flow, and are transparent to CFI enforcement. Normally, the computational capabilities provided by these attacks are limited, requiring the use of control-flow hijacking techniques to inject loops or trigger dispatcher functions in a way to consecutively invoke the attacked function and achieve arbitrary computation. As CFI is in place, control-flow hijacking attacks are limited to the computed CFG. Unfortunately, even under a fine-grained CFI, some functions may be invoked from too many places, creating large sets of valid return targets and causing the resulting CFG to be relatively loose. This fact creates valid paths that enable attackers to manipulate control-flow as needed.

When a shadow stack is in use, it enforces backward-edges to follow a context-based parity policy where each returning function must return to the call site responsible for its invocation. Such restrictiveness breaks loop injections and dispatcher function triggers, leaving attackers solely with computation achievable through non-control-data attacks or CFG-restricted hijacking attacks in forward-edges, as in Control Jujutsu attacks [53]. As demonstrated by Carlini *et al.* [17], shadow stacks are a meaningful protection against this sort of attacks.

5.1.2 Threat Model

Although our current design focuses on the feasibility analysis of a kernel shadow stack, we target a design that is concise in terms of providing defenses against a realistic threat model. Hence, we anticipate its use under the following circumstances.

Adversarial Capabilities. For the ahead prototyped shadow stack scheme, we assume an adversary capable of executing user programs on the OS, seeking to elevate

[1] Or way fewer destinations, if `setjmp/longjmp` compatible shadow stacks are in use

privilege by exploiting a memory corruption vulnerability in kernel code [28, 29]. This opponent is capable of arbitrarily overwriting return addresses stored in a kernel space stack [25, 210, 212], being able to trigger a control-flow hijacking attack through the execution of a return instruction. Adversary's capabilities are not restricted to precise return address corruption — the attacker can overwrite the return address through contiguous memory operations successfully bypassing stack smashing protections. Additionally, we consider that the adversary knows the kernel memory layout, irrespectively of how it was achieved.

Hardening Assumptions. We assume that the system is hardened with W^X policies, preventing the possibility of code injection or replacement in kernel memory space. Also, we assume that protections against *ret2usr* attacks are in place. Particularly for one of the strategies we assessed, we assume that the underlying platform is enhanced with hardware features for address space isolation, such as Intel's SMAP [34]. The shadow stack we propose does not require the use of CFI protections such as kCFI. Although orthogonal, the combination of both schemes generates mutual benefits as the shadow stack refines the backward-edge protection provided by kCFI while kCFI introduces forward-edge protection which is not covered by a shadow stack.

5.2 Background on Linux Kernel Stacks

The concepts we present ahead are architecture-independent and can be replicated on any hardware platform that has memory protection or address space isolation features, such as Intel's Write Protect (WP) bit [37] and SMAP [34]. Similarly, OSes are only required to support these hardware features, not demanding any extra functionality. For our preliminary implementation and further evaluation, we picked Linux running on top of an x86-64 CPU. Before providing details regarding implementation, we provide clarifications regarding intricacies on what concerns kernel stacks.

The Linux kernel keeps one kernel stack for every active thread. Whenever a thread is running in user space, the kernel stack has no frames, only being filled when the execution enters into privileged mode. Each of these kernel stacks holds a structure called `thread_info` which, amongst other data, contains a pointer for the `task_struct` respective to the running thread. This `task_struct` holds information about the process such as states, scheduling, identification, inter-process communication (IPC), timers, virtual memory and file system interaction. On x86-64 architectures, the per-thread Linux kernel stacks have 16KB each.

Besides the per-thread kernel stacks, five specialized stacks with specific purposes exist in kernel space. These stacks are:

- **IRQ Stack:** Used by the kernel for handling external hardware interrupts in a separate memory region than the running kernel stack, avoiding the need of increasing the size of per-thread stacks. Similarly to the per-thread kernel stacks, this data structure has 16KB.

- **Double Fault Stack:** Used for handling cases where handling one exception causes another exception. In these cases, as the system is possibly running in a faulty state,

a secondary stack is useful for enabling more accurate crash reports. This stack has 4KB.

- **NMI Stack:** Used for handling *non-maskable interrupts* (NMI). This stack has 4KB.
- **Debug Stack:** Used for handling hardware and software debug interrupts. This stack has 8KB.
- **MCE Stack:** Used for handling *machine check exceptions*[2] (MCE). This stack has 4KB.

The x86-64 architecture provides a feature called *Interrupt Stack Table* (IST) [84], that can be used to switch stacks efficiently. Whenever an interrupt occurs, its respective gate descriptor is loaded and verified for the existence of a valid IST code. If this is the case, this code is used to index IST entries which are held into the *Task State Segment* (TSS)[3] and contain pointers to the specialized stacks. Once the right entry is loaded, the hardware is able to set a new stack pointer and proceed with the invocation of the interrupt handler.

Stack selection through IST can be nested as long as their respective triggering events have different IST codes. Since nested interrupts are frequent in kernel execution, switching to the IRQ stack cannot be supported by the IST feature and is done through a software-based mechanism. Using the IST is limited to DoubleFault, MCE, Debug and NMI stack switches.

Kernel stacks are allocated in a special kernel space memory region known as *physmap* [86]. This region maps part of the computers available physical memory directly, in a way to allow fast dynamic kernel memory allocation.

Figure 5.1 shows all kernel stacks. On the leftmost side, each active thread has its respective kernel thread, while the right side displays the specialized stacks. The figure also shows the thread_info data structure on the bottom of per-thread stacks and the IST pointing to the specialized stacks.

5.3 Design

Attackers are capable of hijacking the control-flow of an unprotected kernel through corrupting return addresses stored in the stack. In our threat model, W^X and *ret2usr* protections are in place, precluding attackers from redirecting kernel control-flow to maliciously crafted execution payloads, requiring them to employ more sophisticated techniques, such as ROP, to successfully exploit the system.

Shadow stacks pose as a meaningful protection against these threats, as they use context-based information to improve the refinement of the program's backward-edges in a way unachievable through static analysis. Despite that, as the OS kernel is a central

[2] Interrupt 18

[3] TSS is a data structure that holds information about tasks. Linux design reuses TSSs, using one per CPU, instead of one per task.

component that affects all other parts directly in a computer system, incrementing it with a shadow stack mechanism demands accurate fulfillment of certain requirements[4]. First, the overall cost must be minimal, what precludes the use of intermediary layers between kernel and hardware, requiring light-weight approaches such as code instrumentation. Second, the shadow stack must be compatible with basic kernel functionalities, such as self-modifying code and LKMs. Third, events that trigger different control flows, such as interrupts or exceptions, may compliantly return to the halted execution without triggering false-positive violations.

Shadow stacks are known to be computationally expensive mechanisms, especially due to the requirement of protecting its memory pages against malicious write operations [43]. The shadow stack used in our tests consists in a prototype for a feasibility analysis. So, we add on the top of the OS requirements the need for an architecture that enables performance verification of different memory protection management strategies. Our first goal is to understand these overheads and leverage different forms of tackling them, especially through the employment of hardware features whenever possible.

The shadow stack solution we propose was designed in consonance with all the clarified requirements for both OS and feasibility analysis — All OS requirements were considered strictly, in a way to provide a realistic performance assessment and highlight evidence on possible optimizations. In this prototype, all information used to determine control-flow validity is extracted during kernel execution; such dynamic nature turns the compatibility with LKMs and self-modifying code easy to handle. Also, our approach is inherently compatible with events like interrupts and exceptions, not requiring any additional effort for that. Finally, performance costs of shadow stacks are tackled by the use of a fully compiler-based instrumentation approach that precludes the need for any hypervisor or monitoring routine. The adopted compiler-based scheme enables easy experimentation of different strategies for memory protection management, only requiring the addition of new compilation passes at the back-end portion of the compiler.

[4]Such requirements are similar to those imposed to CFI implementations, described in Section 3.2 of this thesis

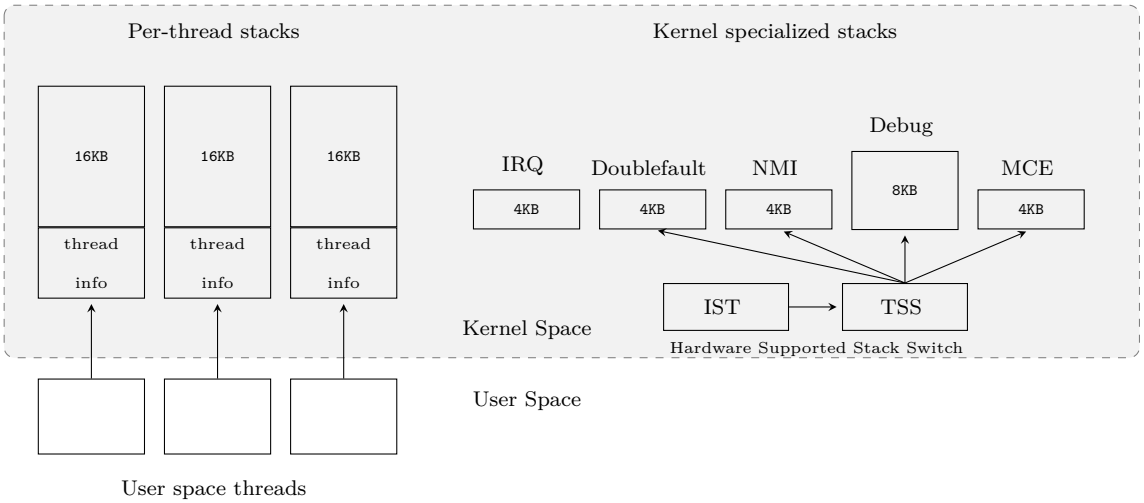


Figure 5.1: Kernel stacks organization

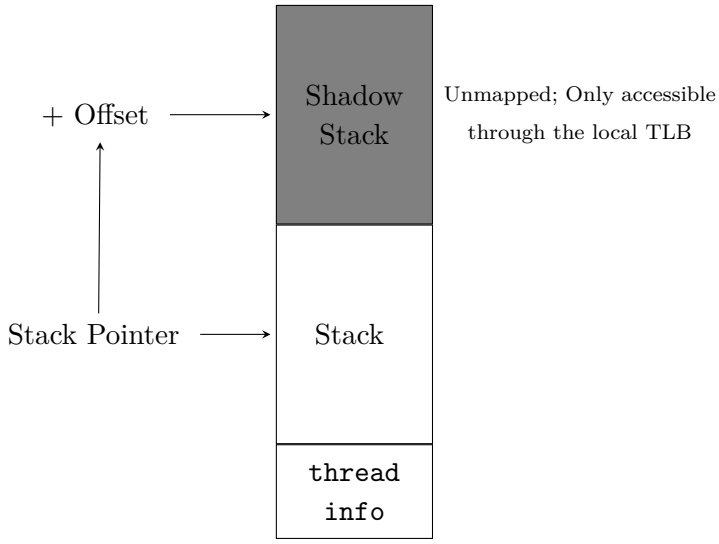


Figure 5.2: Shadow stack access through stack pointer

Some shadow stack implementations tried to improve performance through creating read-only memory areas around the boundaries of the shadow stack, leaving the data structure itself writable [23]. This solution decreases overheads by preventing permission switches prior to adding data to the shadow stack while it still provides protection from contiguous write operations targeted to reach stack contents. Despite the benefits, this solution is insufficient as precise memory overwrite attacks can be employed to bypass the defenses. In our approach, we protect the entire shadow stack structure, as this is plausibly a safe solution.

5.3.1 Kernel Shadow Stacks

To ensure correct backward-edge flow in the protected kernel, our scheme reserves a shadow stack for each execution context existent in the kernel space — similarly to regular stacks, which are not shared amongst different contexts, shadow stacks are also thread-specific. These shadow stacks are allocated right above the regular stacks on memory layout and have the same size as the respective stack. By employing this approach, it is possible to access memory positions inside the shadow stack by adding a fixed offset (which is the stack size) to the regular stack pointer, avoiding the need of allocating and protecting another memory pointer. Dang *et al.* [43] refer to this scheme as *Parallel Shadow Stack*. Figure 5.2 illustrates the scheme, in which the regular stack pointer is incremented by an offset to point to the shadow stack.

By employing this approach, the symmetry between the regular and the shadow stack is always kept. If the kernel uses any feature that unwinds multiple stack frames at once, this behavior is also noticeable on the shadow stack, as the reference pointer for both structures is the same. This design precludes the need of care regarding the equivalence between compared return addresses on both stacks, avoiding the requirement of a relaxed policy that is compliant with features like `setjmp/longjmp`. In comparison with other shadow stack implementations, this strategy is more efficient both in terms of performance and restrictiveness [43].

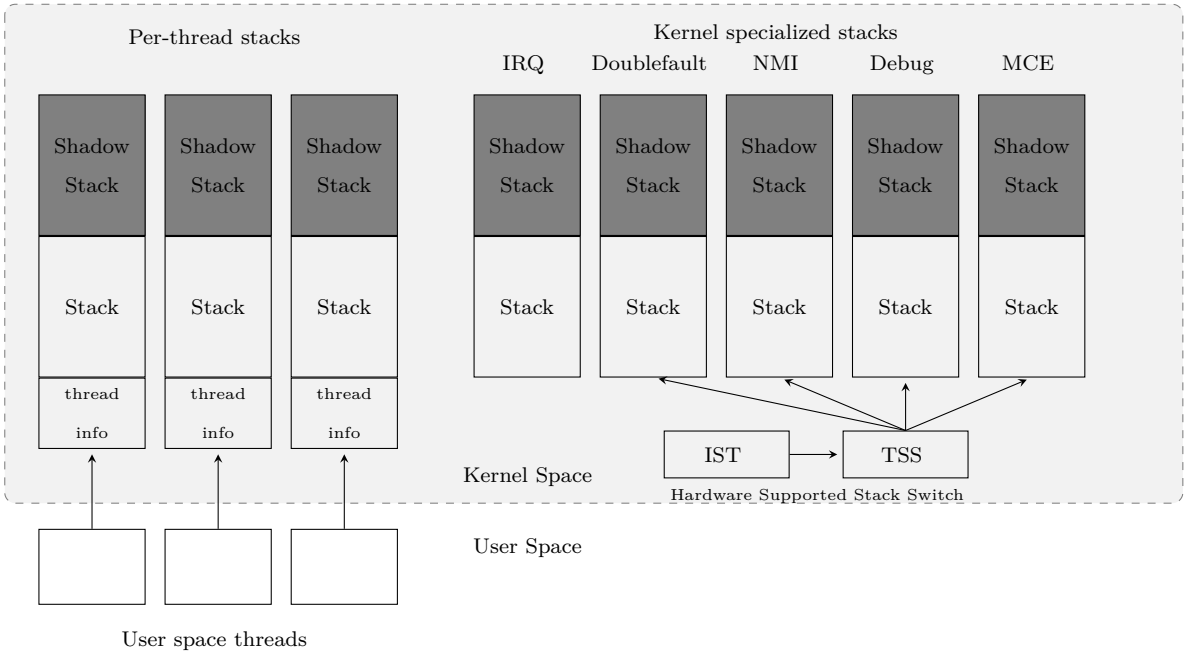


Figure 5.3: Kernel stacks and shadow stacks

Another advantage of using this approach, with fixed positions referenced by the regular stack pointer, is that context switches inherently switch the shadow stack in use with no extra computational cost, decreasing the performance overheads due to shadow stack usage. Hence, whenever kernel switches context and starts running a different execution thread, with a different stack, the shadow stack can already be instantly accessed. The whole scheme architecture, with the multiple shadow stacks positioned above the regular per-thread and specialized stacks, can be seen in Figure 5.3.

Despite the benefits, the approach also introduces a challenge – as the offset is bind to the stack size, different stacks will have different offsets, what requires functions to be precisely instrumented with the offset respective to their stack size. Unfortunately, it is not possible to statically identify which stack is operated by each function, as the same function may be invoked in different contexts. To overcome this difficulty, we redesign the kernel memory layout so that all stacks are equal, having the size of the largest allocated stack[5].

To prevent running threads from interfering with irrespective shadow stacks, we add a modification to the kernel's memory management system. First, we prevent the shadow stack from being mapped into physmap, ensuring that no thread is capable of accessing it. To allow shadow stack operations performed by its corresponding thread, we leverage the use of the processor's *Translation Look-aside Buffer* (TLB) — on the occurrence of a context switch from user to kernel mode, the shadow stack pages are temporarily mapped and accessed, causing them to be loaded into the TLB. Once all shadow stack pages are reachable through the TLB, the respective pages are again unmapped.

Although the modifications to the memory page mapping mechanism ensure that only the respective thread access its own shadow stack, attackers may still be able to corrupt

[5]On x86-64 Linux, 16KB

the shadow stack through exploiting a memory bug on the running thread. To prevent this from happening, we employ hardware mechanisms that forbid write operations on properly marked memory pages. These protections are enabled during most of the system's runtime, only being disabled on demand, whenever a thread needs to save a new address into the shadow stack. Specific details on each mechanism employed are discussed in Section 5.4

5.3.2 Code Instrumentation

The proposed shadow stack design ensures valid returns by asserting addresses used as targets in `ret` instructions. For that, our scheme employs compiler-based instrumentation to insert the proper primitives on function prologues and epilogues. We refer to these instruction sequences as *shadow push* and *shadow assert*.

A shadow push saves the return address referenced by the stack pointer on the shadow stack. For that, this primitive needs to temporarily enable write operations on the shadow stack data structure by disabling the memory protection mechanism used. Shadow pushes are placed on function's prologues, ensuring that every function saves its own return address into the shadow stack before any other functionality. Figure 5.4 illustrates the instruction sequence respective to a shadow push — the displayed sequence is generic and does not specify the memory protection mechanism nor the offset used, leaving these to be set accordingly to the required needs with no major harm to other components on the scheme. For being general, this instruction sequence can be adapted to fit different implementation strategies, what is desired given the analytical goals behind the proposed scheme. Specific protection management with real underlying technologies is described ahead, in Section 5.4.

```
1 <kernel_function>:
2 mov     (%rsp),%rax
3 disable_memory_protection
4 mov     %rax,-offset(%rsp)
5 enable_memory_protection
6 ...
```

Figure 5.4: Shadow Push

Function epilogues are instrumented with shadow asserts, which are illustrated in Figure 5.5. In this Figure, `%reg` denotes any general purpose register which is dead at the function's epilogue. The illustrated primitives precede `ret` instructions and compare the address that will be used by them with the respective address on the shadow stack. If the compared addresses match, then control-flow jumps to the `ret` instruction. Otherwise, a shadow stack violation handler function is invoked, halting the system or performing fail-safe routines. Notice that there is no need to manage memory protections during shadow asserts because no write operation is carried out on the shadow stack. As these operations are costly, this is another efficiency trait of our approach.

```
1 mov     (%rsp),%reg
2 cmp     -offset(%rsp),%reg
3 je      5
4 callq   <ss_vhndl>
5 retq
```

Figure 5.5: Shadow Assert

5.4 Implementation

We built our shadow stack instrumentation tool on top of the LLVM compiler, as a back-end compilation pass — this pass injects shadow pushes and shadow asserts on each function prologue or epilogue, as described in Section 5.3. Differently from kCFI's approach, the shadow stack instrumentation does not require prior analysis and can be applied straightforwardly during IR translation into machine code.

With the purpose of evaluating different shadow stack protection approaches, we implemented shadow pushes using two different methodologies for memory access management. First, we used the WP bit [37] provided on Intel platforms which can be used to allow kernel code to write on read-only pages. The second approach used the SMAP extension [34], also available on Intel platforms, which isolates user space memory from kernel space memory, precluding it to be directly accessible from kernel context.

5.4.1 Write Protect Bit

A straightforward approach for protecting the shadow stack is through using the regular page table entry-based access control mechanism. In this method, the memory pages which hold the shadow stack are marked as read-only and, when a shadow push needs to write on these pages, it first disables the `WP` bit from the control register `CR0`. By disabling this bit, supervisor-level procedures are allowed to write on read-only pages [37]. Right after writing to the shadow stack, the shadow push re-enables the `WP` bit in `CR0`.

`CR0` is a special-purpose register, which cannot be directly operated through arithmetical instructions, requiring its value to be loaded into another general-purpose register, operated and then stored. Although the value in `CR0` can be loaded whenever necessary, doing it on every function prologue is costly, especially because the value loaded rarely changes after system boot. Instead, we reserve two registers `R10` and `R11`, to keep the values of `CR0` with `WP` set and unset so that it can be loaded instantly during the shadow push. Despite the benefits of reserving these registers, such method precludes them from general-purpose uses, forcing the compiler to spill variables into memory more frequently and raising performance overheads.

Figure 5.6 illustrates the shadow push operation which employs the `CR0` approach for protecting shadow stack memory. In the presented instruction sequence, the return address is loaded into `RAX` and `CR0` is updated with a disabled WP bit. After that, the return address is stored in an address referenced by the stack pointer plus an offset, which results in a shadow stack address, and then `CR0` is loaded with a value that sets the WP bit.

```
<kernel_function>:
mov     (%rsp),%rax
mov     %r10,%cr0
mov     %rax,-offset(%rsp)
mov     %r11,%cr0
...
```

Figure 5.6: WP Bit Shadow Push

5.4.2 SMAP

SMAP [34] is an x86-64 architecture extension that enforces isolation between kernel and user space, precluding user data to be dereferenced from kernel context. Although designed with a different purpose, SMAP can be leveraged to protect kernel data structures. For that, protected pages must be marked as user-mode pages, what will turn them inaccessible from the kernel context whenever the feature is enabled. Our scheme marks the shadow stack memory pages as such when these are temporarily mapped into physmap, before priming the TLB with their respective entries.

SMAP includes two new instructions to the processor's instruction set, which are `clac` and `stac`. Through the `clac` instruction, it is possible to disable the isolation mechanism, allowing the kernel to operate user space memory whenever needed. Once the kernel no longer needs to deal with user space, and the isolation is desirable again, the `stac` instruction can be used to re-enable it.

The shadow stack can be written to as illustrated in Figure 5.7. In this figure, the return address is first loaded from the stack, then the isolation mechanism is disabled, the address is saved to the shadow stack by using the stack pointer plus an offset, and finally the isolation is re-enabled. The disadvantage of using SMAP is that this approach also isolates reads from protected memory, requiring SMAP instructions to also be included in the shadow asserts, as illustrated in Figure 5.8.

```
<kernel_function>:
mov     (%rsp),%rax
clac
mov     %rax,-offset(%rsp)
stac
...
```

Figure 5.7: SMAP Shadow Push

```
mov     (%rsp),%reg
clac
cmp     -offset(%rsp),%reg
stac
je      7
callq   <ss_vhndl>
retq
```

Figure 5.8: SMAP Shadow Assert

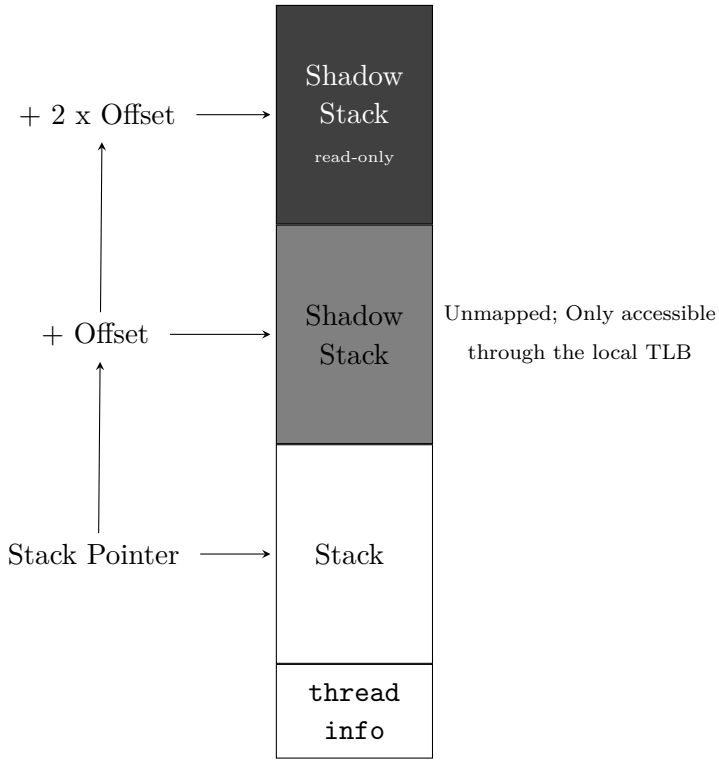


Figure 5.9: Shadow stack layout with read-only mapping

```
1 mov     (%rsp),%reg
2 cmp     -(2*offset)(%rsp),%reg
3 je      5
4 callq   <ss_vhndl>
5 retq
```

Figure 5.10: SMAP Shadow Assert with read-only stack

SMAP With Read-Only Shadow Stack Copy

To avoid the overheads of switching the memory permissions during the assert operation, we propose a third scheme, also based on SMAP. In this scheme, we extend the layout design proposed in Section 5.3 by adding a read-only mapping of the shadow stack above the explicitly unmapped pages. This design provides means through which asserts can perform the required comparisons without exposing the data structure or requiring extra operations, as the read-only pages can be used safely for this purpose.

In the proposed design, the shadow stack can be accessed through dereferencing the stack pointer added by a fixed offset, which is equivalent to the stack size. Similarly, the read-only shadow stack can be accessed for read operation by dereferencing the stack pointer added by a fixed offset, which is equivalent to twice the size of the stack. The memory layout for the stacks can be seen in Figure 5.9, and the instruction sequence respective to the assert, without the memory protection instructions and with the read-only shadow stack correspondent offset, can be seen in Figure 5.10.

5.4.3 Shadow Stack Optimizations

Because shadow stack operations happen during prologues and epilogues, the introduced overhead is bind to the execution of `call/ret` instructions. Hence, a form of reducing the general cost of the scheme is through the use of optimizations that preclude the existence of these instructions, like *tail call elimination* and *inlining*.

Tail Call Elimination

Tail call elimination optimizations convert calls that precede `ret` instructions into direct jumps, avoiding the execution of consecutive returns and promoting stack-frame reuse by the top-most function in the calling chain. Although tail call elimination does not break the shadow stack scheme, the tail-called function redundantly executes a second shadow push — If the stack pointer points to the return address when the tail-called function starts, the shadow push will execute a redundant copy of the return address on the shadow stack. If not, the shadow push will only save values which will never again be used.

This behavior opens space for an optimization. As this function may be called in a context where it does not reuse the caller's stack frame, the shadow push in the tail-called function cannot be removed. Despite that, it is still possible to avoid its execution by adding an offset to the *jmp* instruction generated during the `call` conversion. As shadow pushes are injected at the beginning of the prologues, skipping them won't cause damage to the execution semantics. Figure 5.11 illustrates tail call elimination applied to code instrumented with a shadow stack. On Figure 5.11(a) the shadow push is redundantly called by `anubis_crypt` while, in Figure 5.11(b), an offset is added to the `jmp` target, bypassing the shadow push and branching directly to the function's original prologue.

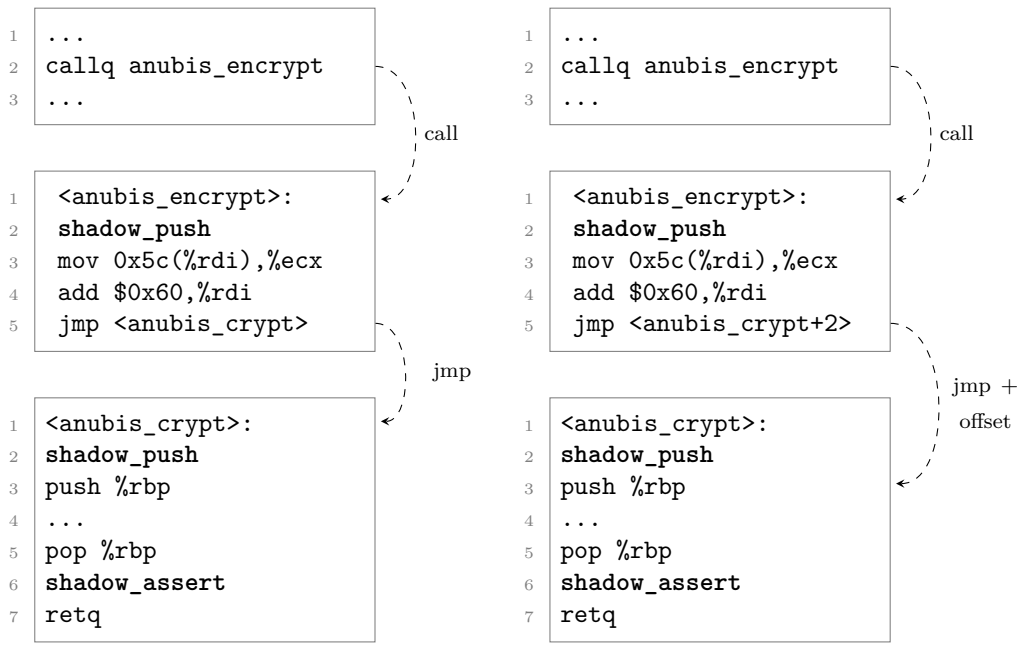


Figure 5.11: Assembly before/after tail call elimination

As explained in Section 4.4, kCFI does not support tail call elimination because it would undesirably relax the enforced policies. As shadow stack restrictiveness is not determined by previously computed CFGs, the implications of tail call elimination on kCFI are not sustained when backward-edges validation is made through this kind of mechanism. Hence, besides the introduced restrictiveness, incorporating a shadow stack to kCFI also restores the scheme's compatibility with tail call elimination.

Inlining

Inline optimizations expand `call` instructions into the function which are called by them, embedding the callee into the caller. This kind of optimization precludes the existence of a `call/ret` pair, what benefits our shadow stack approach.

Despite the benefits, aggressively applying inline optimizations creates binary size overheads. As larger binaries normally present less locality, there is also an effect on the code cache efficiency. Besides, unbounded recursive inlining may result in the explosion of the binary size. Due to these reasons, the use of such optimization must be done in a balanced way. Considering that, our current approach employs inlining optimizations as defined by the compiler's algorithm.

5.5 Evaluation

Willing to identify which of the proposed designs performs better prior to the implementation of a proper shadow stack, we did a preliminary evaluation. During this process, the Linux kernel was instrumented with primitives that mimic the correspondent shadow stack operations and later tested during the execution of the LMbench [109] micro-benchmark, enabling the overhead comparison amongst the underlying designs. The system used for measurement on all tests was an Intel(R) Core(TM) i7-6700 8 core CPU @ 3.40GHz, with 32GB RAM memory, running Debian Linux with GNU kernel v3.19.0. Every program unit inside the benchmark was executed ten times, and the presented results are averages of the observed numbers.

The mimic primitives employ the previously mentioned memory protection hardware features analogously to the cases where the kernel has its memory layout redesigned to provide space for the shadow stacks. These instruction sequences can be seen in Table 5.1. This resulted in a set with four tested kernel versions: (i) *Vanilla*, which is not instrumented and is used as the comparison baseline; (ii) WP Bit, which is compiled with a write-protect bit/page table entry-based protection primitives; (iii) SMAP, which is compiled with SMAP primitives both on prologues and epilogues; and (iv) SMAP+RO, which is compiled with SMAP primitives on the prologues and regular primitives on epilogues, as it would happen with the use of a read-only shadow stack mirror. For the WP Bit case, we also reserved the register `R10` and `R11`, precluding them from being allocated by the compiler — by comparing the *vanilla* kernel with and without this registers reservation, we noticed an overhead of ≈10% when running LMbench.

Figure 5.12 illustrates the performance overheads for the evaluated kernels. The charts presented are in logarithmic scale as the costs for the WP Bit version are significantly

Version	Shadow Push	Shadow Assert

WP Bit	`<kernel_function>:`
	`mov (%rsp),%rax`
	`mov %cr0, %r10`
	`mov %r10, %cr0`
	`mov %rax,(%rsp)`
	`mov %r10, %cr0`
	`...`
		`...`
		`mov (%rsp),%reg`
		`cmp (%rsp),%reg`
		`je 6`
		`callq <ss_vhndl>`
		`retq`

SMAP	`<kernel_function>:`
	`mov (%rsp),%rax`
	`clac`
	`mov %rax,(%rsp)`
	`stac`
	`...`
		`...`
		`mov (%rsp),%reg`
		`clac`
		`cmp (%rsp),%reg`
		`stac`
		`je 8`
		`callq <ss_vhndl>`
		`retq`

SMAP RO	`<kernel_function>:`
	`mov (%rsp),%rax`
	`clac`
	`mov %rax,(%rsp)`
	`stac`
	`...`
		`...`
		`mov (%rsp),%reg`
		`cmp (%rsp),%reg`
		`je 6`
		`callq <ss_vhndl>`
		`retq`

Table 5.1: Shadow Stack primitives

higher, reaching 24.72x in the *Select TCP* test, which is the worst case. On average, WP Bit presented an overhead of 8.6x for latencies and 5.01x for communication throughputs. The overheads were drastically decreased by employing SMAP instead of using the WP Bit mechanism — SMAP presented average overheads of 2.31x and 1.8x for latencies and communication throughput while SMAP+RO reduced these even further, respectively inducing overheads of 1.69x and 1.49x. The maximum overheads presented by SMAP and SMAP+RO were of 5.04x and 3.28x, both in the *Select TCP* test.

5.6 Discussion and Future work

During the development of this work, we noticed some limitations and opportunities regarding our shadow stack approach. Prior to turning the observed results into a real implementation, we envision that some topics need to be considered. We highlight the most relevant amongst these here, in a way to foment discussion and proper solutions to the matters.

Assembly Support. Our approach does not support Assembly functions yet. Nevertheless, developing such support follows the same method used in kCFI— most functions can be automatically patched through source code rewriters, while a small set of macros that do not follow a clear pattern will require manual patching. As instrumenting Assembly functions with shadow stack primitives do not need CFG information for determining tag values, this process is even more straightforward in this context. Nevertheless, differently from kCFI, not instrumenting Assembly code won't trigger false-positive violations.

Better Inlining Strategies. Although we currently only employ inlining optimizations as defined by the compiler's algorithm, there is a clear opportunity for improving the

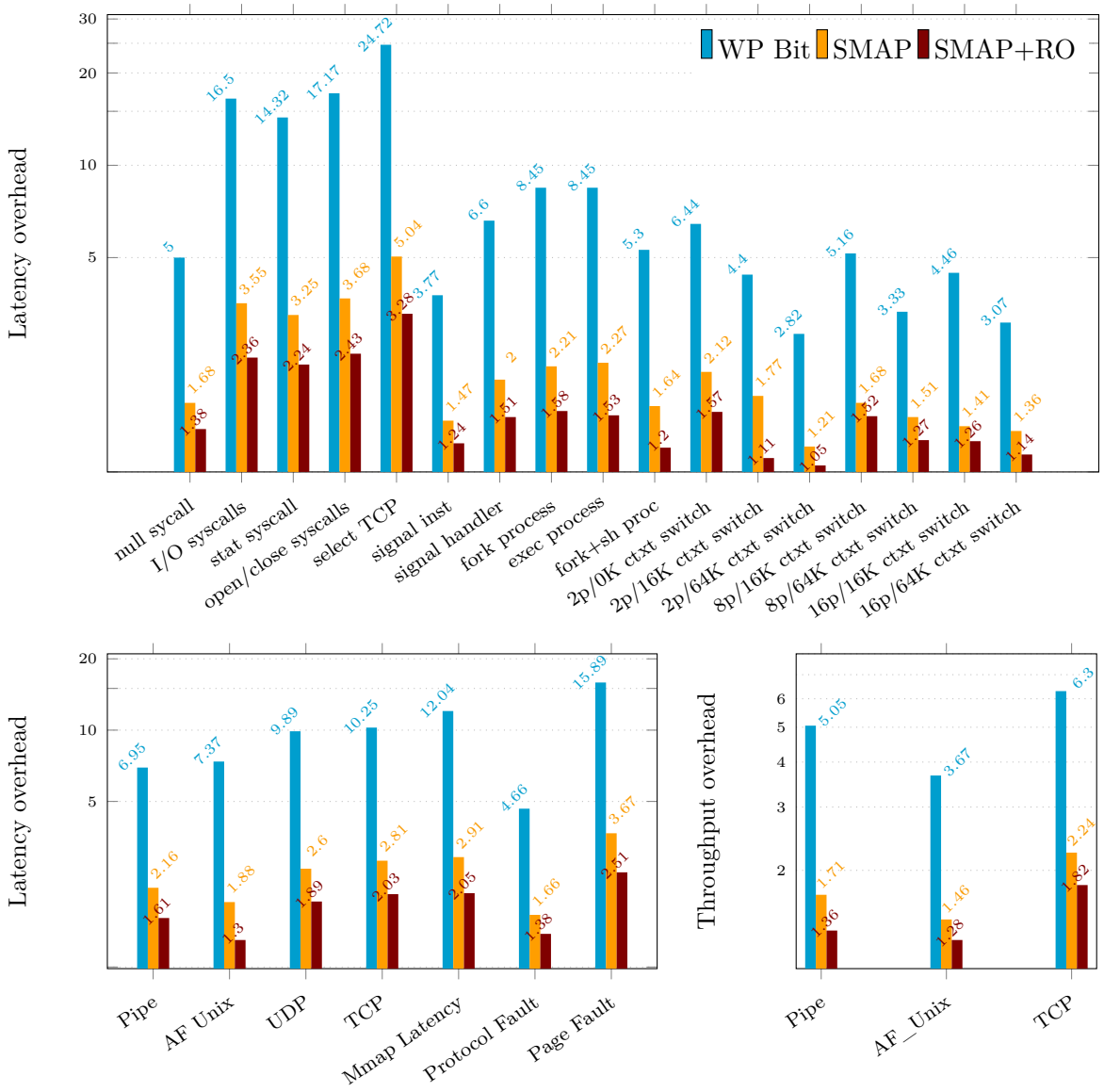


Figure 5.12: Performance overhead introduced by each memory protection approach (logarithmic scale)

system's performance by applying these optimizations more aggressively or by targeting them with the assistance of profiling information. By employing dynamic analysis in the identification of the most frequently executed `call` instructions, it is possible to select the most beneficial targets for inlining regarding runtime cost. By combining this information with the callee size, which is already used by the compiler's algorithm, the equation used to select inlining candidates can be keenly reformulated, increasing its benefits under the shadow stack context.

Memory Protection Keys Extension. Intel recently announced MPK [35], a x86-64 architecture extension that enables setting page permissions to an entire group of pages with a single operation. Unfortunately, by the time we developed this research, MPK was not yet available in any Intel product and, for this reason, we did not assess its efficiency. Despite that, future shadow stack implementations following our proposed design can clearly benefit from this technology.

User space Shadow Stacks. The presented shadow stack design is strictly applicable in the kernel context, especially because of the memory protection mechanisms employed. During the research which culminated in this thesis, we developed a shadow stack implementation based on DBI methods for user space applications – this implementation relies on an asynchronous architecture and employs efficient communication methods to achieve better performance. For not being focused on the kernel domain, we moved the details regarding this implementation to the Appendix A. Yet, its design and use are orthogonal to kernel solutions discussed in this thesis.

Chapter 6

Related Work

> You must understand that there is more than one path to the top of the mountain
>
> *Miyamoto Musashi*

As kernel exploitation through *ret2usr* methodologies was addressed by different protection technologies [34, 89, 113, 132–134, 168, 232], attackers were pushed into evolving control-flow hijacking techniques for reusing code confined to the kernel address space. From these techniques, ROP and its variants [11, 142] were widely employed for enabling Turing-complete computation even when launched over small code bases [15]. Although widely adopted to attack user space programs, ROP-based attacks were shown to work in kernel software [77, 122].

Trying to protect the kernel against all sorts of control-flow hijacking attacks, including ROP-based variants, researchers proposed different schemes that consist of CFI implementations or approaches largely based on its idea of CFG enforcement. The CFI concept was originally introduced by Abadi *et al.* [4], and has been largely explored for protecting user space software [10, 22, 44, 108, 110, 131, 235, 236].

kCFI is a kernel CFI mechanism designed to support OS features and intricacies, like system calls, LKMs and mixed Assembly/C source code. Despite being inspired by the original, user space centered, proposal by Abadi *et al.* [4], kCFI makes better use of architectural traits, generating close to zero memory contention. Additionally, kCFI introduces CGD, as an alternative for increasing CFI restrictiveness for the cases where shadow stacks are not feasible.

Coarse-grained CFI for OSes

KCoFI [41] was the first solution to fully implement CFI for OS kernels. This solution employs a tag-based coarse-grained mechanism built on top of the secure execution layer SVA [42]. Besides being insufficient as a security mechanism for enforcing a weak policy that was proved incapable of breaking control-flow hijacking attacks [18, 45, 67, 68, 161], the overheads introduced by the scheme range from 2x to 3.5x, what is considered high for a central system component as the kernel.

When compared to KCoFI, kCFI is more efficient both in terms of performance and security — it directly instruments code with fine-grained assembly primitives which are

more restrictive than coarse-grained approaches and much less costly than approaches dependent on execution layers. While not susceptible to attacks against coarse-grained CFI, kCFI can also be considered more robust than KCoFI, as it was implemented and evaluated over the Linux kernel while KCoFI limits its coverage to the less complex code-base which is the FreeBSD OS.

Fine-grained CFI for OSes

HyperSafe [226] employs *non-bypassable memory lockdown* and *restricted pointer indexing* to implement CFI for hypervisors. While memory lockdown protects the hypervisor code and static data from being compromised, restricted pointer indexing creates a code structure that enforces all indirect branches to target addresses present in a pre-computed destination table.

While HyperSafe is restricted to hypervisors, a similar approach is employed by Ge *et al.* [65] to implement fine-grained CFI for kernel software. As the restricted pointer indexing approach requires a pre-computation of valid target addresses, this scheme inherently breaks LKM support, which is an important feature of most modern kernels. Besides, achieving good performance in this implementation requires amortizing the costs introduced by execution flow indirections, what is done through code optimizations that may not be applicable in a given context and that are more scarce on larger code bases. In fact, Ge's implementation restricts its support to the OSes FreeBSD and MINIX.

kCFI implements fine-grained CFI for kernels, being capable of supporting the Linux kernel with its basic OS functionalities, such as LKMs. For this reason, kCFI can be considered more robust and compliant to the OS requirements than Ge's implementation, that only supports FreeBSD and MINIX and completely precludes LKMs. On what concerns the enforced restrictiveness, Ge's implementation is presented with smaller AIA values than kCFI. Yet, we argue that the code-base used for computing the metric is much smaller than the one used in kCFI (Linux), turning unfair the direct comparison through AIA. Additionally, if extended with a shadow stack as the one describe in Chapter 5, kCFI becomes more restrictive than Ge's implementation, even under the unfair criteria comparison. Performance-wise, kCFI is slightly better than Ge's implementation and such efficiency is achieved without the dependency on code optimizations, thus, being less coupled to the underlying code-base.

Return-Address Protection

The Linux patch set PaX [173] implements a solution called RAP [174] as a way of tackling control-flow hijacking attacks. This solution reserves a general-use register for keeping an XOR cookie, which is used as an encryption key during kernel execution. Although the scheme protects itself against brute-force attacks by using different cookies for different running kernel threads, this measure is not enough as it may still be exploitable through memory disclosure attacks that can unveil encrypted values on the stack and enable the derivation of the cookie.

kCFI does not rely on secrecy in any aspect — it remains a solid defense even in the presence of memory disclosure bugs. If not compromised, the granularity enforced

by RAP is equivalent to that respective to shadow stacks, imposing similar protection to kCFI when it is extended with one. Also, RAP requires a register to be reserved for its exclusive use, what leads to less efficient code as the compiler's register allocation will be required to perform spills more frequently. Performance numbers regarding RAP are either from protected user space applications or not specific enough when referring to kernel protection[1], precluding a direct performance comparison between both mechanisms.

Hardware-enforced CFI

CET [38] is an architecture extension proposed by Intel to provide hardware enforced CFI. CET includes a transparent shadow stack structure which is implicit to `call` and `ret` instructions, providing efficient and simple protection for return edges. To enable forward-edge protection, CET introduces new instructions to be used for marking valid targets for indirect calls, enabling compiler-supported coarse-grained CFI. As CET was not yet released in any Intel product, little is known regarding its overheads, OS compatibility and support for features like `setjmp/longjmp` or code optimizations.

kCFI imposes finer-granularity on what concerns forward-edge protection, being, overall, more restrictive than CET when extended with a shadow stack. In fact, the coarse-grained model used by CET for protecting forward-edges was already proven flawed [18, 45, 67, 68, 161]. Additionally, CET is a vendor-specific x86 extension whose functionalities were not announced by any other fabricant – its benefits are limited to specific CPUs and cannot be used for the protection of legacy systems.

Cryptographic CFI

Cryptographic CFI [108] is a method based on the computation of *Message Authentication Codes* (MACs) for each code pointer. Through verifying the MAC prior to using the pointer it is possible to prevent the use of corrupted values. MAC computation also uses the memory address in which the pointer is stored, preventing the use of a pointer copied from a different location along with its MAC. For performance, this method uses cryptographic hardware instructions.

Despite its efficiency in terms of granularity, this approach presents some limitations. First, it remains vulnerable to replay attacks that use a pointer which was previously stored on the attacked address. For using cryptographic hardware extesions, this method reserves the respective registers to store keys, preventing them from being used along with cryptographic functionalities in different contexts. Finally, as the MAC computation uses memory addresses as one of its inputs, pointers copied through functions like `memcpy` will have its MAC invalidated. In the kernel context, the last two issues are critical, as the first breaks compatibility with handwritten Assembly functions that use the mentioned hardware resources and `memcpy`-like operations are frequent during kernel execution.

[1] In the kernel setting, a 25% overhead is reported for when the kernel is fully protected, but information regarding kernel version and which benchmarks were used during the tests is not provided

Kernel Shadow Stack

DRK [56] is a security tool that employs DBT for instrumenting kernel entry points dynamically. Through this instrumentation, DRK builds a shadow memory of kernel data structures, including the stack, enabling the detection of corrupted return addresses. Although this method does not require source code recompilation, shadow memory instrumentation introduces very high overheads that can reach 10x when compared to native performance, being unfeasible for real world scenarios.

The kernel shadow stack design presented in Section 5 is followed by a realistic cost analysis which indicates that the model proposed outperforms DRK significantly. Both implementations are equivalent in terms of restrictiveness.

The work by Dang *et al.* [43] describes an approach called *parallel shadow stacks* in which the stack pointer respective to the regular stack is incremented by an offset and reused to access the shadow stack structure, that is in a fixed known position. The approach is implemented on user space applications and shown to be the most efficient form for accessing a shadow stack.

The design we propose employs a similar approach, providing a similarly efficient access to the kernel shadow stacks. Yet, although the mentioned work affirms that shadow stack performance cannot be substantially improved, we go further and optimize it by taking advantage from tail call elimination to prevent the execution of unnecessary shadow stack operations.

Fine-grained CFI Attacks

As previously discussed in Section 2.3.2, Fine-grained CFI may be vulnerable to attacks under very particular circumstances. Although the described attacks do redirect control-flow, they perform such redirection in a way that do not diverge the enforced CFG, thus, not causing policy violations. In Section 4.2 we argue that further understanding regarding the feasibility of these attacks in the kernel context is required, illustrating that, for example, the printf-oriented programming technique employed in the Control-Flow Bending [17] attack is not applicable in the setting due to differences between `printf()` and `printk()`.

Even if we assume that the kernel is vulnerable to these attacks, we state that the solutions we propose raise the bar for successful exploitation. As demonstrated by Carlini *et al.* [17], shadow stacks stand as a substantial defense against Control-Flow Bending attacks, as these mechanisms protect against return address corruption and prevent attackers from backwardly traversing the enforced CFG. In the absence of a shadow stack, kCFI also makes such task harder, as it creates more restrictive CFGs through the use of the CGD technique. Although the attack demonstrated by Evans *et al.* [53] cannot be prevented through a shadow stack[2], the CFG enforced during their attack is different from ours and the example exposed would not work against code protected through our heuristics. Whereas this fact does not fully close the problem, it highlights that there exist room for improvement on what concerns forward-edge restrictiveness, either through

[2]For being based on a forward-edge corruption

heuristics, source-code annotation or even through requiring code compliancy in a way to allow more precise pointer analyses. In fact, kCFI stands as a solid framework for developing this study in the kernel setting.

Chapter 7

Conclusions

> We can only see a short distance ahead, but we can see plenty there that needs to be done
>
> *Alan Turing*

In this thesis we investigated the hypothesis that the construction of fine-grained CFI can be leveraged through multi-level program analyses in a way to enable efficient implementations for the kernel setting, not precluding code coverage, OS functionalities and good performance.

To this end, we first presented kCFI, a fine-grained tag-based CFI implementation capable of supporting the Linux kernel and protecting it against modern control-flow hijacking techniques, including ROP attacks. kCFI works by instrumenting the kernel source code with indirect branch assertions that verify the validity of a control-transfer before its execution. kCFI's design takes advantage of traits in the x86-64 architecture, introducing only low-cost memory operations and making use of instructions with negligible overhead to mark code – kCFI outperforms previously implemented CFI mechanisms for the kernel setting, presenting overheads of 8% and 2% respectively on micro and macro benchmarks. Unlike previous approaches, kCFI achieves its goals without using restricted pointer indexing or converting indirect branches into direct ones. For this reason, kCFI does not harm system features and is the first kernel CFI approach to support LKMs. On top of that we also state that kCFI is robust to the point of fully supporting the Linux kernel, being the first implementation to fully implement and evaluate fine-grained CFI in this OS.

Additionally, we pave the way for an OS shadow stack implementation through the presentation of a first study on the design and performance of these mechanisms in the kernel setting. The proposed design is realistic considering the internals of Linux, only depending on features that already exist or can be extended to provide the needed resources. In this study we show that the overheads introduced by the shadow stack can be significantly lowered through the employment of better self-protection strategies — in fact, we demonstrate that for the shadow stack scenario, where access permissions must be consecutively reseted, memory pages can be efficiently protected by using a hardware feature originally developed for address space isolation.

7.1 List of Publications

Throughout the development of the research that resulted in this Thesis, we have published the papers listed below. The first two papers cover an implementation of user space shadow stacks through the use of DBT tools, which is detailed in the Appendix A. FLOW, a kernel CFI prototype that later became kCFI, was described in a paper that was published and awarded as best paper on SBSeg 2016. As FLOW was later extended with CGD, Assembly support and `nopl`-based tags, this new version was described in a paper which is currently under review.

- João Moreira, Divino Lucas, Guido Araujo, Edson Borin, Sandro Rigo. Asynchronous Program Flow Verification Through Binary Instrumentation in QEMU. In Workshop on Architectural and Microarchitectural Support for Binary Translation 2012. - AMAS-BT 2012.

- João Moreira, Lucas Teixeira, Edson Borin, Sandro Rigo. Leveraging Optimization Methods for Dynamically Assisted Control-Flow Integrity Mechanisms. In 26th International Symposium on Computer Architecture and High Performance Computing 2014 - SBAC-PAD 2014.

- João Moreira, Sandro Rigo. Go With the FLOW: Fine-Grained Control-Flow Integrity for the Kernel. In XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2016. **Best paper award**

Under review:

- João Moreira, Sandro Rigo, Michalis Polychronakis, Vasileios Kemerlis. kCFI: Fine-Grained Control-Flow Integrity for Commodity Operating System Kernels. In Elsevier Computers & Security Journal.

7.2 Future Directions

Considering statistics from public vulnerability databases [39, 125], it is unrealistic to think that memory corruption bugs will cease from existing anytime soon, or even in a distant future. Memory-safe programming languages, like Java or Ruby, aren't capable of covering the full set of domains to which software is applied. Especially in low-level settings, like the OS kernel or embedded systems, the use of languages that do not provide memory access abstractions is still a constant that occasionally leads to the occurrence of exploitable programming bugs. Yet, modern methods for identification and prevention of software bugs in unsafe programming languages [145] remain distant from fully closing the matter.

Given this context, we believe that methods for software self-protection are a promising research field, specially on what concerns finding or improving efficient solutions for accurately defeating attacks without depending on the superposition of new methods. In this sense, we envision that it is better to look for solid and concise methods, fitted to a

particular domain and that do not trade effectiveness for generality. We understand that CFI is an attempt in this direction, as it defends against control-flow hijacking attacks irrespectively to the presence of other security augmentations, only depending on execution environments that respect policies like W^X and address space isolation. This notion is even stronger when we think about a CFI mechanism that is specific to the kernel setting. A clear example of what we intend to avoid is the fragile relationship between ASLR and stack canaries, in which bypassing the first consequently diminishes the effectiveness of the second.

Control-Flow Integrity

Accordingly to our vision, we intend to pursue efficiency while enforcing control-flow correctness. Specifically, we plan to improve CFI methods in a way to make them more accurate and reliable, preventing possible circumventions. The first logical step towards this direction is finishing a *de facto* shadow stack implementation for the kernel, providing the most accurate CFI mechanism in terms of return granularity. This implementation must take into account the introduction of the CET [38] extension, recently proposed by Intel and much expected by the security community.

On what concerns forward-edge protection, we envision a few improvements to kCFI. First, it is possible to combine the already employed heuristics with program analyses to reach narrower CFGs. To this end, pointer analyses and source code information can be used to identify indirectly unreachable targets, enabling the reduction of valid target sets. Simultaneously, we did not explore any dynamic approach for computing or pruning the used CFGs, what leaves an interesting open topic. Finally, we also consider the possibility of developing user-assisted identification of valid targets through the introduction of new function attributes.

Non-Control Data Attacks Mitigation

As previously exposed, non-control-data attacks remain a threat against software. Considering this fact, an important direction to pursue is the early identification and possible mitigation of these attacks. We believe that by employing multi-level analyses methods, as done in kCFI, it may be possible to tackle these problems. To this end, we envision the addition of relevant dynamic methods like runtime profiling, fuzzing [228] and *Data Flow Tracking*-based analyses [88] on top of the already employed analyses.

Operating Systems Security

Considering the central relevance of the kernel as a system component and that critical security flaws remain submerged in its internals, we understand that this is a meaningful ground for security improvements. Along our future work we intend to investigate how new features and hardware extensions may influence the kernel security. In the near future we expect the release of CET and MPK [35, 38] extensions to evaluate how our solutions stand or can be improved in the respective new systems.

Bibliography

[1] Control Flow Guard. `https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx`.

[2] Control Flow Integrity. `http://clang.llvm.org/docs/ControlFlowIntegrity.html`.

[3] *Non-Control-Data Attacks Are Realistic Threats*. USENIX, August 2005.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[5] Anonymous. Once upon a free(). `"http://www.phrack.org/issues.html?issue=57&id=9&mode=txt`, 2001. [Online; accessed 2-August-2016].

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, May 2000.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[8] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[9] blackngel. MALLOC DES-MALEFICARUM. *Phrack*, 66, 2009. [Online; accessed 2-August-2016].

[10] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating Code-Reuse Attacks with Control-Flow Locking. In *Proc. of ACSAC*, pages 353–362, 2011.

[11] 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, pages 30–40, New York, NY, USA, 2011. ACM.

[12] Marcus Botacin, Paulo de Geus, and André Grégio. Detecção de ataques por rop em tempo real assistida por hardware. In *SBSeg 2016 - Artigos completos ()*, Niterói, nov 2016.

[13] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2000.

[14] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[15] 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, pages 27–38, New York, NY, USA, 2008. ACM.

[16] Kil3r Bulba. Bypassing stackguard and stackshield. `"http://phrack.org/issues.html?issue=56&id=5"`, May 2000. [Online; accessed 2-August-2016].

[17] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[18] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association.

[19] 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, pages 559–572, New York, NY, USA, 2010. ACM.

[20] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[21] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. *Computers and Communications, IEEE Symposium on*, 0:749–754, 2006.

[22] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *NDSS*. The Internet Society, 2014.

[23] Tzi-Cker Chiueh and Fu-Hau Hsu. Rad: a compile-time solution to buffer overflow attacks. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 409–417, Apr 2001.

[24] Stephen Chong. Static single assignment form (and dominators, post-dominators, dominance frontiers?). http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec04-SSA.pdf, 2011. [Online; accessed 2-August-2016].

[25] Common Vulnerabilities and Exposures. CVE-2009-3234. `"https://cve.mitre.org/cfgi-bin/cvename.cfgi?name=CVE-2009-3234"`, September 2009. [Online; accessed 15-November-2016].

[26] Common Vulnerabilities and Exposures. CVE-2010-3301. `"https://cve.mitre.org/cfgi-bin/cvename.cfgi?name=CVE-2010-3301"`, September 2010. [Online; accessed 15-November-2016].

[27] Common Vulnerabilities and Exposures. CVE-2010-3904. `"https://cve.mitre.org/cfgi-bin/cvename.cfgi?name=CVE-2010-3904"`, October 2010. [Online; accessed 15-November-2016].

[28] Common Vulnerabilities and Exposures. CVE-2015-3036. `"https://cve.mitre.org/cfgi-bin/cvename.cfgi?name=CVE-2015-3036"`, April 2015. [Online; accessed 15-November-2016].

[29] Common Vulnerabilities and Exposures. CVE-2015-3290. `"https://cve.mitre.org/cfgi-bin/cvename.cfgi?name=CVE-2015-3290"`, April 2015. [Online; accessed 15-November-2016].

[30] Common Vulnerabilities Exposure. CVE-2010-3437. `"https://cve.mitre.org/cfgi-bin/cvename.cfgi?name=CVE-2010-3437"`, September 2010. [Online; accessed 15-November-2016].

[31] Common Vulnerabilities Exposure. CVE-2013-6282. `"https://cve.mitre.org/cfgi-bin/cvename.cfgi?name=CVE-2013-6282"`, October 2013. [Online; accessed 15-November-2016].

[32] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, New York, NY, USA, 2015. ACM.

[33] Jonathan Corbet. An updated guide to `debugfs`. `http://lwn.net/Articles/334546/`, May 2009.

[34] Jonathan Corbet. Supervisor mode access prevention. `http://lwn.net/Articles/517475/`, October 2012.

[35] Jonathan Corbet. Memory protection keys. "https://lwn.net/Articles/643797/", 3 2015. [Online; accessed 7-August-2016].

[36] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual (volume 1). "http://www.intel.com.br/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf", 9 2016. [Online; accessed 7-November-2016].

[37] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual (volume 3). "http://www.intel.com.br/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf", 9 2016. [Online; accessed 7-November-2016].

[38] Intel® Corporation. Control-flow enforcement technology preview. "https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf", 6 2016. [Online; accessed 7-August-2016].

[39] The MITRE Corporation. Common vulnerabilities and exposures. https://cve.mitre.org/. [Online; accessed 2-August-2016].

[40] 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 conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[41] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, pages 292–307, May 2014.

[42] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 351–366, New York, NY, USA, 2007. ACM.

[43] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[44] 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 *Proc. of NDSS*, 2012.

[45] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[46] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63`, August 1997.

[47] Advanced Micro Devices. AMD-V Nested Paging. Technical report, Advanced Micro Devices, July 2008.

[48] Yaozu Dong, Shaofan Li, Asit Mallick, Jun Nakajima, Kun Tian, Xuefei Xu, Fred Yang, and Wilfred Yu. Extending Xen with Intel Virtualization Technology. *Intel Technology Journal*, 10(03):193–204, 2006.

[49] Tyler Durden. Bypassing pax aslr protection. `"http://www.phrack.org/issues.html?issue=59&id=9#article"`, Jul 2002. [Online; accessed 2-August-2016].

[50] DynamoRio. Dynamorio. `"http://www.dynamorio.org/"`, Apr 2012. [Online; accessed 2-August-2016].

[51] Jake Edge. Kernel address space layout randomization. `"https://lwn.net/Articles/569635/"`, 9 2013. [Online; accessed 2-August-2016].

[52] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro. The cornell commission: on morris and the worm. *Commun. ACM*, 32(6):706–709, June 1989.

[53] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, New York, NY, USA, 2015. ACM.

[54] Exploit Database. EBD-31346, February 2014.

[55] Exploit Database. EBD-33516, May 2014.

[56] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 135–146, New York, NY, USA, 2012. ACM.

[57] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 62–, Washington, DC, USA, 2003. IEEE Computer Society.

[58] James C. Foster. *Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals*, chapter 12, pages 517–518. Syngress, April 2005.

[59] Free Software Foundation. Gcc, the gnu compiler collection. `https://gcc.gnu.org/`. [Online; accessed 2-August-2016].

[60] Free Software Foundation. Gnu operating system. `https://www.gnu.org/`. [Online; accessed 2-August-2016].

[61] Mike Frantzen and Mike Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

[62] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, October 2003.

[63] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[64] Robert Gawlik and Thorsten Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proc. of ACSAC*, pages 396–405, 2014.

[65] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-grained control-flow integrity for kernel software. In *IEEE European Symposium on Security and Privacy 2016*, Euro S&P, Washington, USA, 2016. IEEE Computer Society.

[66] Varghese Geroge, Tom Piazza, and Hong Jiang. Technology insight: Intel® next generation microarchitecture codename ivy bridge. `"http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf"`, 9 2011. [Online; accessed 7-August-2016].

[67] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, Washington, DC, USA, 2014. IEEE Computer Society.

[68] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA, August 2014. USENIX Association.

[69] Trusted Computing Group. Tpm main specification. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, March 2011. [Online; accessed 2-August-2016].

[70] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3 – 14, dec. 2001.

[71] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 155–174, New York, NY, USA, 2009. ACM.

[72] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. ShrinkWrap: VTable Protection Without Loose Ends. In *Proc. of ACSAC*, pages 341–350, 2015.

[73] Dave Hansen. Intel® memory protection extensions (intel® mpx) for linux*. `"https://01.org/blogs/2016/intel-mpx-linux"`, 3 2016. [Online; accessed 2-August-2016].

[74] Kim Hazelwood. *Dynamic Binary Modification: Tools, Techniques, and Applications*. Morgan & Claypool Publishers, March 2011.

[75] John L. Henning. SPECCPU 2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[76] Greg Hoglund and Gary Mcgraw. *Exploiting Software : How to Break Code*. Addison-Wesley Professional, February 2004.

[77] 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*, SSYM'09, pages 383–398, Berkeley, CA, USA, 2009. USENIX Association.

[78] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proc. of NDSS*, 2014.

[79] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 235–246, New York, NY, USA, 2013. ACM.

[80] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection and monitoring through vmm-based out-of-the-box semantic view reconstruction. *ACM Trans. Inf. Syst. Secur.*, 13(2):12:1–12:28, March 2010.

[81] jp. Advanced doug lea's malloc exploits. `"http://phrack.org/issues/61/6.html`, 2003. [Online; accessed 2-August-2016].

[82] Michel Kaempf. Vudo - An object superstitiously believed to embody magical powers. *Phrack*, 57, 2001. [Online; accessed 2-August-2016].

[83] 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, pages 272–280, New York, NY, USA, 2003. ACM.

[84] AK Keith Owens. Kernel stacks on x86-64 bit. https://www.kernel.org/doc/Documentation/x86/kernel-stacks. [Online; accessed 2-August-2016].

[85] Vasileios P. Kemerlis. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. Phd thesis, Columbia University, 07 2015.

[86] Vasileios P. Kemerlis. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. PhD thesis, Columbia University, 7 2015.

[87] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 957–972, San Diego, CA, August 2014. USENIX Association.

[88] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[89] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 459–474, Bellevue, WA, 2012. USENIX.

[90] Thomas J. Killian. Processes as Files. In *Proc. of USENIX Summer*, pages 203–207, 1984.

[91] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Execution model enforcement via program shepherding. Master's thesis, Massachusetts Institute of Technology, 2003.

[92] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[93] Thomas Kittel. Design and implementation of a virtual machine introspection based intrusion detection system. Diploma thesis, Technische Unversität München, October 2010.

[94] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[95] Kenichi Kourai and Shigeru Chiba. Hyperspector: virtual distributed monitoring environments for secure intrusion detection. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 197–207, New York, NY, USA, 2005. ACM.

[96] Greg Kroah-Hartman. `udev` – A Userspace Implementation of devfs. In *Proc. of OLS*, pages 263–271, 2003.

[97] Michael Larabel and Matthew Tippett. Phoronix Test Suite. `http://www.phoronix-test-suite.com/`, 2011. [Online; acesso 07/08/2016].

[98] Mike Larkin. Kernel W^X Improvements In OpenBSD. In *Hackfest*, 2015.

[99] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, Washington, USA, 2004. IEEE Computer Society.

[100] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[101] 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)*, 2010.

[102] Siarhei Liakh. NX protection for kernel data. `http://lwn.net/Articles/342266/`, July 2009.

[103] LLVMLinux. `http://llvm.linuxfoundation.org/`. [Online; accessed 2-August-2016].

[104] Linux Kernel Mailing List. [PATCH] ACPI: fix acpi_debugfs_init prototype. `https://lkml.org/lkml/2015/8/1/72`.

[105] Linux Kernel Mailing List. [tip:x86/mm] module: fix () used as prototype in include/linux/module.h. `https://lkml.org/lkml/2010/2/17/250`.

[106] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[107] H. Marco-Gisbert and I. Ripoll. Preventing brute force attacks against stack canary protection on networking servers. In *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 243–250, Aug 2013.

[108] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 941–951, New York, NY, USA, 2015. ACM.

[109] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[110] Zhang Mingwei and R. Sekar. Control flow integrity for cots binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., 2013. USENIX.

[111] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque control-flow integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, San Diego, California, February 2015.

[112] Ingo Molnar. Lkml: [announce] [patch] nx (no execute) support for x86, 2.6.7-rc2-bk2. `"http://lkml.iu.edu/hypermail/linux/kernel/0406.0/0497.html"`, 6 2004. [Online; accessed 7-August-2016].

[113] James Morse. arm64: kernel: Add support for Privileged Access Never. `https://lwn.net/Articles/651614/`, July 2015.

[114] K. Nance, M. Bishop, and B. Hay. Virtual machine introspection: Observation or interference? *Security Privacy, IEEE*, 6(5):32 –37, sept.-oct. 2008.

[115] National Vulnerability Database. Kernel Vulnerabilities. `https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&query=kernel&cvss_version=3`, October 2016. [Online; accessed 2-August-2016].

[116] D. Nebenzahl, M. Sagiv, and A. Wool. Install-time vaccination of windows executables to defend against stack smashing attacks. *IEEE Transactions on Dependable and Secure Computing*, 3(1):78–90, Jan 2006.

[117] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation.* Phd thesis, University of Cambridge, November 2004.

[118] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV03)*, 2003.

[119] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[120] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of NDSS '05*, San Diego, California, USA, February 2005.

[121] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In *Proceedings of the 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2013)*, pages 177–196, July 2013.

[122] Vitaly Nikolenko. Linux kernel rop - ropping your way to # (part 1). `https://www.trustwave.com/Resources/SpiderLabs-Blog/Linux-Kernel-ROP---Ropping-your-way-to---(Part-1)/`. [Online; accessed 16-August-2016].

[123] Ben Niu and Gang Tan. Modular Control-flow Integrity. In *Proc. of PLDI*, pages 577–587, 2014.

[124] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. In *Proc. of CCS*, pages 914–926, 2015.

[125] National Institute of Standards and Technology. National vulnerability database. `"https://nvd.nist.gov"`, Apr 2012. [Online; accessed 2-August-2016].

[126] 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, pages 49–58, New York, NY, USA, 2010. ACM.

[127] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. [Online; accessed 2-August-2016].

[128] Linux Kernel Organization. The linux kernel archives. `https://www.kernel.org/`. [Online; accessed 2-August-2016].

[129] Hewlett Packard. Data execution prevention. `"http://h10032.www1.hp.com/ctg/Manual/c00387685.pdf"`, 05 2005. [Online; accessed 7-August-2016].

[130] pakt. Ropc, a turing complete rop compiler. `https://github.com/pakt/ropc`. [Online; accessed 2-August-2016].

[131] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, Washington, D.C., 2013. USENIX.

[132] PaX Team. UDEREF/i386. `http://grsecurity.net/~spender/uderef.txt`, April 2007.

[133] PaX Team. UDEREF/amd64. `https://goo.gl/iPuOVZ`, April 2010.

[134] PaX Team. Better kernels with GCC plugins. `http://lwn.net/Articles/461811/`, October 2011.

[135] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-Grained Control-Flow Integrity through Binary Hardening. In *Proc. of DIMVA*, pages 144–164, 2015.

[136] Jannik Pewny and Thorsten Holz. Control-flow Restrictor: Compiler-based CFI for iOS. In *Proc. of ACSAC*, pages 309–318, 2013.

[137] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *Proceedings of the 1st ACM workshop on Virtual machine security*, VMSec '09, pages 1–10, New York, NY, USA, 2009. ACM.

[138] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Exploiting the x86 architecture to derive virtual machine state information. In *Proceedings of the Fourth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2010)*, pages 166–175, Venice, Italy, July 2010. IEEE Computer Society. Best Paper Award.

[139] Phrack. `http://www.phrack.org/`, 1985. [Online; accessed 2-August-2016].

[140] Georgios Portokalidis and Angelos D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 41–48, New York, NY, USA, 2010. ACM.

[141] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proc. of NDSS*, 2015.

[142] M. Prandini and M. Ramilli. Return-oriented programming. *Security Privacy, IEEE*, 10(6):84–87, Nov 2012.

[143] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *In Proceedings of the USENIX Annual Technical Conference*, pages 211–224, 2003.

[144] Clang Project. Clang: a c language family frontend for llvm. `http://clang.llvm.org/`. [Online; accessed 2-August-2016].

[145] Clang Project. Clang static analyzer. `http://clang-analyzer.llvm.org/`. [Online; accessed 2-August-2016].

[146] LLVM Project. opt - llvm optimizer. `http://llvm.org/docs/CommandGuide/opt.html`. [Online; accessed 2-August-2016].

[147] The Linux Documentation Project. Linux interprocess communication. `"http://www.tldp.org/LDP/tlk/ipc/ipc.html"`, 3 1996. [Online; accessed 7-November-2016].

[148] PUC-Rio. The Programming Language Lua. `http://www.lua.org`.

[149] QEMU. Qemu tcg readme. "http://git.qemu.org/?p=qemu.git;a=blob;f=tcg/tci/README". [Online; accessed 2-August-2016].

[150] QEMU. Qemu. "http://wiki.qemu.org/Main_Page", Apr 2012. [Online; accessed 2-August-2016].

[151] Ramu Ramakesavan, Dan Zimmerman, Pavithra Singaravelu, George Kuan, Brian Vajda, Scott Gibbons, and Gautham Beeraka. Intel memory protection extensions enabling guide. "https://software.intel.com/sites/default/files/managed/9d/f6/Intel_MPX_EnablingGuide.pdf", 4 2016. [Online; accessed 7-August-2016].

[152] G. Ramalingam. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[153] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[154] Dan Rosenberg. kptr_restrict for hiding kernel pointers. http://lwn.net/Articles/420403/, December 2010.

[155] Emílio Rubens, Mateus Tymburibá, and Fernando Pereira. Inferência estática da frequência máxima de instruções de retorno para detecção de ataques rop. In *Simpsósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, SBSEG XV, 2015.

[156] Joanna Rutkowska. Introducing stealth malware taxonomy. "http://blog.invisiblethings.org/papers/2006/rutkowska_malware_taxonomy.pdf", November 2006. [Online; accessed 2-August-2016].

[157] Tasker P. S. Trusted computer systems. In *In Proc. IEEE Symp. Sec. and Privacy*, pages 99–100, 1981.

[158] Jonathan Salwan. Ropgadget - gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/. [Online; accessed 2-August-2016].

[159] Karen Scarfone, Karen Scarfone, Scarfone Cybersecurity, Peter Mell, Rebecca M. Blank, and Acting Secretary. Guide to intrusion detection and prevention systems (idps), 2007.

[160] Karen Scarfone, Karen Scarfone, Scarfone Cybersecurity, Peter Mell, Rebecca M. Blank, and Acting Secretary. Guide to intrusion detection and prevention systems (idps), 2007.

[161] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Proc. of RAID*, pages 88–108, 2014.

[162] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 209–, Washington, DC, USA, 2002. IEEE Computer Society.

[163] Scut and Team Teso. Exploiting Format String Vulnerabilities. Technical report, March 2001.

[164] 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, pages 552–561, New York, NY, USA, 2007. ACM.

[165] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[166] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[167] 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 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

[168] Brad Spengler. Recent ARM security improvements. `https://goo.gl/H8HkwD`, February 2013.

[169] Brad Spengler. Enlightenment Linux Kernel Exploitation Framework. `https://grsecurity.net/~spender/exploits/enlightenment.tgz`, December 2014.

[170] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.

[171] Andrew S. Tanenbaum. *Modern Operating Systems*, pages 60–61. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[172] PaX Team. Address space layout randomization. `"https://pax.grsecurity.net/docs/aslr.txt"`, Mar 2003. [Online; accessed 2-August-2016].

[173] PaX Team. Pax. `"https://pax.grsecurity.net"`, Apr 2013. [Online; accessed 2-August-2016].

[174] PaX Team. Rap: Rip rop. "https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf", 10 2015. [Online; accessed 7-August-2016].

[175] PaX Team. Close, but no cigar: On the effectiveness of intel's cet against reuse attacks. "https://forums.grsecurity.net/viewtopic.php?f=7&t=4490", 6 2016. [Online; accessed 7-August-2016].

[176] 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 *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[177] 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 *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[178] Mateus Tymburibá, Rubens E. A. Moreira, and Fernando Magno Quintão Pereira. Inference of peak density of indirect branches to detect rop attacks. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 150–159, New York, NY, USA, 2016. ACM.

[179] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48 – 56, may 2005.

[180] Arjan van de Ven. Debug option to write-protect rodata: the write protect logic and config option. http://lkml.indiana.edu/hypermail/linux/kernel/0511.0/2165.html, November 2005.

[181] Arjan van de Ven. Add `-fstack-protector` support to the kernel. http://lwn.net/Articles/193307/, July 2006.

[182] Vendicator. Stackshield. "http://www.angelfire.com/sk/stackshield/", 1 2000. [Online; accessed 7-August-2016].

[183] VMware. VMware fusion. https://www.vmware.com/products/fusion/. [Online; accessed 2-August-2016].

[184] VMware. VMware player. https://www.vmware.com/products/player/. [Online; accessed 2-August-2016].

[185] VMware. VMware workstation. https://www.vmware.com/products/workstation/. [Online; accessed 2-August-2016].

[186] Sebastian Vogl. A bottom-up approach to VMI-based kernel-level rootkit detection. Diploma thesis, Technische Unversität München, October 2010.

[187] Common Vulnerabilities and Exposures. Cve-2005-0736. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0736", 2005. [Online; accessed 7-August-2016].

[188] Common Vulnerabilities and Exposures. Cve-2005-2555. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-2555", 2005. [Online; accessed 7-August-2016].

[189] Common Vulnerabilities and Exposures. Cve-2007-4315. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4315", 2007. [Online; accessed 7-August-2016].

[190] Common Vulnerabilities and Exposures. Cve-2008-1335. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1335", 2008. [Online; accessed 7-August-2016].

[191] Common Vulnerabilities and Exposures. Cve-2008-3875. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3875", 2008. [Online; accessed 7-August-2016].

[192] Common Vulnerabilities and Exposures. Cve-2009-1897. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897", 2009. [Online; accessed 7-August-2016].

[193] Common Vulnerabilities and Exposures. Cve-2009-2675. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2675", 2009. [Online; accessed 7-August-2016].

[194] Common Vulnerabilities and Exposures. Cve-2009-2692. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692", 2009. [Online; accessed 7-August-2016].

[195] Common Vulnerabilities and Exposures. Cve-2009-2698. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2698", 2009. [Online; accessed 7-August-2016].

[196] Common Vulnerabilities and Exposures. Cve-2009-3002. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3002", 2009. [Online; accessed 7-August-2016].

[197] Common Vulnerabilities and Exposures. Cve-2009-3547. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3547", 2009. [Online; accessed 7-August-2016].

[198] Common Vulnerabilities and Exposures. Cve-2010-2959. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2959", 2010. [Online; accessed 7-August-2016].

[199] Common Vulnerabilities and Exposures. Cve-2010-3437. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3437", 2010. [Online; accessed 7-August-2016].

[200] Common Vulnerabilities and Exposures. Cve-2010-4073. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4073", 2010. [Online; accessed 7-August-2016].

[201] Common Vulnerabilities and Exposures. Cve-2010-4347. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4347", 2010. [Online; accessed 7-August-2016].

[202] Common Vulnerabilities and Exposures. Cve-2012-4530. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4530", 2012. [Online; accessed 7-August-2016].

[203] Common Vulnerabilities and Exposures. Cve-2013-0349. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0349", 2013. [Online; accessed 7-August-2016].

[204] Common Vulnerabilities and Exposures. Cve-2013-0402. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0402", 2013. [Online; accessed 7-August-2016].

[205] Common Vulnerabilities and Exposures. Cve-2013-1848. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1848", 2013. [Online; accessed 7-August-2016].

[206] Common Vulnerabilities and Exposures. Cve-2013-2851. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2851", 2013. [Online; accessed 7-August-2016].

[207] Common Vulnerabilities and Exposures. Cve-2013-2852. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2852", 2013. [Online; accessed 7-August-2016].

[208] Common Vulnerabilities and Exposures. Cve-2013-3136. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3136", 2013. [Online; accessed 7-August-2016].

[209] Common Vulnerabilities and Exposures. Cve-2013-4164. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4164", 2013. [Online; accessed 7-August-2016].

[210] Common Vulnerabilities and Exposures. Cve-2015-0570. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0570", 2015. [Online; accessed 7-August-2016].

[211] Common Vulnerabilities and Exposures. Cve-2015-2666. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2666", 2015. [Online; accessed 7-August-2016].

[212] Common Vulnerabilities and Exposures. Cve-2015-3036. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3036", 2015. [Online; accessed 7-August-2016].

[213] Common Vulnerabilities and Exposures. Cve-2016-0264. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0264", 2016. [Online; accessed 7-August-2016].

[214] Common Vulnerabilities and Exposures. Cve-2016-0859. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0859", 2016. [Online; accessed 7-August-2016].

[215] Common Vulnerabilities and Exposures. Cve-2016-1790. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1790", 2016. [Online; accessed 7-August-2016].

[216] Common Vulnerabilities and Exposures. Cve-2016-1887. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1887", 2016. [Online; accessed 7-August-2016].

[217] Common Vulnerabilities and Exposures. Cve-2016-2068. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2068", 2016. [Online; accessed 7-August-2016].

[218] Common Vulnerabilities and Exposures. Cve-2016-2117. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2117", 2016. [Online; accessed 7-August-2016].

[219] Common Vulnerabilities and Exposures. Cve-2016-3256. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3256", 2016. [Online; accessed 7-August-2016].

[220] Common Vulnerabilities and Exposures. Cve-2016-3272. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3272", 2016. [Online; accessed 7-August-2016].

[221] Common Vulnerabilities and Exposures. Cve-2016-4653. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4653", 2016. [Online; accessed 7-August-2016].

[222] Common Vulnerabilities and Exposures. Cve-2016-4794. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4794", 2016. [Online; accessed 7-August-2016].

[223] Common Vulnerabilities and Exposures. Cve-2016-4805. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4805", 2016. [Online; accessed 7-August-2016].

[224] Common Vulnerabilities and Exposures. Cve-2016-5829. "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5829", 2016. [Online; accessed 7-August-2016].

[225] Perry Wagle and Crispin Cowan. Stackguard: Simple stack smash protection for gcc. In *Proc. of the GCC Developers Summit*, pages 243–255, 2003.

[226] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395, May 2010.

[227] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *In Proceedings of the USENIX Annual Technical Conference*, 2002.

[228] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 511–522, New York, NY, USA, 2013. ACM.

[229] Jun Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 260–269, Oct 2003.

[230] 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. *Commun. ACM*, 53(1):91–99, January 2010.

[231] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

[232] Fenghua Yu. Enable/Disable Supervisor Mode Execution Protection. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=de5397ad5b9ad22e2401c4dacdf1bb3b19c05679, May 2011.

[233] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proc. of CCS*, pages 29–40, 2011.

[234] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proc. of NDSS*, 2015.

[235] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Stephen McCamant, and Laszlo Szekeres. Protecting Function Pointers in Binary. In *Proc. of ASIACCS*, pages 487–492, 2013.

[236] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, Washington, DC, USA, 2013. IEEE Computer Society.

[237] Qin Zhao, Ioana Cutcutache, and Weng-Fai Wong. Pipa: Pipelined profiling and analysis on multi-core systems. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 185–194, New York, NY, USA, 2008. ACM.

[238] Dino Zovi. Practical return-oriented programming. `https://www.trailofbits.com/resources/practical_rop_slides.pdf`. [Online; accessed 2-August-2016].

Appendix A

A DBI-Based Shadow Stack

A.1 Introduction

Many of the solutions proposed to fix security problems are based on recompilation or source code modification, what is unfeasible under contexts where programs are distributed in the binary form. Besides, it is not a novelty that applications are frequently distributed with intentionally hidden flaws. As DBI tools operate programs directly at the binary level, these tools are seen as an interesting enabler for security enhancement under these specific contexts. In this Appendix we present a shadow stack implementation for user space applications built on top of a DBI tool.

The proposed shadow stack implementation, which we call LORD, employs an asynchronous design capable of separating the analysis overheads into different execution threads without stalling the protected application, decoupling the translation-implicit and protection costs. LORD was implemented as an experiment on shadow stacks design and was developed in the early days of the research presented in this thesis. Despite being implemented in a different domain, these experiments leveraged our knowledge regarding the costs of DBI strategies employed on the implementation of hardening features, leading us to adopt more efficient approaches in terms of performance for achieving the requirements of a kernel design. As LORD is not a kernel solution, we fit it as an Appendix.

A.2 Design

Prior to designing our shadow stack, we built a DBI platform which supports asynchronous analyses of binaries running on top of it. To enable these analyses, the platform instruments code during runtime, adding code snippets that will forward information to externally plugged modules capable of leveraging information about the execution context.

Because communication also introduces overheads, we designed the tool to also support a hybrid approach — analysis can be performed both online and asynchronously, being prone to balancing accordingly to the context. Through exploring this architecture, we could understand better the cases in which the communication costs more than managing the data inside the platform.

When an application is running on top of our platform, the three main components of

the scheme are (i) the **Main process**, which is the application being instrumented and protected; (ii) the **Verifier**, which analyses information regarding the main process and (iii) **DBI platform**, which is responsible for running, translating and instrumenting the main process with functionalities that will analyze or forward information to be analyzed by the verifier process.

A.2.1 Asynchronous Approach vs Online Approach

Both asynchronous and online approaches introduce different difficulties and advantages to the analyses built on top of the platform. The contrast on the most relevant aspects of each approach is presented below.

Parallelizable architecture: By using an asynchronous scheme, the whole design is inherently parallelizable, being suitable for multi-processed and heterogeneous environments. This characteristic enables features such as (i) the exploitation of resources such as GPUs or idle processors in different machines on the same data-center; (ii) load balancing reconfiguration through the modification of the CPU time of the verifier processes whenever higher priority tasks need to be executed; (iii) capability of plugging new analysis routines to the verifier process without the need to restart the analyzed process and, finally, (iv) the possibility of analyzing processes that are executed on geographically distant machines, even after an extended period of time.

Decoupling analysis overhead from the main process. The asynchronous approach also decouples the verification costs from the main process execution, opening space for sophisticated analysis that would be too heavy for online execution.

Delay on reaction mechanisms. For being asynchronous, the identification of anomalies is delayed on this approach. The main process execution won't be halted until its validity is verified. This characteristic implies in different reaction mechanisms, focused on fixing attack consequences, instead of avoiding them.

Additional communication overheads. The communication between the two processes introduces additional overhead that is inexistent when using the online approach. To better understand these costs, we leverage the use of *pipes* and *shared memory* as a communication mechanism, even thought the later poses a slightly more invasive implementation, requiring additional verification of the communication interfaces to ensure its safety.

Memory isolation. The distributed design of the asynchronous approach, which employs different processes on the achievement of its tasks, inherently builds access barriers between its components. Depending on how the communication between these components is done, memory safety comes at no additional cost.

The described characteristics highlight that each approach is better under certain circumstances. As the DBI-generated code is translated during runtime, the whole scheme enables the implementation of a platform that can benefit from both, changing the employed strategy for performing the intended analyses based on context information. These guidelines led the development towards a hybrid, context-sensitive, platform.

A.3 mQEMU: Binary Instrumentation with QEMU

QEMU is an open source machine emulator capable of executing cross-platform applications and operating systems [8, 149, 150], thus being considered a Dynamic Binary Translation (DBT) tool. The DBT process in QEMU first translates each instruction of the original program into corresponding simpler instructions called micro-operations, storing them in an IR of the original code. After that, a dynamic code generator is invoked to replace the micro-operations with host instructions. Each micro-operation had its behavior coded in C and was previously compiled to be used by the dynamic code generator. In this process, that can be seen in Figure A.1, the host instructions are concatenated, forming translation blocks, which are stored in a code cache.

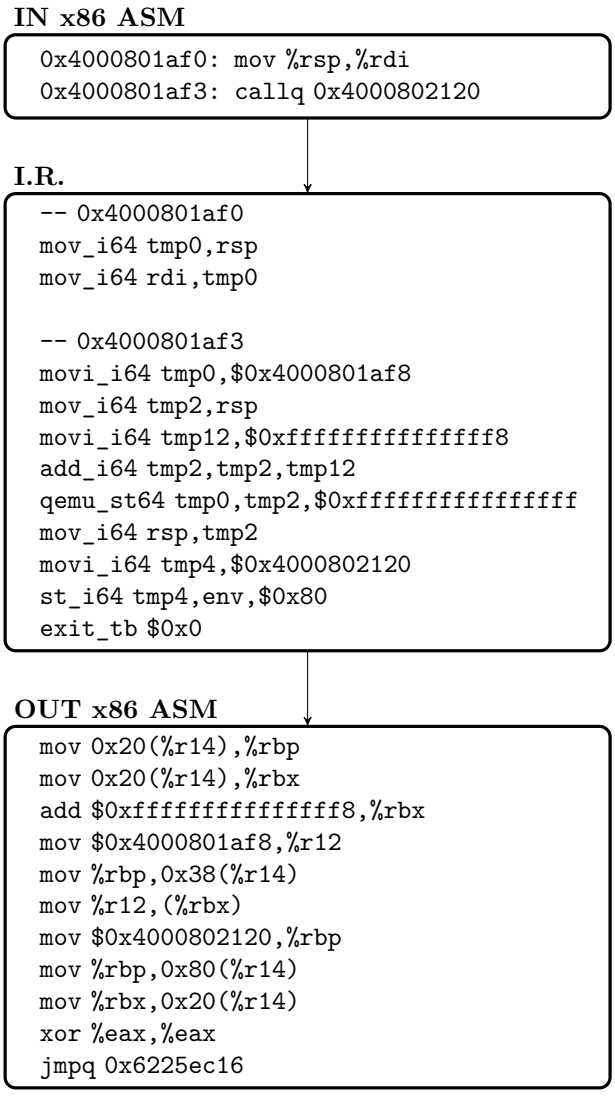
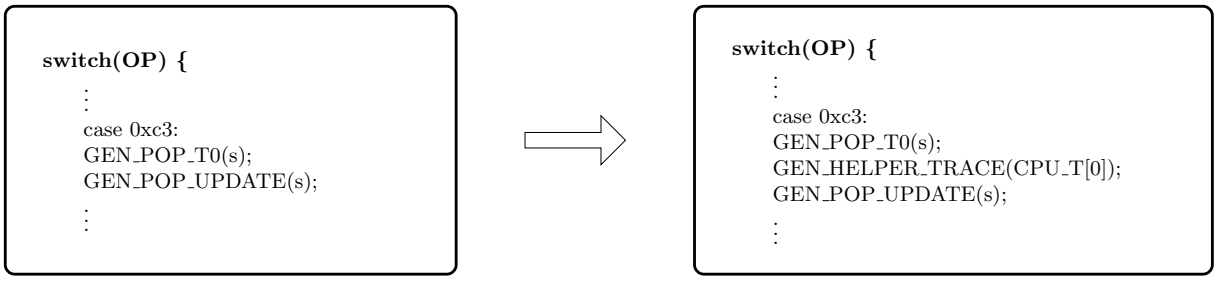


Figure A.1: QEMU binary translation

We built our platform on top of QEMU, in a way to extend its DBT capabilities to also instrument code meant to be protected, turning it into a DBI tool. Whenever translating code, the tool identifies specific contexts to which it injects instructions which augment the application's original capabilities. We call the modified QEMU that supports DBI as mQEMU.



a. Originally generated code b. Instrumented generated code

Figure A.2: Micro-operations generated for instruction 0xc3 before and after QEMU modification

Instrumentation support on QEMU was enabled through leveraging the use of *helper functions* to create interfaces between instrumentation and instrumented code. Due to the simplicity of the QEMU's micro-operations, it is hard to implement the behavior of complex instructions[1]. Instead of encoding complex instructions in micro-operations and later translating them into native machine code, QEMU uses *helper functions*. These functions are written in C along with QEMU's source code and are compiled to the host architecture during the compilation of QEMU itself. When QEMU is translating code and one of these instructions is found, instead of emitting micro-operations, QEMU generates a `call` instruction to the corresponding *helper function*. Since the *helper function* is already compiled into native machine code, the `call` itself suffices to emulate the instruction's behavior without requiring any additional translation.

The first step to enable DBI on QEMU was to write the instrumentation functions in C, following the standards used to write *helper functions*. The second step consisted in modifying the front-end stage of the translation process, introducing `call` instructions to the custom *helper functions* along with instructions that should be instrumented. Figure A.2 shows how the translation process was modified to emit helper function calls: each operation has its case inside a `switch` statement, thus, instrumenting consists in adding the emission for the *helper function* in the particular code snippet.

Since the CPU state is visible from the *helper function*'s scope, it is possible to implement almost any functionality that depends on processor information. It is also feasible to modify the CPU state from the *helper function*, enabling instrumentation that changes information like register values of even fix erroneous and unexpected behaviors.

A.4 LORD: An Asynchronous Shadow Stack

LORD is a shadow stack implementation based on the mQEMU, which instruments the code with functions to log and output the target of `call` and `ret` instructions, enabling parity verification. Since the `call` to the instrumentation function is emitted before the control-flow instruction itself, its execution will take place first, ensuring that the analysis

[1]For example, the x86 `div` is implemented on QEMU in the form of a helper function

relevant information is always logged. LORD targets the validation of return branches only, precluding indirect call target analyses.

The verifier process, external to mQEMU execution, is responsible for collecting and analyzing the information emitted by the instrumentation code. It builds a data structure of valid returns to executed `calls` and uses this information to assert every return operation as it happens. The verifier also implements two different Shadow Stack policies, one that enforces strict parity between `call` and `ret` instructions, and one based on a hash map validation, which is more relaxed allowing returns to target addresses after any previously executed `call`. Subsection A.4.2 goes through both policies in detail.

The communication channel between the two running processes was evaluated using two different approaches: Linux *pipes* and *shared memory*. Both implementations follow a minimal protocol and rely on very simple data structures that avoid all unnecessary overhead. This enables easy hardening of the interfaces, making the corresponding source code unlikely to be a new target for attacks.

A.4.1 Leveraging the Asynchronous Shadow Stack

To enable a better understanding of overheads and bottlenecks in the asynchronous approach, we conducted an experiment where the efficiency of the same solution with different optimizations was compared. These various optimizations consisted mainly in project decisions about the communication protocol used between the main and the verifier processes. All the different versions are described below, the efficiency and benefits of each one are later explained in Section A.5.2.

Raw communication. In this implementation, a regular pipe is opened between the main and the verifier processes. Every time a monitored instruction is executed, a message with an instruction code and its target is sent through the pipe to the verifier process for analysis.

Call buffer. In this version, every `call` instruction executed generates a message that is stored into a local buffer in the main process. Once a `ret` instruction is executed, all messages in the buffer are sent to the external process together with a final message containing information about the `ret`. When compared to the raw communication method, this implementation replaces a big number of small write operations with fewer but larger write operations. The total amount of data exchanged is the same in both implementations.

Hybrid verification. Considering the *Call buffer* implementation, there are two worst case scenarios where a big number of small messages are generated. In the first one, many leaf functions are consecutively called, creating a large number of messages with information about two instructions. In the second one, functions with only one call in its scope are successively called, generating a first long message with information about all call instructions when the leaf function returns, but then creating many small messages with information about single return instructions. These scenarios reduce significantly the benefits obtained by the *Call buffer* implementation. In such cases, the full overhead of sending a message is introduced for messaging a tiny set of data.

To analyze how often the bad scenario may happen we conducted an experiment in which we counted the number of leaf functions into the same execution instance for

applications on the Mibench benchmark [70]. We consider as leaf one function that does not call any function during its execution. Notice that, even if this function has calls to other functions, it may be considered leaf or non-leaf, depending on its runtime context, making it possible that the same function is counted as a leaf in certain conditions and non-leaf in different ones. Table A.1 shows the numbers observed for leaf and non-leaf functions, unveiling a large percentage of leaf functions for every application.

Due to this huge number of leaf functions on each execution instance, we developed a hybrid scheme that combines the asynchronous and online Shadow Stack approaches. In this scheme, parity between a `call` to a leaf function and its respective return instruction are locally analyzed, avoiding the need to exchange tiny messages with the external process.

Application	Leaf functions	Non-leaf functions	Leaf functions %
adpcm-c	39972	39	99.90
adpcm-d	39972	39	99.90
basicmath	20216164	9408100	68.24
crc32	26617774	19549	99.93
dijkstra	192803	178566	51.92
fft -i	1893252	1234169	60.54
fft	3457660	1657267	67.60
gsm-t	1865737	138615	93.08
gsm-u	1666416	102689	94.20
jpeg-c	155183	27334	85.02
jpeg-d	16893	6401	72.52
lame	2266102	405699	84.82
mad	820691	310413	72.56
patricia	4609645	1768438	72.27
qsort	2041625	451996	81.87
sha	51465	1585	97.01
stringsearch	29532	6538	81.87
susan1	1902	826	69.72
susan2	1546	601	72.01
susan3	1546	603	71.94
typeset	2039328	1022059	66.61
Leaf functions average			79.22%

Table A.1: Mibench applications - Number of leaf functions - Number of non-leaf functions - Percentage of leaf functions

Shared memory. We also implemented a communication scheme in which messages are exchanged through a shared memory area between both processes, instead of a pipe. Whenever information about an instruction executed by the main process needs to be recorded, this data is directly written to a shared buffer and is automatically visible by the verifier process. In comparison with the pipe-based implementation, this version avoids the need of a second memory copy from the local main process buffer throughout the pipe.

Even though this implementation is more straightforward, the access to the buffer requires synchronization, what was made with the use of Linux IPC semaphores [147].

The external process was implemented as a polling mechanism that sleeps and checks the buffer for new data. The goal of this implementation was to analyze if and how a shared memory based mechanism would outperform the pipe based scheme.

A.4.2 Policy modules

Two policy modules for verifying the parity between `call` and `ret` instructions were implemented on LORD. These modules are implemented as part of the LORD's verifier process and operate building data structures and verifying the correctness of the program flow with information received from mQEMU.

The first policy implements a stack that is incremented whenever a message for a `call` instruction is received. When a message for a `ret` instruction is received, the target value on the top of the shadow stack is compared with the `ret` target, validating the program flow or emitting a warning message in cases of corruption. This approach is quite restrictive and enforces strict parity between `call` and `ret` instructions.

The second policy is based on a hash map implementation. The idea is to implement program flow verification in a slightly more relaxed way, not enforcing `call/ret` order. Once a `call` is executed, the address of the instruction immediately after it is added to the hash map, being considered valid. If a return targets this address, even out of parity, it is considered valid and no warning is emitted. This policy is efficient against attacks that try to call injected functions or code snippets not previously executed but will not work against attacks that merely subvert the program flow by changing a return target to a different address that was added to the hash map previously and wasn't yet removed.

Both policies worked smoothly on the x86_64 architecture.

A.5 Experimental Evaluation

The proposed implementation was evaluated using two different sets of applications. The first set consisted in synthetic exploit implementations based on published vulnerabilities and focused on verifying if the proposed solution was able to correctly detect attack attempts on the monitored software. The second set consisted in a subset of applications present in the Mibench benchmark [70] and focused on the analysis of the introduced overheads, allowing better understanding and leveraging of the techniques used to speed up the entire solution. All tests were executed on an Intel Core 2 Quad 2.4GHz with 4Gb RAM memory machine, running Ubuntu Server 10.04.3 LTS.

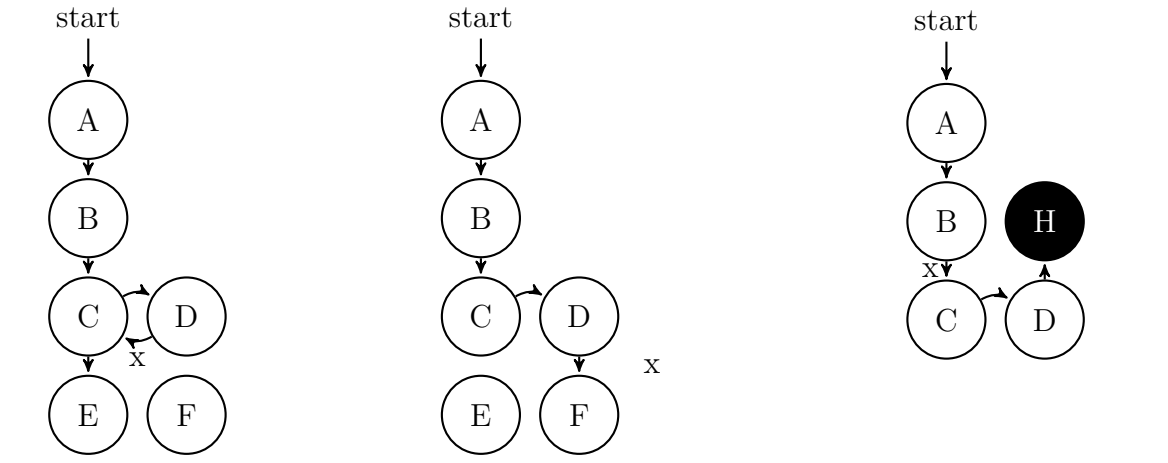
A.5.1 Attack detection

Figure A.3 shows three different scenarios for a program execution. In Figure A.3a the program follows its regular course. In Figure A.3b, the program flow was modified to return from the node D towards F, and not towards C, as it would be expected. The edge x depicts the corrupted return. In Figure A.3c, the program flow was modified to execute code that was injected in the program's memory. The injected code is represented by the black node *H*.

We implemented the described scenarios over two synthetic applications with buffer manipulation errors that led to stack smashing based exploitations and one implementation that could be exploited through a format string vulnerability. Launching both attacks shown in Figures A.3a and A.3b against QEMU was not a significant difficulty, showing that, even in different execution environments, these attacks still work. The vulnerable binaries were compiled without stack smashing protection for a more straightforward test — As applications built this way remain vulnerable to techniques used for bypassing [5, 9, 16, 81, 82, 107, 153], its use would only increase the complexity of the test without adding conclusive results. This test set covered the whole group of attacks that overwrite return addresses stored on the program's stack.

LORD correctly detected all attacks with both the shadow stack and the hash map based policies. Attack detection also worked correctly with all implemented communication approaches.

A small detail about the attack described in Figure A.3b should be noticed. It is possible to manage an attack against the hash map based policy, in which the node F is an already executed node which was not yet returned to. In this case, F corresponds to an address that was a return target before the x redirection. Since the policy only verifies if the return destination is inside the hash map without caring about `call/ret` parity, it would be considered a valid return and the attack would be successful.



(a) Normal program flow (b) Corrupted program flow (c) Corrupted program flow towards injected code

Figure A.3: Control-flow examples

A.5.2 Performance

The performance of our solution was evaluated using applications from the Mibench benchmark suite, covering all communication approaches described in Subsection A.4.1. Table A.2 shows the execution time for native execution, followed by the respective slowdown when executed on top of our implementation with the specified communication approach. In this table, all programs were executed ten times and the arithmetic means of the execution times for each configuration were used in the slowdown analysis. We show

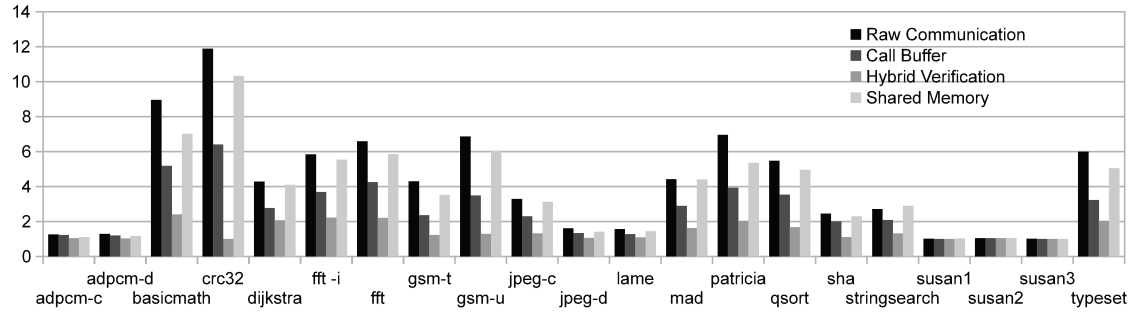


Figure A.4: Slowdowns of mQEMU x QEMU

the standard deviation of time in mQEMU native execution to show that the variance among different executions is subtle, and the same occurred for all configurations.

Figure A.4 shows the slowdown of our mechanism when comparing executions on the original QEMU and mQEMU. Figure A.5 shows a comparison of the observed speedups in our system achieved by each enhanced communication approach when compared to the simplistic pipe communication scheme.

As observed, our implementation reached an average slowdown of 1.46x on its most efficient method, which is the Hybrid Verification, when compared to the execution on QEMU. In this approach, the biggest slowdown noticed was 2.22x, and it achieved no slowdown in a few cases (see Figure A.4). The Hybrid Verification also performed very well when used along with the CRC32 application, achieving a speedup of 13.84x in comparison to a non-optimized execution (see Figure A.5). A closer analysis of this application reveals that a significant part of its execution consists of successive calls to leaf functions (99.9%). In this case, the use of the Hybrid Verification combined with runtime optimizations present in QEMU, such as its block chaining mechanism, drastically reduces the instrumentation overhead.

It is important to notice that the Hybrid approach is sensitive to the complexity of the analysis. Moving too many operations to the application thread would certainly harm performance. So, it is important to carefully divide computation, so that operations allocated in the main process thread do not become a greater overhead than communication.

The performance on CRC32 shows that, if the trade-off between computation and communication is carefully designed, the Hybrid approach has a significant potential as an optimization. Finally, the Hybrid Verification never performed worse than the Raw Communication implementation in our tests.

The Shared Memory approach performed better than the Raw Communication version, as expected. Since both threads use the same memory area as buffers, this implementation reduces the number of memory operations. Although the benefits are clear, the mechanism used for synchronizing the access to the shared resource combined with successive memory operations resulted in a blocking scheme that introduced relevant slowdown to the system, especially if compared to the Hybrid implementation which avoids a fair number of communication operations.

Application	QEMU Time (s)	Std. Deviation	Raw Comm.	Call Buffer	Hybrid Verification	Shared Memory
adpcm-c	0.641	0.003	1.25	1.22	1.04	1.11
adpcm-d	0.483	0.007	1.28	1.2	1.03	1.16
basicmath	9.339	0.032	8.95	5.18	2.4	7.02
crc32	4.833	0.044	11.88	6.4	1	10.32
dijkstra	0.261	0.003	4.28	2.76	2.06	4.1
fft -i	1.368	0.004	5.83	3.68	2.22	5.53
fft	2.084	0.010	6.58	4.25	2.2	5.85
gsm-t	1.507	0.009	4.29	2.36	1.22	3.52
gsm-u	0.67	0.000	6.86	3.48	1.28	6.05
jpeg-c	0.16	0.000	3.29	2.29	1.31	3.11
jpeg-d	0.076	0.019	1.61	1.34	1.05	1.41
lame	11.37	0.169	1.56	1.27	1.09	1.44
mad	0.625	0.016	4.41	2.89	1.62	4.4
patricia	2.615	0.005	6.95	3.92	2.03	5.34
qsort	1.295	0.018	5.47	3.53	1.66	4.95
sha	0.09	0.000	2.44	2	1.11	2.29
stringsearch	0.038	0.004	2.71	2.08	1.32	2.89
susan1	0.235	0.005	1.01	1	1	1.03
susan2	0.135	0.005	1.04	1.04	1.04	1.04
susan3	0.08	0.000	1.01	1	1	1
typeset	1.542	0.014	5.98	3.23	2.04	5.04
average			4.22	2.67	1.46	3.74

Table A.2: Application execution on QEMU and execution slowdown for each version of asynchronous communication

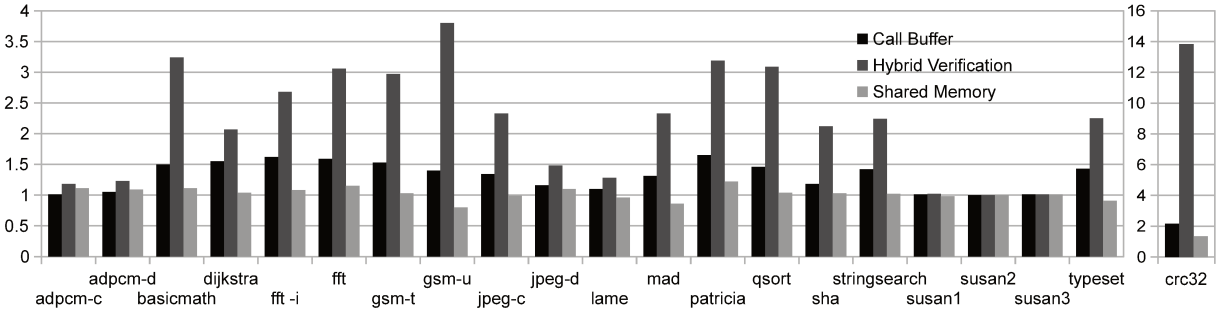


Figure A.5: Communication speedups x non-optimized instrumented version

A.6 Conclusions

We described LORD, a shadow stack implementation for user space processes that supports asynchronous analysis and is built on top of QEMU. The proposed system was tested with different approaches for communication between its components, allowing the development of optimizations to increase its efficiency.

The experiments show that our mechanism can perform, on average, 1.46x slower than the execution on QEMU and correctly detected attacks based on the modification of return addresses stored on the program's stack. It is also shown that the proposed optimizations brought an average speedup of 2.73x in comparison to a raw implementation and that, when particular characteristics are met, the optimized system can reach the significant speedup of 13.84x.