

**PROTEÇÕES CONTRA ATAQUES BASEADOS EM
PROGRAMAÇÃO ORIENTADA A RETORNOS**

MATEUS FELIPE TYMBURIBÁ FERREIRA

PROTEÇÕES CONTRA ATAQUES BASEADOS EM PROGRAMAÇÃO ORIENTADA A RETORNOS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

Julho de 2020

MATEUS FELIPE TYMBURIBÁ FERREIRA

**PROTECTIONS AGAINST ATTACKS BASED ON
RETURN-ORIENTED PROGRAMMING**

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

July 2020

Ferreira, Mateus Felipe Tymburibá

F383p Protections against attacks based on return-oriented programming [manuscrito] / Mateus Felipe Tymburibá Ferreira. — 2020.
 xxvii, 122 f.; il.; 29cm.

Orientador: Fernando Magno Quintão Pereira.

Tese (doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Referências: f. 107-122

1. Computação – Teses. 2. Compiladores (Programas de computador) – Teses. 3. Programação (Computadores) – Teses. 4. Algoritmos de computador – Teses. 5. Programação Orientada a Retornos. 6. Computação - Medidas de segurança - Teses.
I. Pereira, Fernando Magno Quintão. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*33(043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Protections against attacks based on Return-Oriented Programming

MATEUS FELIPE TYMBURIBÁ FERREIRA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. SANDRO RIGO
Instituto de Computação - UNICAMP

PROF. HENRIQUE COTA DE FREITAS
Departamento de Ciência da Computação - PUCMG

PROF. ÍTALO FERNANDO SCOTÁ CUNHA
Departamento de Ciência da Computação - UFMG

PROF. LEONARDO BARBOSA E OLIVEIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 17 de Julho de 2020.

To my grandparents for their efforts in building extraordinary families.

Acknowledgments

First of all, I thank God for offering me endless amazing opportunities throughout my life. This PhD was another divine privilege that allowed me to enjoy valuable learning.

I would like to thank my parents for emphasizing the value of education and for being an example of character, dedication and perseverance. You are the heroes I am inspired by in the most difficult moments!

I owe a special thanks to Fernando, my advisor. Not just for mere formality, but because he is, in fact, the best tutor that a student can expect. His technical and professional qualities do not require comments and he generously shares all that experience and wisdom. But more than that, he treats all his students indiscriminately in a fair, affectionate and understanding manner. There were times when personal problems prevented me from developing my maximum productivity. In all those situations, he knew how to support and demand at the right time and in the right measure. I hope that my future students will be as proud to work with me as I am to have worked with him.

I thank my colleagues in DECOM and CEFET-MG for the financial support and for releasing me from my duties so that I could dedicate myself entirely to this project.

To the friends at the Compilers Lab, especially Rubens and Hugo, I thank you for the competent partnership.

Finally, I am grateful to all my friends and family, who sent good energy to my success on this journey.

*“Live as if you were to die tomorrow;
learn as if you were to live forever.”*
(Mahatma Gandhi)

Resumo

Programação Orientada a Retornos (ou ROP, de *Return-Oriented Programming*) é o nome de uma técnica usada para o desenvolvimento de códigos maliciosos que vem sendo amplamente utilizada para forçar a execução de códigos arbitrários em aplicações vulneráveis. Em função de seu vasto emprego em investidas contra sistemas computacionais modernos, proteções contra ROP têm sido extensamente estudadas. Apesar disso, ainda não existe uma solução definitiva contra esses ataques. Esta tese introduz abordagens que dificultam a consolidação de ataques dessa natureza.

Criamos uma análise estática de código-fonte que estima a densidade máxima de instruções de desvio indireto executável por um programa. Sua importância advém do fato de que diversas proteções monitoram a densidade de desvios indiretos, a fim de detectar ataques ROP. Contudo, esses mecanismos utilizam limiares únicos: a frequência de desvios indiretos que caracteriza um ataque é a mesma para qualquer aplicação. Neste trabalho, mostramos que limiares padrão podem ser superados facilmente. Como alternativa, criamos um algoritmo que estabelece o melhor valor de limiar para cada aplicação. Nossa análise foi implementada no compilador LLVM e utilizada para encontrar limiares específicos para cada benchmark que compõe o SPEC CPU2006. Comprovamos a acurácia da nossa abordagem comparando os limiares estimados com valores reais obtidos durante a execução desses benchmarks no framework Pin. Apesar de percorrer estaticamente todos os possíveis caminhos de execução de um programa ser um problema teoricamente indecidível em situações gerais, para o contexto específico de proteção contra ataques baseados em ROP nós conseguimos desenvolver um algoritmo que encontra a solução para programas com até 700 mil instruções em 25 minutos.

Em nossa segunda abordagem, apresentamos um sistema multicamadas para proteger programas contra ataques ROP. As camadas superiores validam a maior parte do fluxo de controle de um programa a um baixo custo computacional; portanto, não comprometendo o tempo de execução. As camadas inferiores fornecem fortes garantias para lidar com fluxos mais suspeitos; aumentando assim a segurança. Nossa solução

multicamadas combina técnicas já descritas na literatura com uma verificação que introduzimos: checar se a instrução de chamada de função que precede um endereço de retorno aponta para um segmento executável do programa. Cuidamos para que nosso projeto utilize unidades microarquiteturais já disponíveis nas versões modernas dos processadores x86. Nosso modelo impõe uma sobrecarga no tempo de execução de apenas 0,57% em um conjunto de benchmarks compostos por todos os 29 programas do SPEC CPU2006 e 209 benchmarks do LLVM Test Suite. A nova verificação que introduzimos para validar as instruções `call` que precedem os endereços de retorno também é efetiva. Ela reduz em quase 35x o número de gadgets disponíveis para criar um ataque, quando comparada a uma técnica semelhante tradicionalmente empregada na literatura. Também analisamos o número de falsos positivos que atravessam nossas camadas de proteção. Nesse experimento, enriquecemos nosso conjunto de benchmarks com os três navegadores de desktop mais populares e quatro aplicativos vulneráveis para os quais existem explorações ROP conhecidas. Apenas 3,14% de todos os endereços de retorno são considerados inválidos nesses benchmarks. Demonstramos que o custo para tratar esses falsos positivos não afeta o tempo de execução dos programas.

Palavras-chave: Programação Orientada a Retornos, proteção, segurança, análise estática de código, multicamadas.

Abstract

Return-Oriented Programming (ROP) is the name of a technique used for the development of malicious code that has been widely used to force the execution of arbitrary code on vulnerable applications. Because of its wide use in attacks against modern computing systems, protections against ROP based exploits have been widely studied. Nevertheless, there is still no definitive solution against these attacks. This thesis introduces approaches that hinder the consolidation of such attacks.

We created a static source code analysis that estimates the maximum density of indirect branch instructions executable by a program. Its importance stems from the fact that several protections monitor the density of indirect branches to detect ROP attacks. However, these mechanisms use unique thresholds: the frequency of indirect branches that characterizes an attack is the same for any application. In this work, we show that standard thresholds can be easily overcome. As an alternative, we created an algorithm that establishes the best threshold value for each application. Our analysis was implemented in the LLVM compiler and used to find specific thresholds for each benchmark of the SPEC CPU2006 suite. We proved the accuracy of our approach by comparing the estimated thresholds with real values obtained during the execution of these benchmarks in the Pin framework. While statically traversing all possible execution paths of a program is a theoretically undecidable problem in general situations, for the specific context of protection against ROP based attacks we managed to develop an algorithm that finds the solution for programs with up to 700,000 instructions in 25 minutes.

In our second approach, we present a multilayer system to guard programs against ROP attacks. Upper layers validate most of a program's control flow at a low computational cost; thus, not compromising runtime. Lower layers provide strong enforcement guarantees to handle more suspicious flows; thus, enhancing security. Our multilayer solution combines techniques already described in the literature with a verification that we introduce: checking if the function call instruction that precedes a return address points to an executable segment of the program. We took care

that our project uses micro-architectural units already available in modern versions of x86 processors. Our model imposes a small execution time overhead of 0.57% in a set of benchmarks composed by all 29 programs from SPEC CPU2006 and 209 benchmarks from the LLVM Test Suite. The new check we introduced to validate `call` instructions that precede return addresses is also effective. It reduces by almost 35x the number of gadgets available to build an attack, when compared to a similar technique traditionally employed in the literature. We also analyzed the number of false positives that slip through our layers of protection. In this experiment, we enriched our benchmark suite with the three most popular desktop browsers and four vulnerable applications for which there are known ROP exploits. Only 3.14% of all return addresses are considered invalid in these benchmarks. We demonstrate that the cost to treat these false positives bears no impact in the execution time of programs.

Keywords: Return-Oriented Programming, protection, security, static code analysis, multilayers.

List of Figures

1.1	Number of buffer error vulnerabilities and code injection attacks cataloged between 2001 and 2017 [NIST, 2018].	1
1.2	Occurrence of the attack classes cataloged in the CVE between 2001 and 2017 [MITRE, 2018].	2
2.1	Stack structure and flow between gadgets of a ROP exploit designed to attack "My MP3 Player" software.	13
2.2	General scheme of a JOP exploit: a trampoline gadget interconnects functional gadgets.	16
2.3	JOP version of the ROP exploit illustrated in Figure 2.1.	17
3.1	Protection strategy based on building a shadow stack [Davi et al., 2011b]. .	25
4.1	(a) Input string used in a stack buffer overflow attack that triggers a ROP exploit. Data used in the exploit is written in light grey, and return addresses are written in black. Letters in front of addresses indicate the gadget that will be invoked. (b) Gadgets that are chained during this attack. . . .	49
4.2	(a) Example program written in C. (b) Simplified x86 assembly of example program. (c) δ -CFG produced for the example program.	55
4.3	Brute-force inference of indirect branches using window of size K.	56
4.4	Intuitive view of the algorithm from Figure 4.3, when applied on the δ -CFG of Figure 4.2 (c). Numbers in parentheses above block letters indicate the highest frequency of RETs that our algorithm obtains starting the search from that block, considering a window of 16 instructions.	57
4.5	Minimal size of a no-op gadget needed to bypass universal (a) and specific (b) thresholds in a window of 32 instructions. Specific thresholds require longer gadgets, which are harder to find, as shown in Figure 4.6.	59

4.6	Percentage of potential no-op gadgets of sizes ranging from 3 to 25. There are 75% less potential no-op gadgets of size 6 (used to bypass the specific threshold) than 4 (used to bypass the universal threshold).	60
4.7	Specific threshold of indirect branch instructions for programs in the SPEC CPU2006 benchmark suite. Grey bars show specific thresholds found with the algorithm of Section 4.2, and black bars show values found dynamically. 1/4 is the density that we have assumed for library functions, which we cannot analyze statically.	61
4.8	Program size vs runtime for $K = 32$. Points in the X axis are benchmarks from Figure 4.7, ordered by number of instructions. This figure indicates that even though our algorithm has a theoretical exponential complexity, its execution time has a linear behavior and is practically feasible.	62
5.1	Abstract overview of our multilayer protection mechanism.	68
5.2	The second protection layer.	71
5.3	Example of the checks performed by the second layer on a program running on a 16-bit address space. Arrows on the left indicate the targets of indirect branches. Numbers on the arrows indicate which branch executes first. . .	72
5.4	CET’s Indirect Branch Tracking (IBT) operation [Intel, 2019].	75
5.5	Hit rate (%) of the branch predictors for the programs in the SPEC CPU2006 and LLVM Test benchmark suites. Numbers next to labels show the weighted average accuracy of each predictor. The LLVM bar subsumes results of 209 programs in the LLVM test suite.	87
5.6	RET-validation overhead.	92
5.7	Percentage of overhead imposed by CET to validate RETs	93
5.8	Minimum percentage of overhead imposed by CET to validate indirect Jumps and CALLs.	94
5.9	Cumulative overhead of first two layers, with e-CC.	96
5.10	Cumulative overhead of first two layers, with shadow stack.	97
5.11	Percentage of CET’s IBT overhead reduction (upper bound) due to the multilayer protection system.	98
5.12	Percentage of CET’s IBT overhead reduction (lower bound) due to the multilayer protection system.	99
5.13	To carry out the experiment described in this Section, we compile a program such as (a) into its assembly version (b). Then, we replace, manually, occurrences of RET with the two-instruction sequence seen in (c).	100

5.14 Execution overhead of replacing RET with the POP and JMP sequence in programs of the LLVM Shootout suite. Each box contains eight executions. The higher the average, the higher the overhead.	101
---	-----

List of Tables

- 3.1 Comparison of common ROP defenses. 39
- 4.1 Proportion of indirect branch instructions in cataloged exploits. Each gadget has, on average, 2.46 instructions. This results in a high frequency of indirect branches. 50
- 5.1 Gadget Reduction on four applications vulnerable to known ROP exploits (top) and on the three most popular desktop browsers in November 2018 (bottom) [NetMarketShare, 2018]. 83
- 5.2 Percentage of gadget reduction in SPEC CPU2006 benchmarks. 85
- 5.3 Validations that slip from Layer 2 to Layer 3. 89

Contents

Acknowledgments	xi
Resumo	xv
Abstract	xvii
List of Figures	xix
List of Tables	xxiii
1 Introduction	1
1.1 Motivation	3
1.2 Thesis Statement	4
1.3 Objectives	4
1.4 Contributions	5
1.5 Publications	6
1.5.1 Awarded Publication	6
1.5.2 Other Publications	6
2 Return-Oriented Programming: An Overview	9
2.1 Memory Protections	9
2.1.1 Non-Executable Stack (nx-stack)	9
2.1.2 Address Space Layout Randomization (ASLR)	10
2.1.3 Write Xor Execute (NX/XD)	10
2.2 Operation	10
2.2.1 ROP Example	12
2.2.2 Jump-Oriented Programming	15
2.2.3 Call-Oriented Programming	17
2.3 Chapter Summary	18

3	Related Work	19
3.1	Common ROP defenses	19
3.1.1	Evaluation Metrics	20
3.1.2	Protection Strategies	22
3.1.3	Implementation Approaches	31
3.1.4	Discussion	38
3.2	Control-Flow Integrity	40
3.3	Defenses Based on Indirect Branch Frequency	41
3.4	Multilayer Protection	42
3.5	Chapter Summary	45
4	Inferring The Peak Density Of Indirect Branches	47
4.1	Universal ROP-Detection Thresholds	48
4.1.1	Evading Universal Thresholds	50
4.1.2	Using Specific Thresholds To Improve Protections	52
4.2	Static Inference Of Specific Thresholds	53
4.2.1	The Supporting Data-Structure	53
4.2.2	Algorithm To Estimate The Peak Density Of Indirect Branches	54
4.3	Experiments	58
4.3.1	Specific Thresholds Lead To Less Available Gadgets	58
4.3.2	On The Accuracy Of Our Estimates	60
4.3.3	Performance Of The Inference Algorithm	62
4.4	Chapter Summary	63
5	Multilayer Protection	65
5.1	Layers Of Protection	67
5.1.1	First Level: Branch Predictors	69
5.1.2	Second Level: e-CC and CET	70
5.1.3	Third Level: False Positives Filtering	76
5.2	Evaluation	78
5.2.1	Gadget Reduction: a false negatives assessment	81
5.2.2	Filtering statistics	86
5.2.3	Overhead	91
5.3	Chapter Summary	101
6	Final Remarks	103
6.1	Future Work	104
6.2	Conclusion	105

Chapter 1

Introduction

Software is considered the weakest link in the chain of components targeted by attackers in hostile acts against computer systems [Hoglund and McGraw, 2004]. As the quantity and complexity of software increases, the volume of identified vulnerabilities also increases, as shown in the statistics provided by NIST (National Institute of Standards and Technology) and presented in Figure 1.1.

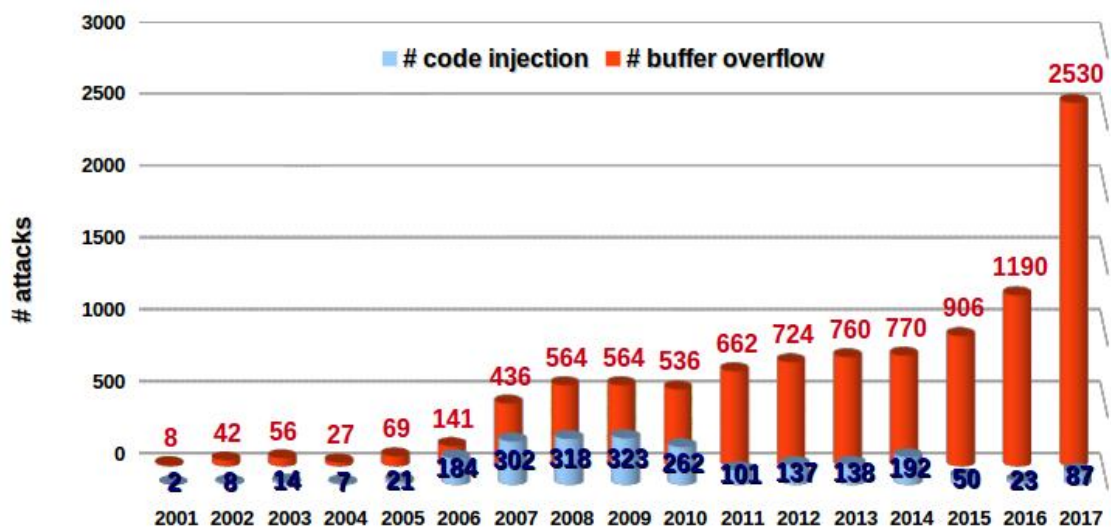


Figure 1.1. Number of buffer error vulnerabilities and code injection attacks cataloged between 2001 and 2017 [NIST, 2018].

This graph illustrates, in the red bars, the growing trend in the discovery of vulnerabilities classified by NIST as buffer errors¹. Also known as buffer overflow,

¹Buffer: a limited and continuous memory space.

this type of software failure is considered critical to system security because it allows the attacker to subvert the normal execution flow of the application under attack and execute arbitrary code without the user’s consent. For this reason, vulnerabilities that lead to arbitrary code execution are classified as critical [Microsoft, 2018a].

Paradoxically, the blue bars in Figure 1.1 demonstrate a recent reduction in code injection attacks, a technique that in the past was the primary weapon used to consolidate arbitrary code execution. If we add the statistics of other types of software failures that can lead to the execution of arbitrary code (eg, use-after-free, integer overflow, double free, uninitialized variable usage, format string, etc. [Koziol et al., 2004]) in the red bars of Figure 1.1, the discrepancy becomes even greater! In Section 1.1, we discuss the reason for this gap.

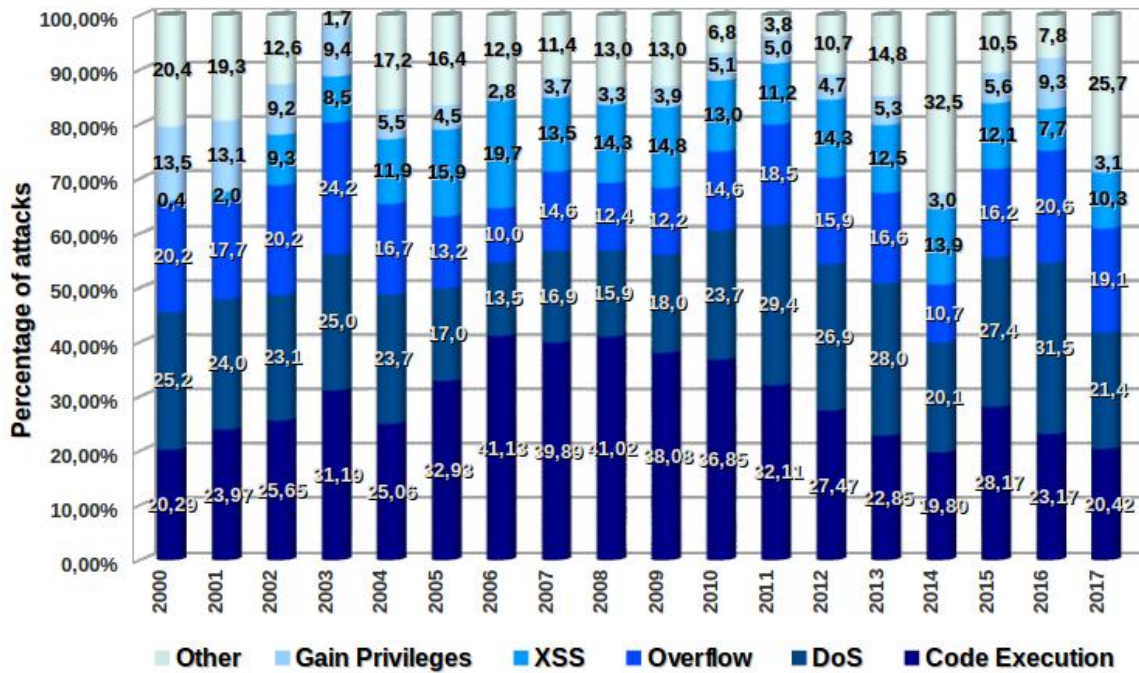


Figure 1.2. Occurrence of the attack classes cataloged in the CVE between 2001 and 2017 [MITRE, 2018].

Due to the power they provide to an attacker, exploitations that execute arbitrary code have won invaders’ preference. This behavior can be observed in Figure 1.2, which presents the percentage, over the years, of this type of attack in relation to the other classes cataloged in CVE (Common Vulnerabilities and Exposures). All the time, exploiting vulnerabilities that lead to the execution of arbitrary code remains one of the most frequent types of attack, even though it is more difficult to execute than other categories.

1.1 Motivation

As shown in Figure 1.1, there is a clear discrepancy between the number of code injection attacks and the amount of vulnerabilities that can result in arbitrary code execution. This is due to the emergence of protection mechanisms, such as Non-Executable Stack and Write $\oplus$ Execute (detailed in Section 2.1.3), that prevent the consolidation of code injection attacks. Despite this, arbitrary code execution attacks continue to rank among attackers' favorites, as noted in Figure 1.2. To do so, attackers have developed new mechanisms, which reuse the targeted application code to perform malicious activities, rather than inserting a payload into the memory of the affected process. Among these techniques, the mechanism called Return-Oriented Programming (ROP) stands out.

ROP, described in detail in Chapter 2 of this text, has aroused great interest from the scientific community and the system security industry, due to its wide use in recent attacks on computer systems [Roemer et al., 2012]. Among many examples of high-impact malicious code using the ROP technique, we can mention the Stuxnet and Conficker malware [Matrosov et al., 2010; F-Secure Response Labs, 2013; Minh and Minh, 2009], as well as a code that violated an electronic voting machine used in various locations [Checkoway et al., 2009]. The relevance of ROP was also boosted by the development of automated techniques for building code attacks, which drastically reduced the time and complexity for the preparation of ROP-based malicious artifacts [Kornau, 2010; Chen et al., 2011b; Schwartz et al., 2011; Buchanan et al., 2008; Hund et al., 2009; Roemer, 2009; Dullien et al., 2010; Corelan Team, 2011; Rapid7, 2012; Solé, 2010].

Because it has become one of the main techniques used by attackers to develop exploits² [Marpaung et al., 2012], mitigations against ROP have been extensively studied. Windows 8, for example, has added a new protection mechanism against ROP, which prevents the call of APIs (Application Programming Interfaces) typically used in ROP attacks if the parameters are not stored in the process stack area. However, a few days after the release of the preliminary version of the system (Windows 8 Developer Preview), researchers presented demonstrations of relatively simple strategies capable of circumventing this defense [Rosenberg, 2011; Son, 2011; Tung, 2011]. All this related to a system that had not even been officially released.

Another point of concentration of efforts in the search for mechanisms of contention of attacks has been the tool EMET (Enhanced Mitigation Experience Toolkit) [Microsoft, 2018b]. This is a free utility that can be installed on Windows

²Exploit: artifact developed for the purpose of exploiting a vulnerability present in a system.

systems. Over time, this tool has added a number of protections against exploit development techniques. Currently, it includes, among other security mechanisms, mitigations against ROP, including a version of ROPGuard, a defense against ROP presented by Ivan Fratric in the contest BlueHat Prize [CSO, 2012]. This implementation of ROPGuard was surpassed two weeks after its announcement [Snake, 2012].

Even the result of the "BlueHat Prize" competition, promoted to reward works that aim to defend the safety of computers, highlight the relevance of ROP in the current scenario of systems security. The three best solutions, chosen for awards, presented mitigation techniques against ROP, which reinforces the realization that software manufacturers are in a state of alert regarding this type of attack [CSO, 2012]. In fact, several protections against ROP attacks have been proposed, but there is no definitive solution yet.

1.2 Thesis Statement

Return-Oriented Programming remains a threat because effective protection techniques tend to slowdown programs beyond acceptable levels; hence, they are not used in practice. We show that efficiency and effectiveness do not need to coexist in a single protection technique designed to prevent ROP based attacks. Rather, we can achieve them by combining existing approaches in *layers*. This view implies a change of perspective: instead of focusing on malicious execution streams as previous work does [Abadi et al., 2005; Jang et al., 2014; Gawlik and Holz, 2014; Tice et al., 2014; Prakash et al., 2015; Zhang et al., 2015; Davi et al., 2012; Payer et al., 2015; van der Veen et al., 2015; Ding et al., 2017], we certify authentic streams. Upper layers cheaply certify most of the program flows and drastically reduce the number of tests executed in lower layers, which check eccentric codes via more expensive verifications. This novel multilayer system can be implemented using components that already exist in current versions of the x86 architecture.

1.3 Objectives

Our goal in this thesis is to develop new protection mechanisms against attacks based on the Return-Oriented Programming technique. To achieve it, we created two distinct approaches:

- An algorithm for static analysis of source codes capable of identifying the maximum density of indirect branches achievable by an application. As we show in Section 4.1.1, standard thresholds for density of indirect branches can be easily overcome. Thus, the identification of the best threshold value for each application, offered by the algorithm proposed in Section 4.2.2, allows the strengthening of several protections based on this principle.
- A layered protection model that merges various defense strategies to combine effectiveness and efficiency in real-time prevention of ROP attacks. In this second approach, described in Chapter 5, we also propose a new strategy to combat ROP attacks, to compose one of the model layers. This strategy, detailed in Section 5.1.2.1, is to prevent "function returns" to addresses that are not preceded by `CALL` instructions that point to an executable target. Our results indicate that this strategy reduces by almost 35 times the capacity to construct a ROP attack, when compared with a similar defense traditionally employed in the literature, as presented in Section 5.2.1.

By adapting malicious code available in a public exploit repository [Offensive Security, 2020], we demonstrate that standard thresholds of indirect branches density can be overcome. The effectiveness and efficiency of the proposed approaches were evaluated by running experiments with vulnerable applications for which there are public ROP exploits, popular applications - not necessarily vulnerable - and well known benchmarks. The occurrence of false positives and false negatives were also analyzed. The details of these experiments are presented in Sections 4.3 and 5.2.

1.4 Contributions

To reach the objectives listed in the previous Section, we provide the following contributions:

1. A demonstration of the fragility of protections supported by standard thresholds of indirect branches density.
2. The construction of an exponential algorithm, with linear behavior, to solve a subset of a theoretically undecidable problem: the identification of the maximum threshold of density of indirect branches achievable by an application through the static analysis of its source code.

3. The implementation, in the LLVM compiler [The llvm-admin team, 2018], of a static analysis prototype that identifies the maximum threshold of indirect branches density in a source code.
4. The proposition of a new strategy of protection against ROP attacks capable of drastically reducing the possibilities of building an attack of this kind.
5. The definition of a multilayer model that takes advantage of the best characteristics inherent in different ROP protections to compose a solution that combines effectiveness and efficiency using hardware already available in current processors.
6. A performance analysis of Control-Flow Enforcement Technology protection, under development by Intel for future processor models, regarding the runtime overhead and the degree of security offered by this solution.

1.5 Publications

This section lists the scientific publications resulting from the progress of the thesis.

1.5.1 Awarded Publication

Tymburibá, M., Moreira, R. and Pereira, F. "Inference of peak density of indirect branches to detect ROP attacks". Proceedings of the ACM International Symposium on Code Generation and Optimization (CGO). 2016.

This paper presents the algorithm capable of inferring the maximum frequency of indirect branch instructions for an application from its source code. It received the ACM Student Research Competition award in an important event for the dissemination of research results related to code analysis and optimization (CGO).

1.5.2 Other Publications

Tymburibá, M. "Counting the Frequency of Indirect Branches to Detect Return-Oriented Programming Attacks". Student Forum Supplementary Volume of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2015.

This paper describes a hardware implementation for the sliding window protection strategy, which controls the frequency of indirect branches to protect applications

from ROP attacks.

Tymburibá, M., Moreira, R. and Pereira, F. "RipRop: A Dynamic Detector of ROP Attacks". Anais da Sessão de Ferramentas do Congresso Brasileiro de Software (CBSOft). 2015.

This paper describes the optimizations and implementation details of a prototype of the sliding window protection mechanism.

Emilio, R., Tymburibá, M. and Pereira, F. "Inferência Estática da Frequência Máxima de Instruções de Retorno para Detecção de Ataques ROP". Anais do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg). 2015.

This paper presents a preliminary version of the algorithm published in CGO 2016.

Sousa, H. and Tymburibá, M. "Análise Dinâmica de Programas Binários". XXXVII Jornada de Atualização em Informática. Congresso da SBC (CSBC). 2018.

This book chapter is the textbook of a mini-course that presents the fundamentals of Dynamic Binary Instrumentation. It is focused on Pin, a framework widely used in this thesis.

Tymburibá, M., Sousa, H. and Pereira, F. "Multilayer ROP Protection Via Microarchitectural Units Available in Commodity Hardware". Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2019.

This paper presents our multilayer system designed to efficiently protect programs against ROP attacks reusing hardware structures already in place in commodity processors.

Chapter 2

Return-Oriented Programming: An Overview

In this chapter we revisit the notion of Return-Oriented Programming. We start, in Section 2.1, by briefly describing the main aspects of memory protections that motivated or directly influenced the ROP attack technique. In Section 2.2 we shall explain, through an example taken from an actual exploit, how ROP works. Thus, this chapter shall provide the reader unfamiliar with the notion of Return-Oriented Programming a minimum background that will enable him or her to understand the rest of this thesis.

2.1 Memory Protections

Just as strategies to exploit vulnerabilities have evolved over time, protection mechanisms have been designed to hamper the consolidation of these attacks. In this Section we briefly discuss the main aspects regarding memory protections that motivated or directly influenced the ROP attack technique.

2.1.1 Non-Executable Stack (nx-stack)

Because stack overflow attacks were the first to become popular, there was initially a need to protect the stack against code execution, mainly because the most obvious strategy of exploiting this type of vulnerability is to insert malicious code into the stack itself, along with the code that generates the buffer overflow. Due to the realization of this possibility, in 1996, devices arose to prevent the execution of instructions from the stack area (Non-Executable Stack or nx-stack). Currently, this type of protection is

present by default on most operating systems, including Linux, OpenBSD, Mac, Solaris and Windows, among others [Koziol et al., 2004].

2.1.2 Address Space Layout Randomization (ASLR)

Another protection strategy is based on the principle that if memory addresses are assigned randomly to modules and their segments, an attacker will not know to which memory address the execution flow can be diverted. This strategy was named Address Space Layout Randomization (ASLR) [PaX Team, 2003b]. ASLR is adopted by Linux since the 2.6.12 kernel. In this system, each time a process is loaded, its segments, base address, and libraries receive a different range of addresses. On Windows, this protection has been deployed from Vista and Server 2008. In the case of Microsoft systems, address scrambling is renewed with every reboot of the operating system [Cugliari and Graziano, 2010].

2.1.3 Write Xor Execute (NX/XD)

A natural extension of the mechanism for prohibiting the execution of instructions stored on the stack, designed to block such attempts in other areas of memory – such as heap and BSS – has been given the generic $W\oplus X$ designation (or Writable or Executable) [PaX Team, 2003a]. Subsequently, processor manufacturers renamed it NX (No eXecute, by AMD) and XD (eXecute Disable, by Intel). It can also be referenced by other names, according to its implementation in each operating system (eg. DEP – Data Execution Prevention – in Windows). Windows has been using DEP since the release of Service Pack 2 for XP. The $W\oplus X$ technology is also present in Linux since kernel 2.6.8 [Cugliari and Graziano, 2010].

This protection mechanism marks the memory pages with one execution bit. Thus, the Operating System can ensure that areas of memory destined to store data, such as the stack and heap, are not executable. This prevents an attacker from transferring the execution flow of the target program directly to malicious code injected into those memory areas.

2.2 Operation

As soon as the first protections against code injection attacks began to be incorporated into computer systems, alternative attacks based on the reuse of original application codes were proposed. On UNIX-based operating systems, virtually all programs load

the standard C library (libc). This library contains several routines that redirect execution to system calls that are useful in attacks. Therefore, libc was a target of choice in the first reported code reuse attacks. These attacks became known as “return-into-libc” [Tran et al., 2011].

Although the libc library has long been the preferred target of code reuse attacks, any available code can be used. Since the functions available in the libc library limit code execution possibilities, ideas have emerged for linking functions [McDonald, 1999; Newsham, 2000; Wojtczuk, 2001] or executable code snippets [Newsham, 1997; Spyrit and Jack, 1999; Richarte, 2000, 2001; Krahmer, 2005]. By demonstrating that the chaining of original code snippets allows the execution of arbitrary computations [Shacham, 2007], this technique became popular among attackers and came to be referred to by the Return-Oriented Programming nomenclature, or simply ROP.

ROP attacks are based on code reuse to overcome the protection offered by the execution bit (NX / XD). They chain several small pieces of code, known as *gadgets*, to perform a certain task. To achieve this chaining, the last instruction of each chosen code snippet must be a branch. The original ROP idea uses gadgets finalized with return (RET) instructions to interconnect the chosen code fractions. That’s the reason for the name of the technique. Later, it has been shown that it is also possible to interconnect gadgets through unconditional jump (JMP) [Checkoway et al., 2010; Chen et al., 2011c] or procedure call (CALL) [Carlini and Wagner, 2014; Goktas et al., 2014] indirect branch instructions. Through this chained sequence of code, attackers are able to overcome the execution bit protection to execute arbitrary malicious code.

Gadgets can be composed of instructions found in any module, as long as these instructions are executable, located at a known address, and do not contain invalid bytes (bytes filtered by the application during the input). In addition, because the x86 architecture uses instructions of varying sizes, there is no requirement to conform to some sort of alignment while accessing the instruction segment (.text). Thus, an attacker can use parts of an original instruction as if they were new instructions. An "ADD AL, 0x58" instruction (0x80C058), for example, if it is referenced from its third byte (0x58), will be interpreted as a "POP EAX" instruction (0x58). This trick is constantly used by attackers when searching for useful gadgets to build a chain for an attack.

Gadget chaining, performed in ROP attacks before diverting the execution stream to shellcode¹, can be used to accomplish several tasks: enable the execute bit for the memory region where the shellcode is located, copy the shellcode to a memory area

¹Shellcode: sequence of instructions that perform malicious activity.

with execute permission or disable the protection provided by the NX/XD bit. It turns out that to call functions that allow an attacker to overcome the execution bit protection through these strategies, an intruder needs to prepare the parameters to be submitted for these APIs. When exploiting a stack overflow vulnerability, for example, as arguments are also written down in the stack, theoretically it would suffice to insert them into the stack along with the data that lead to the overflow and perform the direct function call. However, previously unknown addresses (such as the exact location of the shellcode, which may be inserted in heap, for instance) or values containing improper bytes (such as 0x00 null byte, string terminator) usually require the prior execution of instructions to load the desired values into the stack. Therefore, the preparation of the arguments requires a thorough selection of gadgets for the instruction chaining, as well as the precise choice of the values to be written during the overflow. These gadgets must manipulate the data entered by the attacker into memory in order to generate the appropriate parameters to call the chosen API. The great challenge imposed by the ROP technique lies in the fact that in most cases, when executing instructions to prepare and write a value in the stack, it ends up changing other values in registers or in the stack itself. Therefore, the choice of gadgets should be very careful.

To allow an understanding of the aspects considered by the attackers when elaborating ROP attacks, the explanation of a real ROP exploit is presented in detail below. In this artifact, designed to attack "My MP3 Player" software when run by x86 processors on Windows XP SP3, the ROP technique is used to exploit a stack buffer overflow vulnerability [Blake, 2011].

2.2.1 ROP Example

Figure 2.1 illustrates the gadgets used in the aforementioned exploit, designed to attack the "My MP3 Player" software. The dashed arrows indicate when each address or data stored in the stack is used by an instruction. The continuous arrows represent the running sequence of the gadgets. The value displayed at the top of the stack (0x5D091399) corresponds to the address of the first gadget to be run during the attack. This value is positioned exactly where the original return address of the application was located, initiating the subversion of the control flow. The value 0x42424242 corresponds to a simple adjustment of displacement in the stack, since the return instruction "RETN 4" (address 0x0050A1DD), which starts the exploitation when jumping to the address overwritten in the stack, also performs the increment of the stack pointer (ESP) in 4 bytes. 0x5D09398C is the address of the second gadget. At this point, the values written in the stack are addresses of gadgets or values that the gadgets use to con-

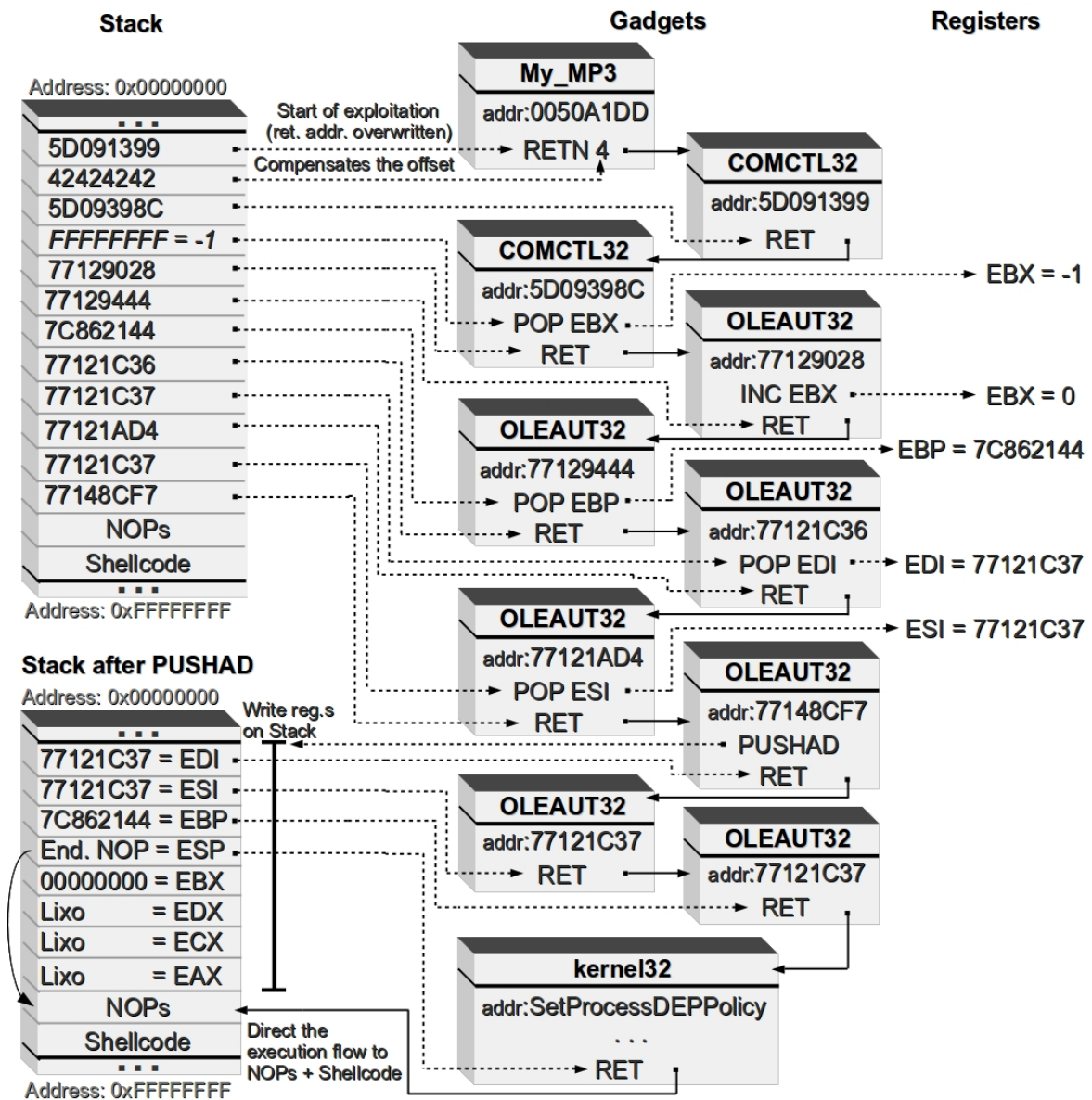


Figure 2.1. Stack structure and flow between gadgets of a ROP exploit designed to attack "My MP3 Player" software.

struct the parameters for the "SetProcessDEPPolicy" function on the stack. At the end of the gadget chain, this function is called using a parameter with a value of 0 (zero) to disable the NX/XD protection. By convention of the x86 architecture, the return address of a function is stored in the stack, above the parameters received by the function. Therefore, the attacker needs to ensure that the starting address of the NOP (No Operation) instruction sequence preceding the shellcode will be written to the stack immediately above the value 0 (zero). This address is used by the "SetProcessDEPPolicy" function as the return address, which guarantees the execution of the

shellcode immediately after the execution bit protection is disabled.

To establish this structure, the exploit uses the `PUSHAD` instruction, which stacks the value of all general purpose registers on the stack in the following order: `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI`, `EDI`. This stacking is represented in Figure 2.1 by "Stack after `PUSHAD`". The `EAX`, `ECX`, and `EDX` values are irrelevant. `EBX` is used to store the value zero (0), passed as parameter to the "`SetProcessDEPPolicy`" function. Since in "My MP3 Player" it is not possible to insert null bytes directly into the stack, the value `0xFFFFFFFF` (-1) is loaded into `EBX` by a gadget containing the "`POP EBX`" instruction. Next, a gadget with the instruction "`INC EBX`" is used to increment the value of this register, making it equal to zero (4 bytes null).

The `ESP` value written on the stack corresponds to the stack region exactly prior to the values written by the `PUSHAD` instruction. This value is used as the return address to be passed to the "`SetProcessDEPPolicy`" function, forcing it to return the execution flow directly to the shellcode (preceded by `NOPs`). The `EBP`, `ESI`, and `EDI` registers are overwritten by gadgets composed of `POP` statements, which load stack values into these registers. `EBP` is used to store the address of the function to be called (`0x7C862144`). The `ESI` and `EDI` registers store the address of a gadget composed only of the `RET` instruction. This gadget is used twice to move the pointer to the top of the stack in 8 bytes (two 4-byte shifts). After these shifts, the pointer to the top of the stack will be pointing to the address of the "`SetProcessDEPPolicy`" function, which is finally called when executing the `RET` instruction contained in the penultimate gadget.

At the end of its execution, the "`SetProcessDEPPolicy`" function uses the return address stored in the stack by the `PUSHAD` instruction (corresponding to the value contained in `ESP` at the time the `PUSHAD` instruction executes), which diverts the execution flow to the shellcode. Since the `NX/XD` protection has already been disabled by the "`SetProcessDEPPolicy`" function, the shellcode executes normally, even though it is located on the stack.

Besides Windows, ROP is used to develop exploits directed to other operating systems that implement protections based on the `NX/XD` bit. Usually, just like in Windows, the ROP technique is employed in an early stage of the attacks necessary to organize the parameters to be passed to the functions. The difference, therefore, lies in the functions available for the completion of the attacks and in how these functions receive the parameters [Le, 2010]. The use of ROP to execute arbitrary code was initially demonstrated for the x86 architecture running the Linux operating system. Later, its employment was demonstrated in attacks on other environments, such as: Atmel AVR [Francillon and Castelluccia, 2008], PowerPC [Lindner, 2009], Z80 and Sequoia AVC Advantage [Checkoway et al., 2009], ARM [Kornau, 2010] and iPhone [Miller and

Iozzo, 2009; Naraine, 2010].

2.2.2 Jump-Oriented Programming

As explained in Section 2.2, gadget chaining occurs through the return instruction, which simultaneously updates the stack and determines the program's execution flow. There are other combinations of instructions that make it possible to update the ESP (stack pointer) and EIP (program counter) registers. The leanest option is to use the `POP reg; JMP reg;` sequence of instructions. In this case, the `POP` instruction accumulates the functions of moving the stack pointer and loading the branch target address into a register, while the `JMP` instruction is in charge of diverting execution flow. This sequence of operations is the basis of the first work that proposed a variation of ROP attacks where return instructions are not used [Bletsch et al., 2011]. Because it was anchored in the use of the `JMP` instruction, it was named Jump-Oriented Programming, or JOP. Subsequently, Chen et al. [2011c] also presented ways to automate the process of creating JOP exploits.

Although convenient, the `POP reg; JMP reg;` instruction sequence occurs much less often than the `RET` instruction. This observation is true due to two main reasons. First, the probability of finding a single instruction (`RET`) is greater than the chance of finding two close operations. Although `POP` and `JMP` do not need to be consecutive, intermediate operations must not change the "state" of the data storage areas used in the attack (registers and memory locations). Therefore, it is almost always necessary for the `POP` and `JMP` instructions to be close. Secondly, the `RET` instruction is represented by a single byte, just like the `POP` instruction, but the indirect `JMP` instruction is formed by two bytes. Thus, the chance of finding misaligned jumps is lower than the chance of finding the byte that represents the return instruction in the binary text of a program.

Example 1 *A direct consequence of the scarcity of `POP reg; JMP reg;` gadgets is the eventual inability to produce an exploit solely with such gadgets. To overcome this limitation, a single gadget with this format—called trampoline—is used. This scheme is illustrated in Figure 2.2. The trampoline links functional gadgets, which effectively perform the operations necessary to execute the malicious action. Unlike the trampoline, which carries the value of the next target via the `POP` instruction, functional gadgets only need to be terminated with an indirect `JMP` instruction. However, the attacker must first save the trampoline address in a register and ensure that this value is not overwritten. In the hypothetical example of Figure 2.2, a `POP X; JMP X;` gadget is initially used to save the trampoline address in the `X` register. This implies that all*

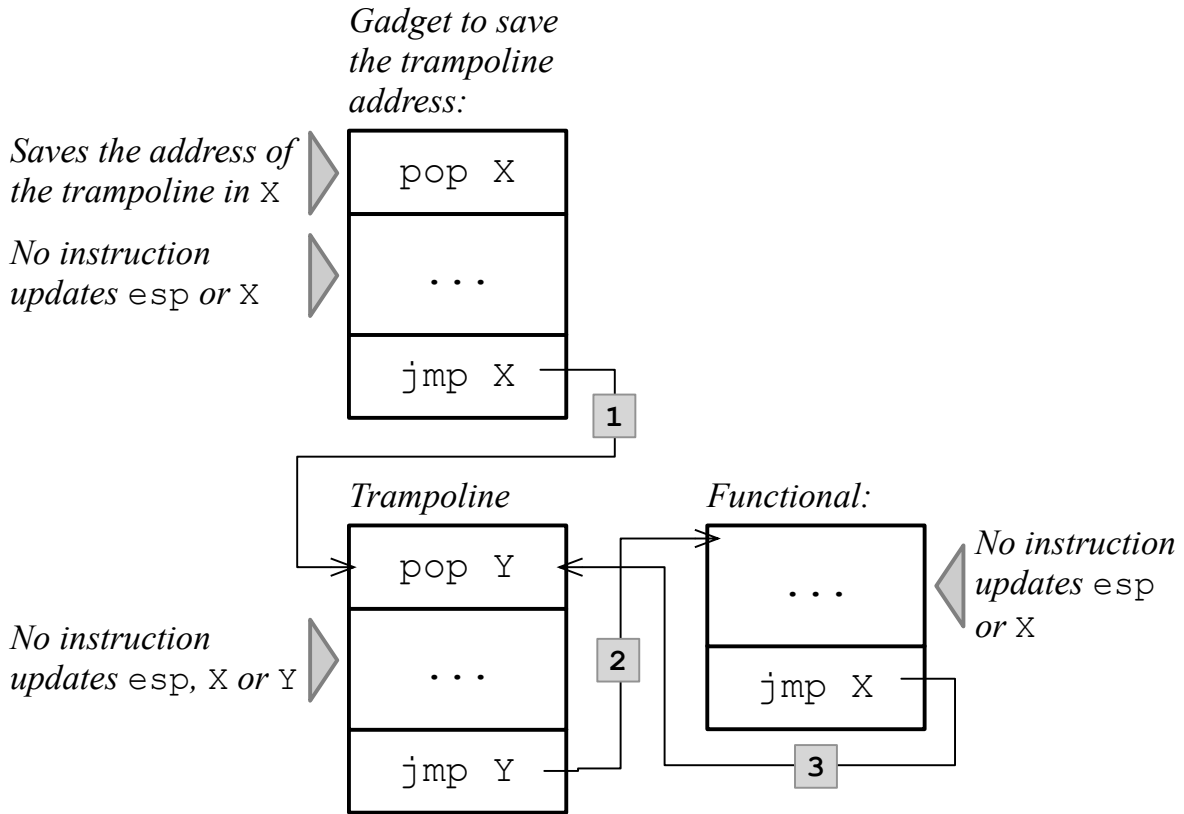


Figure 2.2. General scheme of a JOP exploit: a trampoline gadget interconnects functional gadgets.

functional gadgets must be terminated with the `JMP X` instruction, so that the flow always returns to the trampoline. The trampoline must be a `POP Y; JMP Y;` gadget where `Y` can be any register except `X` and `ESP`. Because the control flow is redirected to the trampoline after the execution of each functional gadget, the attacker can use it to read malicious addresses written in the stack through the `POP Y` instruction, and then force the branch to the next functional gadget, through the `JMP Y` instruction. The only caveat is to make sure that the remaining gadgets do not change the values of the `X`, `Y`, and `ESP` registers.

Example 2 Figure 2.3 shows how the exploit earlier seen in Figure 2.1 looks like, once implemented as a JOP attack. We used Mona [Corelan Team, 2011] to search the target process's executable memory region; hence, collecting gadgets that fill the role of initial loader, trampoline and functional sequences. For the sake of simplicity, we have omitted intermediate gadget instructions. In addition to the inclusion of the initial load and trampoline gadgets, a main departure from Figure 2.1 involves using the `ESI`

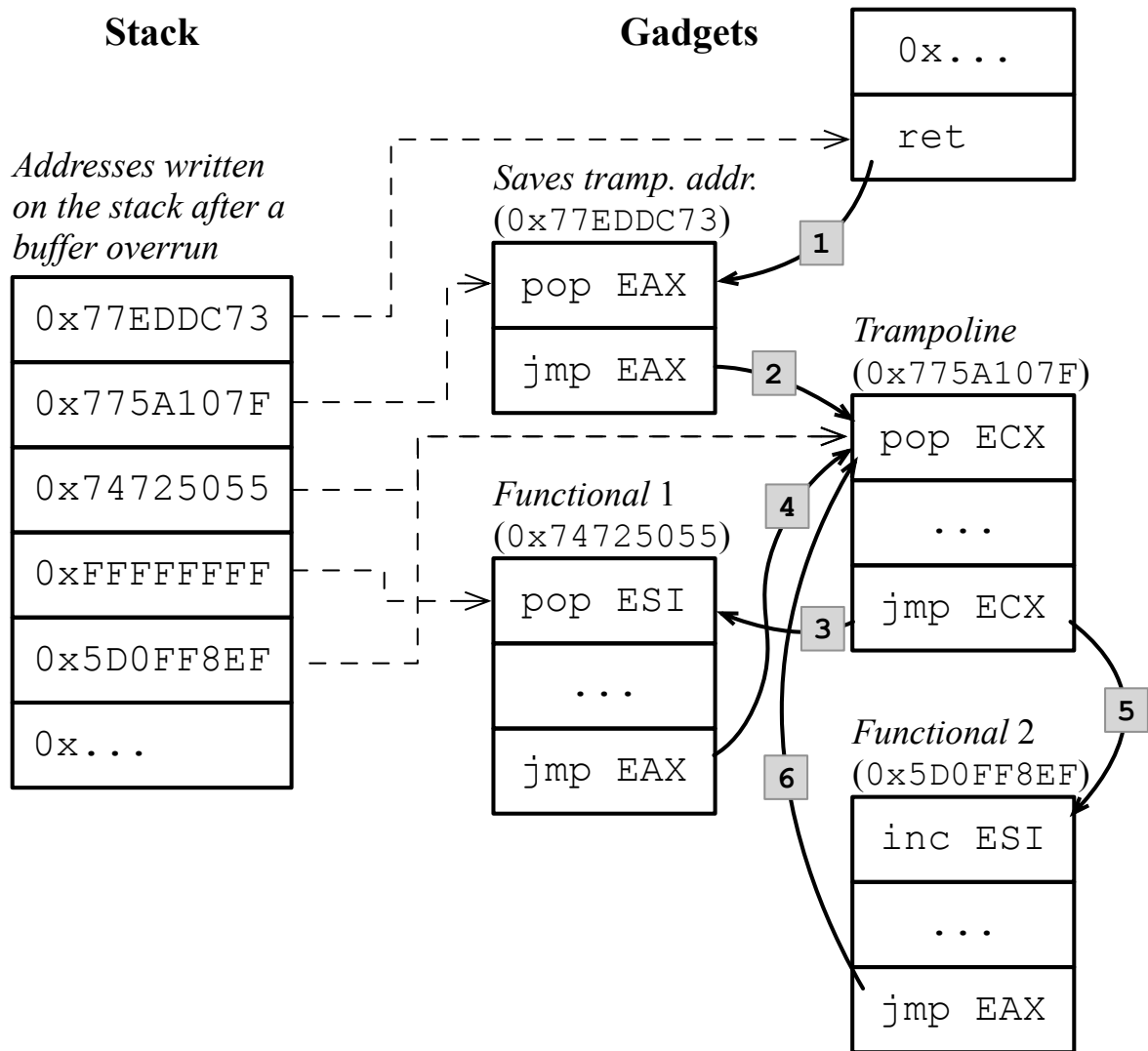


Figure 2.3. JOP version of the ROP exploit illustrated in Figure 2.1.

register to unstack and increment the value “-1” (0xFFFFFFFF) instead of *EBX*, because we could not find a JOP gadget to increment *EBX*.

2.2.3 Call-Oriented Programming

It may seem simple to adapt the JOP technique so that it uses indirect `CALLs` instead of indirect `JMPs`. Theoretically, an attacker only has to select gadgets with an instruction that can fix the side effect caused by executing a `CALL`: stacking a return address. Instructions like `ADD ESP, 4` or `POP` can play this role. However, practice shows that

the availability of gadgets with these features is quite limited [Carlini and Wagner, 2014; Goktas et al., 2014]. This makes it difficult to build attacks using only gadgets terminated by an indirect `CALL` instruction, the so called Call-Oriented Programming (COP) attack. Nevertheless, researchers have already used sequences of gadgets built with the COP technique on specific parts of exploits as a way to overcome protection mechanisms [Carlini and Wagner, 2014; Goktas et al., 2014].

Goktas et al. [2014] use a 4 gadgets COP attack in the first stage of their proof-of-concept exploitation against the CCFIR protection [Zhang et al., 2013]. The first phase of the attack exchanges the ability to control the target of an indirect `JMP` for the ability to control a return address used by a `RET`. This strategy is used to avoid the severe restrictions on indirect `JMPs` imposed by CCFIR protection. Although there are also restrictions on the abuse of `RET` instructions, Goktas et al. [2014] were able to overcome them more easily.

In another work, Carlini and Wagner [2014] use the COP technique to hide the history of executed ROP gadgets, eluding kBouncer [Pappas et al., 2013b], a well known technique that prevents ROP attacks from succeeding. The strategy used in this research is based on the finding that indirect `CALLs` almost always use memory addresses as pointers rather than registers. In order to take advantage of the greater availability of this type of branch instruction, in this COP attack, gadgets are chained by pointing the memory-indirect locations to the next gadget in the sequence. To implement this chaining, the authors use a traditional ROP attack to set up these memory locations before they run the history hiding exploit. COP gadgets restore the pre-attack state of the registers to effect this hiding.

2.3 Chapter Summary

In this chapter we have reviewed basic notions related to Return-Oriented Programming. As we have discussed, this technique enables a wide range of attacks, which have been shown to be very effective against typical defense mechanisms such as $W \oplus X$. In the next chapter, we shall discuss previous work that has dealt with ROP and proposed defenses against it.

Chapter 3

Related Work

Confirming the traditional dispute between attackers and defenders in various subjects in the area of computer systems security, since the ROP attack technique was introduced, a large amount of work has been developed both to propose mitigation strategies and to refute solutions presented. In this Chapter, we analyze the main lines of research involved in construction of the state-of-the-art defense mechanisms. Common ROP protection strategies are discussed and classified in Section 3.1. We dedicate Section 3.2 to briefly present Control-Flow Integrity, currently the most popular class of protections. Then, in the rest of the Chapter we analyze the works that relate to the approaches introduced in this thesis. In Section 3.3, we focus on work related to the inference of the peak density of indirect branches. In Section 3.4, we detail the publications that deal with aspects related to our multilayer protection.

3.1 Common ROP defenses

To oppose protection classes, as well as the specific works pertaining to each of the classes, in Section 3.1.1 we define metrics that aim to reflect the quality of ROP defenses. In Section 3.1.2 we discuss the general strategies to protect against ROP attacks that have been more often used by the works already published. In addition to classifying the main strategies, in Section 3.1.3 we group solutions according to the most used approaches to implement the presented strategies. Both classifications group protections according to similar characteristics, which facilitates the understanding of the factors that influence the quality of protection schemes.

3.1.1 Evaluation Metrics

When we started this project, there was no set of metrics designed to express the main factors to be considered when assessing protections against ROP attacks. Therefore, we proposed metrics that we used during the analysis of the characteristics, virtues and deficiencies inherent to each of the evaluated works. Likewise, we believe they are a valuable tool in the process of analyzing the benefits and limitations inherent in new protections against ROP attacks. The following metrics were considered relevant:

- Types of attacks blocked.
- Average overhead.
- Exceptions.
- Practical feasibility of the prototype.

In the following Sections, we detail each of these metrics.

3.1.1.1 Types of Attacks Blocked

As we mentioned in Section 2.2, ROP attacks were initially conceived through the interconnection of gadgets terminated by the `RET` instruction. The ability to control the contents of the stack in arbitrary code injection attacks, coupled with the vast availability of this instruction in binary files, made this operation the ideal tool to link the execution flow between the pieces of code that make up a ROP attack [Roemer et al., 2012]. As they all had this characteristic in common, the first proposals for protections against these malicious codes were intended to verify specific aspects related to the behavior of the return instruction [Chen et al., 2009; Davi et al., 2009; Bania, 2010; Li et al., 2010].

As we saw in Section 2.2, new works later demonstrated the possibility of using indirect "jump" (`JMP`) [Checkoway et al., 2010; Chen et al., 2011c] and "function call" (`CALL`) [Carlini and Wagner, 2014; Goktas et al., 2014] instructions to link gadgets. Due to the existence of this variety of instructions useful in the construction of ROP exploits, this work proposes to distinguish the types of attacks blockable by each protection as the first metric for evaluating ROP defenses. When analyzing the functioning of a defense mechanism, we suggest indicating whether the protection is able to identify gadget chains interconnected by `RET`, `JMP` or `CALL` instructions. This distinction makes it possible to clearly separate the types of blocked malicious code, a major factor in defining the degree of security offered by each solution.

3.1.1.2 Overhead

One of the main factors impacted by the implementation of security mechanisms in computer systems is the increase in workload (overhead) and its consequent reduction in system performance. This aspect also affects protections against ROP attacks. For this reason, we propose to establish the loss of performance that the solution imposes on protected applications as the second evaluation metric to be considered.

Since there is already a consensus that performance degradation is an important factor for the practical feasibility of computer security solutions, most ROP defenses present studies of the impact of these protections on systems performance. In all the works we evaluate, the overhead is expressed through the average percentage of additional execution time imposed by the solution. That is, if an application ran on average in 10 seconds and, after the implementation of a protection against ROP attacks, starts to run in 11 seconds, the average overhead imposed by this solution is 10%. This is the overhead representation adopted throughout this work.

3.1.1.3 Exceptions (False Positives)

Many ROP protections are based on coding conventions or compilation standards to apply their detection strategies. However, it is common to experience situations that are outside the execution patterns established for an architecture or operating system. In general, these cases stem from custom codes written directly in assembly language. The option to develop assembly code is usually related to a need for performance optimization or the requirement to adapt to some specific hardware.

Unfortunately, the conventions that sustain some ROP protections are not respected on a vast number of occasions. This disparity between the standard expected by a defense and the different format adopted by many codes leads to situations in which authentic codes are mistakenly classified. In other words, excerpts of instructions that should normally be executed can be blocked or corrupted, resulting in false positives

Although some protections address the exceptions found in the environments where their prototypes have been tested, it is difficult to predict which exceptions may arise in environments with other operating systems and applications. Therefore, in general, the greater the number of exceptions cataloged for a given solution, the lower its reliability. With that in mind, we establish the number of exceptions (false positives) of each protection as our third assessment metric.

While some studies present examples of situations that resulted in exceptions during the testing of their prototypes, other work do not indicate this peculiarity,

despite being clearly subject to the same situations. Therefore, recording only the cases of false positives explained in the respective publications would not result in an accurate estimate. Instead, we catalog the cases of false positives reported in the literature and relate them to the protection strategies (discussed in Section 3.1.2). Then, we analyze the operation of the protection mechanisms to identify which exceptions affect each of the evaluated defenses.

It is important to note that the exceptions we have identified do not account for cases that represent possible security breaches (false negatives), as the evaluated works do not test these possibilities. However, we discuss potential flaws in Section 3.1.2, while protections are described.

3.1.1.4 Practical Feasibility of the Prototype

Some of the defense mechanisms have a tolerable protection quality when analyzed from the perspective of the first three evaluation metrics, but they lack practical feasibility for protecting real applications. On the other hand, there are solutions that are able to offer immediate protection for systems in production environments despite having deficiencies in some of the previously defined metrics. Therefore, we included a metric of practical feasibility of the prototype in our evaluation model. It reflects the possibility of using the protection in full operation immediately. This can be an important characteristic in situations in which the evaluator wishes to point out the best solution among those that allow immediate employment. This scenario can occur, for instance, in the case of the need to protect applications that contain vulnerabilities for which security updates have not yet been made available (the so-called 0 days).

Among the solutions we have evaluated, some have only theoretical protections and do not implement prototypes. Therefore, we consider them unfeasible in practice. The protections that require the availability of the table of symbols for executables were also considered impractical, since most executables do not keep this information. In addition, protections that require new hardware structures were also considered unfeasible, since they cannot be used immediately.

3.1.2 Protection Strategies

We noticed that ROP protections have been using similar strategies for detecting attacks. Then, we propose to group the solutions into classes to facilitate the understanding of aspects common to the protections belonging to the same segment. We suggest a taxonomy composed of the following elements:

- Randomization.
- Constructing Shadow Stacks.
- Checking instructions prior to return addresses.
- Checking targets of branches.
- Checking the entry points of functions.
- Controlling the frequency of return instructions.
- Checking the top of the stack pointer.

We detail each of these classes and their protections in Sections 3.1.2.1 to 3.1.2.7.

3.1.2.1 Randomization

The randomization strategy is based on the idea of shuffling the memory addresses assigned to program instructions. Thus, attackers are unable to define the address to which the flow of execution should be transferred. This strategy is actually an improvement on the ASLR technique [PaX Team, 2003b], presented in Section 2.1.2. ASLR is implemented in most modern operating systems. It works well when properly implemented and integrated with the applications. However, it can offer the following flaws:

- Some libraries are not compatible with ASLR or can be compiled without this option. In these scenarios, it has already been demonstrated that it is possible to use binary codes from these libraries to infer the location of other libraries useful for an attack, such as the libc library [Roglia et al., 2009].
- Instruction addresses are fixed in most application binaries. With the exception of Position Independent Code (PIC), only the addresses of loaded libraries vary. Therefore, attackers can use the alternative of redirecting the control flow to gadgets built with instructions from the application itself [Schwartz et al., 2011].
- When an attacker is able to retrieve some address from the process during an information leakage attack [Sotirov and Dowd, 2008; Vreugdenhil, 2010; The H Security, 2009], it obtains enough information about the memory layout to proceed with a ROP attack.

- As ASLR scrambles only a portion of each memory address (32-bit Windows 7 changes bits 17 to 24, for example), a brute force scheme can be used to test all possible addresses where the shellcode might be located [Shacham et al., 2004].
- Another problem arising from the partial change of addresses is the possibility of partial overwriting of the return address during the exploitation of a buffer overflow vulnerability in the stack. In 32-bit architectures, for example, ASLR changes only the 16 most significant bits of the address. Therefore, in specific situations, attackers can precisely calculate the size of the malicious entry to overwrite only the least significant fraction of the return address [Durden, 2002; Rapid7, 2007], making it point to a chosen gadget chain.
- There is also another class of techniques to overcome ASLR generically called Spraying [Blazakis, 2010; Wei et al., 2010; Eeckhoutte, 2011]. These strategies are based on forcing a sequential allocation of several copies of the shellcode preceded by large amounts of NOP instructions. For performance reasons, operating systems ensure that new blocks are always aligned during allocations. With this, an attacker can induce the allocation of new blocks and predict the relative position of an address within those blocks.

Due to these weaknesses identified in ASLR implementations several proposals for ROP protections return to the idea of inserting randomness to instructions addresses [Bania, 2010; Wartell et al., 2012; Pappas et al., 2013a]. When well implemented, the randomization strategy considerably increases the complexity of developing ROP attacks. However, there are deficiencies inherent in this strategy related to the possibility of exploitation through information leakage or brute force attacks.

Most solutions that apply randomization seek to do it at the level of blocks of instructions or individual instructions, precisely to increase the degree of scrambling (entropy) and, consequently, increase the number of attempts to be tested in brute force attacks. The problem is that this randomization of small blocks of instructions can corrupt signed codes, blocks of instructions that perform checks on themselves – like checksums – or codes that change during execution – like obfuscated codes. In addition, the randomization strategy is unable to protect code compiled-on-demand (Just In Time), common in applications that support environments with interpreted languages – such as Java, JavaScript, Flash, .Net and SilverLight [Wei et al., 2010].

3.1.2.2 Building Shadow Stacks

Long before the emergence of ROP attacks, the strategy of building shadow stacks to protect against arbitrary code execution attacks had already been devised. Initially, this solution was proposed to avoid the consolidation of code injection attacks in the stack segment [Payer and Payer, 2001; Vendicator, 2000; Chiueh and Hsu, 2001]. However, recently the use of shadow stacks has also been widely used in the design of protections against ROP attacks. In some proposals the additional stack is implemented via software [Sinnadurai et al., 2008; Shuo et al., 2012; Chen et al., 2011a; Davi et al., 2011a, 2012; Huang et al., 2012b; Davi et al., 2011b]. In others it is built directly on the hardware [Kayaalp and Ozsoy, 2012; Kayaalp et al., 2012; Intel, 2019].

This strategy consists of creating a safe copy of the return addresses written on the stack for comparison when a `RET` is executed. As shown in Figure 3.1, in addition to writing in the traditional stack, the return address is also written in an additional stack structure (known as shadow stack) whenever a `CALL` is executed. This additional stack must be protected against writing by user level instructions to prevent an attacker from changing the return addresses on the two stacks. The matching of addresses at the top of the two stacks is checked at the moment when return instructions are executed. Thus, if a return address stored in the original stack is changed, the attack is identified.

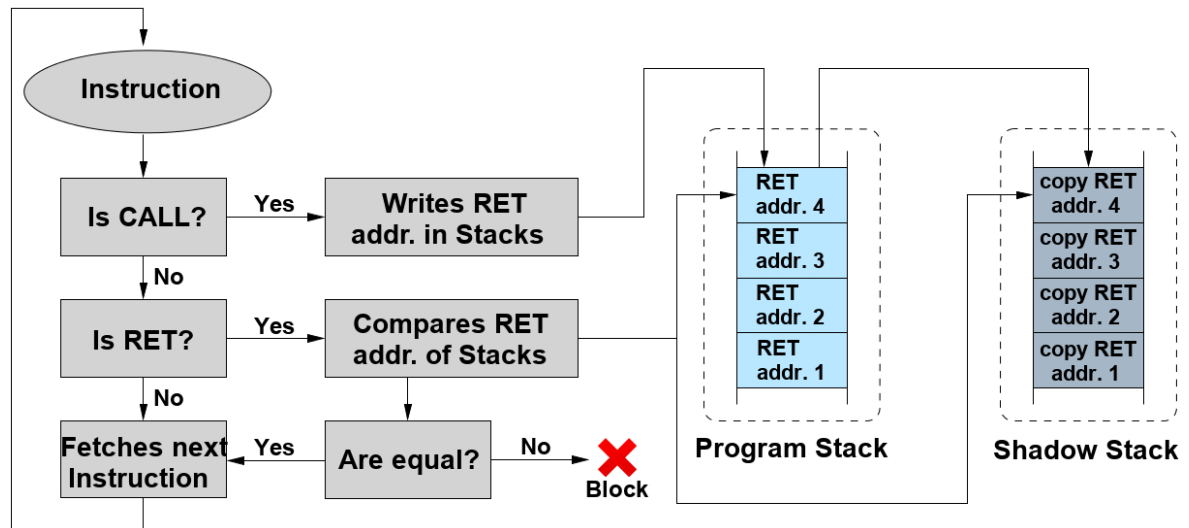


Figure 3.1. Protection strategy based on building a shadow stack [Davi et al., 2011b].

In addition to ROP, this strategy is able to prevent the consolidation of “return-into-libc” [Tran et al., 2011] attacks, mentioned in Section 2.2. However, it has the drawback of classifying some authentic codes as attacks, incurring false positives. Typ-

ically, these misclassified codes are related to unconventional programming practices used to handle low-level tasks or to manually optimize assembly code [Onarlioglu et al., 2010]. As highlighted by Davi et al. [2011b], these false positives can occur in three situations:

- When a function does not return or does so using a branch instruction other than a `RET`.
- When a function is called without using a `CALL`.
- When the return address is genuinely overwritten by the application.

These situations conflict with the coding standard established for the x86 architecture, since it was designed to optimize the process of calling and returning from functions using the traditional `CALL` and `RET` instructions. Unfortunately, there is a considerable amount of code that does not follow this convention. The following examples of exceptions have already been cataloged by works that implement the strategy of building a shadow stack:

- Codes using the `Setjmp` and `Longjmp` system calls [Davi et al., 2011b; Payer and Payer, 2001; Sinnadurai et al., 2008; Kayaalp and Ozsoy, 2012; Kayaalp et al., 2012].
- Position Independent Code (PIC) [Chen et al., 2011a].
- The signal processing code in UNIX-derived systems [Davi et al., 2011b].
- The code that implements the lazy binding of dynamic libraries in UNIX-derived systems, where this is the standard procedure [Davi et al., 2011b].
- The exception handling code inserted by the GNU C++ compiler [Payer and Payer, 2001; Davi et al., 2011b].
- Threads created at the end of a sequence of function calls that begin to return shortly after creation [Payer and Payer, 2001].

Although some protections based on the use of the shadow stack deal with the exceptions found in the environments where their prototypes were tested, it is difficult to predict which exceptions may arise in environments with other operating systems and applications. This unpredictability reduces the reliability of this strategy for use in production environments.

3.1.2.3 Checking the instruction positioned before return addresses

Because of the convention used to make function calls in the x86 architecture, the execution flow is usually returned to the calling function at the position immediately following the calling instruction (`CALL`) when a `RET` instruction is executed. Based on this characteristic, some studies proposed the strategy of verifying whether there is a `CALL` instruction in the position immediately before the target address of return instructions. In other words, if the return address indicates a position that is not preceded by a `CALL` instruction, the protection mechanism assumes that the process is experiencing a ROP attack.

As with the strategy of building a shadow stack, checking the instruction positioned before return addresses can result in incorrect classification of authentic codes, causing false positives. This occurs in situations where the return instruction (`RET`) is used as a generic branch instruction, without being related to a function call (`CALL`) instruction. Two examples of exceptions that affect the strategy of building a shadow stack also trigger the occurrence of false positives with the strategy of checking the instruction positioned before the return address:

- The signal processing code in UNIX-derived systems [Davi et al., 2011b].
- The code that implements the lazy binding of dynamic libraries in UNIX-derived systems, where this is the standard procedure [Davi et al., 2011b].

In addition to allowing false positives, the strategy of checking the instruction positioned before return addresses leaves room for an attacker to build a ROP attack in specific situations. An opponent can, for example, use unaligned (unintended) gadgets that start in a position where the previous bytes correspond to a `CALL` instruction. Another possibility is to use gadgets positioned immediately after an authentic `CALL` instruction that did not call the function that is returning.

Authors who use the strategy of checking the instruction positioned before return addresses claim that the number of gadgets that satisfy these prerequisites is low. This scarcity makes building a complete gadget chain for use in ROP attacks impractical in most situations. In Section 5.1.2.1 we present **enhanced Call-preceded Constraint (e-CC)**, an improvement of the strategy of checking the instruction positioned before return addresses. Through this improvement, we were able to reduce almost 35 times the number of gadgets available for building a ROP exploit, as presented in Section 5.2.1. This reduction makes it impossible, for example, to consolidate the proofs of concept presented by Davi et al. [2014] and Goktas et al. [2014].

3.1.2.4 Checking targets of indirect branches

A strategy that has been studied for several years in an attempt to contain various types of arbitrary code execution attacks is to pre-identify the authentic addresses of indirect branches so that these addresses are checked the moment the branches are executed [Abadi et al., 2005]. Since the goal of this strategy is to restrict branches in the execution flow to only those addresses that were originally planned during the development of the program, ROP attacks have also become a target of this type of protection.

This type of protection establishes sets composed of valid targets for indirect branches in the execution flow of programs. These sets can be built during compilation [Demott, 2012; Bletsch et al., 2011], by extracting the information directly from the binary file in a step subsequent to the compilation [Davi et al., 2011a, 2012], or by monitoring the branches executed during the execution of the program, in a training stage of the protection [Xia et al., 2012].

The need to identify valid branch targets during a training stage can lead to the occurrence of false positives at the time the application is actually running. These mistakes occur because it is unlikely that all the flow paths for large and complex applications will be triggered by the mass of tests used during training [Hoglund and McGraw, 2004]. Usually the input data generated to test the application, especially when the program’s source code is not available, will be able to induce the most frequent execution paths during the use of the application. However, some specific combinations of inputs that have not been tested during training can be submitted to the process, causing the execution flow to branch to an uncatalogued path. Consequently, authentic executions of the software can be mistakenly classified as ROP attacks.

The implications of solutions applied during program compilation are discussed in Section 3.1.3.1. Solutions based on static binary instrumentation (detailed in Section 3.1.3.2) are not able to correctly identify all indirect branch targets [Xia et al., 2012], even if the table of symbols is available in the binary file – which is unusual in commercial software. In addition, inserting instructions in the binary code to check conditions before branch instructions [Davi et al., 2011a, 2012; Bletsch et al., 2011] only works against ROP attacks on architectures where the size of the instructions is fixed. Only in these environments an attacker can’t access unaligned memory addresses to force the execution of originally unintended instructions. To enable the use of this strategy in the x86 architecture, where the size of the instructions is variable, Bletsch et al. [2011] avoid the exploitation of unaligned instructions through a mechanism that forces the code alignment.

An undesirable characteristic common to protections that use the strategy of checking authentic targets of indirect branches is the high computational overhead they impose [Tran et al., 2011]. This performance deficiency stems from the need to make many comparisons, depending on the number of possible branch targets, and the complexity of implementing the checking mechanisms. Despite this, researchers point to the strategy of checking authentic targets of indirect branches as a promising scheme for protection against ROP attacks [Roemer et al., 2012]. Indeed, Control-Flow Integrity (CFI) [Abadi et al., 2005] is probably the most prominent subclass of protections and it overlaps with the idea of checking the authenticity of branch targets. Considering its vast appearance in the literature, we dedicated Section 3.2 exclusively to discuss CFI.

3.1.2.5 Checking the entry points of functions

The strategy of checking the entry points of functions consists of verifying if the destination of an indirect branch corresponds to the first instruction of the target routine when transferring the execution flow from one procedure to another. For this, the proposed solutions perform checks whenever an indirect branch instruction is executed. Most protections perform this verification ensuring that the destination of the branch to be executed corresponds to an instruction belonging to the function currently being executed or to the entry point of another routine [Huang et al., 2012b; Kayaalp and Ozsoy, 2012; Kayaalp et al., 2012; Xia et al., 2012]. Huang et al. [2012a] chose to check if the execution flow passed through the entry point of the current procedure whenever an indirect branch instruction is executed.

Although the works that use the strategy of checking the entry position of functions do not indicate the existence of false positives, there are customized codes that use indirect branches to jump to another routine in destinations that do not correspond to their entry point. Meng and Miller [2016] identifies two situations in which this exception occurs: procedures sharing code and non-contiguous functions. In these cases, this strategy will mistakenly indicate the occurrence of a ROP attack.

3.1.2.6 Controlling the frequency of return instructions

Typically, the gadgets used in a ROP attack are extremely short, hardly containing more than five instructions. This is an inherent characteristic of ROP attacks because the longer the sequence of instructions, the greater the probability of an operation between these instructions that changes the status of the memory or of a register, compromising the attack. This status change is commonly referred to by attackers as

the "side effect" of a gadget. To avoid these side effects, almost always the gadgets chosen by the attackers are extremely short.

Given this observation, several authors have invested efforts in a strategy to control the frequency of return instructions as a way to detect the execution of gadget chains. Some have proposed work that verifies the peak frequency of return instructions by real-time monitoring of each executed return instruction [Chen et al., 2009; Davi et al., 2009]. Other solutions adopted the posture of accounting for the frequency of previously executed return instructions, through the analysis of a branch buffer available in hardware (Branch Trace Store buffer) [Yuan et al., 2011] or by checking the distance between the stored return addresses in the most recently released area of the stack [Han et al., 2013; Jiang et al., 2011].

In Section 4.1.1 we generalized the strategy of interposing long gadgets with small gadgets [Carlini and Wagner, 2014; Göktas et al., 2014] to demonstrate that it is possible to overcome these protections based on controlling the frequency of return instructions executed. This strategy also has the following deficiencies:

- A sequence of close function returns, typical of functions with tail recursion ¹, can induce protections based on the strategy of controlling the frequency of return instructions to mistakenly block the execution of authentic codes.
- In the case of protections that analyze the frequency of return instructions traversing the stack, an attacker can forge the stack with any values between the return addresses in order to overcome this defense. It is enough that the gadgets have some instruction that increases the top of the stack pointer, for instance.
- In this same scenario (analysis of the proximity of return addresses in the stack), functions that contain few variables and parameters can present close return addresses, leading to the occurrence of false positives. This possibility may even increase if the compilation optimization that employs the base pointer register as a general purpose register has been used, as it forces the release of spaces reserved in the stack to store function frame pointers (also known as base pointers).

3.1.2.7 Checking the top of the stack pointer

In ROP-based arbitrary code execution attacks, an attacker must be able to control the contents stored in the program counter (EIP register on 32-bit architectures, or RIP on 64-bit architectures) and the pointer to the top of the stack (ESP register on

¹Tail recursive functions are those in which the recursive `CALL` is the last instruction executed.

32-bit architectures, or RSP on 64-bit architectures) [Erlingsson, 2007; Dai Zovi, 2010]. Controlling the program counter allows the attacker to hijack the execution flow to the first gadget. Controlling the pointer to the top of the stack, on the other hand, allows other gadgets to be linked through return addresses chosen by the intruder.

Depending on the vulnerability exploited, the attacker inserts the malicious entry into another segment of the targeted program's memory, rather than the stack. In such cases, the pointer to the top of the stack needs to be adjusted during the ROP attack. This adjustment happens running gadgets that make the stack pointer reference the exact position of the malicious entry, positioned outside the address limits reserved for the stack. This operation is known by the expression "stack pivoting".

Considering that the pointer to the top of the stack should never store an address that does not belong to the range of addresses reserved for the stack, Bania [2010] and Fratric [2012] proposed the strategy of checking the pointer to the top of the stack. An attack is identified if the ESP/RSP register contains an address that does not belong to the stack segment.

Although effective in combating attacks that corrupt other memory segments, the strategy of checking the pointer to the top of the stack is not useful in attacks that affect the stack (eg, buffer overflow in the stack). This is because the attacker's input data will already be positioned in the stack region and therefore it is not necessary to move the stack pointer to another segment. In addition, as mentioned in Section 1.1, in 2011 an implementation of this strategy was incorporated into the Windows 8 Developer Preview and surpassed a few days after its release [Rosenberg, 2011; Son, 2011; Tung, 2011]. Since the check is performed the moment a Windows API is called, the attacker can run gadgets that copy the malicious entry to an area belonging to the stack before executing any API calls. Therefore, this strategy may become ineffective if the verification of the ESP/RSP is not performed regularly, as the attacker may reposition the malicious code on the stack before the check is carried out. We are not aware of protections that employ frequent checks of this type, probably because such a procedure should result in inviable overheads at runtime.

3.1.3 Implementation Approaches

The various protection strategies discussed in Sections 3.1.2.1 to 3.1.2.7 can be applied to systems to be defended through various approaches. These implementation approaches directly impact the quality of the proposed solution and have peculiar characteristics. Therefore, we propose the following taxonomy for the classification of protections according to the approaches chosen to apply preventive measures against

ROP attacks:

- Compilation.
- Static Binary Instrumentation.
- Dynamic Binary Instrumentation.
- Hardware Adaptation.
- Virtualization.
- Code Emulation.

Some solutions use a combination of these classes of approaches to implement their protection strategies against ROP attacks.

3.1.3.1 Compilation

As the name implies, the compilation approach is used by solutions that make modifications to an application's binary code when its source code is compiled. The main disadvantage of this approach stems from the need to obtain the source code, which is not available for most commercial software. In addition, protections based on compilation require that all system developers choose to employ the solution for it to be effective. Otherwise, a single library that does not have the protection enabled can be exploited by an attacker.

Unfortunately, history has shown that many software vendors prioritize performance over security. This observation can be confirmed by the evolution of some protections incorporated by the Visual Studio compiler. Microsoft has released a mechanism called "SafeSEH" (Safe Structured Exception Handler) to restrain attacks that exploit Structured Exception Handlers (SEHs). SafeSEH added new stack protection mechanisms to safeguard the records for exception handling used by Microsoft operating systems [Microsoft, 2012]. For this, the `/SafeSEH` linking flag has been available in the Visual Studio compiler since the ".Net 2003" version. However, a strategy often used by attackers to overcome SafeSEH is simply to use a library loaded by the process that has not been compiled with the `/SafeSEH` option, as this is a recurring scenario in many vulnerable applications.

The exploitation of applications and libraries compiled without the `/SafeSEH` option became so common that in 2009 an extension for SEH security called Structured Exception Handler Overwrite Protection (SEHOP) was launched. This new protection operates through checks made only during the execution of a process [Microsoft,

2010]. Thus, this mechanism is independent of the choices made during the software compilation stage.

A similar evolution has occurred with ASLR (described in Section 2.1.2) and DEP protections (described in Section 2.1.3), which originally depended respectively on the use of the `/DYNAMICBASE` and `/NXCOMPACT` compilation flags to activate the protections. Upon realizing that many vulnerable software continued to be exploited for not adhering to compilation solutions, Microsoft incorporated in its operating systems the possibility of forcing ASLR [Microsoft, 2011] and DEP [Microsoft, 2018] for all processes, despite any compatibility issues.

The anti-ROP protection strategies employed in solutions that use the compilation approach vary widely. However, one of them applies only to this approach. Two works [Li et al., 2010; Onarlioglu et al., 2010] adopt the same strategy, viable only through compilation, as both solutions adjust the binary code to be generated preventing it from presenting useful branch instructions for chaining gadgets. However, while one work prevents only the existence of unwanted return instructions (RET) [Li et al., 2010], the other prevents the abuse of any kind of indirect branch [Onarlioglu et al., 2010].

3.1.3.2 Static Binary Instrumentation

The concept of Static Binary Instrumentation usually designates the manipulation of instructions contained in compiled binary files [Laurenzano et al., 2010]. Normally these modifications are performed with the help of specialized tools in this type of task, such as DynInst [Buck and Hollingsworth, 2000], Vulcan [Srivastava et al., 2001] and DTrace [Cantrill et al., 2004]. These softwares replace the original instructions in the executable file with instructions known as "trampoline", which deviate the execution flow to a new code indicated by the user.

In this work we expand the meaning of the expression Static Binary Instrumentation to encompass all solutions that analyze or modify executable binary codes after their compilation and before those codes start their execution. So, we included in the group of solutions that use the Static Binary Instrumentation approach the proposals that:

- Extract information from binary files using disassembly tools (disassemblers).
- Make changes to the code, or its structure, at the time of loading it into memory.
- Apply patches to system libraries.

- Insert new instructions into executable binary files.

The solutions belonging to the Static Binary Instrumentation approach differ from the works that compose the Dynamic Binary Instrumentation class (detailed in Section 3.1.3.3) only in the fact that in the Dynamic Instrumentation the manipulations of the binary code happen in real time. In other words, the binary code is analyzed and possibly modified during the execution of the program in works that use the Dynamic Binary Instrumentation approach.

In general, strategies implemented through static approaches offer less overhead for protected applications than dynamic approaches [Laurenzano et al., 2010]. This lower cost happens because the Static Binary Instrumentation introduces only the codes used to check the conditions provided by the chosen protection strategy. In contrast, Dynamic Binary Instrumentation inserts an additional overhead because the instrumentation tool performs context changes with the instrumented application, in addition to disassembling codes and generating instructions. All of this is done during the execution of the instrumented application.

On the other hand, Static Binary Instrumentation solutions that require an accurate disassembly of the binary code, so that they can be modified, only work if debugging information – such as the table of symbols – is available [Pappas et al., 2012; Kayaalp and Ozsoy, 2012; Kayaalp et al., 2012; Huang et al., 2012a]. However, this debugging information is typically removed from software intended for production environments for space saving and intellectual property protection.

Another deficiency related to protections that adopt the Static Binary Instrumentation approach falls on implementations that use the insertion of instructions in the original executable code [Pappas et al., 2012; Kayaalp and Ozsoy, 2012; Kayaalp et al., 2012; Wartell et al., 2012; Pappas et al., 2013a; Davi et al., 2011a, 2012; Bletsch et al., 2011]. This procedure suffers from the same problems related to the randomization strategy. It can corrupt signed codes, blocks of instructions that perform checks on themselves – like checksums – or codes that change during execution – like encoded parts. In addition, static instruction insertion is also unable to protect code compiled on demand (JIT-compiled), common in applications that support environments with interpreted languages, such as Java, JavaScript, Flash, .Net and SilverLight [Wei et al., 2010]. It is also important to emphasize that inserting instructions for checking conditions before branch instructions, as performed by Davi et al. [2011a] and Davi et al. [2012], only works as a protection against ROP attacks in architectures where the size of the instructions is fixed. Otherwise, some scheme that forces the alignment of instructions must be imposed, as established by Bletsch et al. [2011].

3.1.3.3 Dynamic Binary Instrumentation

As mentioned in Section 3.1.3.2, the solutions classified in the Dynamic Binary Instrumentation category differ from the proposals included in the Static Binary Instrumentation class mainly because they perform manipulations in binary codes after the start of their execution. For this, typically JIT (Just-in-time) binary instrumentation tools are employed, as they allow the analysis and eventual modification of the code while it is executed. Pin [Luk et al., 2005], Valgrind [Nethercote and Seward, 2007], Strata [Scott et al., 2003] and DynamoRIO [Bruening, 2004] are examples of systems that support this functionality. For this support to work, before an instruction is executed by the processor, these frameworks intercept the instruction, generate and execute new code, and guarantee that the framework will regain control of the processor after the instruction has been executed.

The major disadvantage of the Dynamic Binary Instrumentation approach stems from the computational cost to execute context changes between the instrumentation tool and the monitored application. Studies show that Pin, Strata, DynamoRIO and Valgrind frameworks present respectively an average overhead of 230%, 260%, 490% and 750% to perform a simple task of counting the number of instruction blocks executed [Luk et al., 2005; Guha et al., 2007]. On the other hand, solutions that use the Dynamic Binary Instrumentation approach do not require additional information about the binary code – such as tables of symbols – nor do they require the source code of the applications. In addition, these systems allow the analysis of programs that generate code compiled on demand (JIT-compiled).

3.1.3.4 Hardware Adaptation

Some authors have proposed adaptations in the hardware structures of the processors [Kayaalp and Ozsoy, 2012; Kayaalp et al., 2012; Demott, 2012] aiming for an efficient implementation of ROP protection strategies regardless of the availability of application codes or tables of symbols. Other works [Xia et al., 2012; Pappas et al., 2013b; Yuan et al., 2011] present schemes that reuse hardware structures and special instructions, designed for other purposes, to enable the implementation of protections against ROP attacks. For classification purposes, both sets of proposals are considered to belong to the Hardware Adaptation approach. However, we do not observe a protection that uses exclusively the Hardware Adaptation approach. Software-level adjustments are always necessary.

The main argument supporting schemes that adopt hardware adaptation is to reduce the execution time overhead imposed over the protected applications. However,

the impact of incorporating changes to the architecture of modern processors must also be considered. Complex modifications may require the inclusion of many circuits, which can harm the overall performance of the processor. In addition, results presented by prototypes may not be reliable, because it is very difficult to simulate all aspects relevant to the final performance of the adapted architecture. In general, the mechanisms used to simulate the operation of hardware protections disregard some characteristics considered less impactful. However, the importance of these characteristics can be underestimated until a complete simulation is performed in a real processor.

3.1.3.5 Virtualization

Virtual machines are also used to implement protection strategies against ROP attacks. Some works [Shuo et al., 2012; Jiang et al., 2011] explore specific features offered by virtualizers to detect attacks in real time. These protections use the virtualization assistance incorporated in some modern processors to trigger the context switch between the application and the virtualizer, which allows the execution of procedures to check the memory area belonging to the monitored process. The processor calls a treatment code previously registered by the virtualization system if the virtualized operating system (guest) tries to perform some privileged operation in environments with hardware assistance for virtualization. Virtualization-based protections [Shuo et al., 2012; Jiang et al., 2011] force the execution of privileged operations at strategic moments in the execution of the monitored application to guarantee the transfer of the execution flow to the check codes. These verification codes are previously added into a handler registered by the virtualizer.

The virtualization approach is limited to architectures where specific structures exist to assist virtualization tasks, such as x86. In architectures where there is no such hardware support, the performance of this approach becomes impractical. In fact, even with hardware support, it is expected that solutions that adopt this approach present an overhead comparable to solutions that use the Dynamic Binary Instrumentation approach, since virtual machines also constantly change context between the monitored application and the virtualizer code. Generally speaking, works that use virtualization to implement protections against ROP attacks present a better average overhead than solutions that use Dynamic Binary Instrumentation (see values in Section 3.1.4). However, it is necessary to carry out additional tests to confirm these overheads, since in the studies that we evaluated the experiments were executed with a limited set of applications and with the protection applied to a small group of libraries.

3.1.3.6 Code Emulation

As far as we know, ROPGuard [Fratric, 2012] was the first protection to suggest using an approach based on code emulation. Among the various detection strategies that ROPGuard uses, this defense adopts a scheme of simulating the instructions to be executed after the return of critical functions. ROPGuard considers critical functions those routines that attackers use to disable the protection of the NX/XD bits or otherwise allow the execution of the shellcode. Code emulation is used in ROPGuard to verify that subsequent return addresses are executable and that these addresses are positioned immediately after a `CALL` instruction (strategy described in Section 3.1.2.3).

To perform code emulation, ROPGuard simulates some instructions commonly used in gadgets to change the stack pointer value: `PUSH`; `POP`; `ADD ESP, value`; `SUB ESP, value`; `RET`. The rest of the instructions are simply passed on. When a return instruction is found, the protection performs the mentioned checks. The simulation ends when an instruction limit set in a configuration file is reached, or when a branch instruction is reached. Note that ROPGuard was designed to block only ROP attacks built with gadgets chained by `RET` instructions. However, the author stresses the possibility of extending code emulation in a future work to simulate other instructions, including indirect `JMPs` and `CALLs`.

Although it is also limited to detecting traditional ROP attacks in which gadgets are interconnected only by return instructions, ROPscan [Polychronakis and Keromytis, 2011] expanded the code emulation approach by adopting it to detect ROP attacks in any type of program entry. This protection uses the “Nemu” code emulation tool [Polychronakis et al., 2006] to check if the program entries correspond to valid gadget addresses that can be linked in a ROP attack. ROPscan establishes a threshold size for a gadget and for the total number of instructions executed to limit the number of instructions to be simulated.

As the code emulation tools already impose a high computational cost [Bellard, 2005], it is expected that protections that use this approach lead to a significant overhead, regardless of the strategies adopted. In the specific case of ROPGuard, the reported overhead corresponds to only 0.48%. However, in addition to running extremely simple simulations, this protection is applied only to Windows APIs that are considered critical, which limits the impact of code emulation. The authors of ROPscan do not report the overhead resulting from their solution, because their tests use a shellcode detection tool to analyze the inputs, instead of interfering with the execution of the process. Therefore, the metric used to measure the quality of the solution was the average throughput, which reached 120 Mb/s in the implementation of ROPscan.

This is a metric commonly used in the evaluation of shellcode detection systems, since these solutions are normally used in the analysis of network traffic.

Therefore, although it is apparently not useful for real-time protection of the memory space of applications, the code emulation approach seems to be useful in other situations. A viable scenario for applying this approach may be the verification of data files downloaded by users. Code emulation could be used to scan data before actually submitting it to the target application, avoiding infections from malicious data files.

3.1.4 Discussion

Table 3.1 provides a summarized comparison between the ROP attack protections used to illustrate the most common strategies and approaches. The overhead values indicated correspond to the rates presented by the works themselves. An accurate comparison of the solutions would require that all protections be implemented in the same environment and subjected to tests with the same set of applications. Despite this diversity, it is possible to ratify more general relations established between the implementation approaches used by the protections. We were able to verify, for example, that the works that employ the Dynamic Binary Instrumentation approach cause greater overheads than the works that use the other approaches. In contrast, solutions that employ Static Binary Instrumentation and Hardware Adaptations tend to cause less overhead.

An interesting finding is that few solutions are compatibility with any type of application. Most conflict with some type of exception, either by mistakenly identifying authentic software as malicious, or by corrupting code that performs some type of manipulation on themselves, such as decryption or signature verification. It is important to emphasize that the column of false positives in Table 3.1 does not illustrate the cases that represent possible security flaws left by the proposed protections. In other words, attackers may eventually overcome some of the listed defenses, but those possibilities are not counted among the exceptions.

The analysis of the evaluated protections allows the formulation of general statements regarding the future of protections against ROP attacks. All solutions based on checks carried out at specific points in the execution of an application, such as the occasions when system calls are triggered, can eventually be overcome by strategies that force the execution of codes without effect to evade future checks. As the scans are limited to the last N instructions executed (since a complete scan is not viable due to performance and storage space), if the code without effect executes N or more instructions, the history of the malicious code will be overwritten and the attack will not

Table 3.1. Comparison of common ROP defenses.

Protection Name	Implementation Approaches	Protection Strategies	Blocked Types	Overhead (%)	# False Positives	Practical Feasibility
G-Free	1	8	R, J and C	3,0	0	Yes
Return-less Kernels	1	8	R	9,7	0	Yes
CFL	1 and 2	4	R, J and C	5,6	3	Yes
Security mitigations for ROP	1 and 2	1, 3, 7 and 8	R	*	4	No
/ROP	1 and 4	3 and 4	R	3,3	2	No
Hardware Virtualization	1 and 5	2	R	10,0	4	Yes
Poster / MoCFI	2	2 and 4	**	1,2	7	No
Marlin	2	1	R, J and C	*	0	Yes
STIR	2	1	R, J and C	1,6	3	Yes
In-place randomization	2	1	R, J and C	0	3	Yes
ILR	2 and 3	1	R, J and C	16,0	6	Yes
ROP Monitor	2 and 3	2 and 8	R, J and C	350,0	6	No
ROPDetector	2 and 3	5	R, J and C	350,0	0	Yes
DynIMA	2 and 3	6	R	*	4	No
BR	2 and 4	2 and 5	R, J and C	2,1	6	No
kBouncer	2 and 4	3	R	4,0	3	Yes
CFIMon	2 and 4	3, 4 and 5	R, J and C	6,1	6	Yes
Eunomia	2 and 4	6	R	4,7	1	Yes
ROPGuard	2 and 6	3,7 and 8	R	0,5	3	Yes
Code Shredding	3	1	R, J and C	365,0	3	Yes
TRUSS	3	2	R	53,4	4	Yes
ROPdefender	3	2	R	217,0	4	Yes
ROPgrind	3	2 and 5	**	900,0	4	Yes
DROP	3	6	R	530,0	1	Yes
CFLC	3	8	R, J and C	374,0	0	Yes
HyperCrop	5	6	R	15,0	1	Yes
ROPscan	6	8	R, J and C	*	1	Yes

Approaches: 1=Compilation; 2=Static Binary Instrumentation; 3=Dynamic Binary Instrumentation; 4=Hardware Adaptation; 5=Virtualization; 6=Code Emulation.

Strategies: 1=Randomization; 2=Constructing Shadow Stacks; 3=Checking instructions prior to return addresses; 4=Checking targets of branches; 5=Checking the entry points of functions; 6=Controlling the frequency of return instructions; 7=Checking the top of the stack pointer; 8=Other.

Blocked attacks: R=chaining via RET; J=chaining via JMP; C=chaining via CALL.

* The overhead is not reported in the publication.

** In the ARM architecture, the branch instructions are different. This protection addresses all branches.

be detected. In addition, solutions that verify the occurrence of the attack at a single point in the process execution period, as in API calls, are unable to verify areas of code with write permission because the data to be checked may already have been changed legitimately since the execution of the branches to be verified. These executable and writable memory areas are used in code compiled on-demand (JIT compiled), for example.

For these reasons, we believe that effective solutions against ROP should check the status of the execution flow throughout the execution of the process. This continuous checking can be achieved with dynamic binary instrumentation tools or by equipping

the hardware with sufficient structures and logic to block attacks without overburdening application performance. Dynamic binary instrumentation solutions can be very useful for the processes of prototyping protections. In addition, they appear as an important tool in blocking new types of attacks and zero-day attacks (those for which security updates are not yet available) in production environments. In such cases, it may be advantageous to cope with the overhead imposed by a dynamic binary instrumentation tool to ensure system security until a security update is available.

Since there is still no effective protection against ROP attacks implemented in production systems, this topic continues to attract the attention of researchers around the world. We believe that a promising solution should combine effectiveness in blocking all variants of ROP attacks, reduced execution time overhead and low rate of exceptions. This and the other observations presented in this Section guided the development of the solutions that we present in the Chapters 4 and 5 of this thesis.

3.2 Control-Flow Integrity

The expression Control-Flow Integrity was created by Abadi et al. [2005] to name a protection that aimed to ensure the coincidence of the execution flow of a program with its Control Flow Graph (CFG). Subsequently, several studies broadened the scope of this term by classifying its solutions as CFI mechanisms, although these protections are not always linked to the CFG of protected programs. Among the vast amount of protections against code reuse attacks that call themselves CFI, some stand out for their effectiveness in making it substantially harder for attackers to gain arbitrary control of vulnerable programs [Burow et al., 2017]. In this Section, we focus on these defenses. They monitor backward and forward branches. Backward branches, performed by `RET` instructions, are typical in ROP attacks. Forward branches characterize COP or JOP attacks, since they are executed by `CALL` or `JMP` instructions.

CFI protections based on the use or emulation of a shadow stack, such as the original CFI [Abadi et al., 2005], MoCFI [Davi et al., 2012], HAFIX [Davi et al., 2015], Lockdown [Payer et al., 2015], PathArmor [van der Veen et al., 2015] and Intel CET [Intel, 2019], are known to be effective in protecting against backward edge attacks [Burow et al., 2017]. The main challenge for the widespread adoption of this protection strategy lies in the overhead it imposes. The first implementation of CFI incurred an overhead of 21% [Abadi et al., 2005]. Said overhead has been since reduced, but it remains noticeable. As pointed by Burow et al. [2017], effective CFI requires the validation of every type of indirect branch, including return instructions. Such

verification has a cost.

A quick survey over recent implementations of CFI reveals an overhead between 2 and 18%. This range includes SafeDispatch [Jang et al., 2014], T-VIP [Gawlik and Holz, 2014], VTV/IFCC [Tice et al., 2014], vfGuard [Prakash et al., 2015], and VTint [Zhang et al., 2015]. These approaches implement a *forward-only* version of CFI. More general techniques, that verify targets of return instructions, tend to be even more expensive because these instructions are more frequent than indirect CALLs and JMPs. Techniques that guard RET instructions, such as the original CFI [Abadi et al., 2005], MoCFI [Davi et al., 2012], Lockdown [Payer et al., 2015], PathArmor [van der Veen et al., 2015] and PittyPat [Ding et al., 2017] present overheads ranging from 7 to 21%. These values are considered prohibitive.

3.3 Defenses Based on Indirect Branch Frequency

There are several previous work that propose ways to detect ROP attacks by inspecting the frequency of indirect branches in the instruction stream [Chen et al., 2009; Davi et al., 2009; Han et al., 2013; Jiang et al., 2011; Pappas et al., 2013b; Yuan et al., 2011; Cheng et al., 2014]. These researchers propose to monitor the sequence of instructions that a program produces during its execution, flagging as exploits instruction streams that show a high frequency of RET operations. For instance, Chen et al. [2009] fire warnings if they observe sequences of three return instructions separated from each other by five or less instructions. Pappas et al. [2013b] and Cheng et al. [2014] use the Last Branch Record (LBR) registers to detect chains of gadgets. Carlini and Wagner [2014] and Göktas et al. [2014] propose ways to bypass the LBR-based defense. However, we believe that our approach based on a sliding window, described in Chapter 4, would be harder to circumvent. For attackers to succeed against a sliding window, they must insert innocuous instructions (no-ops) between gadgets. Just adding them at the end, or after 10 gadgets, like Carlini and Wagner [2014] does against the defenses proposed by Pappas et al. [2013b] and Cheng et al. [2014] is not enough. But even more important, our sliding window approach allows setting a specific threshold of frequency of indirect branches for each application. This greatly increases the protection power offered by strategies based on the control of the frequency of indirect branches.

In a different direction, Yuan et al. [2011] have proposed to count the frequency of instructions that are executed close in time to RET operations. Instructions are assigned an amount of “suspicion” according to their proximity of a return instruction. The closer, the more suspicious it is. Once a given suspicion threshold is achieved,

a warning is issued. Finally, there are approaches to detect ROP-based attacks that count the density of `RET` instructions in the stack [Han et al., 2013; Jiang et al., 2011]. Code that invokes gadgets usually only allocates space on the stack for return addresses. That is to say: the activation record of gadgets do not contain space for local variables, parameters, links to nesting functions, and other data that is usually present in the activation record of legitimate functions. Thus, in face of a ROP attack, the stack is likely to contain many words holding program addresses.

The work that we present in this thesis is not a competitor of all these techniques. Instead, it complements them. They all rely on fixed thresholds to detect ROP exploits. By giving them the means to estimate an specific peak density of indirect branches for each application, we help them to be more precise. Additionally, by showing that a sliding window provides an effective way to recognize ROP attacks, we open new avenues for the design and implementation of protection mechanisms that are simpler and more reliable than the current state-of-the-art detection approaches.

3.4 Multilayer Protection

The idea of combining different defense mechanisms in layers, so that low-overhead validations are used to gradually filter safe control-flows, is an original contribution of this thesis. Nevertheless, except for the verification of executable targets of `CALLs`, discussed in Section 5.1.2.1, our defense layers reuse techniques presented in previous work. Our rationale when choosing which technique to apply on each layer was based on two tenets: (i) upper layers – which are applied first – should have lower computational overhead; and (ii) work done in upper layers should not be redone in lower layers.

Our uppermost layer, which certifies targets of indirect branches via the branch predictor, has already been described as early as 2007, by Shi and Lee [2007], right after Shacham [2007] introduced the concept of Return-Oriented Programming. More recent works have revisited this concept in an attempt to block ROP attacks [Lee and Lee, 2018; Das et al., 2018; Ahn and Lee, 2015]. Other proposals use the number of mispredictions of indirect branches as an attack indication [Tang et al., 2014; Wicher-ski, 2016]. We adopt the inverse perspective: correct predictions of indirect branches indicate authentic executions. Execution flows that we cannot validate using branch prediction flow into lower layers of our system. The previous literature doesn’t talk about this multilayer approach.

The idea of checking if the address before the target of a `RET` is a `CALL` instruction has been independently described by previous works. Carlini et al. [2015] provide a

comprehensive overview of the related literature. The fact that Carlini et al. [2015] have been able to circumvent this kind of defense led us to conceive the *enhanced Call-preceded Constraint* that we describe in Section 5.1.2.1. We designed this test to be easily implemented, once the original CALL-validation mechanism is in place.

The technique that we use to filter out false positives, discussed in Section 5.1.3, was first suggested by Zhang and Sekar [2013]. We adopted the same strategy defined in that work because it is simple and computationally cheap. However, were it used without our upper layers, then it would be too costly to be practical. Finally, sandboxing, one of the alternatives described in Section 5.1 to protect an application, when all the upper layers fail to certify an indirect branch, is well-known in the systems community. We defend its usage, in spite of its heavy runtime overhead, for two reasons. First, the experiments in Section 5.2.2 indicate that its forth protection layer is unlikely to ever be fired during the normal execution of a program. Second, sandboxing, once initiated, has been shown to be effective against virtually any kind of exploit Egele et al. [2012].

How our overhead compares to previous works’. The ultimate goal of this thesis is to provide protection against ROP based attacks at a low computational cost. The literature contains much previous work with similar purpose. For a comparison, we mention the overhead of eight recent systems that enforce some sort of Control Flow Integrity policy Davi et al. [2012, 2015]; Ding et al. [2017]; Kayaalp et al. [2012, 2013]; Li et al. [2019]; Payer et al. [2014]; van der Veen et al. [2015]. We report overheads instead of reproducing them ourselves because these tools have existed as research artifacts only and they aren’t easily available today. Some of these systems were only tested via software-based prototypes, although they are meant to be implemented in hardware; hence, some numbers that we shall mention are estimates from their authors. SCRAP’s authors report the lowest overhead [Kayaalp et al., 2013], between 1% and 2%; however, they use a strategy based on controlling the frequency of small gadgets, which has already been overcome in subsequent work [Carlini and Wagner, 2014; Göktaş et al., 2014; Tymburibá et al., 2016]. Kayaalp et al. [2012] report a 2% overhead for BR –same value reported for HAFIX [Davi et al., 2015] and for Zipper Stack [Li et al., 2019]. The same group had, before, proposed MoCFI, with an overhead of 7% [Davi et al., 2012]. van der Veen et al. [2015] report an overhead of 8.4% for PathArmor. PittyPat’s original presentation reports an overhead of 12.73% [Ding et al., 2017]. Finally, Lockdown’s authors have measured a runtime slowdown of 19% [Payer et al., 2014]. Most of these techniques require some support from the hardware. For instance, HAFIX requires changes in the target architecture’s instruction set. Software-only

approaches, such as PittyPat, tend to present higher overhead, in addition of requiring recompilation of the code that must be protected.

JOP and COP protections. The practical use of JOP and COP protections suffers from the same drawback that limits the use of defenses against ROP attacks: runtime cost. Most solutions impose a high overhead, ranging from 4 to 200% [Pappas et al., 2013b; Tice et al., 2014; Mohan et al., 2015; Wang and Jiang, 2010; Niu and Tan, 2014a; Bletsch et al., 2011; van der Veen et al., 2015; Xia et al., 2012; Niu and Tan, 2014b; Abadi et al., 2005; Prakash et al., 2015; Payer et al., 2015; Pewny and Holz, 2013; Criswell et al., 2014]. There are forward edge protections that promise a lower impact on application performance [Jang et al., 2014; Gawlik and Holz, 2014; Niu and Tan, 2015; Zhang et al., 2013]. Some even claim to reach overhead values of around 1% [Bounov et al., 2016; Zhang et al., 2015]. However, these approaches relax the constraints on CALLs and JMPs targets, which made them known by the expression coarse-grained CFI, in contrast to fine-grained CFI. Coarse-grained CFI is known to be weaker, as it leaves a larger number of gadgets available for constructing an exploit. This number is a traditional indicator of the quality of protections against code reuse attacks [Zhang and Sekar, 2013; Niu and Tan, 2014a]. The lower the number of branch targets and gadgets available, the higher the level of protection.

Efficient fine-grained CFI via multilayers. π CFI [Niu and Tan, 2015] is the work that comes closest to restricting control flow deviations only to options originally intended for the application. This protection dynamically activates edges previously selected during a static analysis. Despite its higher accuracy, π CFI results in a 3.2% overhead due to the operations it performs at runtime. Furthermore, depending on the number of labels cataloged during the static analysis of the program’s control flow graph, the performance of this solution can be severely penalized. RockJIT [Niu and Tan, 2014b] and MCFI [Niu and Tan, 2014a] also offer an above average degree of protection. These works use values on data types to derive context-sensitive information that allows more precise constraints to be set for monitored branches. However, the 14.6% and 5% overhead values reported by these works are considered prohibitive for use in most real scenarios. PathArmor [van der Veen et al., 2015] also raises the degree of protection compared to coarse-grained CFIs. However, PathArmor carries an overhead of 8.4% and does not protect libraries loaded by the application. Our multilayer system reverses the logic traditionally used to certify the control flow of applications by first filtering out authentic execution flows. Because this selection of untainted flows is extremely accurate and efficient, our multilayer model allows for the adoption of the

most precise fine-grained protections in the deepest layers of the system. Since the number of cases that pass through these deeper layers is tiny, the impact of the higher overhead imposed by the most secure CFI protections is not felt.

3.5 Chapter Summary

This chapter presented a review of works that deal with code reuse attacks, emphasizing the bibliography related to the Return-Oriented Programming technique and its variants. To assess the protections proposed in the literature, we defined metrics that express the main quality factors of defenses against ROP-based attacks. Moreover, to facilitate the understanding of protection schemes, we classified similar work according to the most common defense strategies and implementation approaches. Finally, we discussed CFI and the works closely related to the two protection approaches advocated in this thesis, announced in Section 1.3. In the next Chapters, we will present the details and results of experiments with these two approaches.

Chapter 4

Inferring The Peak Density Of Indirect Branches

A ROP exploit is likely to produce a program behavior that is different than its normal execution [Pfaff et al., 2015]. One of the reasons behind this discrepancy is due to the fact that this type of malicious code consists of small sequences of instructions ending with an indirect branch, e.g., a return (`RET`), an indirect call (`CALL`) or an indirect jump (`JMP`). The very nature of this attack technique forces these gadgets to be small. Thus, it is possible to identify ROP exploits by counting the number of indirect branches observed in a sequence of instructions fetched during the execution of a program. This approach is fairly studied in the literature: by tracking the density of indirect branches present in the instruction stream, different research groups have been able to provide ways to detect ROP-based attacks with various degrees of accuracy [Chen et al., 2009; Davi et al., 2009; Han et al., 2013; Jiang et al., 2011; Pappas et al., 2013b; Yuan et al., 2011; Cheng et al., 2014].

Nevertheless, this detection approach is not foolproof, as independently demonstrated by Carlini and Wagner [2014] and Göktaş et al. [2014]. These researchers have shown that it is possible to interpose long gadgets between small ones, thus masquerading the occurrence of the attack. In this thesis we look deeper into techniques such as Carlini’s and Goktas’ and use compiler-related techniques to design protection mechanisms that make it more difficult for them to succeed. With this goal, we bring in two contributions, which we enumerate as follows:

- First, in Section 4.1.1 we show that the most general frequency based mechanism – a sliding window named RIP-ROP [Ferreira et al., 2014] – can be circumvented, depending on compilation choices while generating binaries. To this end, we rely

on some default characteristics of the Gnu C Compiler (`gcc`). This evasion technique generalizes the strategies adopted by Carlini and Wagner [2014] and Göktaş et al. [2014], which we have mentioned before.

- Finally, in Section 4.2 we describe a static analysis that estimates the peak density of indirect branches for a given program. By using this specific threshold, instead of a universal threshold, we show that it is possible to design detection mechanisms that raise the bar for exploits such as those that we introduce in Section 4.1.1.

As we show in Section 4.3, our new static analysis determines conservative, yet precise, approximations of peak density of indirect branches. To support this claim, we have analyzed instruction traces produced by the programs in SPEC CPU2006. Our results are tighter than a universal threshold. Hence, our specific detection thresholds make it harder to produce an undetectable ROP attack with the public exploits that we have been able to reproduce.

4.1 Universal ROP-Detection Thresholds

We have shown in previous work that it is possible to use a window of fixed length of program instructions to identify current Return-Oriented Programming exploits [Ferreira et al., 2014]. This approach generalizes previous frequency-based ROP detection techniques; hence, it is harder to evade than the similar techniques. Being able to use a fixed-size window to detect ROP attacks is an essential requirement of the algorithm that we introduce in Section 4.2. Were we not dealing with a fixed-size window, then estimating peak densities of indirect branches would leave us with an undecidable problem.

The goal of this section is two fold. First, we show that even a window-based detection mechanism can be circumvented by a determined attacker, due to the way that compilers generate code. Second, we demonstrate how specific detection thresholds can improve the security level of frequency-based protections. Before we dive into these two goals, we provide an intuition on why a ROP-exploited program tends to show a high density of indirect branches. To this end, Figure 4.1 shows the ROP attack used in Saif El-Sherei’s tutorial [El-Sherei, 2013].

There are several ways to trigger a ROP attack. One of the most common techniques is to exploit a stack buffer overflow. This vulnerability lets the attacker overwrite the return address of a function with a sequence of new return addresses that

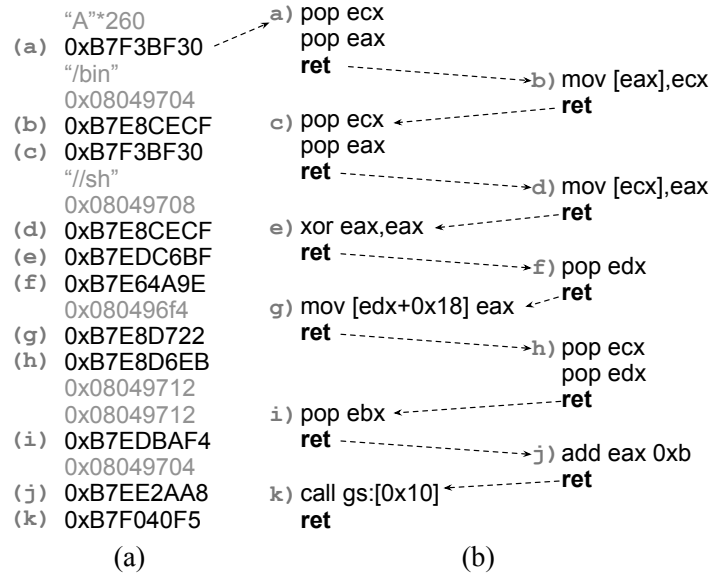


Figure 4.1. (a) Input string used in a stack buffer overflow attack that triggers a ROP exploit. Data used in the exploit is written in light grey, and return addresses are written in black. Letters in front of addresses indicate the gadget that will be invoked. (b) Gadgets that are chained during this attack.

invoke gadgets. The attack in Figure 4.1 uses a particular string to open a shell section (/bin/sh) with the privileges of the exploited program. The exact contents of this string are immaterial to our explanation, except that it contains several addresses within the executable memory segment of the program under attack. Each one of these addresses point to a sequence of instructions that ends in an indirect branch (return operations in the case of Figure 4.1). These sequences, e.g., gadgets, perform some simple task, and then invoke a return operation, which will move control to a new gadget, determined by the next address on the stack. The gadgets used in this attack are seen in Figure 4.1 (b). They have between two and three instructions, including the return operation. Larger gadgets could break the progress of the attack by overwriting some memory slot or register used during the exploitation. This small size is not a particularity of this example, but a characteristic of all documented ROP exploits that we have found in Exploit Database [Offensive Security, 2020], the biggest public repository of malware we know.

Table 4.1 makes it explicit the “fingerprint” of these exploits: all of them show execution traces with a very high density of indirect branches. This is a consequence of having very small gadgets: only 2.46 instructions, on average. The smaller the gadgets, the higher the frequency of indirect branches.

Table 4.1. Proportion of indirect branch instructions in cataloged exploits. Each gadget has, on average, 2.46 instructions. This results in a high frequency of indirect branches.

Application	Instructions in Exploit	Indirect Branches	Ratio (Average Gadget Size)
Windows			
Wireshark	49	24	2.04
DVD X Player	62	28	2.21
Zinf Audio Player	95	38	2.50
D. R. Audio Converter	69	36	1.92
Firefox - use aft free	51	21	2.43
Firefox - integer ovf	36	18	2.00
PHP	53	21	2.52
AoA Audio Extractor	212	81	2.62
ASX to MP3 Conv	150	59	2.54
Free CD to MP3 Conv	47	19	2.47
Linux			
ProFTPD Debian	66	24	2.75
ProFTPD Ubuntu	45	19	2.37
PHP	81	27	3.00
Wireshark	69	30	2.30
NetSupport	45	14	3.21

4.1.1 Evading Universal Thresholds

The use of a universal threshold to separate legitimate and illegal executions raises the following question: is it possible to find gadgets that are large enough to lower the density of indirect branches to a threshold below the standard value? We have tried to build such sequences on top of ten publicly available exploits in the Exploit Database [Offensive Security, 2020]. We chose these 10 exploits because they were all the attacks that we could reproduce successfully; hence, confirming the results described in Exploit Database. Our efforts show that it is possible, although difficult, to find such gadgets.

We define a *no-op* gadget as a sequence of instructions ending with an indirect branch, which neither changes the state of the registers or the stack used in the exploit, nor cause a violation (memory access, privileged instruction, etc). In other words, the instructions in a no-op gadget do not cause side effects that could invalidate an exploit. Our idea is to interpose no-op gadgets between the valid sequences, in order to lower the density of indirect branches observed in an exploit.

We have used the open-source tool Mona [Corelan Team, 2011] to find no-op

gadgets. Mona gave us, on average, 58,000 potential gadgets per each one of the vulnerable applications. Trying one by one all of these gadgets would be too time consuming; hence, we have defined four criteria to select good candidates. We choose gadgets that:

- Have more than five instructions.
- Have no instructions that change the stack (`push`, `pop`, `add esp`, etc).
- Have no instructions only allowed in privileged mode (`cli`, `ltr`, `smsw`, and 21 more).
- Do not write on registers used in the exploit.

Once we had a list of candidates, we sorted them according to two criteria:

1. Increasing order on the number of indirect memory accesses.
2. Decreasing order on the total number of instructions.

Criterion 1 is useful, because a large part of an application's address space is not accessible.

To check which gadgets work in practice, we decided to test all the candidates found in **AoA Audio Extractor** [Samir and Mr_me, 2010] and in **CD to MP3 Converter** [G0M3S, 2011]. We chose these two applications because they have the lowest number of candidates (21 and 52, respectively). We did not test more applications because this process is very time consuming. Checking if a sequence of instructions can be interposed between the other gadgets of an original exploit is a manual process. To this end, we must adjust the return addresses used in the attack, and must check if the final effect of the exploit is still observable. We estimate that testing these 21 + 52 no-op candidates took us more than 70 man-hours of work.

In total, we found (3 + 6) functional no-op gadgets out of a universe of 21 + 52 candidates having between seven and ten instructions. The key conclusion of this experiment is that: *it is feasible to circumvent a window-based detection scheme using no-op gadgets*. All the no-op gadgets that worked belong into one of two categories: (i) static initialization sequences, and (ii) alignment blocks.

In the former category we have long sequences of stores into fixed addresses, which are used to initialize static memory in C. In the latter we have no-op instructions that correspond to code that compilers insert to force an alignment of blocks of instructions. Most processors fetch instructions in aligned 16-byte or 32-byte blocks. Hence, it can

be advantageous to align branch targets by 16 in order to minimize the number of 16-byte boundaries in the code. For instance, in `gcc` this alignment is controlled by the `-falign-labels` and `-falign-jumps` arguments, which are default in optimization levels `-O2` and `-O3`. The three valid no-op gadgets in `AoA Audio Extractor` were in category (ii). `CD to MP3 converter` had three no-ops in category (i), plus three more in category (ii).

We believe that it is possible to modify the compiler to remove these gadgets. Elements in the first group, i.e., static initialization, can be removed by the simple expedient of interposing instructions that write to registers with stores into static memory. This process does not change the semantics of the program, because no register is in use when static data is being initialized. To eliminate the second category of no-op gadgets, it is possible to replace the padding code by no-ops of up to nine bytes, which would reduce the number of instructions in the innocuous gadgets considerably [Intel, 2006]. All the other no-op candidates failed to lead us to valid exploits, either because they write into general purpose registers or in memory addresses necessary for the exploit. The explanation for this failure is straightforward: it is hard to find no-op gadgets that fit anywhere in the ROP chain. The x86 architecture has very few registers, and usually gadgets with more than seven instructions break the progress of the attack by writing in some of these registers.

4.1.2 Using Specific Thresholds To Improve Protections

One way to improve the level of protection offered by mechanisms based on controlling the frequency of indirect branch instructions is to establish a specific threshold for each application to be protected. Because they are smaller than the universal threshold, specific thresholds force an attacker to use longer gadgets. However, as previously discussed, larger gadgets are more likely to corrupt the contents of the memory or any register used during the attack. Alternative attacks using no-op gadgets also become more difficult, since longer no-op gadgets are scarcer, as we show in Section 4.3. Although advantageous, specific thresholds are not yet adopted due to the lack of algorithms to estimate them. In Section 4.2 we describe a static analysis that solves this challenge.

Although unexplored so far, the possibility of using specific thresholds to enhance ROP detection mechanisms has been suggested by several researchers. As an example, the authors of schemes based on the control of indirect branches density stress the possibility of improvement through the use of specific thresholds [Chen et al., 2009; Davi et al., 2009; Jiang et al., 2011; Pappas et al., 2013b; Cheng et al., 2014]. For

instance, Jiang et al. [2011] claim that “the threshold must be carefully chosen in order to balance the false positive rate and the false negative rate” and Chen et al. [2009] point out that “the thresholds can be chosen by the user”. This opinion is also shared by authors who developed methods to overcome protections. Göktas et al. [2014] devote an entire subsection to discuss countermeasures based on per-application parameters. According to them, albeit challenging, one can explore the peculiarities of each application to establish specific values able to consistently detect attacks. Indeed, analysis of publicly available ROP exploits confirms the theory that specific thresholds can improve indirect branch frequency based protections. In Section 4.3, we will show that for the vulnerable applications analyzed, the use of specific thresholds reduces by 75% the availability of no-op gadgets.

4.2 Static Inference Of Specific Thresholds

This section discusses the problem of inferring the peak density of indirect branches in a window of K instructions that can be observed during the execution of a program. Definition 1 states the problem of *Inference of Peak Density of Indirect Branches*, henceforth referred as IPD.

Definition 1 (IPD) *Given a program P , and two integers: K and R , is there an execution trace of P with no more than K instructions that contains R indirect branches?*

To solve IPD, we could, in principle, interpret P , logging the maximum number of indirect branches observed at any point during this interpretation. However, P may not terminate, and we would never be sure to have found or not its peak density. In other words, IPD is undecidable in the general case, as a consequence of Rice’s Theorem [Rice, 1953]. To see why, notice that we could ask if the program P ever reaches a return instruction at any point of its execution. “Reaching a return instruction” qualifies as an *interesting property* in Rice’s sense. Given that the IPD is undecidable, we shall solve it via a conservative, partially context-sensitive, flow-sensitive static analysis. We describe this static analysis in the rest of this Section.

4.2.1 The Supporting Data-Structure

For any instruction I in the program control flow graph, our algorithm traverses every possible path of K instructions starting from I , counting up every indirect branch visited in this process. The worst case for the performance of this approach happens when each instruction may branch to each other, a situation that would give us searches

as costly as $O(N \times N^K) = O(N^{(K+1)})$ starting at each of the N instructions that constitute the program. Even though we shall not reduce this asymptotic complexity, we will try to alleviate its constant factor by abstracting away every element of the Control Flow Graph (CFG) that has no influence in the solution of IPD. Therefore, we will not work on the program's CFG, but in a data-structure henceforth called δ -CFG.

The δ -CFG has three kinds of nodes: BBL, RET and FUN. Each one of these nodes represent a different kind of *basic block*, e.g., a sequence of instructions with only one entry-point, and only one exit-point. The difference between these blocks of instructions is the operation that ends it. Nodes BBL represent basic blocks that end with conditional or unconditional JMP instructions. For simplicity, we shall assume that each such block has two targets, even though unconditional direct branches only have one target. Nodes RET represent blocks that end with return instructions. Each RET node that is part of a function f is associated with a list of instructions that succeed function calls to f . Nodes FUN represent blocks that end with function calls. These nodes are associated with an entry-point, and a return-point. The entry point of a node FUN that invokes a function g is the first basic block of g . The return point of a node FUN at program address ℓ is the instruction at label $\ell + 1$. Therefore, these nodes let us represent the entire control flow graph of a program, considering direct and indirect branch instructions. Each of these three kinds of nodes is associated with an integer n , which denotes the number of instructions present in that basic block.

Figure 4.2 (a) shows an example of program written in C, which we shall use to explain how our algorithm works. A simplified assembly version of this program is given in Figure 4.2 (b). Figure 4.2 (c) shows the δ -CFG of the example program. In this section, we shall use this – rather artificial – program, to illustrate our technique to estimate peak density of indirect branches. The δ -CFG eliminates most of the instructions present on the original CFG, but it keeps the program flow. Thus, we can use this concise structure to find paths within the program code. This search is performed by a brute-force algorithm that we describe in the the rest of this section.

4.2.2 Algorithm To Estimate The Peak Density Of Indirect Branches

Given a program P , the function deduce in Figure 4.3 estimates P 's peak density of indirect branch instructions by visiting every path of size K starting at any node of P 's δ -CFG. This implementation is written in SML/NJ's syntax, and it has no omissions: Figure 4.3 is a running prototype of our algorithm. Pattern `top::stack` denotes a list with head `top` and tail `stack`. We use `[]` to represent empty lists. The construct `fn x =>`

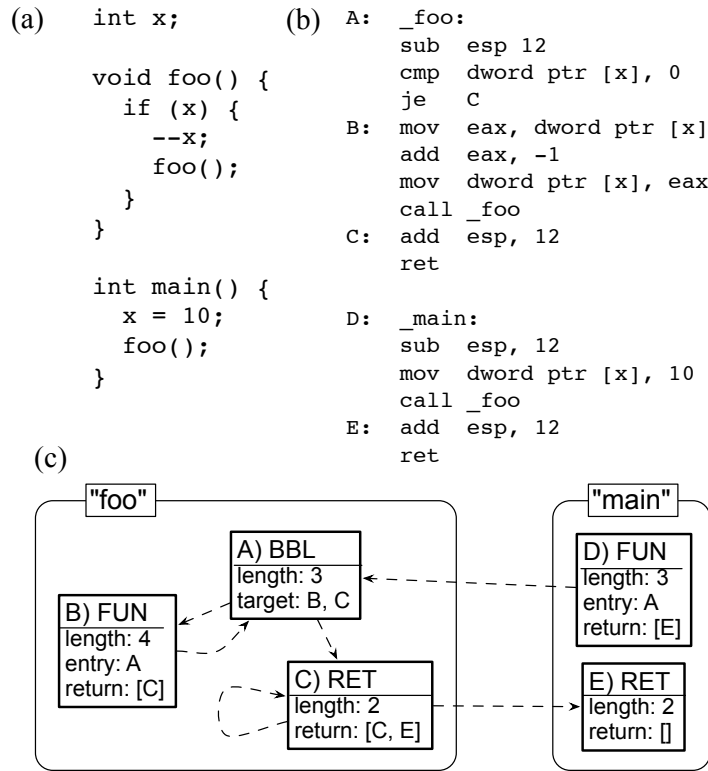


Figure 4.2. (a) Example program written in C. (b) Simplified x86 assembly of example program. (c) δ -CFG produced for the example program.

e, at line 43, denotes an anonymous function that has formal parameter x and body e . The datatype in lines 1-3 represents the δ -CFG. Each one of the three possible types of nodes is associated with an integer n , which represents the number of instructions in that node. Nodes of type BBL are also associated with two successors, *left* and *right*. Nodes FUN are also associated with an *entry point* and a *return point*. Finally, nodes RET are associated with a list of return points.

In line 43 of Figure 4.3, function `deduce` performs an exhaustive search bounded by the window size K starting from any node in the program. The function `map` applies `explore` onto every element of the list P , which represents the input program. Function `explore` receives a *trip budget* T , a *return stack* and a node of the δ -CFG. While `explore`'s trip budget is larger than 0, it will visit more nodes in the δ -CFG. Hence, the trip budget represents the part of the window that we must still fill up with instructions while searching for a peak density. If `explore` visits a FUN node, then it will store this node's return point onto the stack, as seen in line 40 of Figure 4.3. Upon reaching an indirect branch, e.g., a RET node, `explore` resumes its search at the node on the top of the stack, in case its trip budget is still positive, as seen in line 23-26 of

```

1  datatype  $\delta$ -CFG = RET of int  $\times$  list  $\delta$ -CFG
2                | BBL of int  $\times$   $\delta$ -CFG  $\times$   $\delta$ -CFG
3                | FUN of int  $\times$   $\delta$ -CFG  $\times$   $\delta$ -CFG
4
5  fun max a b = if a > b then a else b
6
7  fun explore T [] (RET (n, return_addresses)) =
8      if T  $\geq$  n
9      then
10         let
11             fun max_of_all_return_paths [] = 0
12             | max_of_all_return_paths (next_addr::addresses) =
13                 let
14                     val max_so_far = max_of_all_return_paths addresses
15                     val dens_of_next = explore (T - n) [] next_addr
16                 in
17                     max max_so_far dens_of_next
18                 end
19         in
20             1 + max_of_all_return_paths return_addresses
21         end
22     else 0
23 | explore T (top::stack) (RET n) =
24     if T  $\geq$  n
25     then 1 + explore (T - n) stack top
26     else 0
27 | explore T stack (BBL (n, left, right)) =
28     if T  $\geq$  n
29     then
30         let
31             val bestLeft = explore (T - n) stack left
32             val bestRight = explore (T - n) stack right
33             fun max a b = if a > b then a else b
34         in
35             max bestRight bestLeft
36         end
37     else 0
38 | explore T stack (FUN (n, entry, next)) =
39     if T  $\geq$  n
40     then explore (T - n) (next::stack) entry
41     else 0
42
43 fun deduce P K = map (fn b => explore K [] b) P

```

Figure 4.3. Brute-force inference of indirect branches using window of size K .

Figure 4.3. Otherwise, `explore` continues the search at every return address that may succeed a call to the function that is being traversed, as seen in lines 7-22.

We shall use Figure 4.4 to provide an intuitive description of our algorithm. The figure shows another view of the δ -CFG seen in Figure 4.2 (c). The weights on the edges represent number of program instructions. We say that the *weight of a path* is the sum of all the weights on the edges that constitute this path. The IPD problem asks for a path weight K containing R indirect branches. Our algorithm starts a graph traversal at each node of the δ -CFG, stopping each search upon finding a path of weight K . During this search, every edge can be traversed; the only special cases are the RET

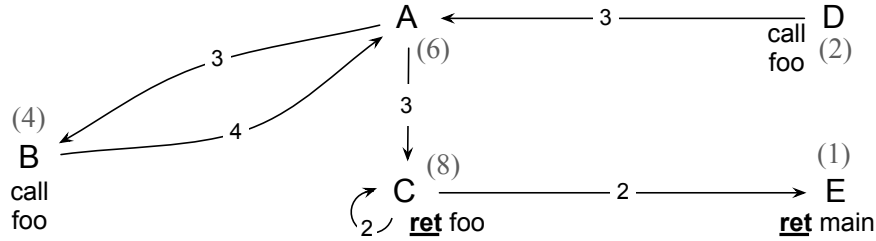


Figure 4.4. Intuitive view of the algorithm from Figure 4.3, when applied on the δ -CFG of Figure 4.2 (c). Numbers in parentheses above block letters indicate the highest frequency of RETs that our algorithm obtains starting the search from that block, considering a window of 16 instructions.

nodes. Upon visiting a FUN node that calls a function f , we store f and its return address onto the stack. When leaving a RET node that returns from a function f , we check if f is on the top of the stack. If that is the case, then we can only continue the search at the return point of the last call to f that we have seen, which is naturally onto the stack. In this example, the peak density that our algorithm finds for a window of 16 instructions is 8, which corresponds to eight visits to the edge $C \rightarrow C$ in Figure 4.4. This scenario happens when `foo` (Fig. 4.2) returns from eight or more recursive calls.

4.2.2.1 Sources Of Imprecision

Our algorithm works on machine code; however, inferring the exact control flow graph of a binary program is not always possible due to indirect branches [Theiling, 2000]. We have implemented our algorithm on the x86 back-end available in the LLVM compiler. LLVM uses an intermediate representation that makes explicit the control flow of programs. We use this information, still available during the translation from LLVM’s representation to x86’s binaries, to build the δ -CFG. One source of imprecision left is due to functions called through pointers. We assume that any function whose address has been taken can be the target of an indirect call. Other heuristics, such as Bacon’s Rapid Type Analysis [Bacon and Sweeney, 1996] could be used to refine our δ -CFG; however, we have not experimented with such approach yet.

Our algorithm incorporates some context sensitivity by propagating a stack of return points during the traversal of the δ -CFG. Whenever this traversal meets a FUN node, we push the address immediately after this node onto the stack. Upon finding a RET node, we continue our traversal at the address that is on the top of the stack. Notice that our algorithm is not fully context-sensitive, because we assume that a stack-less RET, within a function f , may divert execution to any program point that could succeed a call to f .

One last source of imprecision is the fact that our actual implementation of `deduce` works at the intermediate representation of the LLVM compilation infra-structure. Therefore, we can only analyze code that we can compile. However, this code often calls library functions, whose source code is not available. To handle these functions, we must assume a peak density for them. In Section 4.3 we have experimented with different assumptions. By assuming that each unknown function gives us no more than one return per five instructions we have observed no false positives.

4.2.2.2 A Word On Asymptotic Complexity

The δ -CFG can be constructed in linear time on the size of a program’s control flow graph. Function `deduce` has higher complexity. As stated in the beginning of this section, our algorithm has a worst case complexity of $O(N^{(K+1)})$, where N is the number of instructions in the program, and K is the size of the detection window. However, most of the instructions in a typical program are actually sequential – branches correspond to less than 15% of the total ¹. Furthermore, multi-target branches are scarce and most of the branches that have more than one target have exactly two targets (conditional branches). These observations reduce the complexity of our algorithm to a $O(N \times 2^K)$ when all the instructions are 2-target branches – still a conservative assumption.

4.3 Experiments

This section validates the following statements: (i) specific thresholds are better than universal thresholds to prevent ROP attacks (Section 4.3.1); (ii) we find good approximations to peak density of indirect branches (Section 4.3.2); (iii) our algorithm scales well to large program sizes (Section 4.3.3).

4.3.1 Specific Thresholds Lead To Less Available Gadgets

ROP attacks require a fair number of available gadgets in the executable address space of the program under attack [Schwartz et al., 2011], otherwise it may not be possible to establish a chain of gadgets able to consolidate the exploitation. Several strategies to combat ROP attacks try to limit the number of available gadgets. The most common technique is to ensure that return operations can only divert execution to addresses that follow `CALL` instructions [Cheng et al., 2014; Pappas et al., 2013b; Zhang et al., 2013; Xia et al., 2012; Fratric, 2012; Demott, 2012; Bania, 2010; Kiriansky et al., 2002].

¹Source: http://www.strchr.com/x86_machine_code_statistics

Thus, the amount of gadgets that an attacker can use to build an attack is restricted to only those preceded by a **CALL**. Another strategy consists in replacing instructions that incidentally contain the code of indirect branches within their binary representation with alternative instructions with the same effect. G-Free [Onarlioglu et al., 2010] and Return-less Kernels [Li et al., 2010] do it at compile time. Our specific thresholds also try to limit the amount of gadgets available to attackers. To camouflage an exploit, attackers will need gadgets greater than those capable of bypassing a universal threshold. This happens because an specific threshold is strictly lower than an universal one. However, our experiments show that larger gadgets are rare.

If an attack uses gadgets of size 2, which is the average found in the publicly available exploits seen in Table 4.1, then we would need no-op gadgets of size 4 to conceal an attack from a universal detection threshold. Figure 4.5 (a) illustrates this issue. In the figure, we interpose no-op gadgets of size 4 among effective gadgets of size 2; hence, obtaining a final density of 11 indirect jumps per 32 instructions. In contrast, if we use specific detection thresholds, then gadgets of size four are not enough to disguise an attack. For most of the benchmarks (9 out of 14), which have a specific threshold of 8 or lower, Figure 4.5 (b) shows that a no-op gadget must have a length greater than or equal to 6.

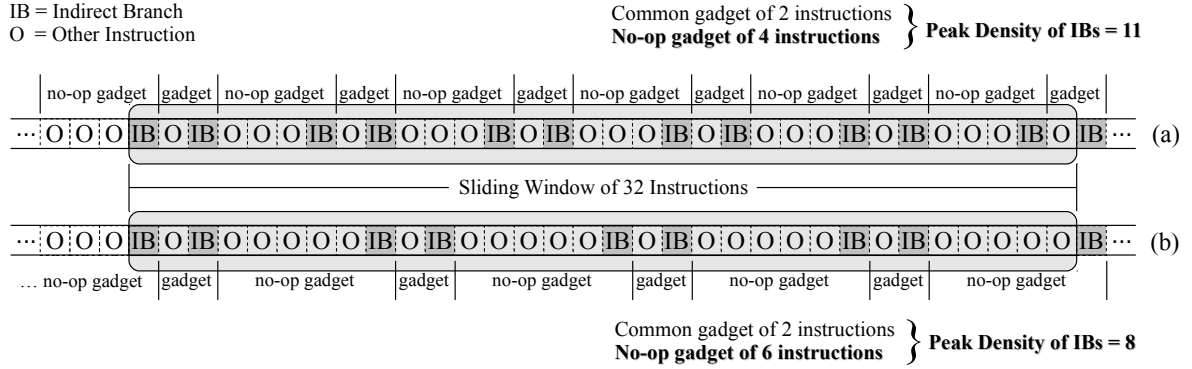


Figure 4.5. Minimal size of a no-op gadget needed to bypass universal (a) and specific (b) thresholds in a window of 32 instructions. Specific thresholds require longer gadgets, which are harder to find, as shown in Figure 4.6.

The need to use larger no-op gadgets implies stricter restrictions because the availability of no-op gadgets decreases as the length of these sequences grows. For sizes of no-op gadgets ranging from 3 to 25, Figure 4.6 shows the amount of these no-op gadgets found in the 9 vulnerable applications studied in Windows (note that Firefox has two different exploits). We use the four criteria of Section 4.1.1 to find no-op gadgets, but we have not tested their effectiveness by crafting exploits with all

of them. It was found that the number of no-op gadgets of size 4, able to circumvent a universal threshold of 12, is 75% greater than the amount of no-op gadgets of size 6, useful in overcoming a specific threshold of 8. Therefore, the use of specific thresholds drastically reduces the amount of artifacts available for building actual ROP attacks.

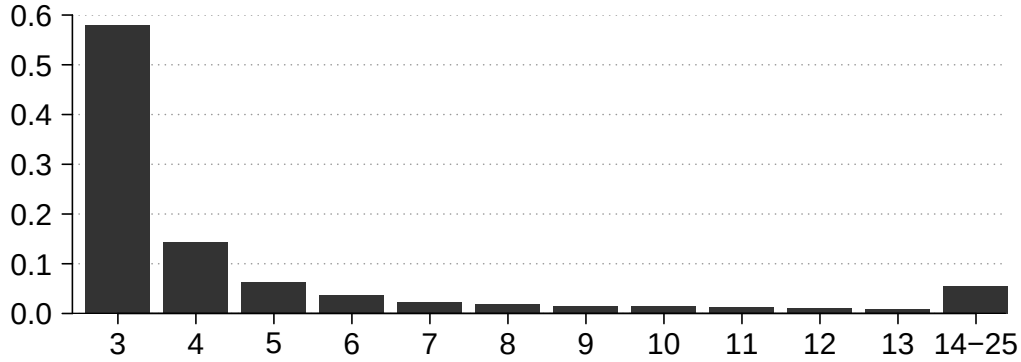


Figure 4.6. Percentage of potential no-op gadgets of sizes ranging from 3 to 25. There are 75% less potential no-op gadgets of size 6 (used to bypass the specific threshold) than 4 (used to bypass the universal threshold).

4.3.2 On The Accuracy Of Our Estimates

Aiming to measure the precision of our algorithm, we compared our estimates of maximum density for the SPEC CPU2006 suite of benchmarks with those observed when running these applications. The execution traces were monitored using the dynamic binary instrumentation framework Pin [Luk et al., 2005]. We have used the reference input of each program. Figure 4.7 shows that the peak values of the frequency of indirect branch instructions indicated by our algorithm are equal to or higher than the values recorded dynamically. This means that our algorithm is more conservative than the dynamic analysis, as it explores all possible program execution paths. In contrast, the control flow paths traversed during dynamic analysis depends on the input values sent to the application. As is often difficult to establish an input set able to exhaust all possible execution paths of a program, our static analysis exceeds dynamic monitoring in 64% of the benchmarks. In other cases, the two analyses presented the same accuracy.

The discrepancy between the values observed through static and dynamic analysis to *gcc* and *omnetpp* is due to tail recursive functions, which were not repeatedly activated by the reference input used in the dynamic analysis. Tail recursion results in a peak density of half the window size ($K/2$); hence, it gives us a density of $16/32$ indirect branches. This corresponds to a repetition of the sequence of instructions `ADD`

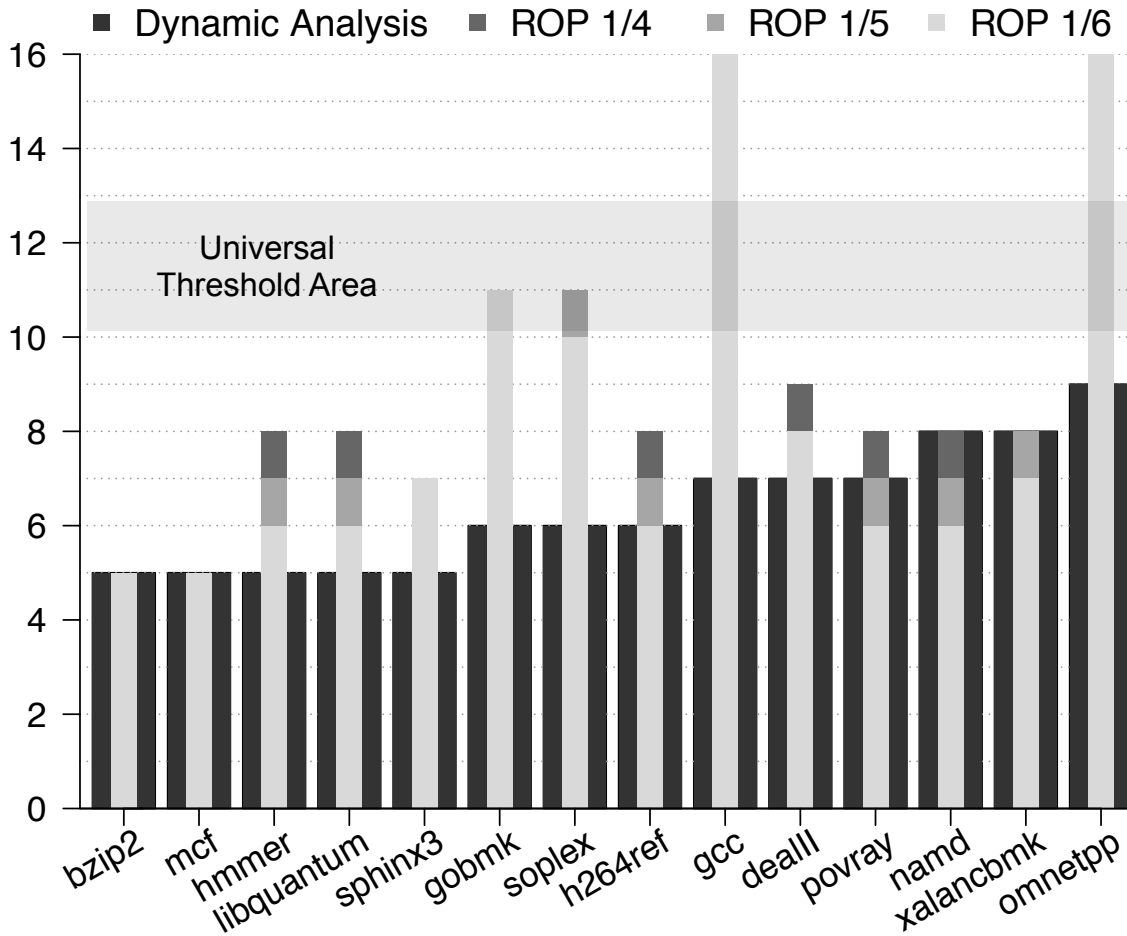


Figure 4.7. Specific threshold of indirect branch instructions for programs in the SPEC CPU2006 benchmark suite. Grey bars show specific thresholds found with the algorithm of Section 4.2, and black bars show values found dynamically. 1/4 is the density that we have assumed for library functions, which we cannot analyze statically.

ESP, #value $\rightarrow$ RET that usually appears in function epilogues. The ADD is responsible for unstacking the function frame, while the RET returns the execution flow to the caller function.

Figure 4.7 also illustrates the impact of the maximum density assumed for the library functions, whose code is not known. For these functions, we assumed that they have at most 1 return every 4, 5 or 6 instructions executed. These values were chosen since the average of the values obtained for the assessed functions is 1 return statement every 5 instructions. It was observed that this choice had no impact in 43% of the cases. For the remaining benchmarks, a maximum density of 1/5 would ensure that there are no false positives. That is, the estimated specific threshold is greater

than or equal to the maximum density monitored dynamically.

4.3.3 Performance Of The Inference Algorithm

We also measured the performance of our algorithm to infer detection thresholds. Figure 4.8 shows the execution time of the benchmarks listed in Figure 4.7 as a function of the number of assembly instructions contained in each program. This experiment uses a window size of 32 instructions. The line in Figure 4.8, which approximates the points by a coefficient of determination of 0.91491 on a logarithmic scale on both axes, indicates that, given a fixed window, our algorithm is linear on the program size. The largest SPEC program, *xalancbmk*, with approximately 700 thousand instructions, was analyzed in 25 minutes. On average, each benchmark has approximately 200,000 instructions and was analyzed in 7 minutes. Therefore, we believe that our solution can be used in practice to estimate specific thresholds for general applications. Our tool is publicly available on our web server (<http://cuda.dcc.ufmg.br/rip-rop-deducer/>).

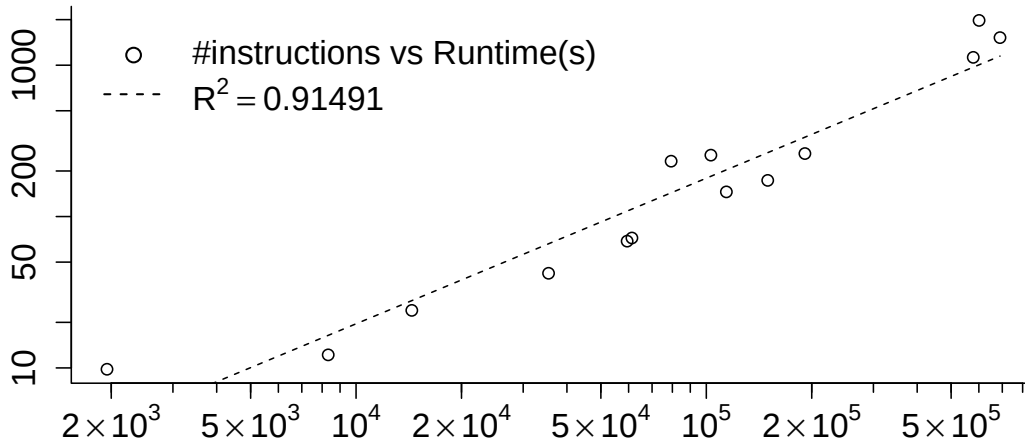


Figure 4.8. Program size vs runtime for $K = 32$. Points in the X axis are benchmarks from Figure 4.7, ordered by number of instructions. This figure indicates that even though our algorithm has a theoretical exponential complexity, its execution time has a linear behavior and is practically feasible.

Runtime vs window size. We have run the benchmarks with different window sizes: $K = 2, 4, 8, 16, 32$. Despite the costly theoretical complexity of our algorithm, discussed in section 4.2.2, the exponential behavior has not surfaced in practice. Our runtime depends on the number of instructions in basic blocks. If a basic block contains more than K instructions, we will not touch its branch when starting the search at its first instructions. More than 50% of our basic blocks contain more than 5 instructions.

Runtimes observed are roughly constant up to $K = 16$. With $K = 32$, the only exception observed was GCC, whose analysis time jumped from 613s with $K = 16$ to 1910s with $K = 32$.

4.4 Chapter Summary

This Chapter has presented a technique that supports more precise detection of Return-Oriented Programming attacks. One of the most common detection mechanism is based on the monitoring of the frequency of indirect branches in the instruction stream of programs. We give this method a way to compare this frequency against density thresholds that are specific per application. We have showed that our method is practical, scaling up to programs with more than 600 thousand assembly instructions. It also decreases effectively the number of gadgets that are available for an exploit. The recent history has shown that, thus far, there is not foolproof method to prevent ROP-based exploits. Nevertheless, we believe that our specific detection thresholds raise the bar for the successful accomplishment of an attack. In the next chapter, we shall discuss another approach to hinder ROP-based attacks: our multilayer protection system.

Chapter 5

Multilayer Protection

Recent code reuse exploits rely on a simple observation: the most effective protection mechanisms are computationally too expensive to be practical [Szekeres et al., 2013]. As we discuss in Section 3.4, the majority of anti-ROP protection techniques used in practice are based on Control Flow Integrity (CFI). This observation: “effective CFI is computationally too expensive to be practical”, has been key for the demonstration of recent ROP-based exploits [Szekeres et al., 2013; Carlini et al., 2015; van der Veen et al., 2017]. Additionally, two other factors contribute to make it difficult to design and implement a definitive CFI mechanism. First, the more constrained the implementation of CFI, the more false positives it raises, often rendering the execution of authentic programs an impractical task. Second, it has been demonstrated recently that even these more restrictive versions of CFI can be surpassed by determined attackers. Carlini et al. [2015] have shown how to circumvent fully-precise CFI, and van der Veen et al. [2017] have shown how to bypass context-sensitive CFI. Therefore, the design and implementation of a system that prevents ROP attacks from succeeding with efficiency and effectiveness remains an open problem.

In this thesis we claim that to be practical, protections against code reuse attacks must ally efficiency and effectiveness. To this end, we advocate a multi-layered runtime protection mechanism, that we describe in Section 5.1. This multilayer mechanism not only allows the implementation of expensive protections, but allows the combination of several of these defenses. The result is a more robust protection system with an acceptable overhead. The high level defense layers must be able to check most of the code cases at a low computational overhead. Lower-level protections are in charge of hindering more eccentric codes. They execute expensive verifications, from a computational standpoint; and due to this more complex checking, they are also harder to

circumvent.

As a proof of concept, we propose a three-layered defense system to prevent code reuse attacks that abuse indirect branch instructions to hijack the control flow of a program. The first layer (Layer 1), explained in Section 5.1.1, uses the hardware branch predictors to filter authentic execution flows. If the target of an indirect branch is correctly predicted, then it can safely be considered valid control flow. This statement is sound because predictions are based on the previous history of the same branch to guess its current target. Therefore, if a branch already has a history that allowed the prediction to be correct, then this branch is part of a legitimate execution of the program. As we have shown in Section 3.4, this observation has already motivated previous work. Our Layer 1 filters out most of the valid program flows, because the branch predictor is very accurate, even when applied onto indirect branches, as we show in Section 5.2.2.1. Therefore, few branches slip from the first to the second layer, where more expensive checks happen.

Layer 2, the subject of Section 5.1.2, implements an original detection strategy that reduces the quantity of gadgets available to attackers. It extends Kiriansky *et al.*'s concept of `CALL`-preceded return address verification [Kiriansky et al., 2002]. However, in addition to checking if a return address is preceded by a `CALL` instruction, we also verify if that `CALL` is valid. This new step was necessary since Carlini et al. [2015] have already showed that Kiriansky's method is not difficult to circumvent. To deal with JOP and COP attacks, we also adopted in our second layer the strategies provided in Control-Flow Enforcement Technology. Our third layer, the subject of Section 5.1.3, avoids false positives. The goal of this layer is to reduce the number of false alarms that would be raised by genuine control flows.

We evaluate our ideas in Section 5.2. In Section 5.2.3 we report the execution time overhead for a set of benchmarks composed by all 29 programs from SPEC CPU2006 [Henning, 2006] plus 209 benchmarks from the LLVM Test Suite [Lattner and Adve, 2004]. We also assess the effectiveness of CET and the new check introduced in Layer 2 to validate `CALL` instructions that precede return addresses. These results are presented in Section 5.2.1. To analyze in Section 5.2.2 the number of false positives that slip through our first two layers of protection, we enriched our benchmark suite with the three most popular desktop browsers and four vulnerable applications for which there are known ROP exploits. We demonstrate the viability of our ideas using a prototype implemented in software, via Pin [Luk et al., 2005]; however, an important part of this work is to show that all these layers could be built at the hardware level, via microarchitectural units already in place in modern versions of x86 processors. Such units include the branch prediction machinery, branch recording structures,

and executable space protection mechanisms. To support this claim, the description of every protection layer contains a discussion on how it can be implemented in today’s hardware.

The change of perspective mentioned in our Thesis Statement (Section 1.2) fosters a very modular design. It is conceptually possible to replace the second layer of our model with other techniques that validate branches, as long as the necessary architectural units are in place. One candidate that fits well our second layer is Intel’s Control-flow Enforcement Technology (CET) [Intel, 2017] – a technique that also thwarts Jump-Oriented [Bletsch et al., 2011; Checkoway et al., 2010] (JOP) and Call-Oriented [Carlini et al., 2015; Goktas et al., 2014] (COP) Programming attacks. The basic principle is the same: the high-accuracy of the branch predictor leaves less Jumps and Calls to be verified by CET. And, similar to the Return Address Stack, the CALL and JMP predictors cannot be tampered with, as they are based on the previous execution of the instruction. When a CALL or JMP is executed for the first time, the branch predictor always misses and, therefore, forces the validation of that instruction in Layer 2. To demonstrate modularity, in Section 5.1.2 we also discuss CET details and in Section 5.2 we show empirically the benefits of implementing Layer 2 using CET.

5.1 Layers Of Protection

This work aims to demonstrate that the Thesis of using multilayers to protect against ROP, JOP and COP attacks is effective. The deeper layers of the system perform more rigorous and effective checks at a higher cost. However, these costly layers are only enabled for a tiny number of cases, as the model’s shallow layers tend to filter out virtually all authentic execution flows. As a result, the ultimate impact of the system on application performance is unnoticeable.

Figure 5.1 illustrates the idea of filtering execution streams in layers to reduce the volume of checks performed by the deeper layers of the system. This idea is anchored in a paradigm shift: instead of trying to quickly identify suspicious operations, the multilayer approach attempts to validate legitimate ones. The first layer of our model is responsible for this task, as described in Section 5.1.1. Because of the inexpensive identification of authentic execution flows, it is possible to perform more thorough checks for the few remaining cases in the second layer, detailed in Section 5.1.2. We also propose a third layer for the treatment of occasional false positives. For these cases, we describe in Section 5.1.3 the appropriate solutions to be implemented in the

third layer of the system.

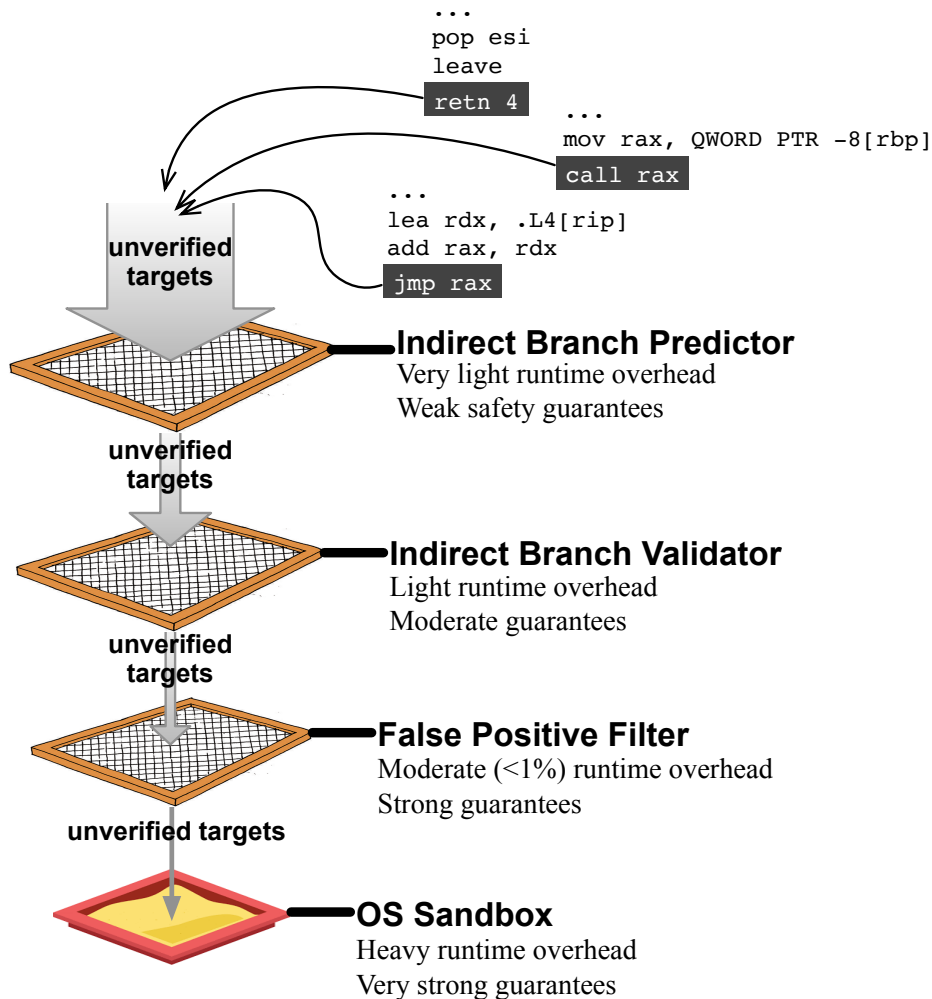


Figure 5.1. Abstract overview of our multilayer protection mechanism.

An additional procedure is necessary to give a final destination to the malicious operations that slip through the three layers proposed. This fourth layer is meant to deal with instructions classified as part of an attack. The literature describes several strategies to handle attacks. The most common solution is to throw an exception that is handled by the operating system. This approach is adopted, for instance, in the implementation of the protection $W \oplus X$ [PaX Team, 2003a]. In this case, most operating systems interrupt the execution of the compromised process, alerting the user about what happened [Cugliari and Graziano, 2010]. More complex solutions to handle attacks include sandboxing, an approach that Figure 5.1 illustrates.

Sandboxing has been widely used in the dynamic analysis of malware behavior [Egele et al., 2012]. It is usually implemented through emulated or virtualized run-

time environments and often detect malicious behavior by monitoring system calls [Rajagopalan et al., 2006]. However, this approach is not normally considered for real-time situations, due to the high overhead it entails. Previous reports indicate that the computational cost of emulating a program ranges from 2 to 40%, depending on implementation choices and the amount of system calls executed by the monitored application [Kolbitsch et al., 2009]. Nevertheless, given that our multilayer model releases authentic execution flows before they reach this level of verification, even costly validations like sandbox analysis can be incorporated into our system at its deepest layers. Even so, we emphasize that the implementation of sandboxes is a research topic already widely explored, not being the focus of this work. Our goal is to show that through efficient filtering of authentic execution flows, it is possible to design multilayer protection against ROP, COP and JOP attacks that combine a high degree of security with a low computational cost.

5.1.1 First Level: Branch Predictors

The first layer of protection relies on the branch predictors to validate targets of indirect branches. Modern computer architectures try to predict the target of indirect branch instructions to avoid stalls in the pipeline. For instance, current implementations of x86 use the so called *Return Address Stack* (RAS) to predict the address targeted by such instructions. Whenever a function is invoked through a `CALL` instruction, the address immediately following this invocation point is pushed onto the RAS. When a return operation executes, the predictor attempts to guess its target as the address on top of the RAS. This modus operandi is simple, yet, very effective and costless.

The use that we make of the RAS resembles the protection known as *Shadow Stack*, described in Section 3.1.2.2. A shadow stack does exactly what the RAS does: matches the target of `returns` against the addresses following `CALL` instructions, in a stack-like fashion. However, shadow stacks were conceived before the RAS became part of current x86 processors; hence, it could not benefit from this hardware. We claim that this idea, the shadow stack, can be implemented at zero-overhead, at the hardware level, by capitalizing on the RAS infrastructure, which is already in place.

All branch target predictors are designed to achieve high hit rates. Therefore, after undergoing numerous improvements, the other types of indirect branch predictors have the same characteristics of efficiency and effectiveness exhibited by RAS. These predictors can also be used to identify authentic execution flows. It is possible to state that an indirect `CALL` or `JMP` is valid whenever the target of that branch is correctly predicted. This statement is sound because predictions are based on the previous

history of the same branch to guess its current target. Therefore, if a branch already has a history that allowed the prediction to be correct, then this branch is part of a legitimate execution of the program.

The first layer of our model leverages the high accuracy of branch predictors: correctly predicted branches are considered valid flows. Thus, the second layer is activated upon a `RET`, indirect `JMP` or indirect `CALL` whenever there is a failure in predicting the target of that branch. The first execution of any indirect `JMP` or `CALL` always results in a prediction miss, as there is still no history for that instruction. Therefore, the first execution of any indirect `JMP` or `CALL` is always validated by the second layer of our model. Considering the high precision of modern predictors, reported in Section 5.2.2.1, few indirect branches are forwarded to the second layer. The implementation of this protection layer only requires one extra comparison at the level of the branch predictor. This check can be fully implemented in hardware with no impact on the execution time of programs, since it does not directly impact the instructions pipeline.

5.1.2 Second Level: e-CC and CET

Not every indirect branch that is mispredicted is a symptom of a ROP based attack. Mispredicted return addresses happen due to overflows in the RAS buffer, an incident caused by very long sequences of function invocations. The first execution of any indirect `JMP` or `CALL` also causes mispredictions. Also, context switches between processes that compete for the CPU can generate mispredictions for all indirect branches. Therefore, if the target of an indirect branch is mispredicted, our protection resorts to further validations in the second layer. Many different implementations of this second layer are possible. In Section 5.1.2.1, we propose enhanced Call-preceded Constraint (e-CC), a new strategy to defeat ROP attacks. In Section 5.1.2.2, we discuss how Intel’s Control-flow Enforcement Technology (CET) can be used in our model to block JOP and COP attempts.

5.1.2.1 enhanced Call-preceded Constraint

Our next verification step, the second layer of our system, relies on a simple idea: we check if the target of a `RET` instruction is preceded by a `CALL` instruction, *and* if the target of the `CALL` instruction that precedes it lays in executable memory. The first verification is the cornerstone of modern implementations of control flow integrity (CFI) [Tice et al., 2014]. The second, however, is a new idea. We had to add further verification on top of traditional CFI, to mitigate shortcomings of this technique, already well-known in the literature [Davi et al., 2014; Goktas et al., 2014]. Figure 5.2

provides an overview of all these interactions to guide the discussion in the rest of this Section.

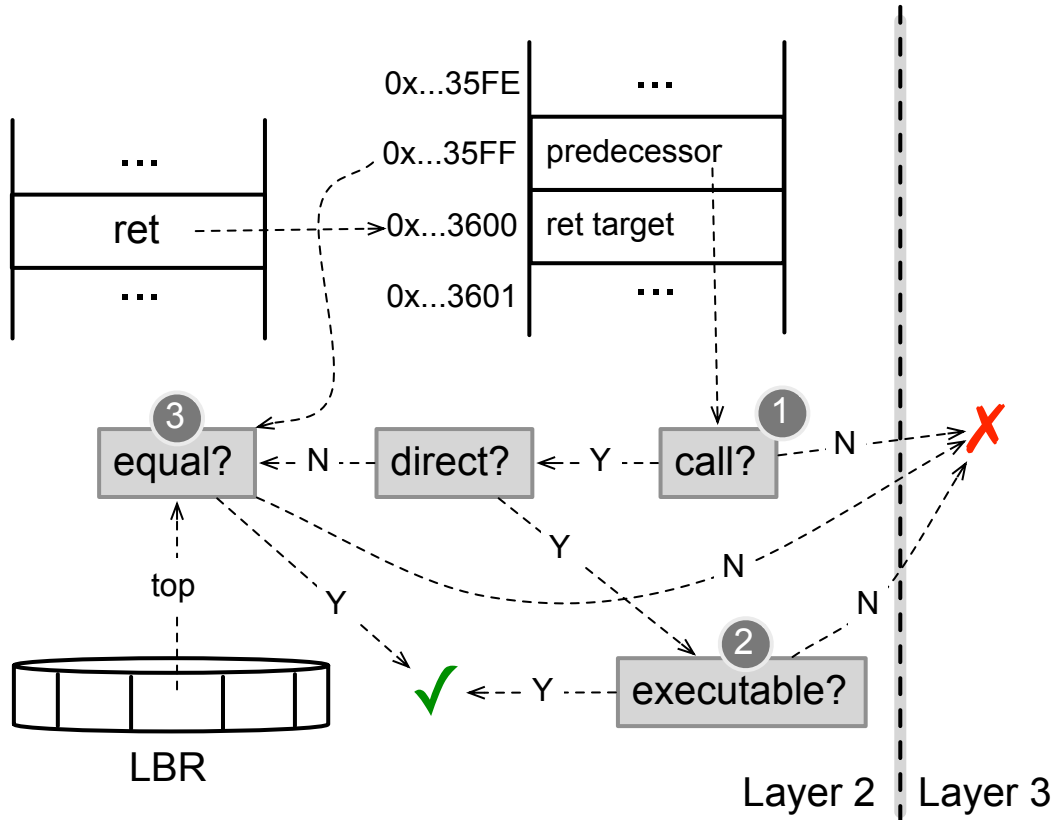


Figure 5.2. The second protection layer.

The CALL-preceded constraint. If the target of a return instruction is not an address that succeeds a `CALL` instruction, then it is possible that such flow has been artificially created by an attacker. This observation has already been extensively explored in the literature [Kiriansky et al., 2002]. Thus, in the absence of this condition, we immediately move the application to the third verification layer of our system (Step 1 in Fig. 5.2), where we apply stronger checks to certify the legitimacy of the flow. However, even when this condition is satisfied, we might still be facing a ROP attack. As an example, Goktas et al. [2014] have demonstrated how to build ROP exploits under the `CALL`-preceded constraint. Therefore, to prevent exploits such as the one carried out by Goktas et al. [2014], we augment this constraint with an extra check, which we claim as an original contribution of this thesis: the *Executable Target* constraint.

The Executable Target (XT) Constraint. Gadgets whose first instruction is preceded by `CALL` operations might emerge by chance in a large binary program, due to

unintended instructions. These are instructions formed by unaligned sequences of bits within the binary code. We can show that the vast majority of such gadgets are invalid through a simple expedient: we verify if the **CALL** instruction targets an executable memory segment (Step 2 in Fig. 5.2). Because the executable area in the address space of a program is limited to its “text” segment, the chance that a misaligned **CALL** instruction targets this segment is very small, especially in 64-bit architectures.

Example 3 *The Linux version of Chrome 59.0.3071.115 (64-bits) has a large amount of code, resulting in an equally long (163 MB) text segment. Yet, the chance that a random address matches an executable address is only 1 in 10^{11} .*

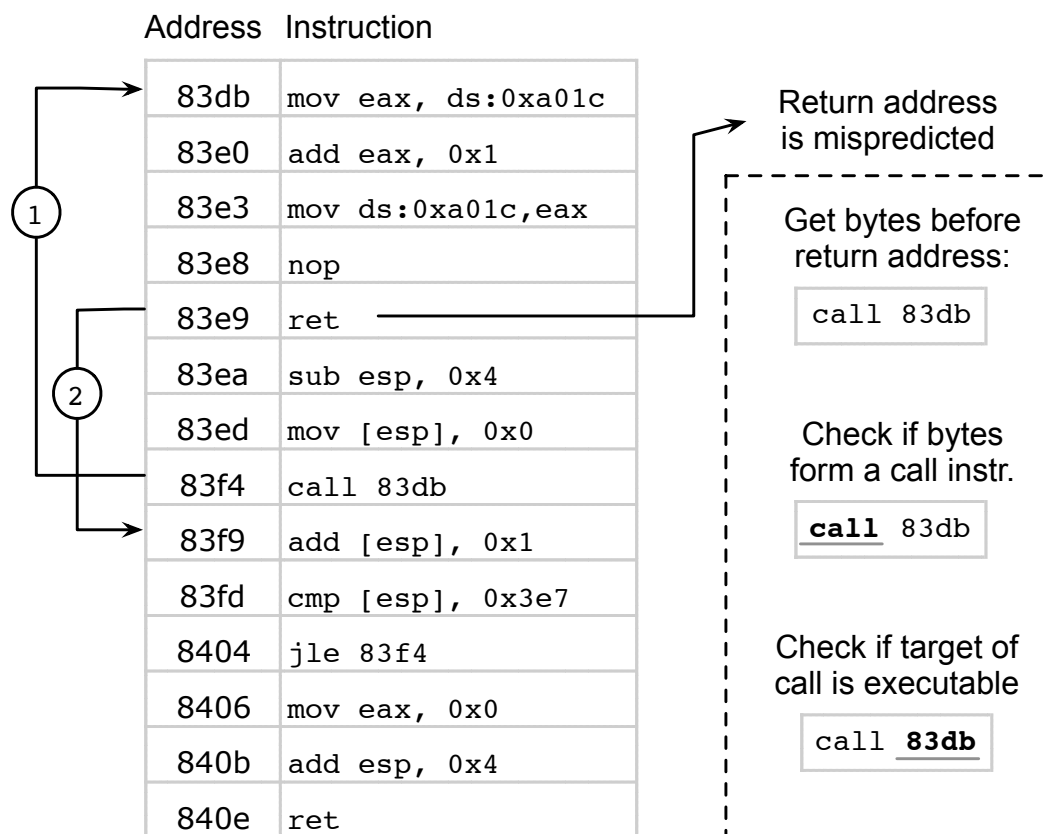


Figure 5.3. Example of the checks performed by the second layer on a program running on a 16-bit address space. Arrows on the left indicate the targets of indirect branches. Numbers on the arrows indicate which branch executes first.

The different types of **CALL** instructions available in the x86 range from 2 to 7 bytes in size. Thus, when executing a **RET** instruction, we look for the 7 bytes that precede the instruction located in the return address stored at the top of the stack. We then decode the instruction formed by these bytes and check if it corresponds to a **CALL**.

If it is a **CALL**, then we calculate the target address of that instruction. To obtain this address, we differentiate the types of **CALL** by their opcodes and the byte “ModR/M” in the instructions, as defined in the manual of the x86 instruction set [Intel, 2018]. Finally, we verify if the target address of the **CALL** instruction has the no-execute (NX) bit enabled. Notice that processors already check the NX flag in the page-structure entry of any fetched instruction. Therefore, the microarchitectural structures that already exist for this purpose can be reused by our XT Constraint Checker.

Example 4 *Figure 5.3 shows a concrete example of checks performed by layer 2 in a hypothetical 16-bits architecture.*

XT Enforcement on Indirect CALLs. The verification of the target address of indirect **CALLs** is not trivial, because when a **RET** operation executes, the address used by the **CALL** that gave origin to it might no longer be available in either registers or memory. To deal with this shortcoming, we resort to a hardware structure called *Last Branch Record* (LBR). This feature, available in Intel processors, lets the CPU log the “from” and “to” address of each branch into specific registers. These registers form a ring-buffer, which is continuously overwritten. Thus, it traces only the most recent branches. We choose to only record data about **CALL** in the LBR. In addition, we use this hardware structure in its stack mode. Thus, whenever a **CALL** executes, its data is stacked in the LBR. When executing a **RET**, the most recently stacked data is removed from the LBR. The main difference between the LBR stack and RAS is that LBR enforces data savings on context switches. This allows the LBR to treat some cases not covered by the RAS. Different processor models have different LBR sizes: Netburst and Merom have buffers with 4 cells; Nehalem up to Haswell have buffers with 16; Skylake features 32 positions and the Atom only 8. When faced with an indirect **CALL** preceding a return address, we scan the last filled entry of LBR looking for a match (Step 3 in Fig. 5.2). Notice that only mispredicted return addresses undergo this verification, because correctly predicted instructions have already been filtered out in the first layer of our system.

The XT constraint can be enforced in processors featuring Return Address Stack and Last Branch Record capabilities. For instance, current Intel/AMD machines feature an extension called *eXecute Disable*: XD in Intel, and NX (of no-execute) in AMD. This mechanism marks virtual pages with a bit denoting executable memory. This bit is consulted at zero overhead. The same is expected for a hardware comparison

with the value stored at the top of LBR.

In Section 5.2 we present results for simulating the implementation of this layer in software via a Pintool, i.e., a tool created with Pin. The Pintool checks if the targets of direct `CALL` instructions that precede return addresses lay in executable memory. It also simulates the LBR operation and matches the top of LBR with indirect `CALL` instructions that precede return addresses.

5.1.2.2 Control-Flow Enforcement Technology

As we show in Section 5.2.2.1, the branch predictor is accurate enough to legitimize most of the indirect `JMPs` and `CALLs` observed during the execution of a typical program. However, invariably, some of these branches will not be validated by this approach, e.g., their first execution, for instance. To deal with these omissions, our model provides a second layer of validation. Many different implementations of this second layer are possible. In this work, we have experimented with Intel’s Control-flow Enforcement Technology (CET) [Intel, 2019].

We chose CET because there is a high expectation for this technology. Windows [Sun et al., 2019] and Linux [Corbet, 2018] operating systems, for instance, have already received initial adjustments to support it. Furthermore, some popular compilers, such as GCC [GCC, 2020], LLVM [LLVM, 2020], Visual Studio [Microsoft, 2020] and ICC [Intel, 2020], already generate CET compliant code. All this expectation is due to the belief that protection implemented in hardware will present less performance degradation and more effectiveness against bypasses [Sun et al., 2019].

CET: a Brief Overview. Control-Flow Enforcement Technology is an initiative by Intel that aims to add a mechanism for validating indirect branches. This mechanism plans to use a shadow stack [Vendicator, 2000] to protect against ROP attacks (review Section 3.1.2.2 for an overview of shadow stacks). In the case of the JOP and COP attacks, CET will adopt the original idea of Control-Flow Integrity [Abadi et al., 2005]: adding a marker to the targets of indirect `JMPs` and `CALLs` at compile time, and validating these markers at run time. CET’s specification establishes the inclusion of a marker instruction in the x86 instruction set architecture [Intel, 2019]. This special instruction is named `Endbranch` and is referenced in assembly by the mnemonic `ENDBR32`, on 32-bit architectures, or `ENDBR64`, on 64-bit architectures.

Once in production, the `Endbranch` instruction will be identified by the opcode `0xF30F1EFA`. Said opcode is a *no-operation* (NOP) instruction in legacy systems. Thus, applications compiled to use CET are backwards compatible with older systems that

do not support this technology. According to the specification, even in processors with CET support, `Endbranch` will work as a NOP, because it will not affect the program’s execution state, including registers and flags. Designers also claim that `Endbranch` will have a minimal impact on both performance and energy consumption. We evaluated this claim and present an estimate of the lower and upper bound overheads of this approach in Section 5.2.3.

Example 5 Figure 5.4 illustrates how indirect `JMPs` and `CALLs` are tracked in CET. The C code executes a call to the function `test` via the function pointer `f`. When compiled, this function invocation is transformed into an indirect `CALL`, as illustrated in the assembly code on the right side of Figure 5.4. If the code is compiled with CET support, `ENDBR` instructions are added at the beginning of each function, as they correspond to indirect `CALL` targets. On the right side of Figure 5.4, these instructions appear at the beginning of functions `main` and `test`. When executing the instruction `CALL *%rdx`, a processor with CET enabled will check if the target of that branch corresponds to an `ENDBR` instruction. If the comparison matches, the program resumes normal execution. Otherwise, the processor throws a control protection exception (`#CP`).

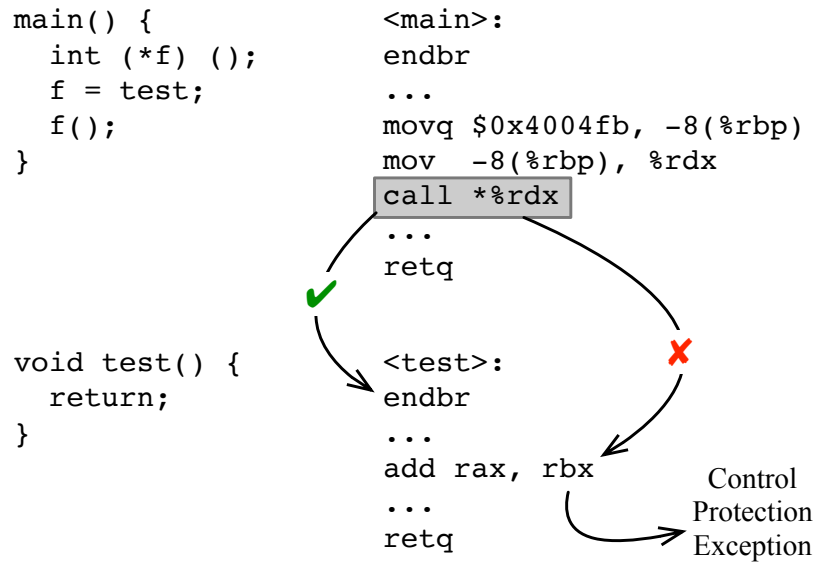


Figure 5.4. CET’s Indirect Branch Tracking (IBT) operation [Intel, 2019].

Figure 5.4 illustrates the application of CET to guard indirect function calls. The same procedure is used for indirect `JMPs`. In both cases, throwing the exception is only avoided if the legacy compatibility treatment mode is active. This mechanism exists to address cases of false positives, which we discuss in Section 5.1.3.

5.1.3 Third Level: False Positives Filtering

The two first layers of the proposed system might fail to identify authentic flows under special circumstances. Therefore, additional filtering is necessary for the occasional false positive. In the case of CET, its specification already provides solutions to treat false positives resulting from both the use of a shadow stack and the comparison of targets of indirect `JMPs` and `CALLs` with the `ENDBR` marker instruction.

The use of the shadow stack can cause false positives in situations where the parity between `CALL` and `RET` instructions is not respected. `Setjmp` and `Longjmp` operations, or exception handling in languages like C++, for instance, lead to mismatches between `CALLs` and `RETs`. Under such circumstances, CET provides additional instructions to unwind the stack, making it follow the irregular displacement of the program stack pointer that occurs in these scenarios [Intel, 2019].

The verification of the existence of an `ENDBR` instruction in the target of indirect `JMPs` and `CALLs` can also lead to false positives when some legacy code—which does not support CET—is running. To avoid this problem, CET offers a mode of compatibility with legacy systems. This compatibility treatment allows CET-compliant software to use software that has not been compiled or instrumented with `ENDBR` instructions. Before moving a CET enabled program to compatibility mode, the processor checks whether:

- The bit that enables legacy compatibility handling is active in a model-specific register (MSR) of the current privilege level (user or supervisor modes). These MSRs are created for the exclusive use of CET.
- The bit corresponding to the target address of the indirect branch is enabled in a linear bitmap created in the program memory specifically to indicate codes incompatible with CET. Each bit in the bitmap represents a 4 KB page in linear memory. If the bit is enabled it indicates that the corresponding code page is not compatible with CET. The processor uses the linear address of the indirect branch target to lookup the bitmap and decide whether the program can be moved to legacy compatibility mode, avoiding a false positive.

It is also possible that legitimate return addresses pass unfiltered through all our previous sieves because some authentic program flows contain `RET` instructions targeting addresses not preceded by any `CALL` instruction. The typical situations where this event happens have been cataloged by previous work [Xia et al., 2012; Davi et al., 2011b; Zhang and Sekar, 2013]: lazy binding of dynamically shared libraries and signal handling in Unix-based systems.

False Positives due to Lazy-Binding. Lazy binding optimizes the loading time of a program. It consists in delaying the resolution of the initial address of functions belonging to shared libraries until these functions are called for the first time. Thus, the dynamic linker does not waste time solving the address of functions that are never called. In Linux, for instance, false positives can occur because a return instruction is “improperly” used by the OS during lazy binding. On Linux systems, lazy binding is enforced by a combination of functions that are part of the `linux-ld.so` dynamic linker library. After resolving the address of the target function, the OS transfers the execution flow to the code of that function through a return instruction. False positives can occur because the execution flow is diverted to the first instruction of the requested function and there is no guarantee that a valid `CALL` exists immediately before that first instruction.

False Positives due to Signal Handling. In Unix-based systems, signals alert a process about exceptional events such as the decoding of illegal instructions, the execution of invalid arithmetic operations and unauthorized accesses of memory segments. Upon receiving a signal, the OS stacks the address of the current instruction of the paralyzed process and transfers the execution flow to the code responsible for signal handling, without executing a `CALL` instruction. When the signal handler returns, it removes the address stacked by the OS and transfers the execution back to the normal program flow through a `RET` instruction. A process can be stopped by the OS at any instruction; hence, the address just removed can refer to anywhere. Therefore, the return address stacked by the OS may not be preceded by a valid `CALL` operation – a fact that might generate false positives.

Detection of False Positives. The two categories of false positives caused by the disparity between `CALLs` and `RETs` can be avoided at the compiler level or at the hardware level. At the hardware level, false positives can be identified via extra checks, such as those proposed by Xia et al. [2012] in the CFIMon system. CFIMon records performance samples using the *Branch Trace Store* (BTS) mechanism. The BTS makes it possible to monitor executed branches and correlate the traces with sets of valid targets obtained via static analysis. To avoid triggering false alarms, Xia *et al.*’s monitor is informed by the operating system whenever the signal handler is invoked.

At the compiler level, it is possible to change problematic `RET` statements by a `POP rg; JMP rg` sequence. The return address r is dumped in register `rg`. Then the control flow is transferred to r via an indirect `JMP`. Zhang and Sekar [2013] propose a

similar solution to circumvent the idiosyncrasy related to using a return instruction for lazy symbol resolution. In our case, this modification must be done in the dynamic linker and in the signal handler. As Section 5.2.3 will show, this replacement does not incur into any overhead of statistical significance on a modern x86 machine.

The strategy used to handle false positives depends on the implementation of Layer 2. In Section 5.2 we shall compare two implementations to monitor return instructions: one based on the XT Constraint validation technique seen in Section 5.1.2, and another based on the shadow stack used by CET. We also analyze the behavior of CET’s Indirect Branch Tracking, Intel’s system aimed to protect indirect `JMPs` and `CALLs`. The implementation discussed in Section 5.1.2 let us avoid false positives at the compilation level. In this case, no runtime check is required¹. On the other hand, the CET-like implementation of a shadow stack requires runtime validation at the level of Layer 3. In this case, we can reuse either the techniques employed in the CFIMon tool [Xia et al., 2012], or the rules discussed by [Qiu et al., 2018, Sec.V]. Both approaches require circuitry not yet in place in current x86 processors.

In Section 5.2.3.4 we shall implement the compiler-level enforcement of Layer 3 via manual replacement of instructions in assembly codes. To experiment with CET, we shall simulate it in Pin, following a description publicly available of the Intel implementation [Intel, 2019].

5.2 Evaluation

The goal of this Section is to answer three research questions, which we enumerate below:

- **RQ1:** How effective is our multilayer method to reduce the number of gadgets available for an exploit?
- **RQ2:** What is the proportion of targets of indirect branches that are filtered at each protection layer?
- **RQ3:** What is the overhead of our approach?

We answer these questions empirically. In Section 5.2.1 we use an assortment of dynamic analysis tools to count how many gadgets our system leaves available to

¹If false positives are avoided at the compilation level, then our Layer 3 becomes inexistent at runtime. We preferred to leave it in our model, to emphasize that it exists to handle false positives produced at Layer 2.

an attacker. In Section 5.2.2 we use Linux Perf [de Melo, 2010] to count how many branches slip from Layer 1 to Layer 2, and Pin to measure leaks from Layer 2 to Layer 3. Finally, in Section 5.2.3 we evaluate the overhead of our approach using an analytical argument supported with data from Pin. We also resort to manual interventions in assembly programs to evaluate the overhead of a compiler-based implementation of Layer 3. The tools used in the evaluation are publicly available on our web server (<http://cuda.dcc.ufmg.br/multilayer-rop-protection/>).

Benchmarks. We used several benchmark suites, because each set fits different purposes. To count branch mispredictions (**RQ2**) and measure overhead (**RQ3**) we use all the programs in SPEC CPU2006 [Henning, 2006] and in the LLVM Test Suite [Lattner and Adve, 2004]. As highlighted by Burow et al. [2017], SPEC is prevalent as a performance measurement instrument. Additionally, we also use the LLVM Test Suite due the large number of benchmarks that it contains (209, in contrast to the 29 programs available in SPEC CPU2006). Furthermore, some of the LLVM benchmarks are even larger than SPEC CPU2006 applications. The diversity found in the LLVM Test Suite favors a more complete analysis of the proposed model, since it widens the range of experiments performed.

To probe gadget reduction—in an effort to answer **RQ1**—we deal with two scenarios. To measure JOP and COP gadget reduction we only need to assess the **ENDBR** instructions inserted by the compiler. Since it is a static approach, we use all the programs in SPEC CPU2006. We did not use the benchmarks from the LLVM test suite in this experiment because, unlike SPEC, this test infrastructure does not offer mechanisms to automate the processes of running auxiliary tools on the generated binaries. Probing ROP gadget reduction and false positives, on the other hand, is an execution time procedure. For each **RET** that is about to execute, we need to check the current state of the machine, like the return address and its characteristics (e.g. if it is a **CALL**, direct or indirect, etc). Running this dynamic analysis for the SPEC and LLVM benchmarks requires an impracticable execution time. Therefore we used 4 vulnerable applications for which we could reproduce 5 public ROP exploits (exploits 17634, 17672, 17974, 12189, 13834 at www.exploit-db.com) and the 3 most popular desktop web browsers [NetMarketShare, 2018].

Following Hennessy and Patterson’s methodology [Hennessy and Patterson, 1994, Sec-1.8], averages that we report to summarize results are weighted by the number of instructions executed: programs that run more instructions contribute proportionally more towards the final average.

System Setup. To investigate JOP and COP gadget reduction, we use the following environment:

- Intel Core i5-3230M 2.60GHz, 4 cores, 32K (L1d) / 32K (L1i) / 256K (L2) / 3072K (L3) caches, 8GB RAM (2x4096MB DDR3 1600MHz).
- Ubuntu 18.04.3 LTS Linux 4.15.0-72-generic
- GCC 9.2.1
- Ropper 1.13.4

The other experiments were carried out in the following environment:

- Intel Xeon E5-2620 2.00GHz, 12 cores, 32K (L1d) / 32K (L1i) / 256K (L2) / 15360K (L3) caches, 16GB RAM (4x4096MB DDR3 1333MHz).
- Ubuntu 18.04.2 LTS Linux 4.15.0-65-generic
- GCC 9.1.0
- Pin 3.4 Kit 97438
- Perf version 4.4.134

A note on the choice of tools. We perform all the experiments that require interventions in hardware via simulation in Pin. We have considered instrumenting the benchmarks using LLVM [Lattner and Adve, 2004] or using a hardware simulator, such as gem5 [Binkert et al., 2011]. We discarded instrumentation via LLVM because it would not allow us to evaluate codes of libraries loaded by the program under analysis. As for gem5, its overhead makes it impractical for our experiments. A recent study with gem5 recorded a rate of 50-500 KIPS [Zhang et al., 2018], but our experiments with SPEC CPU2006 alone would require processing approximately 50 TB of instructions. Furthermore, we cannot replicate hardware optimizations that are not disclosed by manufacturers in a precise cycle simulator, which would result in inaccurate results. We know, for example, through our own experiments, that the hit rate achieved by the Return Address Stack (RAS) existing in modern processors is much higher than the hit rate achieved by a conventional stack. This is probably due to the use of optimizations as the RAS speculative management technique [Vandierendonck and Sezne, 2008]. This difference has also been reported for experiments with the memory system in hardware simulators [Akram and Sawalha, 2016]. Intel’s Indirect branch predictors

also make use of undisclosed optimizations; thus, the omission of these improvements in our simulation would distort our results. Therefore, we preferred to capture the behavior of a real processor through hardware performance counters. We used the “perf” [de Melo, 2010] utility for this task.

5.2.1 Gadget Reduction: a false negatives assessment

In the context of this work, *false negatives* happen whenever we fail to detect a ROP-based attack. As Burow et al. [2017] point out, there is no exact mechanism for establishing the degree of security offered by a given protection against code reuse attacks. The most used metric has been to quantify the number of gadgets available for the construction of an attack [Zhang and Sekar, 2013; Niu and Tan, 2014a]. The greater the availability of chunks of code, the greater the probability that it will be possible to interconnect these pieces to perform malicious activity. Therefore, a good ROP defense mechanism leaves few gadgets still accessible to an attacker. In Section 5.2.1.1, we assess the reduction in gadget availability imposed by our new protection strategy: the enhanced Call-preceded Constraint (e-CC). In Section 5.2.1.2, we measure the impact of using CET on this same gadget availability.

5.2.1.1 enhanced Call-preceded Constraint

Attackers can overcome our multilayer system equipped with our enhanced Call-preceded Constraint (false negatives) using gadgets that have any of the following properties:

1. the gadget does not trigger a branch misprediction;
2. the gadget is preceded by an indirect CALL instruction that is at the top of LBR;
3. the gadget is preceded by a direct CALL instruction that targets an executable memory region;
4. the gadget passes the false positive checks.

Finding gadgets with property (1) is impracticable because branch predictors record data about instructions already executed by the program. Thus, the first subverted branch instruction causes a misprediction, unless the attacker can reuse a program flow already executed. The impossibility of finding gadgets with property (1) is corroborated by previous studies [Liu et al., 2014; Santos and Fei, 2008; Shi and Lee, 2007]. It is also highly improbable to abuse RAS (property 1) or LBR (property 2) because for each

gadget it would be necessary to force the execution of a **CALL** instruction positioned immediately before the return address. Even more: this **CALL** operation must be the most recent instruction of this type to be executed. This would only be possible in a scenario where useful gadgets for the attack, terminated with an indirect **CALL**, are sequentially placed in an executable address space so that an attacker can merge them with gadgets ending in **RET**s. This event is equivalent to accidentally having attack code ready within the application itself. Finally, notice that our multilayer approach does not introduce new gadgets in the process of eliminating those already known – a shortcoming present in techniques that shrink source code, as pointed by D. Brown and Pande [2019].

In principle, a user can combine ROP with an attack similar to Spectre against RAS to change the execution flow of a program from a process of another user. However, even if a malicious user succeeds in polluting RAS with fake return addresses, the context switch between the malicious program and the vulnerable process would need to happen exactly between the original **CALL** and the vulnerable **RET**. The inability of an attacker to interfere in those switches yields this scenario improbable. Finally, item (4) filters the cases enumerated in Section 5.1.3, which otherwise would be misclassified as attacks. Using gadgets with those characteristics is virtually impossible, as they involve Operating System code. Thus, we shall count how many gadgets meet property (3) by chance (i.e. direct **CALL**s whose random target is executable).

Measurement methodology. To count ROP gadgets we follow a methodology proposed by Bowne [2014], which mirrors the procedure adopted by attackers during the process of building ROP exploits. Firstly, we pause the program of interest at a particular execution point (right before hijacking the control flow, for instance) with help of Immunity Debugger [Immunity, 2020]. Secondly, we use our own crafted version of a Mona [Corelan Team, 2011] plug-in to scan the whole process memory and list all available **CALL** instructions. This list lets us identify which gadgets are preceded by a **CALL** operation, even when this instruction is the result of an unaligned memory access. Finally, we use Pin to simulate the **CALL** validation mechanism of Section 5.1.2.1. Pin lets us inspect each instruction upon its execution. We use it to verify (i) that the destination address of a **CALL** that precedes a candidate gadget is executable and (ii) to simulate the presence of the LBR.

Analysis of Results. Table 5.1 shows the results of this experiment on seven large applications – four of which are vulnerable to known ROP exploits. The number of gadgets left by our *enhanced Call-preceded Constraint* (e-CC) is 0.06% of the total

number of gadgets in the binary. To give the reader some perspective on this number, this constraint leaves 34.94 times less gadgets than the *original call-preceded constraint* when applied onto the same suite. The original call-preceded constraint is central to many previous defense mechanisms [Kiriansky et al., 2002; Xia et al., 2012; Fratric, 2012; Pappas et al., 2013b; Zhang et al., 2013; Zhang and Sekar, 2013].

Table 5.1. Gadget Reduction on four applications vulnerable to known ROP exploits (top) and on the three most popular desktop browsers in November 2018 (bottom) [NetMarketShare, 2018].

Application	#Gadgets	CPG	VCPG	BS(MB)	AS(KB)
PHP 6.0	269,318	2.77%	0.04%	0.03	0.01
Free CD to MP3 Converter 3.1	346,415	2.63%	0.09%	1.08	0.99
Internet Explorer 8.0.6001.18702	624,638	2.64%	0.14%	2.45	3.59
Mozilla Firefox 3.6.16	1,444,959	2.61%	0.10%	0.87	0.88
Mozilla Firefox Quantum 62.0.3	1,779,989	2.22%	0.05%	1.31	0.73
Google Chrome 69.0.3497.100	3,098,708	2.08%	0.02%	0.39	0.09
Internet Explorer 11.64.16299.0	3,332,556	1.05%	0.04%	0.79	0.29

CPG: call-preceded gadgets (as a percentage of #Gadgets). **VCPG:** valid call-preceded gadgets. Averages are CPG = 1.94% and VCPG = 0.06%. **BS** binary size, in MBs, and **AS** apparent size, in KBs, after gadget reduction.

In practice, the reduction that our new constraint achieves makes it very difficult to build effective ROP attacks. We support this statement with a statistical argument first used by Schwartz et al. [2011]. Their tool, Q, usually requires binaries with at least 100KB to be able to execute calls to any function in libc. Similarly, it requires binaries of at least 20KB to invoke linked functions and update arbitrary locations. In other words, the number of gadgets expected to be found within 20KB of code is enough to carry out an attack. Our *enhanced Call-preceded Constraint* reduces the density of gadgets in the binary. Pragmatically, this is equivalent to reducing the size of the binary available to the attacker. Table 5.1 shows the size of the binary representation of each application. The last column of Table 5.1 shows the “apparent” size of those binaries, in terms of number of gadgets, that Q would see. Notice that this new size is given in KBs, not in MBs. The largest apparent size, found in Internet Explorer 8, is over 5x smaller than the minimum size that Q requires to perform a successful exploit.

5.2.1.2 Control-Flow Enforcement Technology

Reducing the number of gadgets found in the executable segment of protected applications is also one of the central objectives of protections against JOP and COP. For this purpose, as mentioned in Section 5.1.2, we advocate the use of CET in the second layer of our model. Thus, we quantify the protection afforded by CET via the following experiments:

- We count the number of unaligned (unintended) `ENDBR` instructions found in the text segment of the SPEC CPU2006 binaries.
- We compare the number of gadgets available in the SPEC CPU2006’s binaries when no protection is used with the number of these artifacts when CET’s indirect branch tracking mechanism is enabled.

On the Prevalence of Unaligned `ENDBR` Opcodes. Most of the gadgets used in real code reuse attacks against systems equipped with x86 processors correspond to unaligned accesses to the instruction segment. This claim is corroborated by examples of ROP, JOP and COP exploits publicly available in the Exploit Database [Offensive Security, 2020]. The possibility of executing unaligned instructions expands the amount of available gadgets. For this reason, the effectiveness of the strategy adopted by CET in blocking JOP and COP attacks depends on the volume of unaligned `ENDBR` instructions that can be found by accident amid the sequence of executable bytes of the applications.

We used the Ropper [Schirra, 2019] tool to scan the binary code of all benchmarks in the SPEC CPU2006 suite, after compiling them with GCC. The byte sequence `0xF30F1EFA`, the `ENDBR` instruction, was not found in any program. In fact, the probability of randomly finding this specific four-byte sequence is approximately 1 in 4 billion (256^4). In other words, this sequence of bytes is expected to be found once every 4 GB of instructions. This value is considerably less than the chance of finding misaligned `RET` (1 in 256) or indirect `JMP/CALL` (1 in 65536) instructions—branches exploited in attacks against unprotected binaries. The fact that there is no unintended `ENDBR` instruction in the SPEC programs implies that an attacker would have to hijack the flow of execution for some original indirect branch target. Although an attack of this nature is still feasible in specific scenarios [Carlini and Wagner, 2014; Davi et al., 2014; Goktas et al., 2014], several authors have already proposed protections based on the difficulty imposed by this restriction [Ruan et al., 2016].

On the Reduction of Available Gadgets. Considering that an attack can be built from the authentic targets of indirect branches, our second experiment quantifies the reduction in the amount of gadgets available in this case. Again, we used the Ropper utility to list the JOP and COP gadgets found in the binary files of SPEC CPU2006 applications. Ropper was also used to count the number of `ENDBR` instructions. This number gives us the quantity of gadgets available when CET is enabled. The columns labeled `#Gadgets` in Table 5.2 show the number of JOP and COP gadgets in each benchmark. The columns called `CET` indicate the percentage of valid gadgets when the mechanism to track indirect `JMPs` and `CALLs` is enabled. On average, the number of gadgets available fell to 25.48% of the initial volume, a reduction of almost 4 times.

Table 5.2. Percentage of gadget reduction in SPEC CPU2006 benchmarks.

Application	#Gadgets	CET (%)	BS (MB)	AS (KB)	Application	#Gadgets	CET (%)	BS (MB)	AS (KB)
xalancbmk	55383	40.06%	6.99	2,867.99	hammer	2060	21.55%	0.38	83.02
dealII	22169	55.97%	4.48	2,566.67	sphinx3	1040	29.04%	0.23	67.49
tonto	13654	27.29%	5.84	1,630.74	namd	675	15.85%	0.33	53.56
wrf	10154	27.77%	5.52	1,570.51	milc	869	27.04%	0.18	48.91
gobmk	7500	31.04%	4.50	1,430.43	zeusmp	718	11.00%	0.41	46.10
gamess	26584	10.75%	10.02	1,102.72	GemsFDTD	1139	5.27%	0.53	28.70
omnetpp	7460	69.16%	1.00	705.49	sjeng	1189	10.93%	0.19	21.06
calculix	7653	16.75%	1.97	338.49	astar	294	29.93%	0.06	18.85
gcc	56580	6.45%	4.15	273.95	libquantum	299	31.77%	0.06	18.40
cactusADM	4487	27.52%	0.95	267.63	leslie3d	217	9.22%	0.14	13.61
gromacs	4352	21.62%	1.17	260.00	bzip2	462	11.90%	0.11	12.85
povray	10251	14.27%	1.38	202.24	mcf	76	32.89%	0.02	7.34
perlbench	12712	12.35%	1.45	182.99	lbm	74	22.97%	0.02	4.96
soplex	4816	29.94%	0.58	178.00	bwaves	61	9.84%	0.04	4.08
h264ref	3124	16.26%	0.84	139.22	AVERAGE		25.48%		

Q binary size requirements to create a ROP exploit:

>= 100KB: execute calls to any function in libc

>= 20KB: invoke linked functions and update arbitrary locations

< 20KB: can't build ROP exploit

Again, to understand the impact of this reduction, we consider the estimates reported for Q [Schwartz et al., 2011]. We calculated the apparent size of the SPEC binaries to assess the impact that reductions in the number of gadgets (seen in Table 5.2) would have on Q's ability to build ROP exploits.

The BS (binary size) columns in Table 5.2 indicate the size of the binary files for each benchmark. In the same Figure, the AS columns (apparent size) show the

corresponding size of these binaries if we apply the observed reduction in the number of gadgets. BS values are represented in Megabytes (MB), while AS values are shown in Kilobytes (KB). Originally, Q would be able to produce exploits against all 29 benchmarks, with 24 of those exploits having full access to the libc library functions. Considering the reductions in the number of gadgets, Q would build exploits against 22 benchmarks, with 15 of these exploits having the ability to fully access libc. We grouped these 15 benchmarks (from `xalancbmk` to `h264ref`) on the left side of Table 5.2. On the top right side of this Figure are grouped the benchmarks whose exploits would have reduced functionality (from `hmmmer` to `sjeng`) and on the bottom right side the benchmarks for which Q would not be able to create exploits (from `astar` to `bwaves`). When analyzed from this perspective, it is clear the benefit of gadget reduction provided by the tracking of indirect Jumps and Calls proposed in CET.

5.2.2 Filtering statistics

The multilayer approach that we advocate is interesting if a large volume of authentic indirect branches is filtered across successive layers. In this section, we show that the flow of indirect branches to be validated is drastically reduced by going through Layers 1 and 2 of our model. We test two different implementations of Layer 2: in Section 5.2.2.2 we assess our enhanced Call-preceded Constraint (e-CC), and in Section 5.2.2.3 we evaluate Intel’s Control-flow Enforcement Technology (CET).

5.2.2.1 Layer 1: Indirect branch prediction

The Layer 1 of our model, described in Section 5.1.1, uses branch prediction to validate program flows. The greater the number of indirect branches that can be correctly predicted, the smaller the number of cases that reach Layer 2. In this section we shall count this hit rate.

Measurement methodology. To analyze the effectiveness of modern indirect branch predictors, we count the actual number of prediction misses. To this end, we use Hardware Performance Counters (HPCs) available in current x86 processors. In the experiments, we calculate the percentage of indirect branches correctly predicted by monitoring the number of misses and the total amount of indirect branches executed. x86 implements different prediction strategies for different categories of branches. Thus, we had to capture the performance counters for each type of indirect branch. We count the taken speculative and retired RETs, indirect CALLs, and indirect JMPs. We also measure the number of mispredictions for the same instructions. To produce

summarized results, all the average values presented were weighted on the number of instructions executed by the benchmarks: programs with more instructions contribute proportionally more towards the final average.

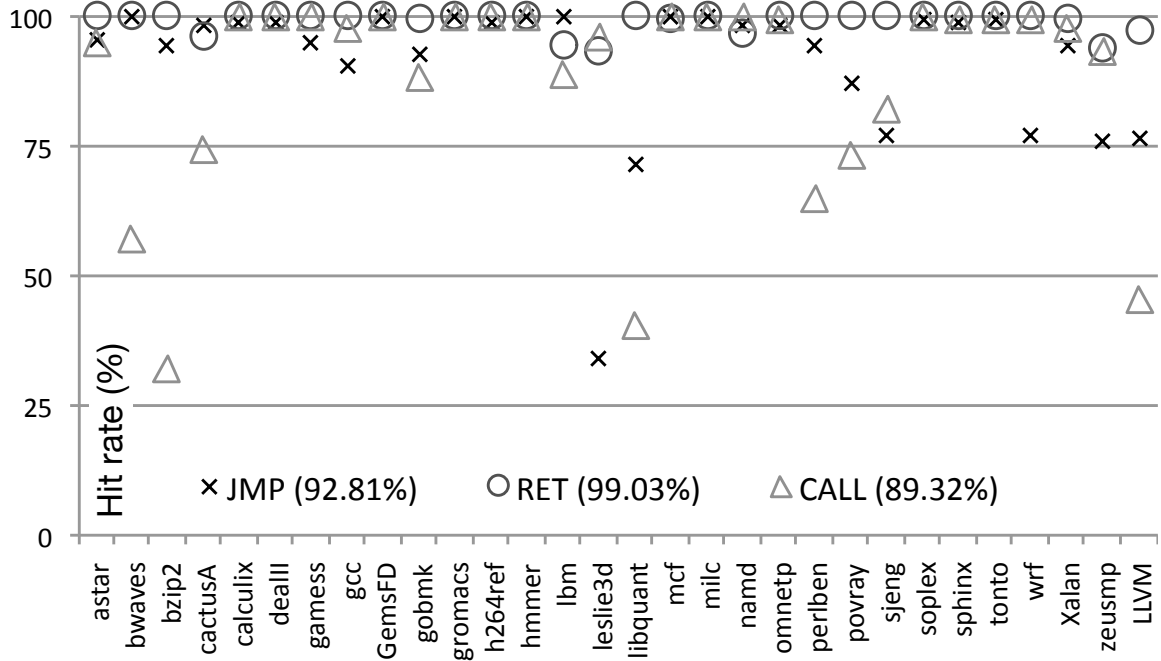


Figure 5.5. Hit rate (%) of the branch predictors for the programs in the SPEC CPU2006 and LLVM Test benchmark suites. Numbers next to labels show the weighted average accuracy of each predictor. The LLVM bar subsumes results of 209 programs in the LLVM test suite.

Analysis of Results. Figure 5.5 reports the accuracy of the branch predictor for the programs in the SPEC CPU2006 collection, and in the LLVM test suite. We have grouped the results for the LLVM test suite into a single bar, called LLVM. This procedure will be also used in the next charts that we present. The benchmark that executed the lowest number of instructions was `gobmk`, with $6.8\text{E}+10$ operations. The largest number of instructions was executed by `calculix`: $6.1\text{E}+12$. In total, we observed $4.8\text{E}+13$ instructions when evaluating SPEC CPU2006. The dynamic predictor succeeds in 97.72% of the indirect branches executed. This number is the average of all the averages, weighted by the number of instructions in each benchmark.

In the specific case of return addresses, the return address stack (RAS) achieved the impressive accuracy of 99.03%. Its lowest hit rate was 93.05%, recorded by the benchmark `leslie3d`, while the highest rate was 99.99%, recorded by the benchmark `h264ref`. 24 of the 30 (80%) results had a hit rate above 99%.

The prediction of the target address of indirect **JMP** instructions also reached an expressive average hit rate of 92.81%. The lowest hit rate observed for indirect **JMPs** was 34.06%, recorded by the benchmark `leslie3d`, while the highest rate was 99.98%, recorded by the benchmarks `lbm` and `bwaves`. 23 out of 30 (76.66%) results had a hit rate above 90%. These values demonstrate the high accuracy of the indirect **JMP** predictor and its consequent effectiveness in filtering authentic execution flows.

Albeit more modest, results observed for the prediction of indirect **CALLs** also show high accuracy. The average number of correct predictions reached 89.32%. The lowest hit rate observed for indirect **CALLs** was 32.07%, recorded by the benchmark `bzip2`, while the highest rate was 99.93%, recorded by the benchmark `dealII`. 20 of the 30 (66.66%) results had a hit rate above 90%.

These results corroborate the hypothesis that dynamic predictors of indirect branches are an effective way to filter out the cases to be monitored. If we consider all the instructions executed, on average one in 100,000 requires action of Layer 2. This is the main reason for the low overhead imposed by our model, as we shall discuss in Section 5.2.3. Moreover, it is already known that the branch prediction mechanism does not cause execution time impacts, since it currently works in parallel with the instructions pipeline.

5.2.2.2 Layer 2: enhanced Call-preceded Constraint

Our Layer 2 applies the *enhanced Call-preceded Constraint (e-CC)* described in Section 5.1.2.1 to validate **RETs** that are mispredicted. This layer is effective as long as most of the return addresses are preceded by valid **CALL** instructions. In this section we quantify them.

Measurement methodology. We use the Pintool prototype described in Section 5.1.2.1 to simulate the *e-CC*. We also simulate the *Instruction Translation Lookaside Buffer* (iTLB). The idea is to check the execution permission of the target of a direct **CALL** instruction (our *Executable Target Constraint*) in the same frequency that a page walk actually happens on a real processor. Using this strategy, we establish an upper bound that better approximates the execution time of a solution implemented in hardware. We have used hardware performance counters to estimate the iTLB miss rate observed when executing SPEC CPU2006 benchmarks. We use this estimate, plus a counter in the Pintool, to mimic the hardware behavior adjusting the frequency that permission checks should actually execute. Our machine's iTLB is parameterized in the following way:

- L1 instruction TLB for 2M/4M pages is fully associative with 8 entries.
- L1 instruction TLB for 4K pages is 4-way associative with 64 entries.
- L2 TLB (instructions and data) for 4K pages is 4-way associative with 512 entries.

Analysis of Results. Table 5.3 displays the results of this experiment, using the same applications seen in Table 5.1. The upper rows in Table 5.3 show cases where return addresses are properly validated (preceded by a valid `CALL`). Lower rows indicate the percentage of cases that will be submitted to a new analysis at Layer 3. As in the prediction of return addresses, our validation of these targets presents a very high percentage of success (96.86%). Checking the `CALLs` that precede the return address adds few cases to the subsequent layer (3.14%). Therefore, considering the benefit provided by this strategy with the reduction in the number of gadgets, the validation of return addresses proposed in this work is worth it.

Table 5.3. Validations that slip from Layer 2 to Layer 3.

return addresses preceded by Valid <code>CALLs</code>	Direct	69.84%	96.86%
	Indirect	27.01%	
return addresses preceded by Invalid <code>CALLs</code>	Direct	0.17%	3.14%
	Indirect	1.54%	
return addresses not preceded by <code>CALLs</code>		1.44%	

96.86% of return addresses are validated at Layer 2. 3.14% will slip to Layer 3.

5.2.2.3 Layer 2: Control-Flow Enforcement Technology

As mentioned, our multilayer approach yields a modular design. Different implementations are possible, for instance, at the level of Layer 2. One such possible implementation is based on the Control-flow Enforcement Technology (CET) from Intel. We have experimented with a CET-like implementation of a shadow stack to protect RETs. This section reports our findings. Since there is not yet an environment with Operating System and libraries compatible with CET’s Indirect Branch Tracking (IBT), we could not experiment with it to assess how many cases will slip from Layer 2 to Layer 3 while validating indirect `JMPs` and `CALLs`. This study is part of our future work list, discussed in Section 6.1.

Measurement Methodology. We have implemented a CET-like validation of `RET` instructions, following public knowledge [Intel, 2019]. A shadow stack was implemented using Pin. Like CET, our Pintool creates a stack in memory and instruments every `CALL` and `RET` executed by the application. Whenever a `CALL` is processed, our prototype pushes the address of the next instruction into the shadow stack. When a `RET` executes, we check if the return address matches the address in the top of the shadow stack. Like CET, our implementation is not affected by context switches. We also create one shadow stack for each thread of the application and use Pin locking system to avoid interferences.

Analysis of Results. Using CET’s shadow stack, we observed that 1.44% of the branches slip from Layer 2 to Layer 3. Our enhanced Call-preceded Constraint, as seen in Table 5.3, gives 3.44%. As recently pointed by Qiu et al. [2018], a CET-like shadow stack has more cases that yield false positives than those we have enumerated for our e-CC. Nevertheless, in our experiments, CET has been able to validate more `RET` instructions. We speculate that this result is due to the fact that Pin does not instrument privileged code and most of the known cases of false positives are related to system calls. This said, our results indicate that both strategies produce a comparably low false positive rate. On the other hand, a shadow stack is more costly, from an implementation perspective, as we discuss next.

A shadow stack has three shortcomings, which do not exist in our strategy. First, e-CC has lower impact on the instruction pipeline since validations happen in parallel with instruction processing. To manipulate the shadow stack, on the other hand, CET adds work onto `CALL` and `RET` instructions. It also adds work onto any `store` operations. Such instructions must be verified to prevent them from tampering with the shadow stack, as pointed out by Sinha et al. [2016]. Second, e-CC only accesses memory when executing `RET` instructions; and only due to iTLB misses, to check if the destination of a `CALL` is executable. In our experiments with SPEC CPU2006 benchmarks, the iTLB miss rate was only 0.005%. In CET, the shadow stack is implemented in memory to prevent overflows. Therefore, each `CALL` and `RET` instruction executed require a memory access. Finally, our schema does not require the creation of mechanisms to protect storage structures, such as the RAS or the LBR, because they already exist in hardware and are inaccessible to the programmer. In the case of CET, since the shadow stack is located in memory, it is necessary to create and manage a protection flag against unauthorized writing [Intel, 2019]. This said, we emphasize that our multilayer approach is equally effective to reduce the overhead imposed on either CET or e-CC. This overhead is the subject of the next Section.

5.2.3 Overhead

Our first layer of protection can be implemented at zero-overhead, because it relies on hardware already in place in modern computer architectures. Therefore, in this section we focus on the overhead of Layers 2 and 3. Like in Section 5.2.2, we analyze two implementations of Layer 2, based on e-CC and CET.

5.2.3.1 Layer 2 Without Layer 1: enhanced Call-preceded Constraint

Our enhanced Call-preceded Constraint is implemented by the combination of two constraints, as explained in Section 5.1.2: the executable target (XT) for return addresses preceded by direct `CALLs` and the LBR matching for return addresses preceded by indirect `CALLs`. In this section we shall measure the impact of enforcing them.

Measurement Methodology. To estimate the overhead of the second layer, we will proceed in two steps. First, we shall approximate the cost of implementing the checks of Section 5.1.2 for every return instruction processed during the execution of a program. Then, in Section 5.2.3.3, we shall reduce this cost proportionally to the quantity of such instructions that slip from the first to the second layer of our model to calculate the overall overhead ($L1 + L2$). As benchmarks, we use SPEC and the LLVM test suite. Our prototype is implemented in Pin, which already imposes a heavy overhead on the code that it instruments. Thus, to estimate the overhead of Layer 2 when running at the hardware level, we need to discount Pin’s runtime cost. To carry out this discount, we also run our applications through Pin without the verifications used to implement Layer 2. Let $ExTime_{pruned}$ be our baseline time. This is the runtime of a simplified Pintool that does not have the verification code. This Pintool has only the LBR structure and the manipulation of counters. Let $ExTime_{complete}$ be the runtime of the Pintool with the implementation of Layer 2. This Pintool has every data structure used in the implementation of Layer 2 (LBR structure, verification of the instruction that precedes a return address, verification of a LBR match for indirect `CALLs`, calculation of targets for direct `CALLs`, verification of permissions of targets of direct `CALLs` and manipulation of counters). The estimated overhead is the difference $ExTime_{complete} - ExTime_{pruned}$.

Analysis of Results. Figure 5.6 shows the result of this analysis. The numbers in Figure 5.6 show a weighted average *upper bound* of 50.64% on the overhead that we can expect. We are emulating in software all the verifications that, in production, would be implemented in hardware. For instance, our Pintool produces sequences with

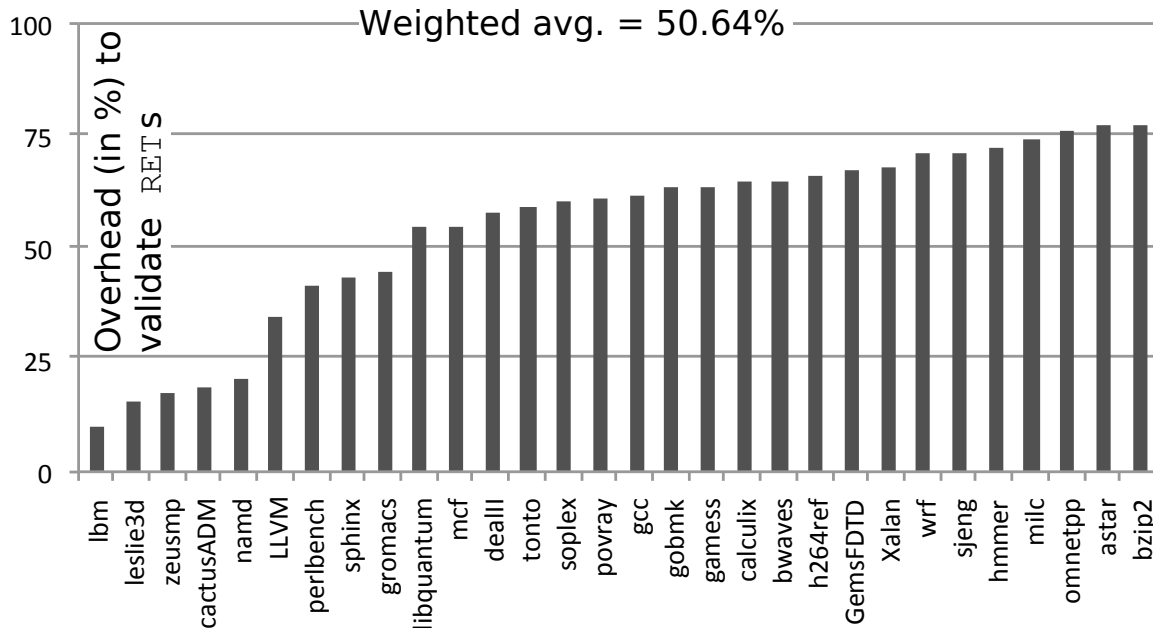


Figure 5.6. RET-validation overhead.

several operations to check if the instruction that precedes the target of a RET is a CALL. Even more instructions are issued by the Pintool to verify if the CALL targets an executable memory area because our prototype needs to track, through data-structures provided by Pin, which memory sections of a process image are executable. In hardware, these verifications would happen in parallel with the instruction pipeline; hence, we speculate that it would be orders of magnitude lower. Nevertheless, as we shall see in Section 5.2.3.4, this overhead shall fade away due to Layer 1 – this is the main contribution of our multilayer approach.

5.2.3.2 Layer 2 Without Layer 1: Control-Flow Enforcement Technology

We also measured the overhead of Layer 2 considering CET in-place. We executed two sets of experiments: one to evaluate CET’s shadow stack, responsible for protecting against ROP attacks, and the other to assess CET’s Indirect Branch Tracking (IBT), designed to block JOP and COP. These experiments simulate the infra-structure seen in Section 5.1.2.2.

Shadow Stack measurement methodology. Like in Section 5.2.3.1, we also discount Pin’s runtime cost to measure overhead. To this end, we created a pruned version of the Pintool. This baseline has all the code of the final prototype: shadow stack manipulation, counters operation, etc. However, it does not perform the compar-

ison between the return address and the address in the top of the shadow stack. We subtract its execution time from the value observed for the complete Pintool.

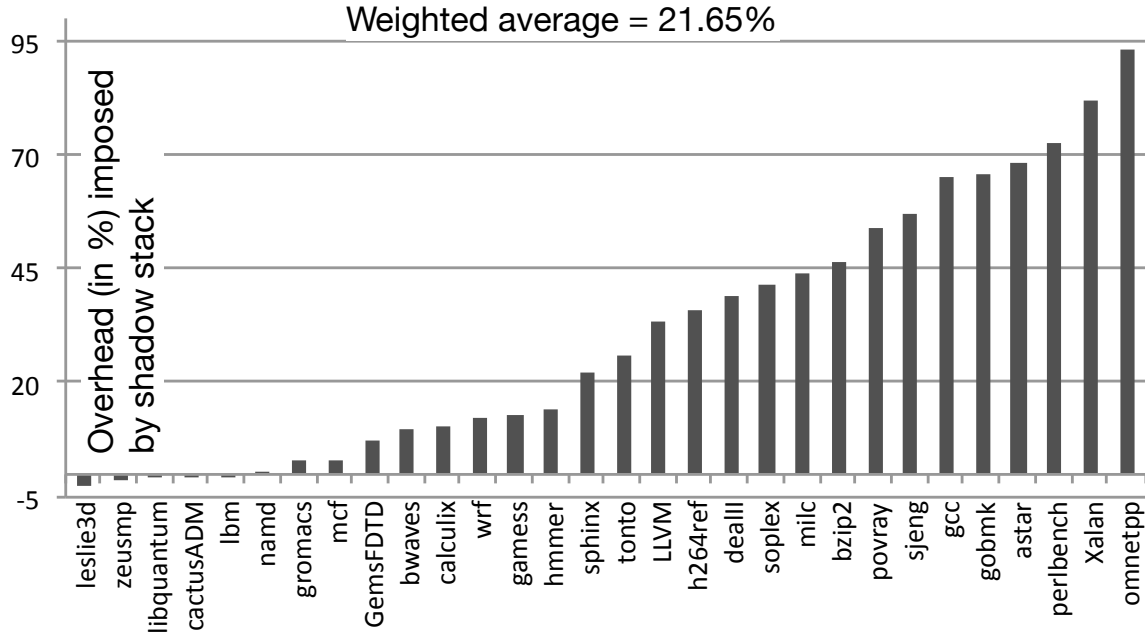


Figure 5.7. Percentage of overhead imposed by CET to validate RETs

Results for the Shadow Stack. Results are presented in Figure 5.7. Our shadow stack implementation slowed down the benchmarks in 21.65% on average. We cannot establish a rigid comparison between the overhead observed in the shadow stack and in our e-CC, because software implementations in Pin are considerably different than hardware implementations. However, these values indicate that both strategies present a comparable cost. When applied to a tiny portion of RETs, as shown in Section 5.2.2.1, both strategies will cause an insignificant overhead, as we show in the next Section.

IBT measurement methodology. Since some details of CET’s hardware implementation are unknown, we devised two experiments to assess the performance of the Indirect Branch Tracking mechanism proposed in that implementation: a test to establish a minimum value of inevitable overhead and another test to bound the maximum overhead. In both cases, we recorded the CPU time consumed by the programs using the hardware performance counter “cpu-clock:u”, made available by the “perf” utility. The execution time of each benchmark corresponds to the average value obtained in 10 repetitions.

Minimum overhead imposed by IBT. The minimum overhead scenario will occur if the CET-enabled hardware is able to implement the ENDBR verification without impacting the instruction pipeline. In this case, the runtime overhead will be limited to the effect of adding ENDBR instructions to the program. Despite functioning as a NOP operation, the simple search and decoding of ENDBR can lead to delays in the flow of instructions through the pipeline. Delay happens because ENDBR instructions, if nothing else, occupy part of the hardware’s pipeline. To assess the impact of inserting the ENDBR instructions, we explored the fact that GCC already generates code compatible with CET, through the “fcf-protection” option. In addition, these programs compiled with CET support are compatible with current processors, as ENDBR instructions also correspond to NOP instructions in these models. Therefore, we compile the benchmarks with and without CET support and subtract the runtime values for the two versions to estimate CET’s minimum overhead. The results are expressed in the dark bars in Figure 5.8.

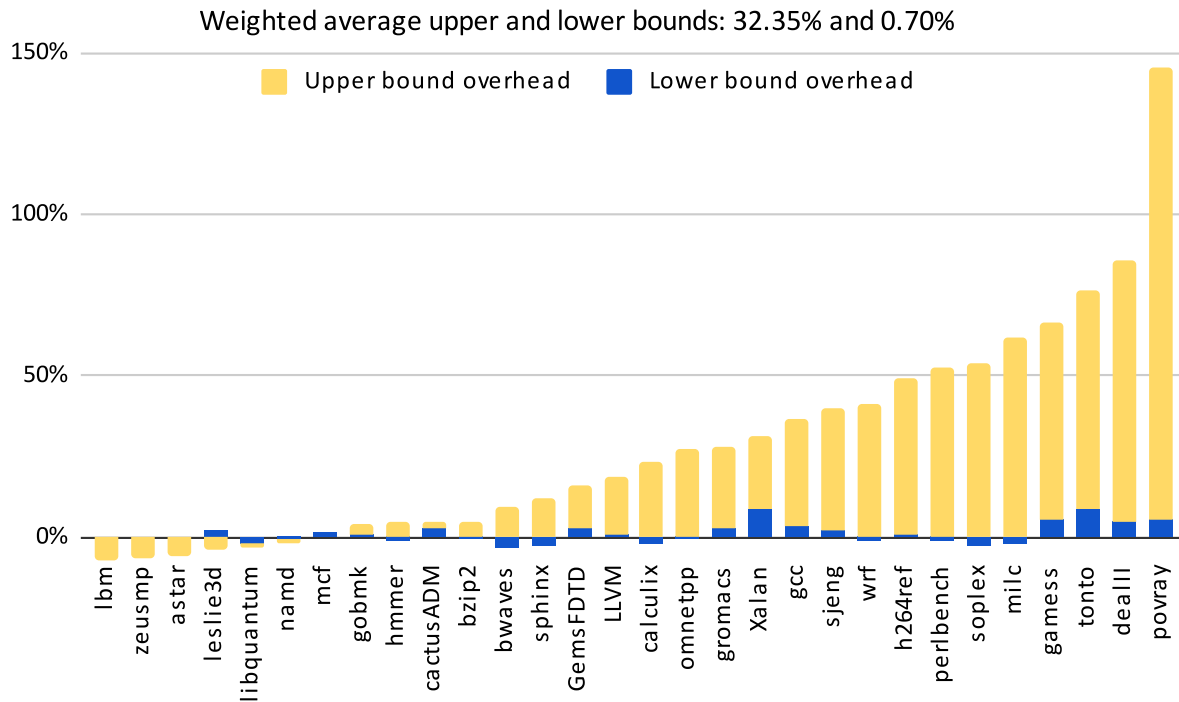


Figure 5.8. Minimum percentage of overhead imposed by CET to validate indirect JMPs and CALLs.

For most benchmarks, the extra ENDBR instructions have imperceptible impact on CPU time. The average overhead was only 0.70%. The values that yield this average range from -3.54%, observed in *bwaves*, to 8.77%, observed in *xalan*. Despite being counter-intuitive, the insertion of NOP instructions in the code can improve the

performance of applications in certain situations, as already demonstrated [Hundt et al., 2011]. This improvement happens because the inclusion of instructions changes the alignment of the code and this realignment can influence positively branch prediction and cache access.

We emphasize that this experiment does not account for the impact of adding ENDBR instructions to library code. The work of making shared libraries compatible with CET is a long process, which is progressing gradually until the release of the first CET-compatible processors. Thus, there exist no CET-compatible libraries available as of today. Despite this unavailability, with the help of Pin, we tracked the number of executed indirect `JMPs` and `CALLs` that did not have a corresponding `ENDBR` instruction. We found that, on average, 15.86% of these branches are executed in libraries. If we consider an impact of the same proportion in the execution time of the benchmarks, then we can expect an increase in runtime of 0.80%. To understand this rather rough estimate, notice that, on average, the branches in user code increase runtime by 0.70%. Extending this impact onto the 15.86% of branches in library code gives us $0.7 + (0.7 \times 0.158) = 0.8106$.

Maximum overhead imposed by IBT . We executed another experiment to establish an upper limit for the overhead imposed by the Indirect Branch Tracking mechanism proposed in CET. We opted to perform the `ENDBR` verification through instructions added to the program. We believe that this is a conservative upper bound, since the CET project aims to implement this check in hardware. We used Pin to instrument the code of the benchmarks at run time. Again, we measured the impact of Pin in the handling of each benchmark and discounted these values from the final result, shown in the light bars of Figure 5.8. The average overhead recorded corresponds to 32.35% of the execution time. This result approaches the numbers reported for similar protection mechanisms Hiser et al. [2012]; Abadi et al. [2005]; De Clercq et al. [2017]; Jiang et al. [2011]. Our result also reinforces the importance of designing a hardware solution and explains the expectation that was created around the CET project. However, in Section 5.2.3.3 we demonstrate that even such a costly solution becomes viable when incorporated into our multilayer system.

5.2.3.3 Layer 2 and Layer 1

The overhead presented thus far, be it due to our enhanced Call-preceded Constraint, be it due to Control-flow Enforcement Technology is high. However, only mispredicted indirect branches need to be verified at Layer 2. In this section we estimate this new

overhead, to conclude that Layer 1 brings a fundamental improvement onto any of these two validation mechanisms.

Measurement Methodology. We use the following equation to calculate the total overhead imposed on each benchmark by our multilayer system:

$$Overhead_{Multilayer} = Missrate_{L1} \times Overhead_{L2}$$

$Missrate_{L1}$ equals the percentage of failed predictions for each indirect branch type, as discussed in Section 5.2.2.1. $Overhead_{L2}$ corresponds to the percentage of overhead recorded to validate each type of indirect branch, as presented in Sections 5.2.3.1 and 5.2.3.2.

Analysis of Results. Figure 5.9 presents the results for the multilayer system based on our enhanced Call-preceded Constraint. Most of the benchmarks present an overhead close to 0%. The weighted average overhead is 0.57%.

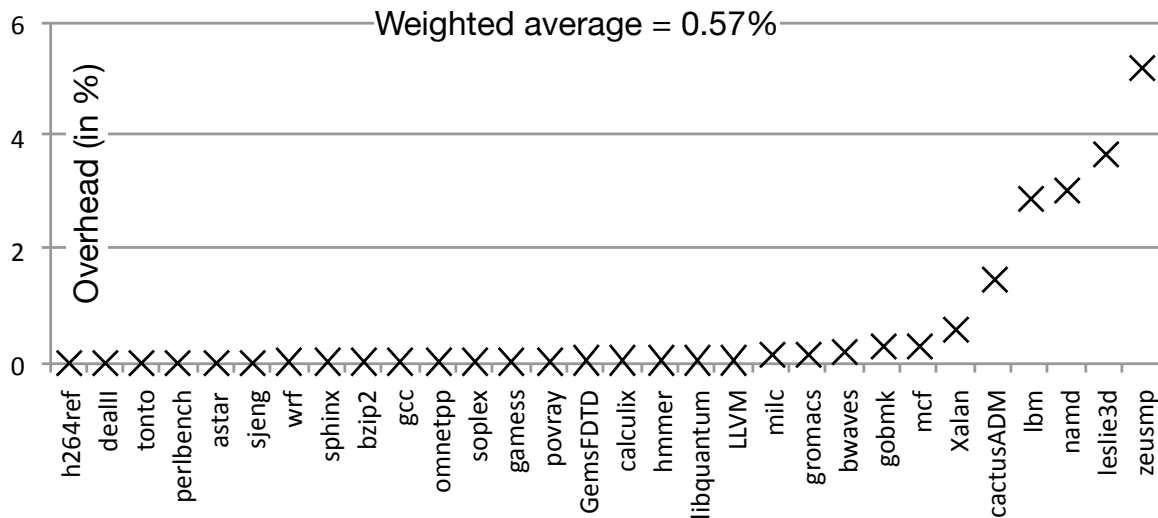


Figure 5.9. Cumulative overhead of first two layers, with e-CC.

The cumulative overhead of a CET-like shadow stack is even lower, due to the lower overhead of our shadow stack implementation (Section 5.2.3.2). Figure 5.10 shows this result. The weighted average in this case is just 0.02%. The numbers outlined in Figure 5.9 and 5.10 reinforces our thesis that a multilayer system allows the combination of costly anti-ROP protections. Notice that our overhead results are

an upper bound limit: they are based in software simulations of hardware solutions. Therefore, we can expect much lower values in a hardware implementation.

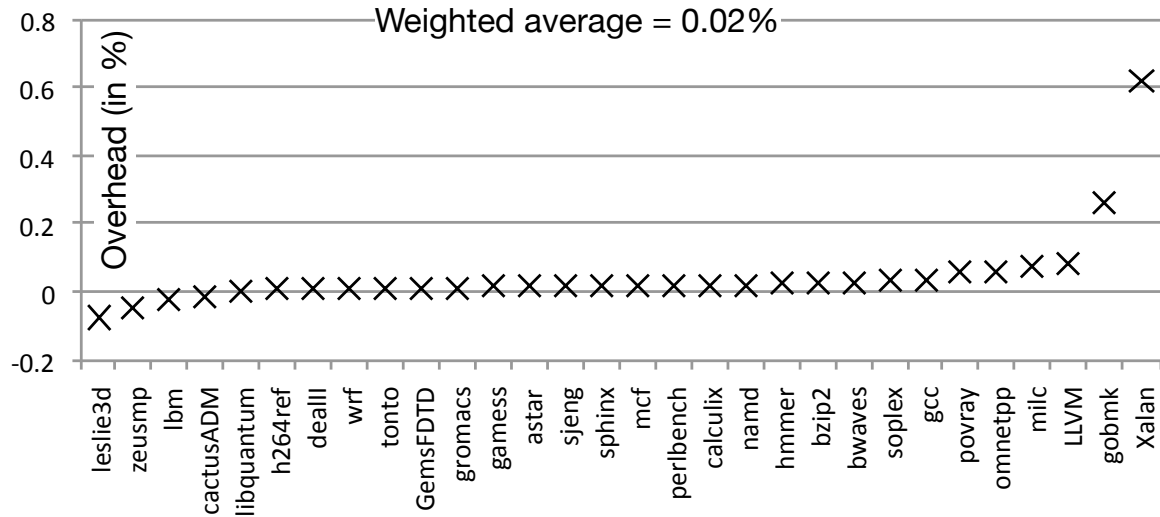


Figure 5.10. Cumulative overhead of first two layers, with shadow stack.

As shown in Section 5.2.3.2, the upper bound for the cost of verifying ENDBR instructions is high. However, in our multilayer system only indirect Jumps and Calls incorrectly predicted need to go through this check. This reduction in the volume of instructions to be verified brings down both the upper and the lower limits of the runtime overhead that CET’s IBT forces onto the protected system.

Figure 5.11 shows the reduction that the multilayer system achieves on the upper bound of execution time. The light bars correspond to the overhead recorded for checking all indirect Jumps and Calls. These values are identical to the upper bound overhead results seen in Figure 5.8. The dark bars report the overhead forced on each benchmark by applying the ENDBR instruction check only on the indirect Jumps and Calls that were incorrectly predicted.

Without the multilayer system, 11 out of 30 benchmarks (36.67%) have experimented an overhead of 5% or less. In Figure 5.11, these are the benchmarks ranging from `lbm` to `bzip2`. Taking into account the reduction provided by the multilayer system, this number raises to 23 out of 30 benchmarks (76.67%). The average overhead without the multilayer system, which was 32.35%, plummeted to 2.29% with it. The lowest overhead was observed in `astar` (-0.37%); the highest was still seen in `povray` (30.11%). The prominent overhead presented by `povray` results from a combination of two factors: a high rate of indirect Jumps and Calls executed, plus a below-average performance in the prediction of the targets of these instructions. Among the 30

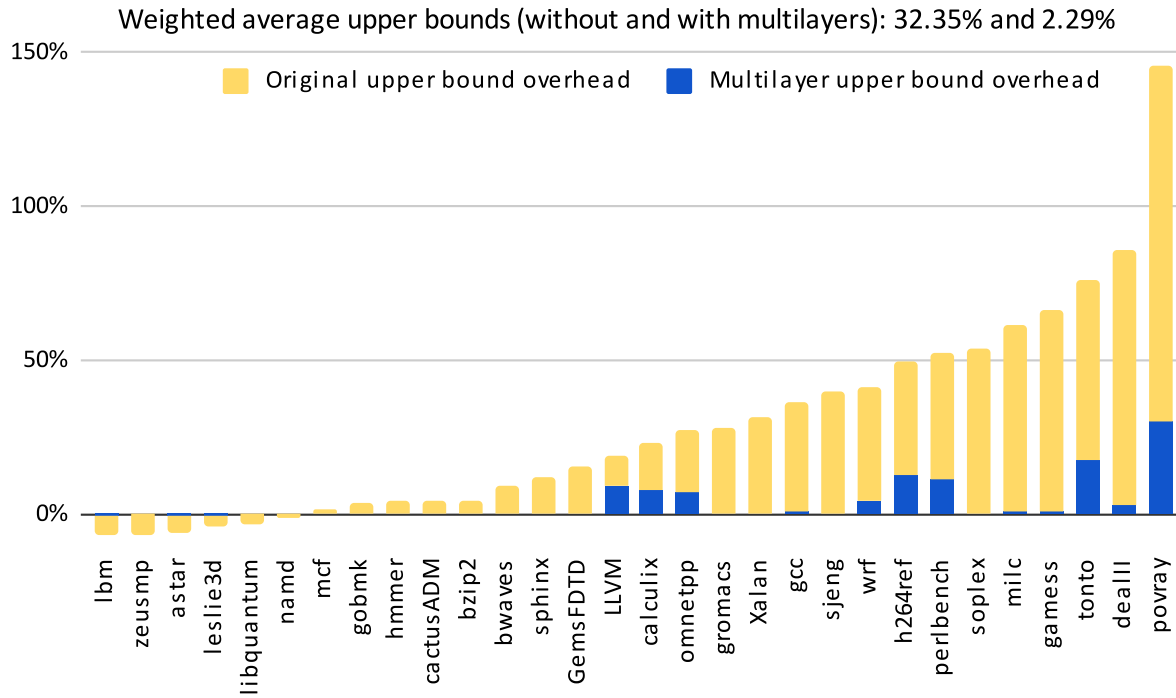


Figure 5.11. Percentage of CET’s IBT overhead reduction (upper bound) due to the multilayer protection system.

benchmarks, `povray` is the fifth with the highest percentage of indirect `JMPs` and `CALLs` executed: 18.32% versus an average of 7.41%. `povray` also has the sixth worst result in predicting the target of indirect `CALLs` (72.87% versus 89.32% of the mean) and the seventh worst prediction for indirect `JMPs` (87.10% versus 92.81% of the mean).

Figure 5.12 shows how much the multilayer system reduces the lower bound of CET’s IBT runtime overhead. Although this minimum cost is already small for most of the benchmarks, as seen in the light bars of Figure 5.12, further reduction is possible. Corroborating this observation, the multilayer system makes CET’s IBT runtime overhead in all the benchmarks negligible, as illustrated in the dark bars of Figure 5.12. The average runtime penalty dropped from 0.70% to 0.05%. The lowest value was registered for `calculix` (-0.56%); the highest, for `tonto` (2.02%). The results presented in Figures 5.11 and 5.12 reinforce our thesis that a multilayer system also makes practical the use of costly protections against JOP and COP attacks.

5.2.3.4 Layer 3

As mentioned in Section 5.1.3, false positives can be avoided either at the hardware level, or at the compiler level. If Layer 2 is implemented via e-CC (Section 5.1.2.1), then a solution at the compilation level is possible. In this Section we show that this solution

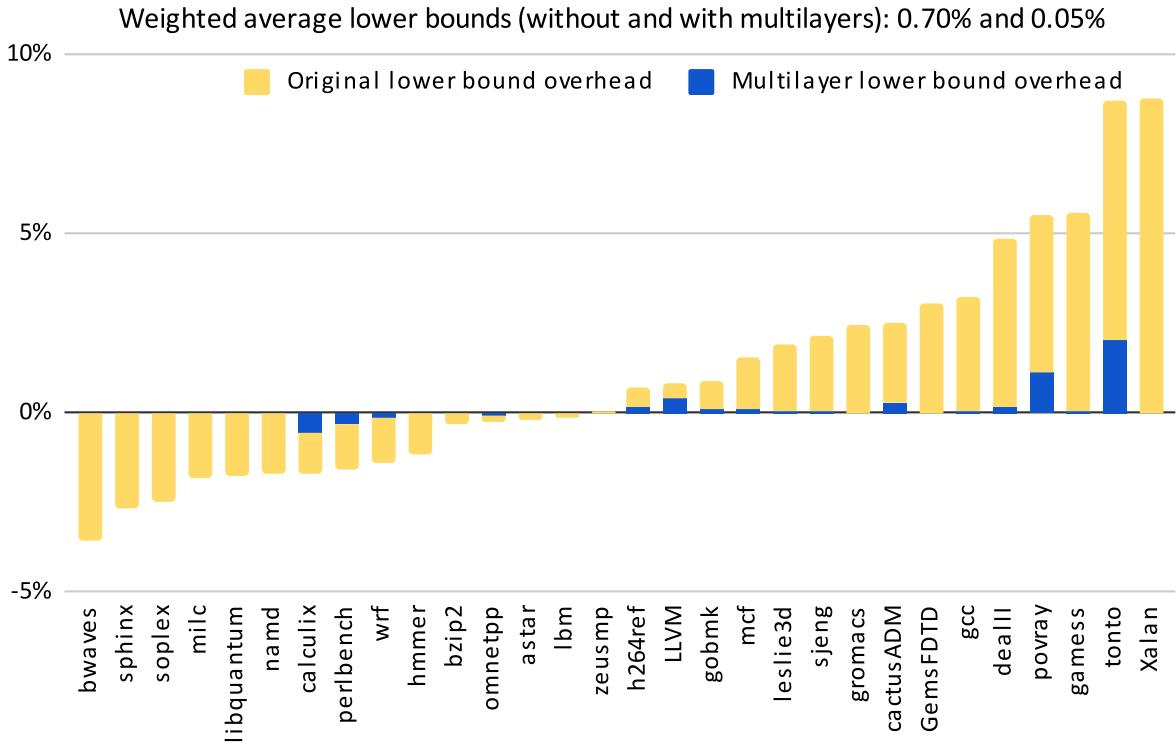


Figure 5.12. Percentage of CET’s IBT overhead reduction (lower bound) due to the multilayer protection system.

doesn’t incur any perceptible overhead. On the other hand, CET’s shadow stack based implementation of Layer 2 requires hardware interventions in Layer 3. At the end of this Section, we discuss the overhead of the hardware-based solutions provided in CET’s specification to treat those cases. As explained in Section 5.1.3, CET’s IBT avoids false positives by disabling the protection for incompatible software. Therefore, it doesn’t impose any additional overhead, despite reducing the system security.

Measurement methodology. As discussed in Section 5.1.3, a compiler-based solution to eliminate e-CC’s false positives consists in replacing the `RET` instruction by a sequence of `POP reg` and `JMP reg`. This replacement is necessary only for functions that might be invoked by means other than the execution of a `CALL` instruction. To estimate the overhead of this replacement, we have performed it manually onto every function of the twelve programs of the Shootout collection, available in the LLVM test suite². We had to perform 52 replacements. Tail call optimization was not allowed.

²Shootout contains the smallest benchmarks in the LLVM test suite that are not unit tests. We chose small benchmarks because we had to perform the instruction replacement manually, directly in the assembly representation. The sole difficulty in this task is finding a free register `reg`.

Figure 5.13 shows an example of code replacement.

(a) <code>void rec(int i)</code> <code>{</code> <code> if(i>0)</code> <code> {</code> <code> rec(i-1);</code> <code> }</code> <code> return;</code> <code>}</code>	(b) <code>rec_ret:</code> <code>.LFB0:</code> <code> cmp [esp+4], 0</code> <code> jle .L4</code> <code> mov eax, [esp+4]</code> <code> sub eax, 1</code> <code> push eax</code> <code> call rec_ret</code> <code> add esp, 4</code> <code>.L4:</code> <code> ret</code>	(c) <code>rec_pop_jump:</code> <code>.LFB0:</code> <code> cmp [esp+4], 0</code> <code> jle .L4</code> <code> mov eax, [esp+4]</code> <code> sub eax, 1</code> <code> push eax</code> <code> call rec_pop_jump</code> <code> add esp, 4</code> <code>.L4:</code> <code> pop edx</code> <code> jmp edx</code>
--	---	---

Figure 5.13. To carry out the experiment described in this Section, we compile a program such as (a) into its assembly version (b). Then, we replace, manually, occurrences of `RET` with the two-instruction sequence seen in (c).

Analysis of Results. Figure 5.14 shows the result of this experiment. All the benchmarks run for more than one second; Hash (`hsh`) runs for 35 seconds – the longest runtime. Every benchmark performs thousands of function calls. Each box shows the runtime of the modified program divided by the runtime of the original program. The t-test applied on the two populations has never given us a p-score lower than 0.1. Thus, it is statistically difficult to distinguish original and modified binaries by just observing their runtimes.

Hardware It is not straightforward to estimate the overhead of false positive detection, at the hardware level, when used in tandem with our system. Considering the filtering statistics of Layers 1 and 2 combined (Section 5.2.2), we can expect that the third layer will be activated for less than 0.03% of the indirect branches in a program. Thus, verification à la CFIMon does not incur a cost that we can measure reliably. Nevertheless, we can rely on previous work to put an upper bound on the expected overhead of a hardware implementation of Layer 3. CFIMon authors claim an overhead of about 6.1% to execute all their solution and point out that 86% of this cost is due only to the code for recording executed branches, carried out through their “pure BTS” implementation. Therefore, the rest of CFIMon, including the tracking code and the treatment of false positives, accounts for an overhead of no more than 0.9%. Notice that every branch target is sanitized in CFIMon. In our case, this overhead will be imposed upon a negligible number of indirect branches, as we explained in Section 5.2.2.

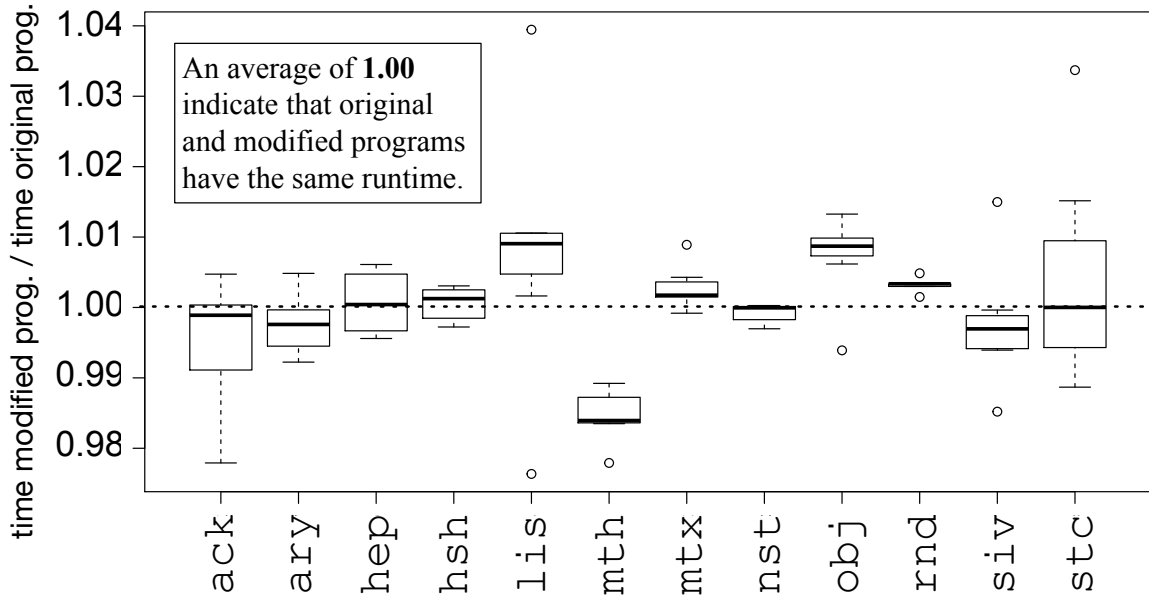


Figure 5.14. Execution overhead of replacing RET with the POP and JMP sequence in programs of the LLVM Shootout suite. Each box contains eight executions. The higher the average, the higher the overhead.

Therefore, we expect a much lower cost. More importantly, this verification does not impose any overhead upon the branches that do not reach the third layer.

5.3 Chapter Summary

This Chapter has presented a multilayer approach to hinder Return-Oriented Programming based attacks. Each layer of the proposed system validates targets of indirect branches; hence, proving that they belong to legitimate program flows. Our key insight is to combine layers, so that layer L_i runs at a lower computational cost than L_{i+1} . A branch certified at L_i does not need to be checked at L_{i+1} . We apply stronger enforcement guarantees only onto cases that are hard to verify, because L_i , by construction, receives more branches than L_{i+1} . In the next chapter, we discuss future work, in addition to presenting the conclusion of this thesis.

Chapter 6

Final Remarks

In this thesis we investigated the following scientific hypothesis:

- A multilayer system can guarantee a high level of protection against ROP based attacks at a negligible computational cost.

Protection strategies were developed, prototypes were implemented and several experiments have been performed. The results obtained corroborate the hypotheses raised. The efficiency of our multilayer system is achieved because our model inverts the logic of control flow validation. Rather than focusing on the detection of malicious executions, our system prioritizes the early identification of authentic execution flows. By doing this filtering in stages, our system allows authentic flows to be identified in the superficial layers of the model, which apply cheap checks by construction. More costly checks are designed to take place in deeper layers; thus, they tend to be rarely activated. Our multilayer system has also been designed to reuse hardware structures already available in modern processors, such as indirect branch prediction units, executable memory pages distinction and LBR. Therefore, we believe that it can easily be incorporated into the design of new hardware mechanisms aimed at software security—CET being one of them.

A secondary contribution that arose from our work to defeat ROP based attacks was the demonstration that it is possible to identify the maximum density of indirect branches achievable by an application through the static analysis of its code. Even though it is a theoretically undecidable problem, we have shown that our method is practical, scaling up to programs with more than 600 thousand assembly instructions. It also decreases significantly the number of gadgets that are available for the construction of an exploit when a protection that controls the frequency of indirect branches executed is used.

6.1 Future Work

Our flexible multilayer model is a work that can be deepened. Below we list some works that we intend to develop in the future to complement our results:

- As presented in Section 5.2.2.3, the number of false positives recorded is limited to cases detectable in user-level codes. Privileged instructions, such as Operating System code, were not computed. We wish to account for these cases since the main potential generators of false positives are in privileged code. To count these false positives, we intend to use Precise Event-Based Sampling (PEBS). PEBS is a special counting mode where Performance Monitoring Counters (PMCs) can be configured to trigger on PMC overflow (every mispredicted RET, for instance) and interrupt the processor. Linux and Windows use Non-Maskable Interrupts (NMIs) to deliver such interrupts and we can create NMI drivers with our own handler for a particular event. This procedure has already been used to improve branch monitoring, for example. [Botacin et al., 2018]. We can use it to implement layers 2 (implemented in Pin) and 3 (false positives filtering). Through these implementations it is possible to both count the volume of false positives that occur in a privileged mode and evaluate the overhead related to the treatment of these cases. The main advantage of using this approach instead of a hardware simulator is to obtain results from a real system where any proprietary optimizations from the processor manufacturers are already naturally present. As we mentioned in Section 5.2, many of these optimizations are not public and therefore there is no way to precisely simulate them.
- Since there is not yet an environment with Operating System and libraries compatible with CET, we could not experiment with CET’s Indirect Branch Tracking (IBT) to assess how many cases will slip from Layer 2 to Layer 3 while validating indirect Jumps and CALLs. When a hardware implementation of CET is available, we intend to use PEBS to assess the number of false positives and the overhead of the exception handling incorporated into CET, similar to what was mentioned in the previous item.
- We also want to study which strategies could be combined in a multilayer system to protect against other types of code reuse attacks (ret-to-libc, JIT-spraying, etc), keeping the principles defined in this work, which allow efficient execution of protected programs. The construction of attacks that circumvent our multilayer protection is also an open problem that we hope to explore in the future.

6.2 Conclusion

The recent history has shown that, thus far, there is no foolproof method to prevent ROP-based exploits. More than that, recent developments have shown that there exists no system that is able to stop any ROP attack [Carlini et al., 2015; van der Veen et al., 2017]. Our multilayer system is not an exception to this rule: it still may face false positives and false negatives. False positives happen if we flag a legitimate branch target as unsafe. The way in which the operating system treats this kind of exception, be it through preemptive termination, be it through sandboxing, impacts user experience. The experiments seen in Section 5.2.2 indicate that exceptions raised by authentic program flows are unlikely events, since our system has a layer specifically designed to prevent such occurrences.

False negatives happen if we allow an attacker to bend the control flow of the protected application. Through a statistical argument, we have shown in Section 5.2.1 that our enhanced Call-preceded Constraint leaves a very small number of gadgets available for the construction of an attack. Consequently, state-of-the-art exploits, such as those carried out by Carlini et al. [2015] are not possible. The construction of attacks that circumvent our protection is, therefore, an open problem.

Bibliography

- Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2005). Control-flow integrity. In *CCS*, pages 340–353, New York, NY, USA. ACM.
- Ahn, D. and Lee, G. (2015). A memory-access validation scheme against payload injection attacks. *IEEE Transactions on Dependable and Secure Computing*, 12(4):387–399.
- Akram, A. and Sawalha, L. (2016). x86 computer architecture simulators: a comparative study. In *IEEE International Conference on Computer Design*.
- Bacon, D. F. and Sweeney, P. F. (1996). Fast static analysis of C++ virtual function calls. In *OOPSLA*, pages 324–341. ACM.
- Bania, P. (2010). Security mitigations for return-oriented programming attacks. Technical report, Kryptos Logic Research.
- Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46.
- Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2).
- Blake (2011). MY MP3 Player 3.0 m3u Exploit DEP Bypass. <http://www.exploit-db.com/exploits/17854/>.
- Blazakis, D. (2010). Interpreter exploitation. In *WOOT*.
- Bletsch, T., Jiang, X., and Freeh, V. (2011). Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362. ACM.

- Botacin, M., Geus, P. L. D., and Grégio, A. (2018). Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Transactions on Privacy and Security (TOPS)*, 21(1):1–30.
- Bounov, D., Kici, R. G., and Lerner, S. (2016). Protecting c++ dynamic dispatch through vtable interleaving. In *NDSS*.
- Bowne, S. (2014). Defeating DEP with ROP. <https://samsclass.info/127/proj/rop.htm>.
- Bruening, D. L. (2004). *Efficient, transparent, and comprehensive runtime code manipulation*. PhD dissertation, Massachusetts Institute of Technology.
- Buchanan, E., Roemer, R., Shacham, H., and Savage, S. (2008). When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM.
- Buck, B. and Hollingsworth, J. K. (2000). An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329.
- Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. (2017). Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys*, 50(1).
- Cantrill, B., Shapiro, M. W., Leventhal, A. H., et al. (2004). Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28.
- Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. (2015). Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium*, pages 161–176. USENIX.
- Carlini, N. and Wagner, D. (2014). ROP is still dangerous: Breaking modern defenses. In *Security Symposium*, pages 385–399. USENIX.
- Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *CCS*, pages 1–14. ACM.
- Checkoway, S., Feldman, A. J., Kantor, B., Halderman, J. A., Felten, E. W., and Shacham, H. (2009). Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage. *EVT/WOTE*, 2009.

- Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). DROP: Detecting return-oriented programming malicious code. In *ISS*, pages 163--177. IEEE.
- Chen, P., Xing, X., Han, H., Mao, B., and Xie, L. (2011a). Efficient detection of return-oriented programming malicious code. In *ICISS*, pages 140--155. Springer.
- Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., and Yin, X. (2011b). Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 20--29. ACM.
- Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., and Yin, X. (2011c). Automatic construction of jump-oriented programming shellcode (on the x86). In *ASIACCS*, page 20. ACM.
- Cheng, Y., Zhou, Z., and Yu, M. (2014). ROPecker: A generic and practical approach for defending against ROP attacks. *NDSS*.
- Chiueh, T.-c. and Hsu, F.-H. (2001). RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, pages 409--417. IEEE.
- Corbet, J. (2018). Kernel support for control-flow enforcement. <https://lwn.net/Articles/758245/>.
- Corelan Team (2011). mona.py - the manual. <https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/>.
- Criswell, J., Dautenhahn, N., and Adve, V. (2014). Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, pages 292--307. IEEE.
- CSO (2012). Microsoft BlueHat Prize Winners. <https://www.csoononline.com/article/2222848/microsoft-subnet/microsoft-bluehat-prize-winners.html>.
- Cugliari, A. and Graziano, M. (2010). Smashing the stack in 2010. <http://www.mgraziano.info/docs/stsi2010.pdf>.
- D. Brown, M. and Pande, S. (2019). Is less really more? why reducing code reuse gadget counts via software debloating doesn't necessarily lead to better security. *arXiv Cryptography and Security (cs.CR)*.
- Dai Zovi, D. (2010). Practical return-oriented programming. *SOURCE Boston*.

- Das, S., Chen, B., Chandramohan, M., Liu, Y., and Zhang, W. (2018). ROPSentry: Runtime defense against ROP attacks using hardware performance counters. *Computers & Security*, 73:374–388.
- Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., and Sadeghi, A.-R. (2011a). Poster: control-flow integrity for smartphones. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS)*, page 749. ACM Press.
- Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., and Sadeghi, A.-R. (2012). MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Network and Distributed System Security Symposium*.
- Davi, L., Hanreich, M., Paul, D., Sadeghi, A.-R., Koeberl, P., Sullivan, D., Arias, O., and Jin, Y. (2015). Hafix: Hardware-assisted flow integrity extension. In *ACM/ESDA/IEEE Design Automation Conference*.
- Davi, L., Sadeghi, A., and Winandy, M. (2009). Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *WSTC*, pages 49–54.
- Davi, L., Sadeghi, A.-R., Lehmann, D., and Monroe, F. (2014). Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*.
- Davi, L., Sadeghi, A.-R., and Winandy, M. (2011b). ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security*.
- De Clercq, R., Götzfried, J., Übler, D., Maene, P., and Verbauwhede, I. (2017). Sofia: software and control flow integrity architecture. *Computers & Security*, 68:16–35.
- de Melo, A. C. (2010). The new linux “perf” tools. <https://pdfs.semanticscholar.org/16ca/fd05fa375dfe370274cd22b4c16c72d6c53b.pdf>.
- Demott, J. (2012). /ROP - BlueHat Prize Submission. http://www.vdalabs.com/tools/DeMott_BlueHat_Submission.pdf.
- Ding, R., Qian, C., Song, C., Harris, B., Kim, T., and Lee, W. (2017). Efficient protection of path-sensitive control security. In *USENIX Security Symposium*.

- Dullien, T., Kornau, T., and Weinmann, R.-P. (2010). A Framework for Automated Architecture-Independent Gadget Search. In *WOOT*.
- Durden, T. (2002). Bypassing pax aslr protection. *Phrack Magazine*, 59(9).
- Eeckhoutte, P. V. (2011). Exploit writing tutorial part 11?: Heap spraying demystified. <https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/>.
- Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):6.
- El-Sherei, S. (2013). Return Oriented Programming (ROP FTW).
- Erlingsson, Ú. (2007). Low-level software security: Attacks and defenses. In *Foundations of security analysis and design IV*, pages 92--134. Springer.
- F-Secure Response Labs (2013). Threat Report. https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2013.pdf.
- Ferreira, M., Santos, A., and Feitosa, E. (2014). Controlando a frequência de desvios indiretos para bloquear ataques rop. In *Anais do XIV SBSeg*, pages 223--236, Porto Alegre, RS, Brazil. Sociedade Brasileira de Computação.
- Francillon, A. and Castelluccia, C. (2008). Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15--26. ACM.
- Fratric, I. (2012). Runtime Prevention of Return-Oriented Programming Attacks. <https://ropguard.googlecode.com/svn/trunk/doc/ropguard.pdf>.
- G0M3S, C. (2011). Free CD to MP3 Converter 3.1 Universal DEP Bypass Exploit. <http://www.exploit-db.com/exploits/15235/>.
- Gawlik, R. and Holz, T. (2014). Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference*.
- GCC (2020). Program instrumentation options. <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>.

- Goktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. (2014). Out of Control: Overcoming Control-Flow Integrity. In *S&P*, pages 575--589. IEEE.
- Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Security Symposium*, pages 417--432. USENIX.
- Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX*, pages 417--432.
- Guha, A., Hiser, J. D., Kumar, N., Yang, J., Zhao, M., Zhou, A., Childers, B. R., Davidson, J. W., Hazelwood, K., and Soffa, M. L. (2007). Virtual execution environments: Support and tools. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1--6. IEEE.
- Han, Y. H., Park, D. S., Jia, W., and Yeo, S. S. (2013). Detecting return oriented programming by examining positions of saved return addresses. In *LNEE*, pages 3:1--3:18. Springer.
- Hennessy, J. L. and Patterson, D. A. (1994). *Computer Architecture, A Quantitative Approach*. Elsevier, 6th edition.
- Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1--17.
- Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. (2012). Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571--585. IEEE.
- Hoglund, G. and McGraw, G. (2004). *Exploiting software: How to break code*. Pearson Education.
- Huang, Z., Zheng, T., Shi, Y., and Li, A. (2012a). A dynamic detection method against ROP and JOP. *ICSAI*, pages 1072--1077.
- Huang, Z. J., Zheng, T., and Liu, J. (2012b). A dynamic detective method against ROP attack on ARM platform. In *IWSEE*, pages 51--57.
- Hund, R., Holz, T., and Freiling, F. C. (2009). Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, pages 383--398.

- Hundt, R., Raman, E., Thuresson, M., and Vachharajani, N. (2011). Mao-an extensible micro-architectural optimizer. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 1--10. IEEE.
- Immunity (2020). Immunity Debugger. <https://www.immunityinc.com/products/debugger/>.
- Intel (2006). IA-32 Architecture Software Developer's Manual. Volume 2B: Instruction Set Reference, N-Z.
- Intel (2017). Control-flow enforcement technology preview. <https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>.
- Intel (2018). Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, 2C, and 2D: Instruction Set Reference, A-Z.
- Intel (2019). Control-flow enforcement technology specification. <https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>.
- Intel (2020). Intel C++ Compiler 19.1 Developer Guide and Reference. <https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-alphabetical-list-of-compiler-options>.
- Jang, D., Tatlock, Z., and Lerner, S. (2014). SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Network and Distributed System Security Symposium*.
- Jiang, J., Jia, X., Feng, D., Zhang, S., and Liu, P. (2011). Hypercrop: a hypervisor-based countermeasure for return oriented programming. In *International Conference on Information and Communications Security*, pages 360--373. Springer.
- Kayaalp, M. and Ozsoy, M. (2012). Efficiently Securing Systems from Code Reuse Attacks. *IEEE Transactions on Computers*, pages 1--14.
- Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., and Ponomarev, D. (2012). Branch regulation: Low-overhead protection from code reuse attacks. In *International Symposium on Computer Architecture*, pages 94--105. ACM.
- Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., and Abu-Ghazaleh, N. (2013). SCRAP: Architecture for signature-based protection from code reuse attacks. In *HPCA*, pages 258--269, Washington, DC, USA. IEEE.

- Kiriansky, V., Bruening, D., and Amarasinghe, S. P. (2002). Secure execution via program shepherding. In *Security Symposium*, pages 191--206. USENIX.
- Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X.-y., and Wang, X. (2009). Effective and efficient malware detection at the end host. In *USENIX security symposium*, pages 351--366.
- Kornau, T. (2010). *Return oriented programming for the ARM architecture*. PhD dissertation, Master's thesis, Ruhr-Universität Bochum.
- Koziol, J., Litchfield, D., Aitel, D., Anley, C., Eren, S. N., Mehta, N., and Hassell, R. (2004). *The shellcoder's handbook: discovering and exploiting security holes*. Wiley London.
- Krahmer, S. (2005). x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. <http://forum.ouah.org/no-nx.pdf>.
- Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75--88. IEEE.
- Laurenzano, M. A., Tikir, M. M., Carrington, L., and Snaveley, A. (2010). Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 175--183. IEEE.
- Le, L. (2010). Payload already inside: datafire-use for ROP exploits. *Black Hat USA*.
- Lee, Y. and Lee, G. (2018). Detecting code reuse attacks with branch prediction. *IEEE Computer*, 51(4):40--47.
- Li, J., Chen, L., Xu, Q., Tian, L., Shi, G., Chen, K., and Meng, D. (2019). Zipper stack: Shadow stacks without shadow.
- Li, J., Wang, Z., Jiang, X., Grace, M., and Bahram, S. (2010). Defeating return-oriented rootkits with return-less kernels. In *EuroSys*, pages 195--208. ACM.
- Lindner, F. (2009). Developments in cisco ios forensics. CONFidence 2.0.
- Liu, C., Yang, C., and Shen, Y. (2014). Leveraging microarchitectural side channel information to efficiently enhance program control flow integrity. In *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*.
- LLVM (2020). Clang command line argument reference. <https://clang.llvm.org/docs/ClangCommandLineReference.html>.

- Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200. ACM.
- Marpaung, J. A., Sain, M., and Lee, H.-J. (2012). Survey on malware evasion techniques: State of the art and challenges. In *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pages 744–749. IEEE.
- Matrosov, A., Rodionov, E., Harley, D., and Malcho, J. (2010). Stuxnet Under the Microscope. https://www.esetnod32.ru/company/viruslab/analytics/doc/Stuxnet_Under_the_Microscope.pdf.
- McDonald, J. (1999). Defeating solaris/sparc non-executable stack protection. <http://seclists.org/bugtraq/1999/Mar/4>.
- Meng, X. and Miller, B. P. (2016). Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35.
- Microsoft (2010). Windows ISV Software Security Defenses. [https://docs.microsoft.com/en-us/previous-versions/bb430720\(v=msdn.10\)](https://docs.microsoft.com/en-us/previous-versions/bb430720(v=msdn.10)).
- Microsoft (2011). An update is available for the ASLR feature in Windows 7 or in Windows Server 2008 R2. <https://support.microsoft.com/en-us/help/2639308/an-update-is-available-for-the-aslr-feature-in-windows-7-or-in-windows>.
- Microsoft (2012). /SAFESEH (Image has Safe Exception Handlers). [https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2012/9a89h429\(v=vs.110\)](https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2012/9a89h429(v=vs.110)).
- Microsoft (2018). How to determine that hardware DEP is available and configured on your computer. <https://support.microsoft.com/pt-br/help/912923/how-to-determine-that-hardware-dep-is-available-and-configured-on-your>.
- Microsoft (2018a). Security Bulletin Severity Rating System. <https://technet.microsoft.com/en-us/security/gg309177.aspx>.
- Microsoft (2018b). The Enhanced Mitigation Experience Toolkit. <https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit>.

- Microsoft (2020). /CETCOMPAT (CET Shadow Stack compatible). <https://docs.microsoft.com/en-us/cpp/build/reference/cetcompat?view=vs-2019>.
- Miller, C. and Iozzo, V. (2009). Fun and games with Mac OS X and iPhone payloads. *BlackHat Europe*.
- Minh, B. Q. and Minh, H. X. (2009). How Conficker makes use of MS08-067. <http://security.bkav.com/home/-/blogs/how-conficker-makes-use-of-ms08-067/normal>.
- MITRE (2018). Vulnerabilities By Type. <https://www.cvedetails.com/vulnerabilities-by-types.php>.
- Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K. W., and Franz, M. (2015). Opaque control-flow integrity. In *NDSS*, volume 26, pages 27--30.
- Naraine, R. (2010). Pwn2own 2010: iphone hacked, sms database hijacked. <http://blogs.zdnet.com/security>.
- Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89--100. ACM.
- NetMarketShare (2018). Browser Market Share. <https://netmarketshare.com/browser-market-share.aspx>.
- Newsham, T. (1997). Re: Smashing the stack: prevention? Bugtraq, Apr. 1997. <http://seclists.org/bugtraq/1997/Apr/129>.
- Newsham, T. (2000). Non-exec stack. <http://seclists.org/bugtraq/2000/May/90>.
- NIST (2018). NVD - Search and Statistics. <https://nvd.nist.gov/vuln/search>.
- Niu, B. and Tan, G. (2014a). Modular control-flow integrity. In *ACM SIGPLAN Notices*, pages 577--587. ACM.
- Niu, B. and Tan, G. (2014b). Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317--1328. ACM.
- Niu, B. and Tan, G. (2015). Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 914--926. ACM.
- Offensive Security (2020). Exploit Database. <https://www.exploit-db.com/>.

- Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., and Kirda, E. (2010). G-free: defeating return-oriented programming through gadget-less binaries. In *ACSAC*, pages 49--58. ACM.
- Pappas, V., Polychronakis, M., and Keromytis, A. D. (2012). Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601--615. IEEE.
- Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013a). Practical software diversification using in-place code randomization. In *Moving Target Defense II*, pages 175--202. Springer.
- Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013b). Transparent ROP exploit mitigation using indirect branch tracing. In *Security Symposium*, pages 447--462. USENIX.
- PaX Team (2003a). Non-executable pages design & implementation.
- PaX Team (2003b). Pax address space layout randomization (aslr). <https://pax.grsecurity.net/docs/aslr.txt>.
- Payer, M., Barresi, A., and Gross, T. R. (2014). Lockdown: Dynamic control-flow integrity. Technical report, ETH Zurich.
- Payer, M., Barresi, A., and Gross, T. R. (2015). Fine-grained control-flow integrity through binary hardening. In *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment*.
- Payer, M. and Payer, G. (2001). StackGhost: Hardware facilitated stack protection. In *Security Symposium*, pages 1--11. USENIX.
- Pewny, J. and Holz, T. (2013). Control-flow restrictor: Compiler-based cfi for ios. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 309--318. ACM.
- Pfaff, D., Hack, S., and Hammer, C. (2015). Learning how to Prevent Return-Oriented Programming Efficiently. In Piessens, F., Caballero, J., and Bielova, N., editors, *ESSoS*, volume 8978 of *LNCS*, pages 68--85. Springer.
- Polychronakis, M., Anagnostakis, K., and Markatos, E. (2006). Detection of intrusions and malware & vulnerability assessment. In *Proceedings of the Third international conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 4064, pages 54--73. Springer.

- Polychronakis, M. and Keromytis, A. D. (2011). Rop payload detection using speculative code execution. In *2011 6th International Conference on Malicious and Unwanted Software*, pages 58–65. IEEE.
- Prakash, A., Hu, X., and Yin, H. (2015). vfGuard: Strict protection for virtual function calls in cots C++ binaries. In *Network and Distributed System Security Symposium*.
- Qiu, P., Lyu, Y., Zhang, J., Wang, D., and Qu, G. (2018). Control flow integrity based on lightweight encryption architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(7).
- Rajagopalan, M., Hiltunen, M. A., Jim, T., and Schlichting, R. D. (2006). System call monitoring using authenticated system calls. *IEEE Transactions on Dependable and Secure Computing*, 3(3).
- Rapid7 (2007). Exploiting the ani vulnerability on vista. <https://blog.rapid7.com/2007/04/01/exploiting-the-ani-vulnerability-on-vista/>.
- Rapid7 (2012). Metasploit: Defeat the Hard and Strong with the Soft and Gentle Metasploit RopDB. <https://blog.rapid7.com/2012/10/03/defeat-the-hard-and-strong-with-the-soft-and-gentle-metasploit-ropdb/>.
- Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366.
- Richarte, G. (2000). Re: Future of buffer overflows. <http://seclists.org/bugtraq/2000/Nov/32>.
- Richarte, G. (2001). Insecure programming by example: Esoteric# 2. <http://pages.cs.wisc.edu/~riccardo/sec/e2.html>.
- Roemer, R., Buchanan, E., Shacham, H., and Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2.
- Roemer, R. G. (2009). *Finding the bad in good code: Automated return-oriented programming exploit discovery*. PhD dissertation, Master’s thesis, University of California, San Diego.
- Roglia, G. F., Martignoni, L., Paleari, R., and Bruschi, D. (2009). Surgically returning to randomized libc. In *ACSAC*, pages 60–69. ACM.

- Rosenberg, D. (2011). Defeating Windows 8 ROP Mitigation. <http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/>.
- Ruan, Y., Kalyanasundaram, S., and Zou, X. (2016). Survey of return-oriented programming defense mechanisms. *Security and Communication Networks*, 9(10):1247-1265.
- Samir, H. and Mr_me (2010). AoA Audio Extractor 2.x - ActiveX ROP exploit. <http://www.exploit-db.com/exploits/15235/>.
- Santos, J. C. M. and Fei, Y. (2008). Leveraging speculative architectures for run-time program validation. In *IEEE International Conference on Computer Design*.
- Schirra, S. (2019). Ropper - rop gadget finder and binary information tool. <https://scoding.de/ropper/>.
- Schwartz, E. J., Avgerinos, T., and Brumley, D. (2011). Q: Exploit hardening made easy. In *USENIX Security Symposium*.
- Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., and Soffa, M. L. (2003). Retargetable and reconfigurable software dynamic translation. In *International Symposium on Code Generation and Optimization*, pages 36--47. IEEE.
- Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, pages 552--561, New York, NY, USA. ACM.
- Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *CSS*, pages 298--307, New York, NY, USA. ACM.
- Shi, Y. and Lee, G. (2007). Augmenting branch predictor to secure program execution. In *DSN*, pages 10--19, Washington, DC, USA. IEEE Computer Society.
- Shuo, T., Yeping, H., and Baozeng, D. (2012). Prevent Kernel Return-Oriented Programming Attacks Using Hardware Virtualization. In *Lecture Notes in Computer Science*, volume 7232, pages 289--300. Springer Berlin Heidelberg.
- Sinha, R., Costa, M., Lal, A., Lopes, N. P., Rajamani, S. K., Seshia, S. A., and Vaswani, K. (2016). A design and verification methodology for secure isolated regions. In *PLDI*.

- Sinnadurai, S., Zhao, Q., and fai Wong, W. (2008). Transparent runtime shadow stack: Protection against malicious return address modifications. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.5702&rep=rep1&type=pdf>.
- Snake (2012). Bypassing EMET 3.5's ROP Mitigations. <https://repret.wordpress.com/2012/08/08/bypassing-emet-3-5s-rop-mitigations/>.
- Solé, P. (2010). Hanging on a ROPE. http://immunitysec.com/downloads/DEPLIB20_ekoparty.pdf.
- Son, N. H. (2011). ROP chain for Windows 8. <http://security.bkav.com/home/-/blogs/rop-chain-for-windows-8/normal>.
- Sotirov, A. and Dowd, M. (2008). Bypassing browser memory protections: Setting back browser security by 10 years. *Black Hat*.
- Spyrit, D. and Jack, A. (1999). Win32 buffer overflows (location, exploitation, and prevention). *Phrack magazine*, 9(55).
- Srivastava, A., Edwards, A., and Vo, H. (2001). Vulcan: Binary transformation in a distributed environment. Technical report, technical report msr-tr-2001-50, microsoft research.
- Sun, B., Liu, J., and Xu, C. (2019). How to Survive the Hardware-assisted Control-flow Integrity Enforcement. <https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf> and <https://www.youtube.com/watch?v=D4-YwGYYcTg>.
- Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). Sok: Eternal war in memory. In *SE&P*, pages 48--62, Washington, DC, USA. IEEE Computer Society.
- Tang, A., Sethumadhavan, S., and Stolfo, S. J. (2014). Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*, pages 109--129. Springer.
- The H Security (2009). Pwn2own 2009: Safari, ie 8 and firefox exploited. <http://www.h-online.com/security/news/item/Pwn20wn-2009-Safari-IE-8-and-Firefox-exploited-740663.html>.

- The llvm-admin team (2018). The LLVM Compiler Infrastructure. <https://llvm.org/>.
- Theiling, H. (2000). Extracting safe and precise control flow from binaries. In *RTCSA*, pages 23--30. IEEE.
- Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. (2014). Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security Symposium*.
- Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P. (2011). On the expressiveness of return-into-libc attacks. In *RAID*, pages 121--141. Springer.
- Tung, L. M. (2011). Advanced Generic ROP chain for Windows 8. <http://security.bkav.com/en/home/-/blogs/advanced-generic-rop-chain-for-windows-8/normal>.
- Tymburibá, M., Moreira, R. E., and Quintão Pereira, F. M. (2016). Inference of peak density of indirect branches to detect ROP attacks. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 150--159. ACM.
- van der Veen, V., Andriesse, D., Göktas, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., and Giuffrida, C. (2015). Practical context-sensitive CFI. In *ACM SIGSAC Conference on Computer and Communications Security*.
- van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., and Giuffrida, C. (2017). The dynamics of innocent flesh on the bone: Code reuse ten years later. In *CCS*, pages 1675--1689, New York, NY, USA. ACM.
- Vandierendonck, H. and Seznec, A. (2008). Speculative return address stack management revisited. *ACM Transactions on Architecture and Code Optimization*, 5(3).
- Vendicator (2000). Stack shield: A stack smashing technique protection tool for linux. <http://www.angelfire.com/sk/stackshield/>.
- Vreugdenhil, P. (2010). Pwn2own 2010 windows 7 internet explorer 8 exploit. <http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf>.
- Wang, Z. and Jiang, X. (2010). Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380--395. IEEE.

- Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. (2012). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *CCS*, pages 157--168, New York, NY, USA. ACM.
- Wei, T., Wang, T., Duan, L., and Luo, J. (2010). Secure dynamic code generation against spraying. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 738--740.
- Wicherski, G. (2016). Threat detection for return oriented programming. US Patent 9,256,730.
- Wojtczuk, R. (2001). The advanced return-into-lib (c) exploits: PaX case study. *Phrack Magazine*, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e.
- Xia, Y., Liu, Y., Chen, H., and Zang, B. (2012). CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1--12. IEEE.
- Yuan, L., Xing, W., Chen, H., and Zang, B. (2011). Security breaches as PMU deviation: Detecting and identifying security attacks using performance counters. In *APSys*, pages 6:1--6:5. ACM.
- Zhang, C., Song, C., Chen, K. Z., Chen, Z., and Song, D. (2015). VTint: Protecting virtual function tables' integrity. In *Network and Distributed System Security Symposium*.
- Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. (2013). Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy*, pages 559--573. IEEE.
- Zhang, J., Liu, Y., Li, H., Zhu, X., and Chen, M. (2018). PTAT: An efficient and precise tool for tracing and profiling detailed tlb misses. *ACM TECS*, 17(3):62.
- Zhang, M. and Sekar, R. (2013). Control flow integrity for COTS binaries. In *USENIX Security Symposium*.