UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

RIP-ROP: uma proteção contra ataques de execução de código arbitrário baseados em Return-Oriented Programming

Mateus Felipe Tymburibá Ferreira

Manaus - Amazonas
2014

Mateus Felipe Tymburibá Ferreira

RIP-ROP: uma proteção contra ataques de execução de código arbitrário baseados em Return-Oriented Programming

Dissertação de mestrado apresentada ao Programa de Pós-graduação em Informática da Universidade Federal do Amazonas como requisito parcial para obtenção do título de Mestre em Informática.
Área de concentração: Segurança de Sistemas.

Orientador: Prof. Dr. Eduardo Luzeiro Feitosa

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

F383r

Ferreira, Mateus Felipe Tymburiba

RIP-ROP: uma proteção contra ataques de execução de código arbitrário baseados em Return-Oriented Programming / Mateus Felipe Tymburiba Ferreira. 2014

87 f.: il. color; 31 cm.

Orientador: Eduardo Luzeiro Feitosa

Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Return-Oriented Programming. 2. ROP - Return-Oriented Programmin. 3. Reúso de código. 4. Proteção de Dados. I. Feitosa, Eduardo Luzeiro II. Universidade Federal do Amazonas III. Título

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

211ª ATA DE DEFESA PÚBLICA DE DISSERTAÇÃO DO MESTRADO

Aos 06 dias do mês de agosto do ano de 2014, às 09:00h, na Sala de Seminários do Instituto de Computação da Universidade Federal do Amazonas, situado na Av. Rodrigo Otávio, 6.200, Campus Universitário, Setor Norte, Coroado, nesta Capital, ocorreu a sessão pública de defesa de dissertação de mestrado intitulada **"RIP-ROP: uma proteção contra ataques de execução de código arbitrário baseados em Return-Oriented Programming"** apresentada pelo aluno **Mateus Felipe Tymburibá Ferreira** que concluiu todos os pré-requisitos exigidos para a obtenção do título de mestre em informática, conforme estabelece o artigo 52 do regimento interno do curso. Os trabalhos foram instalados pelo Prof. Eduardo Luzeiro Feitosa – UFAM, orientador e presidente da Banca Examinadora, que foi constituída, ainda, pelo Prof. Eduardo James Pereira Souto – UFAM – Membro e Prof. Fernando Magno Quintão Pereira – UFMG – Membro. A Banca Examinadora tendo decidido aceitar a dissertação, passou à arguição pública do candidato. Encerrados os trabalhos, os examinadores expressaram o parecer abaixo.

A comissão considerou a dissertação:

( ) Aprovada
( ) Aprovada condicionalmente, sujeita a alterações, conforme folha de modificações, anexa,
( ) Reprovada, conforme folha de modificações, anexa

Proclamados os resultados, foram encerrados os trabalhos e, para constar, eu, Elienai Nogueira, Secretária do Programa de Pós-Graduação em Informática, lavrei a presente ata, que assino juntamente com os Membros da Banca Examinadora.

Prof. Eduardo Luzeiro Feitosa — Assinatura:...

Prof. Eduardo James Pereira Souto — Assinatura:...

Prof. Fernando Magno Quintão Pereira — Assinatura:...

Manaus, 06 de agosto de 2014.

____
Secretária

Aos meus pais, fontes inesgotáveis de amor,
compreensão e apoio incondicionais.

Agradecimentos

Primeiramente, agradeço a Deus. Não apenas por ter me abençoado desde o nascimento com a família maravilhosa que possuo, mas por ter me proporcionado, ao longo de toda a minha vida, oportunidades esplêndidas. A realização deste mestrado é uma dessas oportunidades e, em um país onde tão poucos têm essa chance, sinto-me mais uma vez privilegiado.

A meus pais, entre tantos ensinamentos, neste momento agradeço em especial pelas lições de valorização dos estudos. Esse legado foi e sempre será meu principal fator de incentivo e combustível para os momentos de dificuldade.

À minha amada esposa, pelo companheirismo, incentivo e amparo. Eu jamais teria começado essa jornada sem o seu apoio e compreensão, quanto mais chegado ao final dela com o sentimento de dever cumprido.

Devo também um agradecimento caloroso a todos os integrantes do Programa de Pós-Graduação em Informática da Universidade Federal do Amazonas. Desde a minha chegada, fui extremamente bem recebido por professores, funcionários e alunos. Em especial, agradeço ao professor Eduardo Nakamura, pelos conhecimentos transmitidos e por “abrir as portas” do programa para mim. Ao professor Horácio por me receber como aluno especial e, posteriormente, me apresentar aos professores do grupo ETSS. Aos professores Eduardo Souto e Eduardo Feitosa pela receptividade, ensinamentos, suporte e compreensão. À Elienai e demais funcionários do PPGI, pela disposição e prontidão em ajudar, em todas as circunstâncias.

Agradeço também ao Exército Brasileiro por me proporcionar as oportunidades e experiências necessárias para ingressar na empolgante área de segurança de sistemas computacionais. Os aprendizados colhidos com todos os amigos, colegas e instrutores me prepararam para esta jornada e para outras que virão.

Finalmente, obrigado ao Amazonas pela hospitalidade. “Quem come Jaraqui, nunca mais sai daqui”! Nem que seja em pensamentos...

Resumo

Return-Oriented Programming (ROP) 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. Ela baseia-se na interligação de pequenas frações de código pertencentes aos próprios processos atacados, o que permite a superação de proteções largamente difundidas, como aquela oferecida pelo bit de execução (NX/XD).

Em função de seu vasto emprego em investidas contra sistemas computacionais modernos, proteções contra *exploits* baseados em ROP têm sido extensamente estudadas. Apesar disso, ainda não se conhece uma solução capaz de aliar eficácia contra todas as modalidades de ROP, eficiência computacional e viabilidade de emprego na proteção de aplicações. Com o intuito de facilitar o entendimento desses requisitos, bem como das implicações inerentes a métodos de proteção contra ataques ROP, este trabalho oferece um levantamento bibliográfico do estado-da-arte envolvendo esse tema. Para isso, são propostas neste trabalho: (i) métricas para avaliação e comparação de proteções contra ataques ROP e (ii) taxonomias para classificação dessas proteções em função das estratégias de bloqueio e das abordagens de implementação utilizadas em cada solução.

Esta dissertação provê ainda um novo método de proteção contra ataques de execução de código arbitrário baseados em ROP que busca abarcar os requisitos de eficácia, eficiência e viabilidade. Demonstrou-se que, através do controle da frequência de instruções de desvio indireto executadas pelas aplicações, é possível distinguir ataques ROP de códigos autênticos e, assim, evitar a sua consolidação.

Em um *framework* de instrumentação binária dinâmica, foi desenvolvido um protótipo – denominado RIP-ROP – destinado a ambientes Windows, Linux, Android e OSX. Experimentos realizados com códigos maliciosos disponíveis em repositórios públicos de *exploits* confirmaram a viabilidade do modelo proposto para a proteção de aplicações reais. Além disso, o custo computacional imposto pelo RIP-ROP é comparável e, em alguns casos, inferior àquele alcançado por proteções correlatas.

Palavras-chave: *Return-Oriented Programming*, ROP, reúso de código, proteção.

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 execution of arbitrary code on vulnerable applications. It is based on the interconnection of small fractions of code belonging to attacked processes, which allows overcoming protections widely disseminated, such as that offered by the execute bit (NX/XD).

Because of its wide use in attacks against modern computing systems, protections against ROP based exploits have been widely studied. Nevertheless, it is still not known a solution capable of combining efficacy against all forms of ROP, computational efficiency and feasibility of the employment on applications protection. In order to facilitate the understanding of these requirements and the inherent implications for methods of protection against ROP attacks, this work offers a bibliographic survey of the state of the art about this subject. For this, we propose in this paper: (i) metrics for evaluation and comparison of protections against ROP attacks and (ii) taxonomies to classify these protections depending on blocking strategies and implementation approaches used in each solution.

This dissertation also provides a new method of protection against arbitrary code execution attacks based on ROP that seeks to encompass the requirements of effectiveness, efficiency and viability. It was demonstrated that by controlling the frequency of indirect branch instructions performed by applications it is possible to distinguish ROP attacks from authentic codes and thus prevent their consolidation.

In a dynamic binary instrumentation framework, it was developed a prototype – named RIP-ROP – for Windows, Linux, Android and OSX environments. Experiments conducted with malicious codes available in public repositories of exploits confirmed the feasibility of the proposed model for the protection of real applications. In addition, the computational cost imposed by RIP-ROP is comparable and in some cases lower than that achieved by related protections.

Keywords: Return-Oriented Programming, ROP, code reuse, protection.

Sumário

Resumo v

Abstract vi

Lista de Figuras ix

Lista de Tabelas x

Lista de Siglas xi

1 Introdução 1

1.1 Motivação....2
1.2 Objetivos....4
1.3 Contribuições Esperadas....4
1.4 Organização do documento....4

2 Conceitos 6

2.1 Transbordamento de Buffer na Pilha (Stack Buffer Overflow)....6
2.2 Proteções de Memória....10
2.2.1 Pilha não executável (nx-stack)....10
2.2.2 Espaço aleatório de endereços (ASLR)....11
2.2.3 Bit de execução (NX/XD)....11
2.3 Return-Oriented Programming (ROP)....12
2.3.1 Histórico....12
2.3.2 Funcionamento....13

3 Trabalhos Relacionados 20

3.1 Métricas de Avaliação....20
3.2 Estratégias de Proteção....24
3.2.1 Randomização....24
3.2.2 Construção de uma pilha-sombra....26
3.2.3 Checagem da instrução anterior ao endereço de retorno....28
3.2.4 Checagem dos endereços autênticos para desvios....29
3.2.5 Checagem da posição de entrada nas funções....30
3.2.6 Controle da frequência de instruções de retorno....31
3.2.7 Checagem do apontador para o topo da pilha....32
3.3 Abordagens de Implementação....33
3.3.1 Compilação....33
3.3.2 Instrumentação binária estática....35
3.3.3 Instrumentação binária dinâmica....36
3.3.4 Adaptação do hardware....37
3.3.5 Virtualização....38
3.3.6 Emulação de código....38
3.4 Discussão....40

4 Controle da frequência de desvios indiretos **43**
4.1 Metodologia....44
4.1.1 Bases de dados....45
4.1.1.1 SPEC CPU2006....45
4.1.1.2 Exploit Database....46
4.1.2 Avaliação teórica....47
4.1.2.1 Laços de repetição com estrutura de controle....48
4.1.2.2 Funções recursivas....50
4.1.2.3 Laços de repetição com chamada de função....51
4.1.3 Avaliação empírica....52
4.1.4 Avaliação de desempenho....54
4.2 RIP-ROP....55
4.2.1 Pin....56
4.2.2 Módulo de controle da frequência de desvios indiretos....59
4.2.3 Otimizações....61
4.2.3.1 A instrução POPCNT....61
4.2.3.2 Tamanho da janela....62
4.2.3.3 Linhas de Cache da CPU....62
4.2.3.4 Número de bits deslocados....63
4.3 Discussão....64

5 Resultados **67**
5.1 Protocolo experimental....67
5.2 Ambientes de experimentação....70
5.3 Validação da estratégia de proteção....71
5.4 Eficácia no bloqueio de exploits reais....77
5.5 Desempenho....78

6 Conclusões **84**
6.1 Contribuições....84
6.2 Trabalhos Futuros....85

Referências **88**

Lista de Figuras

1.1 Número de vulnerabilidades de erros em buffers catalogadas entre 2001 e 2012.........................1

1.2 Ocorrência das classes de ataques catalogados no CVE entre 2001 e 2012..................................2

2.1 Estado da pilha antes de iniciar a chamada de uma nova função e após retornar da função..........7

2.2 Estado da pilha após término dos passos executados durante uma chamada de função................8

2.3 Estado da pilha após ataque de estouro de buffer na pilha...9

2.4 Estrutura da pilha e fluxo entre gadgets do exploit apresentado na Listagem 2.1.......................18

2.5 Estratégia de proteção baseada na construção de uma pilha-sombra..27

3.1 Gráficos hipotéticos de densidade máxima de desvios indiretos para 10 aplicações..................44

4.1 Arquitetura do Pin..57

4.2 Lógica de controle das instruções de desvio indireto..60

4.3 Solução para falsos positivos ocasionados por funções com recursividade em cauda................65

5.1 Frequência máxima de desvios indiretos com janela de 32 instruções no Linux........................72

5.2 Frequência máxima de desvios indiretos com janela de 64 instruções no Linux........................72

5.3 Frequência máxima de desvios indiretos com janela de 96 instruções no Linux........................72

5.4 Frequência máxima de desvios indiretos com janela de 128 instruções no Linux......................72

5.5 Frequência máxima de desvios indiretos com janela de 32 instruções no Windows...................73

5.6 Frequência máxima de desvios indiretos com janela de 64 instruções no Windows...................73

5.7 Frequência máxima de desvios indiretos com janela de 96 instruções no Windows...................73

5.8 Frequência máxima de desvios indiretos com janela de 128 instruções no Windows.................74

5.9 Overhead imposto aos benchmarks inteiros no Windows de 64 bits..79

5.10 Overhead imposto aos benchmarks de ponto flutuante no Windows de 64 bits........................80

5.11 Overhead imposto aos benchmarks inteiros no Linux de 64 bits..80

5.12 Overhead imposto aos benchmarks de ponto flutuante no Linux de 64 bits.............................81

Lista de Tabelas

2.1 Opções de configuração do DEP..14

3.1 Comparação das proteções contra ataques ROP..41

4.1 Proporção de instruções de desvio indireto nos exploits catalogados...48

5.1 Distribuição das frequências máximas de desvios indiretos com a janela de 32 instruções........75

5.2 Overhead médio ao executar benchmarks..82

5.3 Proteções que controlam a frequência de instruções de retorno via instrumentação binária dinâmica..82

Lista de Siglas

API	Application Programming Interface
ASLR	Address Space Layout Randomization
BBL	Basic Block
CVE	Common Vulnerabilities and Exposures
DCG	Dynamic Code Generation
DEP	Data Execution Prevention
EMET	Enhanced Mitigation Experience Toolkit
JIT	Just-in-time
JOP	Jump Oriented Programming
NIST	National Institute of Standards and Technology
NOP	No Operation
NX	No eXecute
PC	Program Counter
PIC	Position Independent Code
RILC	Return-into-libc
RIP-ROP	Reuse of Instructions Prevention against Return-Oriented Programmimg
ROP	Return-Oriented Programming
SafeSEH	Safe Structured Exception Handler
SEH	Structured Exception Handlers
SEHOP	Structured Exception Handler Overwrite Protection
TC-RILC	Turing complete Return-into-libc
W⊕X	Write XOR Execute
XD	eXecute Disable

1 Introdução

O software é considerado o elo mais fraco da cadeia de componentes alvejados por atacantes em atos hostis contra sistemas computacionais [1]. À medida que a quantidade e a complexidade dos softwares aumenta, o volume de vulnerabilidades identificadas também cresce, conforme observa-se nas estatísticas disponibilizadas pelo NIST (*National Institute of Standards and Technology*) e apresentadas na Figura 1.1.

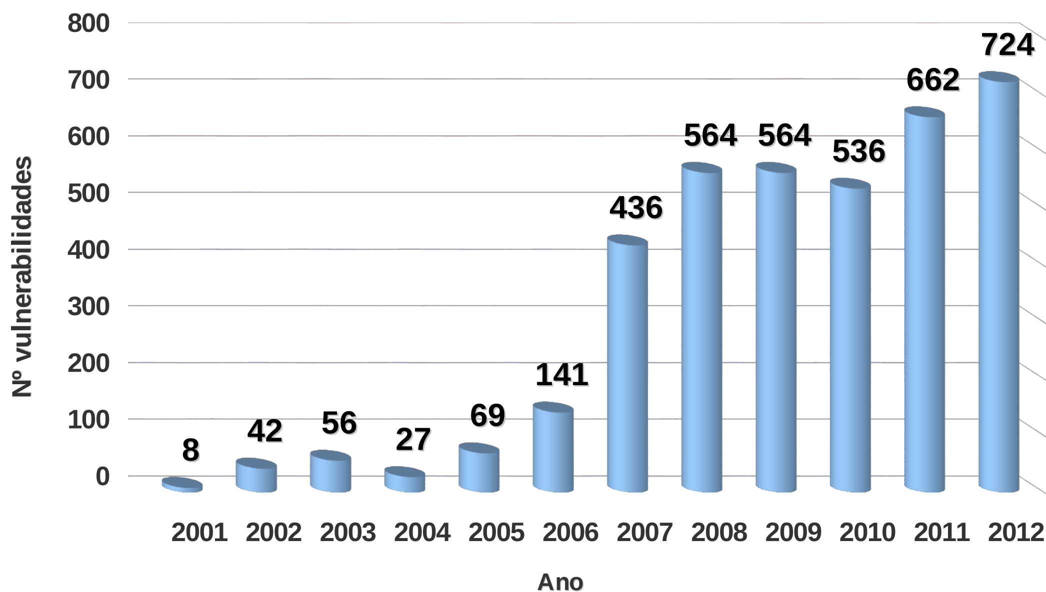

Figura 1.1: Número de vulnerabilidades de erros em *buffers* catalogadas entre 2001 e 2012 [2].

Esse gráfico ilustra a tendência de crescimento na descoberta de vulnerabilidades classificadas pelo NIST como erros em *buffers*[1]. Mais conhecido pela nomenclatura *buffer overflow*[2], esse tipo de falha em softwares é considerado crítico para a segurança de sistemas porque permite que o atacante subverta o fluxo de execução normal da aplicação sob ataque e execute um código arbitrário, sem o consentimento do usuário. Justamente por esse motivo, vulnerabilidades que acarretam na possibilidade de execução de código arbitrário são classificadas como críticas [3].

Em decorrência do poder que proporcionam ao invasor, as investidas que executam código arbitrário conquistaram a preferência dos atacantes. Esse comportamento pode ser observado na Figura 1.2, que apresenta o percentual desse tipo de ataque em relação às demais classes de ataques catalogados ao longo dos anos no CVE (*Common Vulnerabilities and Exposures*). Desde

[1] *Buffer*: um espaço de memória limitado e contínuo.

[2] *Overflow*: transbordamento (também chamado de “estouro”) do espaço disponível.

2001, ano em que as estatísticas iniciais são apresentadas, a exploração de vulnerabilidades que levam à execução de código arbitrário mantem-se como o tipo de ataque mais frequente.

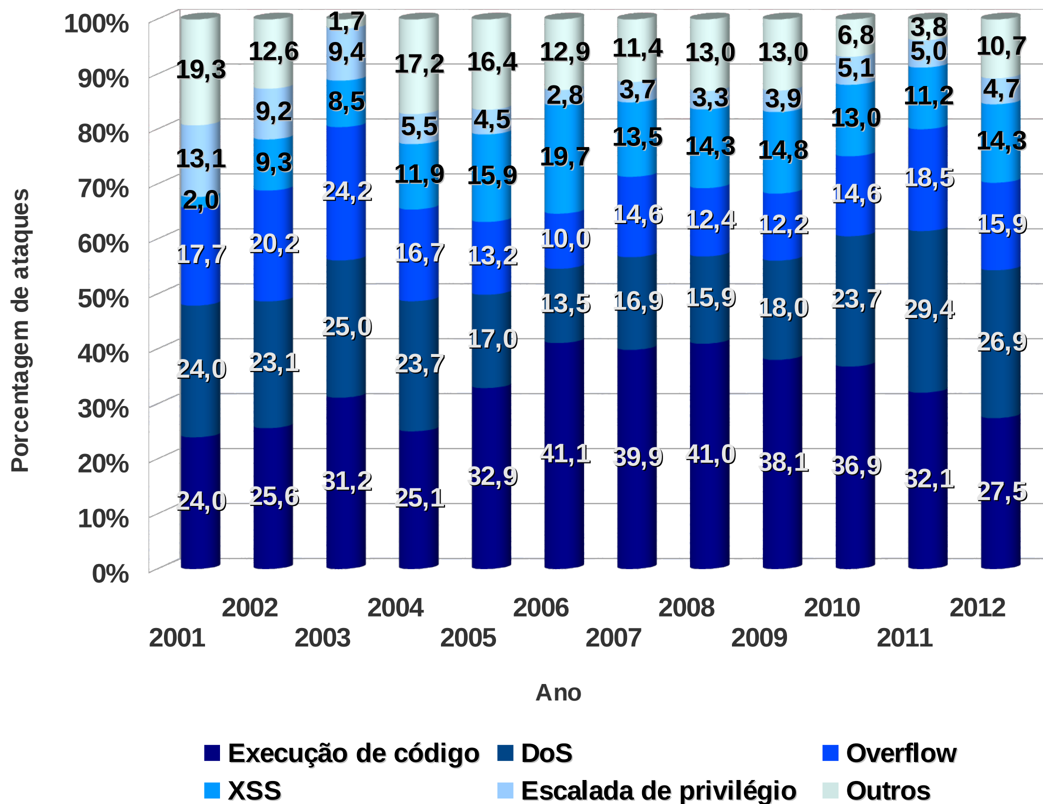

Figura 1.2: Ocorrência das classes de ataques catalogados no CVE entre 2001 e 2012 [4].

Além das vulnerabilidades de *buffer overflow*, outros tipos de falhas em software podem acarretar na execução de código arbitrário. Entre elas, pode-se citar: falhas de formatação de strings (*format string*), estouro de inteiros (*integer overflow*), liberação dupla (*double free*), uso de variável não inicializada (*uninitialized variable usage*) e uso após liberação *(use-after-free*) [5].

1.1 Motivação

Pesquisadores destacam o enorme montante de receitas perdido devido a ataques cibernéticos dos mais variados tipos, o que tem elevado o interesse da comunidade científica e os investimentos de organizações na busca por novas soluções que bloqueiem as recentes técnicas de invasão de sistemas computacionais [6], [7].

A técnica denominada *Return-Oriented Programming* (ROP), detalhada na Seção 2.3 deste texto, tem despertado grande interesse da comunidade científica e da indústria de segurança de sistemas, em função da sua larga utilização em ataques recentes a sistemas computacionais [8].

Entre muitos exemplos de códigos maliciosos de grande impacto que empregam a técnica ROP, pode-se citar os *malwares* Stuxnet e Duqu [9] e o código usado na violação de um tipo de urna de votação eletrônica empregada em diversas localidades [10]. A relevância dos ataques ROP foi impulsionada também pelo desenvolvimento de técnicas automatizadas para a construção de códigos de ataques, o que reduziu drasticamente o tempo e a complexidade para elaboração dos artefatos maliciosos baseados em ROP [11]–[20].

Por ter se tornado uma das principais técnicas utilizadas por atacantes para desenvolver *exploits*[3] [21], mitigações contra o ROP têm sido amplamente estudadas. O Windows 8, por exemplo, agrega um novo mecanismo de proteção contra o ROP, que impede a chamada de APIs (Application Programming Interfaces) tipicamente utilizadas em ataques ROP, caso os parâmetros não estejam armazenados na área de pilha do processo. No entanto, poucos dias depois do lançamento da versão preliminar do sistema (Windows 8 Developer Preview), pesquisadores apresentaram demonstrações de estratégias relativamente simples capazes de burlar essa defesa [22]–[24]. Tudo isso relacionado a um sistema que ainda nem havia sido lançado oficialmente.

Outro ponto de concentração de esforços na busca por mecanismos de contenção de ataques tem sido a ferramenta EMET (*Enhanced Mitigation Experience Toolkit*) [25]. Trata-se de um utilitário gratuito que pode ser instalado nos sistemas Windows. Ao longo do tempo, essa ferramenta tem agregado uma série de proteções contra técnicas de desenvolvimento de *exploits*. Atualmente, ela inclui, entre outros mecanismos de segurança, quatro novas mitigações contra ROP, incluindo uma versão do ROPGuard, uma defesa contra ROP apresentada por Ivan Fratric no concurso BlueHat Prize [26]. Essa implementação do ROPGuard foi superada duas semanas após o seu anúncio [27].

Até mesmo o resultado do concurso “BlueHat Prize”, promovido em 2012 para premiar trabalhos que visam defender a segurança de computadores, evidencia a importância do ROP no cenário atual de segurança de sistemas. As três melhores soluções, escolhidas para premiação, apresentaram técnicas de mitigação contra ROP, o que reforça a constatação de que os fabricantes de software estão em estado de alerta em relação a esse tipo de ataque [28]. De fato, diversas proteções contra ataques ROP já foram propostas, mas ainda não há uma solução definitiva.

[3] *Exploit*: artefato desenvolvido com a finalidade de explorar uma vulnerabilidade presente em um sistema.

1.2 Objetivos

Este trabalho tem os objetivos de introduzir métricas e taxonomias destinadas à avaliação de proteções contra ataques de execução de código arbitrário baseados em ROP e apresentar um novo método de proteção contra esses ataques. A eficácia desse método no bloqueio de ataques reais é demonstrada através de testes com códigos maliciosos disponíveis em repositórios públicos de *exploits*. São analisadas também a ocorrência de falsos positivos e a eficiência computacional do modelo proposto, comparando-o com soluções correlatas.

1.3 Contribuições Esperadas

Para atingir esses objetivos, as seguintes contribuições deverão ser alcançadas:

- Definição de métricas para avaliação e comparação de proteções contra ataques ROP.
- Elaboração de duas taxonomias para classificação das proteções em função das estratégias de proteção contra ataques ROP e das abordagens de implementação utilizadas em cada solução.
- Demonstração da eficácia do controle da frequência de desvios indiretos como estratégia para detecção de ataques ROP, incluindo suas variantes inexploradas por grande parte das soluções atuais.
- Desenvolvimento de um protótipo de proteção contra ataques ROP destinado a ambientes Windows, Linux, Android e OSX em um *framework* de instrumentação binária dinâmica.

1.4 Organização do documento

O restante deste trabalho está estruturado da seguinte maneira:

- **Capítulo 2 [Conceitos]**: são apresentados os conceitos essenciais para a compreensão do funcionamento de ataques ROP. Na Seção 2.1, o conceito de Transbordamento de *Buffer* na Pilha (*Stack Buffer Overflow*) é apresentado para viabilizar a posterior explicação detalhada de um *exploit* ROP real (Seção 2.3). Na Seção 2.2 são apresentadas as proteções de memória amplamente difundidas entre os sistemas computacionais modernos que motivaram ou que têm um impacto significativo na técnica de exploração ROP.

- **Capítulo 3 [Trabalhos Relacionados]**: são discutidas as propostas de proteções contra ataques ROP já publicadas. Para isso, na Seção 3.1 são introduzidas novas métricas para avaliação das proteções contra ataques ROP. Além de debater as vantagens e deficiências inerentes às diversas soluções contra ataques ROP, nas Seções 3.2 e 3.3 são apresentadas duas taxonomias inéditas para classificar as soluções segundo as estratégias de proteção e as abordagens utilizadas por elas para implementar as defesas. A Seção 3.4 encerra o Capítulo apresentando um resumo dos aspectos discutidos e uma discussão sobre as tendências esperadas para o futuro das proteções contra ataques ROP.
- **Capítulo 4 [Controle da frequência de desvios indiretos]**: são detalhados os aspectos relacionados à proteção contra ataques ROP desenvolvida neste trabalho. A Seção 4.1 contém uma descrição da metodologia empregada durante o processo de concepção e validação da estratégia de controle da frequência de instruções de desvio indireto. Em seguida, as bases de dados utilizadas durante a execução dos experimentos são apresentadas na Seção 4.2. Os principais aspectos relacionadas à implementação do protótipo são discutidos na Seção 4.3. Encerrando o Capítulo, a Seção 4.4 resume os maiores desafios impostos ao método de proteção desenvolvido.
- **Capítulo 5 [Resultados]**: são apresentados o protocolo utilizado durante a realização dos experimentos (Seção 5.1), a configuração dos ambientes de teste (Seção 5.2) e os resultados obtidos nos estudos realizados. Esses resultados refletem a validação da estratégia de proteção (Seção 5.3), a avaliação da eficácia do modelo no bloqueio de *exploits* reais (Seção 5.4) e a análise do impacto da solução no desempenho das aplicações protegidas (Seção 5.5).
- **Capítulo 6 [Conclusões]**: são resumidas as principais contribuições deste trabalho (Seção 6.1) e elencadas algumas direções futuras que podem ser seguidas (Seção 6.2), tomando-se como base a experiência adquirida durante a realização deste trabalho.

2 Conceitos

Este capítulo destina-se a elucidar conceitos necessários para o entendimento da técnica ROP. A fim de facilitar a compreensão do processo de exploração de uma vulnerabilidade que acarrete na execução de um código arbitrário, a Seção 2.1 fornece uma breve explicação de uma das técnicas mais usadas pelos atacantes para explorar vulnerabilidades em softwares, o transbordamento de *buffer* na pilha (*stack buffer overflow*). Na sequência, a Seção 2.2 explica a origem histórica dos principais mecanismos de proteção da memória que influenciam na técnica de ataque ROP, bem como suas implicações para o desenvolvimento desse tipo de *exploit*. A Seção 2.3 é dedicada à explicação do funcionamento do ROP, utilizando-se para isso um exemplo público de código malicioso. Finalmente, a Seção 2.4 apresenta a relação da técnica ROP com outros tipos de ataques que reutilizam porções de código da própria aplicação para executar códigos maliciosos.

2.1 Transbordamento de *Buffer* na Pilha (*Stack Buffer Overflow*)

Conforme destacam Hoglund e McGraw [1], as explorações de *buffer overflow* são as principais armas dos atacantes e, provavelmente, essa tendência deve permanecer por vários anos. Em [5], os autores afirmam que o estouro de *buffer* na pilha (*stack buffer overflow*) é o exemplo de *buffer overflow* mais discutido na literatura, sendo, portanto, adequado para ilustrar o procedimento adotado por atacantes com o intuito de desviar o fluxo de execução de um processo para um código arbitrário.

Para entender a estratégia adotada pelos invasores, é preciso lembrar que a pilha é um segmento de memória alocado na área de endereços de um processo. O segmento da pilha é responsável pelo armazenamento de variáveis locais (pertencentes ao escopo de uma única função) e pelo registro de metadados usados pelo processador para controlar chamadas de funções. Um estouro de *buffer* na pilha caracteriza-se pela ultrapassagem dos limites de uma variável alocada na pilha, o que pode acarretar na sobrescrita do endereço de retorno de uma função e no consequente desvio do fluxo de execução do programa para um endereço arbitrário escolhido pelo atacante.

Isso ocorre porque – na popular arquitetura x86 – durante a chamada, execução e retorno de uma função qualquer, a pilha é organizada seguindo alguns passos convencionados. Tomando-se a Figura 2.1 como um exemplo de estrutura da pilha para uma função qualquer, ao ser chamada uma

outra função, a pilha assume a forma ilustrada na Figura 2.2, em decorrência da execução dos seguintes passos:

1. Antes de executar a instrução de chamada de uma função, são anotados na pilha os parâmetros para a função a ser chamada. No exemplo ilustrado, a função a ser chamada recebe dois argumentos;
2. Ao executar a instrução de chamada de função (CALL), o processador empilha o endereço de retorno (instrução subsequente à instrução de chamada da função);
3. O fluxo de execução é então desviado para a primeira instrução da função chamada;
4. O endereço base do *frame* pertencente à função chamadora (*base pointer*) é empilhado;
5. O valor do *base pointer* é atualizado, fazendo-o apontar para a base do *frame* pertencente à função chamada;
6. O ponteiro para o topo da pilha (*stack pointer*) é deslocado, alocando espaço para as variáveis locais pertencentes à função chamada;

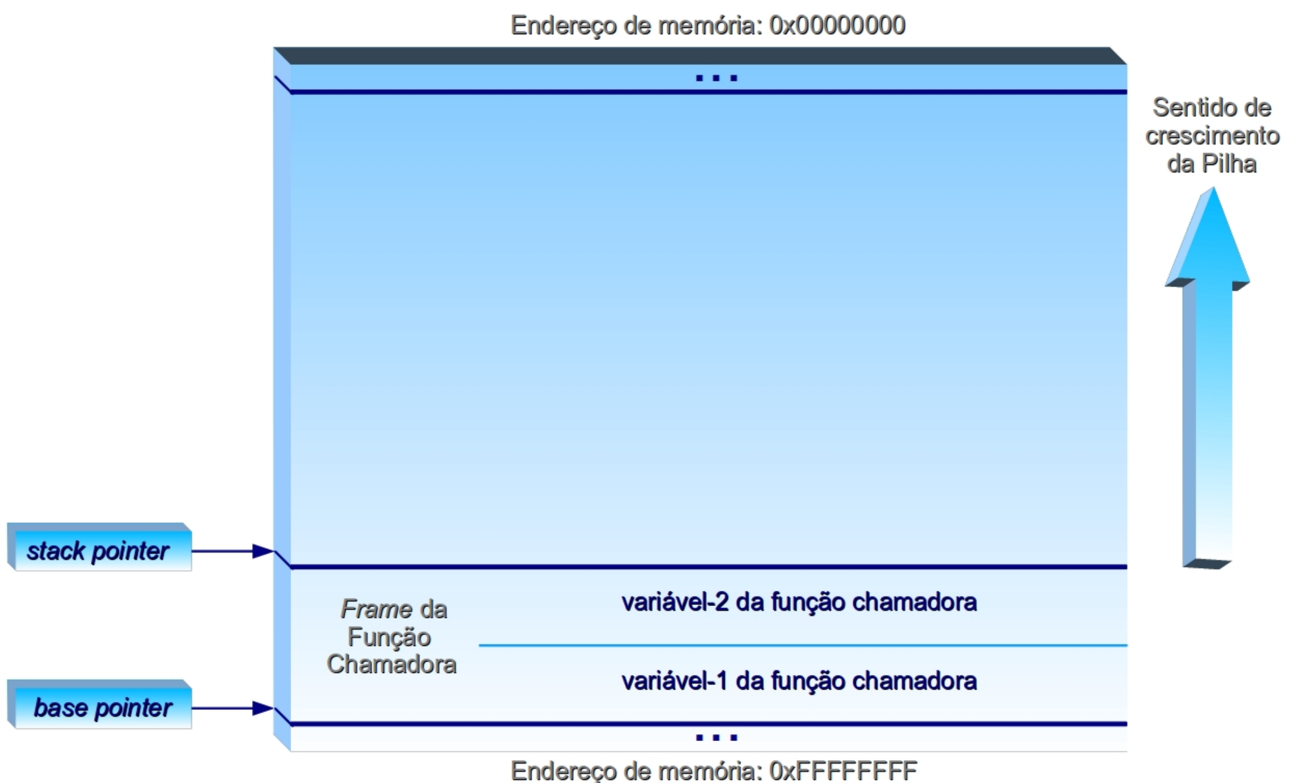

Figura 2.1: Estado da pilha antes de iniciar a chamada de uma nova função e após retornar da função.

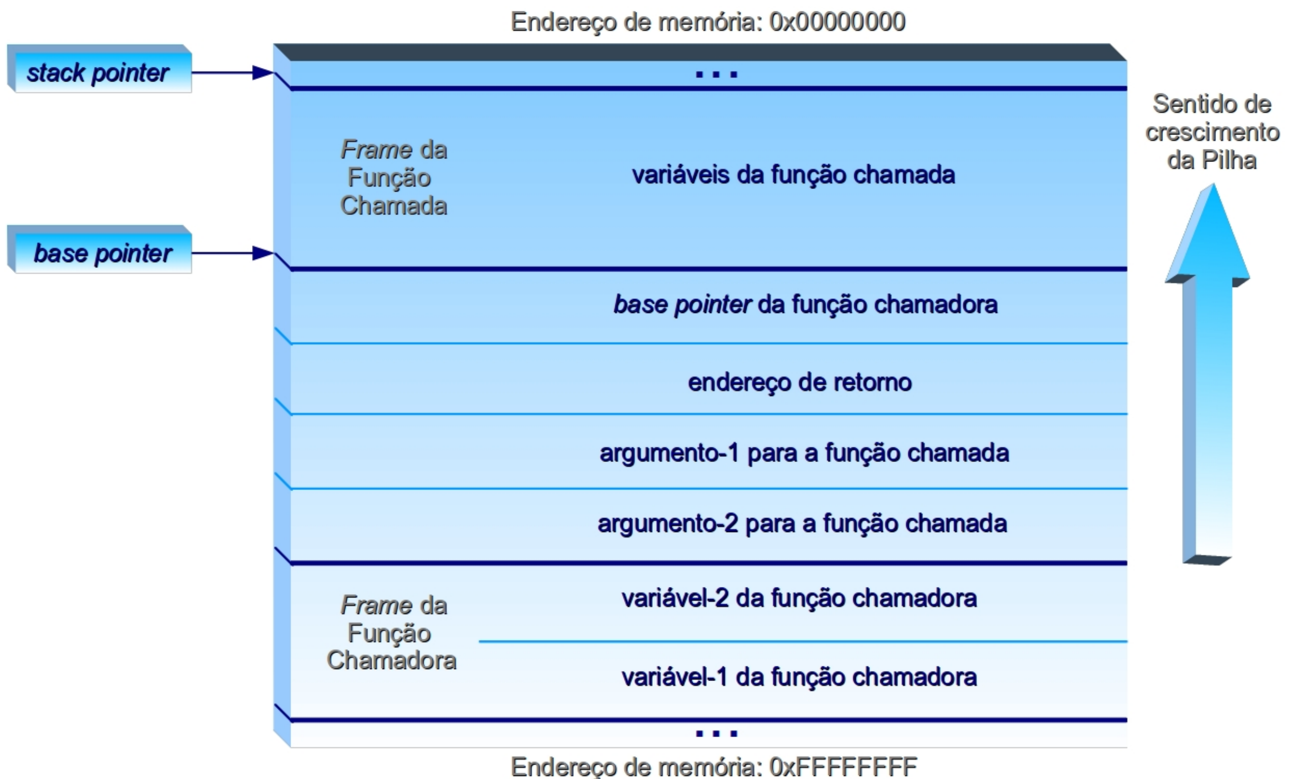

Figura 2.2: Estado da pilha após término dos passos executados durante uma chamada de função.

Após a execução desses passos, a pilha ilustrada na Figura 2.1 se transforma na pilha apresentada na Figura 2.2. O processo de retorno de uma função consiste em realizar as operações inversas ao procedimento de chamada de uma função. Portanto, depois de concluir todo o procedimento de retorno da função chamada, a pilha retorna ao estado indicado na Figura 2.1. Para isso, em um retorno de função, a pilha é organizada conforme os seguintes passos:

1. Depois de executar as instruções previstas em seu código e antes de retornar, a função chamada faz o registrador *stack pointer* apontar novamente para o endereço onde foi armazenado o endereço base do *frame* pertencente à função chamadora;

2. O endereço base do *frame* pertencente à função chamadora é desempilhado e restabelecido no registrador *base pointer*;

3. Finalmente, quando o processador executa a instrução de retorno de função (RET), o endereço de retorno é desempilhado e o fluxo de execução é desviado para esse endereço.

A técnica de exploração através do estouro de um *buffer* na pilha consiste em enviar como entrada para a aplicação uma quantidade de dados maior do que o espaço alocado na pilha para as variáveis locais da função, extravasando os limites do *frame* da função e sobrescrevendo o endereço de retorno. Assim, quando a instrução de desvio para o endereço de retorno (RET) for executada, o

fluxo de execução será transferido para um endereço escrito pelo atacante na pilha, conforme representado na Figura 2.3.

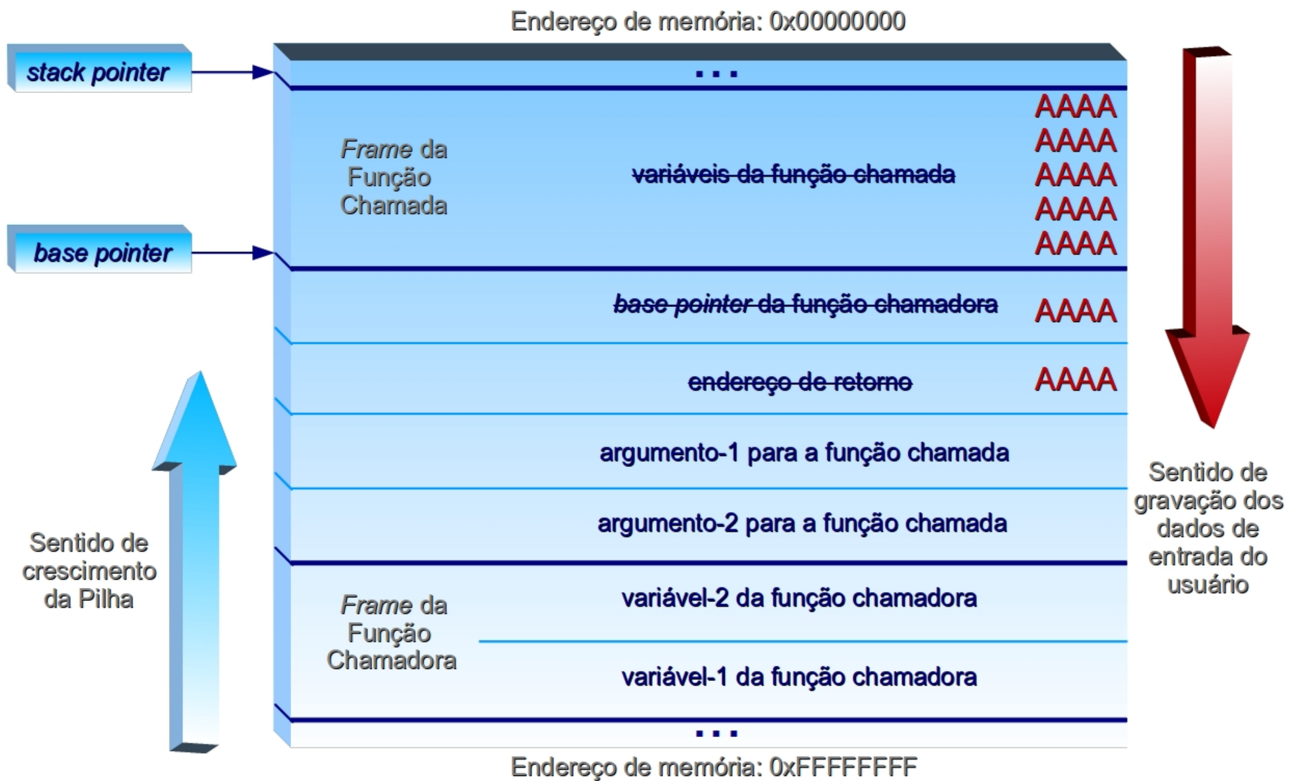

Figura 2.3: Estado da pilha após ataque de estouro de *buffer* na pilha.

Segundo Anley et al. [5], a primeira referência formal à técnica de *buffer overflow* foi apresentada publicamente em 1996 por Aleph One [29], em seu artigo intitulado "Smashing the Stack for Fun and Profit" (Estourando a pilha por diversão e profissão), onde o autor descreve em detalhes os passos para a exploração de vulnerabilidades de estouro de *buffer* na pilha. Além de ser a primeira técnica de *buffer overflow* detalhada em uma publicação científica, o estouro de pilha foi também a primeira vulnerabilidade explorada em um ataque de larga escala, produzido por Morris Worm, em 1988, quando pelo menos uma em cada vinte máquinas conectadas à Internet foi comprometida [30]. Desde então, uma série de variantes de ataques foi desenvolvida, de maneira a explorar também os demais segmentos de memória de um processo, além de outros tipos de vulnerabilidades [31].

Independente do tipo de defeito explorado, todas as falhas em softwares tornam-se cada vez mais severas, à medida que computadores, telefones celulares e outros dispositivos portáteis proliferam-se globalmente [32]. Os ataques globais são ainda impulsionados pela popularização das conexões de alta velocidade, que semeiam um solo fértil para que as investidas remotas se propaguem rapidamente e sejam de rastreamento bem mais complexo. De fato, a exploração remota

de vulnerabilidades tem sido a preferência dos atacantes, em contraposição aos ataques locais a sistemas, totalizando 92% das ocorrências, segundo estudo referente ao primeiro semestre de 2011 [33], realizado pela empresa Secunia, especializada no gerenciamento de vulnerabilidades relacionadas à Tecnologia da Informação.

2.2 Proteções de Memória

Assim como as estratégias para explorar vulnerabilidades evoluíram ao longo do tempo, surgiram e foram aprimorados mecanismos de proteção, que têm a incumbência de bloquear ou – pelo menos – dificultar a consolidação desses ataques. Essas estratégias de defesa focam diferentes etapas do processo de funcionamento de um sistema computacional, variando desde soluções que tratam da fase de compilação de um código-fonte até abordagens que propõem novos mecanismos de hardware.

Entre as classes de mitigações propostas contra ataques de exploração de softwares, os mecanismos de proteção da memória, implementados pelos sistemas operacionais na tentativa de impor barreiras contra a execução de códigos arbitrários, podem ser considerados os dispositivos que mais evoluíram ao longo do tempo. Além disso, a maior parte das técnicas modernas de desenvolvimento de *exploits*, incluindo ROP, surgiram da necessidade de adaptar os ataques para funcionamento em ambientes com proteções de memória.

A seguir, são discutidos sucintamente os principais aspectos relativos às proteções de memória que motivaram ou que influenciam diretamente na técnica de ataque ROP. Discussões sobre outros tipos de proteções de memória eficazes na defesa contra ataques de execução de código arbitrário são encontradas na literatura [34], mas não aparecem neste trabalho por não terem uma relação direta com a técnica ROP, fugindo do escopo deste trabalho.

2.2.1 Pilha não executável (*nx-stack*)

Como os ataques de estouro de pilha foram os primeiros a se popularizar, surgiu inicialmente a necessidade de se proteger a pilha contra a execução de códigos, principalmente porque a estratégia mais óbvia de exploração desse tipo de vulnerabilidade é a inserção do código malicioso na própria pilha, junto com o código que gera o transbordamento do *buffer*. Em decorrência da constatação dessa possibilidade, em 1996, surgiram dispositivos para impedir a execução de instruções oriundas da área de pilha (*non-executable stack* ou *nx-stack*). Atualmente,

esse tipo de proteção está presente por padrão na maioria dos sistemas operacionais, incluindo Linux, OpenBSD, Mac OS X, Solaris e Windows, entre outros [5].

2.2.2 Espaço aleatório de endereços (ASLR)

Outra estratégia interessante de proteção contra ataques baseia-se na ideia de impossibilitar o atacante de descobrir o endereço para o qual o fluxo do programa deve ser transferido, através do embaralhamento dos endereços de memória atribuídos aos segmentos do programa e às bibliotecas do sistema operacional. Essa estratégia recebeu o nome *Address Space Layout Randomization* (ASLR). Ela baseia-se no princípio de que se os endereços de memória forem atribuídos aleatoriamente aos módulos e aos seus segmentos, um atacante não saberá para qual endereço de memória desviar.

O ASLR é adotado pelo Linux desde o *kernel* 2.6.12. Nesse sistema – cada vez que um processo é carregado – seus segmentos, endereço base e bibliotecas recebem uma faixa de endereços diferente. No Windows, essa proteção foi implementada a partir do Vista e do Server 2008. No caso dos sistemas da Microsoft, o embaralhamento de endereços é renovado a cada reinicialização do sistema operacional [35].

2.2.3 Bit de execução (NX/XD)

Uma extensão natural do mecanismo de proibição da execução de instruções armazenadas na pilha (*nx-stack*), concebida para bloquear esse tipo de tentativa em outras áreas da memória – como o *heap* e o BSS –, recebeu a designação genérica “W⊕X” (ou Gravável ou Executável). Posteriormente, os fabricantes de processadores a rebatizaram de “NX” (*No eXecute, na AMD*) e “XD” (eXecute Disable, na Intel). Ela pode ser referenciada, ainda, por outros nomes, de acordo com sua implementação em cada sistema operacional (ex: DEP – *Data Execution Prevention,* no Windows). O Windows implementa o DEP desde o lançamento do *Service Pack* 2 para o XP. A tecnologia “NX/XD” também está presente no Linux, desde o *kernel* 2.6.8 [35].

Essa estratégia de proteção baseia-se na utilização de um recurso incorporado aos processadores, em 2004, para marcar as páginas de memória com um bit de execução [36]. Assim, o sistema operacional pode garantir que áreas de memória destinadas a armazenar dados – como a pilha e o *heap* – não sejam executáveis. Isso impede que, ao explorar vulnerabilidades em softwares, um atacante transfira o fluxo de execução do programa alvo diretamente para códigos maliciosos injetados nessas áreas de memória.

O "NX/XD" torna praticamente impossível a injeção de código externo em programas vulneráveis. Entretanto, técnicas de reutilização de código, como ROP, explicada a seguir, podem obter êxito contra um sistema protegido apenas pelo bit de execução.

2.3 *Return-Oriented Programming* (ROP)

Nesta Seção são apresentadas as motivações históricas para o surgimento dos ataques ROP e discutidos os conceitos essenciais para a compreensão do funcionamento dessa técnica, incluindo a explicação detalhada de um *exploit* real, disponibilizado em um repositório público de códigos de exploração.

2.3.1 Histórico

Tão logo as primeiras proteções contra ataques de injeção de código começaram a ser incorporadas aos sistemas computacionais, surgiram propostas de ataques alternativos baseados no reaproveitamento dos códigos originais das aplicações [37]. Nos primeiros ataques de reúso de código apresentados, a biblioteca padrão de C (*libc*) foi utilizada como alvo dos desvios do fluxo de execução, uma vez que, nos sistemas operacionais derivados do UNIX, praticamente todos os programas carregam essa biblioteca. Além disso, a biblioteca *libc* contém várias rotinas que redirecionam a execução para chamadas de sistema úteis em ataques. Por conta disso, esses ataques ficaram conhecidos como "return-into-libc" (RILC) [38].

Apesar da biblioteca *libc* ter sido durante muito tempo o alvo preferido dos ataques de reúso de código, qualquer código disponível, tanto no segmento de código executável do programa, quanto na área de instruções pertencente a uma outra biblioteca carregada, pode ser utilizado. Baseado nessa constatação, e na crença de que os ataques que desviam o fluxo de execução diretamente para funções disponíveis na biblioteca *libc* limitam as possibilidades de execução de código, surgiram ideias para interligar funções [39]–[41] ou trechos de código executáveis [42]–[46]. Ao ser demonstrado que o encadeamento desses trechos de código permite a execução de computações arbitrárias (*Turing complete computation*) [47], essa técnica se popularizou entre atacantes e passou a ser referida pela nomenclatura *Return-Oriented Programming*, ou simplesmente ROP.

Em 2011, foi publicado um trabalho que apresenta uma nova variante do clássico ataque de reúso de código RILC, denominado TC-RILC (*Turing Complete Return Into Libc*) [38]. Nesse ataque, o autor consegue criar códigos arbitrários (assim como no ROP – daí a expressão *Turing*

Complete) através do encadeamento de chamadas a funções da biblioteca *libc*. Esse ataque tem um impacto relevante em boa parte das proteções elaboradas para detectar ataques ROP, uma vez que essas proteções não são capazes de detectar ataques TC-RILC. No entanto, segundo os próprios autores, a estratégia de defesa baseada em uma pilha sombra, discutida na Seção 3.1.2, é capaz de impedir os ataques TC-RILC. Além disso, a estratégia de checagem da instrução anterior ao endereço de retorno, detalhada na Seção 3.1.3, também é capaz de detectar esse ataque.

2.3.2 Funcionamento

A técnica de desenvolvimento de *exploits* ROP baseia-se no reúso de código para superar a proteção oferecida pelo bit de execução (NX/XD). Ao contrário da tradicional técnica de reúso de código "return-into-libc" (RILC), na qual o atacante desvia o fluxo de execução para o início de alguma função útil para o ataque (normalmente disponível na biblioteca "libc"), o ROP encadeia vários pequenos trechos de código (*gadgets*) a fim de executar uma determinada tarefa. Para conseguir esse encadeamento, a última instrução de cada trecho de código escolhido deve executar um desvio. A ideia original do ROP utiliza *gadgets* finalizados com instruções de retorno (RET) para interligar as frações de código escolhidas [47]. Daí surgiu o nome da técnica.

A vantagem do ROP em relação ao RILC decorre da ampliação das possibilidades para o *shellcode*[4], pois ao usar a segunda técnica, as ações do atacante ficam limitadas às funções carregadas na memória. Na verdade, sabe-se que o ataque TC-RILC (mencionado na Seção anterior) também possibilita a execução de computações arbitrárias (*Turing complete computation*) através do encadeamento de chamadas de sistema disponíveis na biblioteca "libc" [38]. No entanto, até onde se tem notícia, essa técnica ainda não foi empregada em *exploits* públicos, possivelmente porque ela impõe severas restrições de desempenho ao código malicioso. Essas restrições dificultam o emprego prático do TC-RILC em ataques reais, uma vez que um *shellcode* que consome muito tempo de execução pode induzir o usuário a encerrar a aplicação, imaginando que o processo está "travado". Naturalmente, esse encerramento da aplicação implica na interrupção do ataque.

Para elaborar mecanismos de proteção contra qualquer tipo de ataque computacional, é importante entender quais os caminhos podem ser seguidos pelos atacantes, bem como as características que determinam as escolhas desses invasores. Por isso, a fim de permitir um amplo entendimento dos aspectos considerados pelos atacantes ao elaborarem ataques ROP, será apresentado em detalhes o exemplo de um *exploit* real criado para atacar um software executado por

[4] *Shellcode*: conjunto de instruções que, ao serem executadas pelo processador, efetuam alguma atividade maliciosa.

processadores da família x86 em sistemas Windows XP SP3. Contudo, antes de analisar o código do *exploit*, é necessário entender o funcionamento exato do DEP no Windows.

O encadeamento de códigos, efetuado no ROP antes de desviar o fluxo de execução para o *shellcode*, pode ter como objetivo realizar diversas tarefas: habilitar o bit de execução para a região de memória onde o *shellcode* se localiza, copiar o *shellcode* para uma área de memória com permissão de execução ou desabilitar a proteção oferecida pelo bit NX/XD. Nos sistemas operacionais da Microsoft, em função da grande quantidade de incompatibilidade de aplicações com o DEP, por padrão essa proteção não é habilitada para todos os processos. Ao invés disso, o administrador do sistema pode escolher entre quatro políticas de uso, detalhadas na Tabela 2.1.

Tabela 2.1: Opções de configuração do DEP [48], [49]

Opção	Descrição
OptIn	Configuração padrão nas versões XP, Vista, 7 e 8 do Windows. O DEP é habilitado para alguns binários do sistema e para programas incluídos pelo administrador do sistema na lista de opção por usar a proteção (opt-in).
OptOut	Configuração padrão nas versões Server do Windows. O DEP é habilitado para todos os processos, exceto aqueles incluídos pelo administrador do sistema na lista de opção por não usar a proteção (opt-out).
AlwaysOn	O DEP é habilitado para todos os processos, sem exceções.
AlwaysOff	O DEP é desabilitado para todos os processos, sem exceções.

As políticas de uso do DEP definem também se um processo pode alterar sua própria opção de configuração para essa proteção. Se as opções “AlwaysOn” ou “AlwaysOff” estiverem ativas, nenhum processo pode alterar suas configurações relativas ao DEP. Por outro lado, se as opções “OptIn” ou “OptOut” estiverem ativas (e se o “Permanent DEP” – explicado a seguir – estiver desativado), o processo poderá chamar a função "NtSetInformationProcess" [50] ou a função "SetProcessDEPPolicy" [51] para alterar sua opção de configuração. Ressalta-se, porém, que a função “SetProcessDEPPolicy” só pode ser chamada uma vez por cada processo. Portanto, se essa função já tiver sido chamada pelo processo atacado, o *exploit* não funcionará caso efetue uma nova chamada. Isso ocorre, por exemplo, com o Internet Explorer 8, que chama a função “SetProcessDEPPolicy” assim que o programa inicia.

Além das quatro opções de configuração do DEP, a partir do Windows Vista, a Microsoft incorporou um mecanismo denominado “Permanent DEP”, que é ativado automaticamente para os executáveis compilados com a opção /NXCOMPAT. Essa opção também pode ser ativada individualmente pelos processos através de uma chamada à função “SetProcessDEPPolicy”. O “Permanent DEP” tem impacto direto na escolha da estratégia usada para superar o DEP, pois quando esse indicador está ativo, nenhuma função pode ser usada para alterar a política de DEP configurada para o processo [52].

Quando o “permanent DEP” ou a opção “AlwaysOn” estão ativos, os atacantes recorrem a outras estratégias para burlar o DEP. Uma delas é chamar a função “VirtualProtect” para marcar como executável a página de memória onde o *shellcode* se localiza. Outra abordagem consiste em executar a função “WriteProcessMemory”, que permite copiar o *shellcode* para uma localização executável da memória, desde que essa região também seja gravável. Se isso não for viável, existe ainda a possibilidade de utilizar a função “VirtualAlloc” ou a função “HeapCreate” para criar uma nova região de memória com permissões de execução e escrita. Após criar essa área, basta copiar o *shellcode* para lá através de uma chamada a funções como “memcpy” ou “WriteProcessMemory”.

Acontece que, para chamar as funções que permitem superar o DEP através das estratégias mencionadas, é necessário preparar os parâmetros a serem submetidos para essas APIs. Ao explorar uma vulnerabilidade do tipo estouro de *buffer* na pilha, por exemplo, como os argumentos também são anotados na pilha, teoricamente bastaria inseri-los na pilha junto com os dados que acarretam no estouro e executar a chamada direta da função (equivalente à técnica RILC). Entretanto, endereços previamente desconhecidos (como a localização exata do *shellcode*) ou valores que contenham bytes nulos (0x00) usualmente exigem a execução prévia de instruções para carregar o valor desejado na pilha. Por isso, a preparação dos argumentos requer uma seleção minuciosa de *gadgets* para o encadeamento de instruções, além da escolha precisa dos valores a serem escritos durante o *overflow*. Esses *gadgets* devem manipular os dados inseridos pelo atacante na memória a fim de gerar os parâmetros adequados para chamar-se a API escolhida. O grande desafio imposto pela técnica ROP reside no fato que, na maioria das vezes, ao executar instruções para preparar e escrever um valor na pilha, acaba-se alterando outros valores em registradores ou na própria pilha. Por isso, a escolha dos *gadgets* deve ser bastante criteriosa.

Os *gadgets* podem ser compostos por instruções encontradas em qualquer módulo, desde que essas instruções sejam executáveis, localizem-se em um endereço conhecido e não contenham bytes nulos (essa última restrição pode ser desconsiderada caso os dados de entrada não sejam lidos como *strings*). Além disso, como a arquitetura x86 utiliza instruções de tamanhos variados, não há nenhuma exigência para que os acessos ao segmento de instruções (.text) respeitem algum tipo de alinhamento. Assim, é possível utilizar partes de uma instrução original como se fossem novas instruções. Uma instrução “ADD AL, 0×58” (0×80C058), por exemplo, se for referenciada a partir do seu terceiro byte (0x58), será interpretada como uma instrução “POP EAX” (0x58). Esse artifício é constantemente utilizado pelos atacantes durante a pesquisa por *gadgets* úteis para a construção de uma cadeia de instruções.

Um exemplo de *exploit* que utiliza a função “SetProcessDEPPolicy” para desativar a proteção oferecida pelo DEP é apresentado na Listagem 2.1. Nesse artefato, divulgado em 17 de setembro de 2011, é explorada uma vulnerabilidade de estouro de pilha no software “My MP3 Player” [53][5]. Conforme indicado no próprio código, o ataque realizado por esse *exploit* funciona contra versões do Windows XP SP3, com as opções OptIn (incluindo o “My MP3 Player” na lista) ou OptOut do DEP habilitadas.

Listagem 2.1. Código de *exploit* ROP que utiliza a função “SetProcessDEPPolicy” para desativar o DEP.

```
1  # calc.exe - 1014 bytes of space for shellcode
2  shellcode =(
3  "\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49\x49\x51\x5a\x56\x54”
4  "\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58"
5  "\x32\x42\x44\x42\x48\x34\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44"
6  "\x41\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4a\x4e\x46\x44\x42\x30\x42\x50"
7  "\x42\x30\x4b\x48\x45\x54\x4e\x43\x4b\x38\x4e\x47\x45\x50\x4a\x57\x41\x30\x4f\x4e\x4b"
8  "\x58\x4f\x54\x4a\x41\x4b\x38\x4f\x45\x42\x42\x41\x50\x4b\x4e\x49\x44\x4b\x38\x46\x33"
9  "\x4b\x48\x41\x50\x50\x4e\x41\x53\x42\x4c\x49\x59\x4e\x4a\x46\x58\x42\x4c\x46\x57\x47"
10 "\x30\x41\x4c\x4c\x4c\x4d\x30\x41\x30\x44\x4c\x4b\x4e\x46\x4f\x4b\x53\x46\x55\x46\x32"
11 "\x46\x50\x45\x47\x45\x4e\x4b\x58\x4f\x45\x46\x52\x41\x50\x4b\x4e\x48\x56\x4b\x58\x4e"
12 "\x50\x4b\x44\x4b\x48\x4f\x55\x4e\x41\x41\x30\x4b\x4e\x4b\x58\x4e\x41\x4b\x38\x41\x50"
13 "\x4b\x4e\x49\x48\x4e\x45\x46\x32\x46\x50\x43\x4c\x41\x33\x42\x4c\x46\x46\x4b\x38\x42"
14 "\x44\x42\x53\x45\x38\x42\x4c\x4a\x47\x4e\x30\x4b\x48\x42\x44\x4e\x50\x4b\x58\x42\x37"
15 "\x4e\x51\x4d\x4a\x4b\x48\x4a\x36\x4a\x30\x4b\x4e\x49\x50\x4b\x38\x42\x58\x42\x4b\x42"
16 "\x50\x42\x50\x42\x50\x4b\x38\x4a\x36\x4e\x43\x4f\x45\x41\x53\x48\x4f\x42\x46\x48\x35"
17 "\x49\x38\x4a\x4f\x43\x48\x42\x4c\x4b\x57\x42\x45\x4a\x36\x42\x4f\x4c\x38\x46\x30\x4f"
18 "\x35\x4a\x46\x4a\x39\x50\x4f\x4c\x38\x50\x50\x47\x55\x4f\x4f\x47\x4e\x43\x46\x41\x46"
19 "\x4e\x46\x43\x36\x42\x50\x5a")
20
21 buffer = "\x41" * 1024
22 eip = "\x99\x13\x09\x5d"       # RETN - COMCTL32
23 rop = "\x42" * 4                                          # junk to compensate
24 rop += "\x8c\x39\x09\x5d"      # POP EBX, RETN - COMCTL32
25 rop += "\xff\xff\xff\xff"
26 rop += "\x28\x90\x12\x77"      # INC EBX, RETN - OLEAUT32
27 rop += "\x44\x94\x12\x77"      # POP EBP, RETN - OLEAUT32
28 rop += "\x44\x21\x86\x7c"      # SetProcessDEPPolicy
29 rop += "\x36\x1c\x12\x77"      # POP EDI, RETN – OLEAUT32
30 rop += "\x37\x1c\x12\x77"      # RETN - OLEAUT32
31 rop += "\xd4\x1a\x12\x77"      # POP ESI, RETN - OLEAUT32
32 rop += "\x37\x1c\x12\x77"      # RETN - OLEAUT32
33 rop += "\xf7\x8c\x14\x77"      # PUSHAD, RETN - OLEAUT32
34 nops = "\x90" * 20
35 junk = "\x42" * (2000 - len(nops + shellcode + rop))
36
37 print "[+] Creating malicious .m3u file"
38 try:
39     file = open("exploit.m3u","w")
40     file.write(buffer + eip + rop + nops + shellcode + junk)
41     file.close()
42     print "[+] File created"
43 except:
44     print "[x] Could not create file"
45
46 raw_input("\nPress any key to exit...\n")
```

Na primeira linha do *exploit*, desenvolvido na linguagem de programação Python, aparece um comentário que indica o tipo de *shellcode* utilizado: um código que apenas executa o software “calc.exe” (calculadora) do Windows. Nas linhas de 02 a 19, é inicializada a variável que armazena

[5] O endereço da função “SetProcessDEPPolicy” (0x7C862144) foi alterado em relação ao valor contido no *exploit* original para refletir sua localização no Windows XP SP3 English, usado durante os testes.

o código do *shellcode*. Repare que os bytes correspondentes às instruções de máquina são representados em hexadecimal.

Na linha 21 do *exploit*, a variável *buffer* é inicializada com 1024 bytes, necessários para preencher o espaço compreendido entre o *buffer* estourado e o endereço de retorno da função. Em seguida (linha 22), a variável "eip" guarda o valor a ser escrito no endereço de retorno. Como o DEP está ativo para o processo atacado, não é possível usar a estratégia tradicional em estouros de pilha, que consiste em saltar direto para o *shellcode* armazenado na própria pilha (usualmente desviando para o endereço apontado por ESP com uma instrução da forma "JMP ESP"). Antes de saltar para o *shellcode*, o *exploit* trata de desabilitar o DEP. Para isso, o valor que irá sobrescrever o endereço de retorno corresponde ao endereço de uma instrução RETN existente na biblioteca "COMCTL32" (0x5D091399). Esse é o primeiro *gadget* executado pelo *payload* malicioso, assim que o fluxo de execução é subvertido.

Em seguida, a cadeia de instruções ROP é construída na variável denominada "rop" (linhas 23 a 33). A Figura 2.4 ilustra o formato da cadeia ROP estabelecida nesse *exploit*. As setas tracejadas indicam o momento em que cada endereço ou dado armazenado na pilha é utilizado por uma instrução. As setas contínuas representam a sequência de execução dos *gadgets*.

O primeiro valor escrito na variável "rop" (linha 23 da Listagem 2.1) corresponde a um simples ajuste de deslocamento na pilha, pois a instrução de retorno "RETN 4" (endereço 0x0050A1DD), que dá início à exploração ao saltar para o endereço sobrescrito na pilha (0x5D091399), realiza também o incremento do ponteiro de topo da pilha (ESP) em 4 unidades. Na linha 24, é inserido o endereço do segundo *gadget*. A partir desse ponto, os valores escritos na variável "rop", usados para sobrescrever a pilha conforme apontado na Figura 2.4, são endereços para *gadgets* ou valores que os *gadgets* utilizam para construir, na própria pilha, os parâmetros de chamada da função "SetProcessDEPPolicy".

Para desabilitar o DEP, a função "SetProcessDEPPolicy" recebe apenas um parâmetro, de valor zero. Acima desse valor é colocado o endereço inicial da sequência de instruções NOP (*No Operation*) que precedem o *shellcode* (linha 34 do *exploit*). Esse endereço é usado pela função como endereço de retorno, o que garante a execução do *shellcode* imediatamente após o DEP ser desabilitado.

Para estabelecer essa estrutura, o *exploit* utiliza a instrução PUSHAD, que empilha de uma única vez o valor de todos os registradores de uso geral na pilha, na seguinte ordem: EAX,

ECX, EDX, EBX, ESP, EBP, ESI, EDI. Esse empilhamento é representado na Figura 2.4 pela "Pilha após PUSHAD". Os valores de EAX, ECX e EDX são irrelevantes. EBX é usado para guardar o valor zero, passado como parâmetro para a função "SetProcessDEPPolicy". Como não é possível inserir bytes nulos diretamente na pilha, o valor 0xFFFFFFFF (-1) é carregado em EBX por um *gadget* que contém a instrução "POP EBX". Em seguida, é usado um *gadget* com a instrução "INC EBX" para incrementar o valor desse registrador, tornando-o igual a zero (4 bytes nulos).

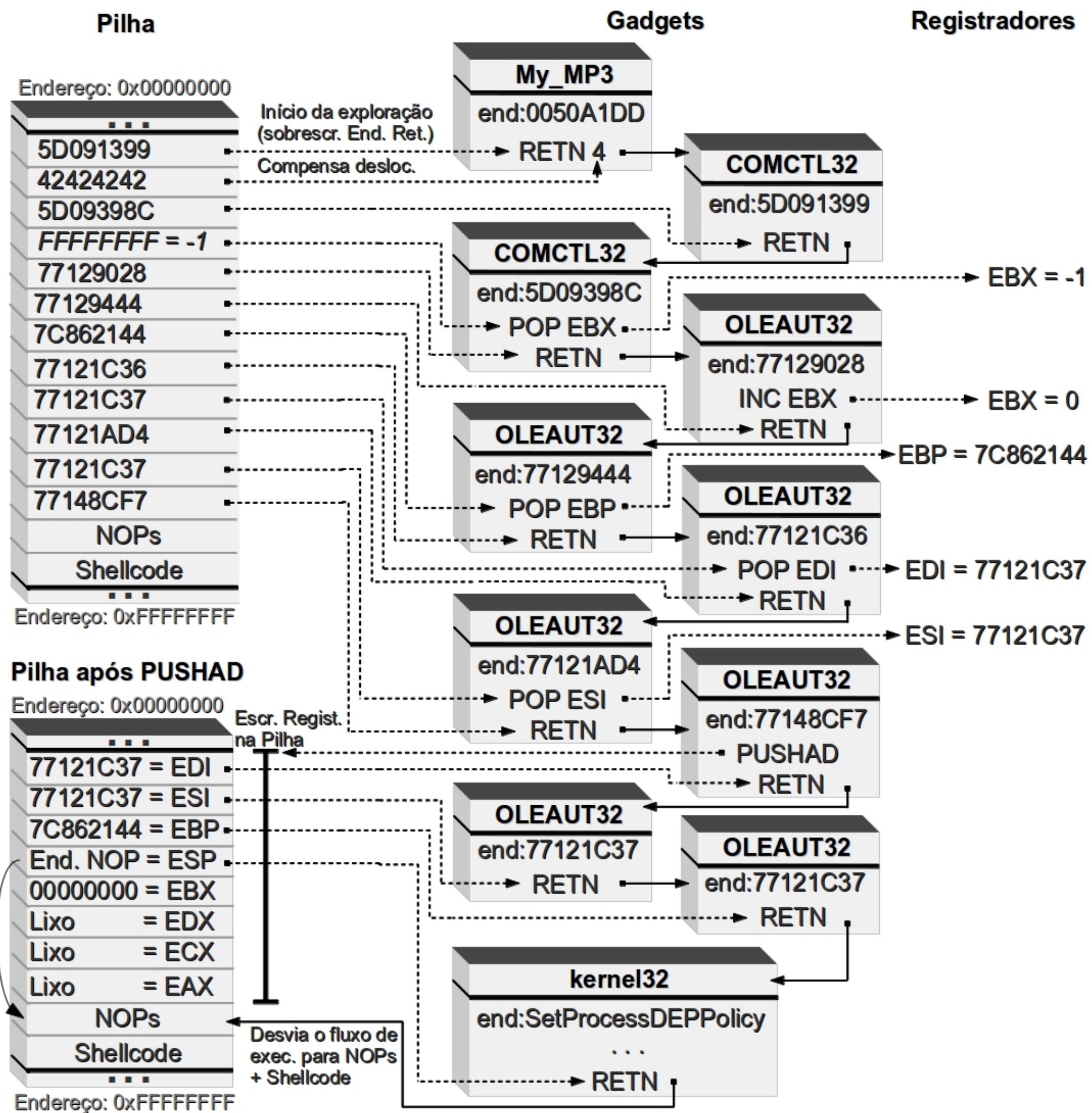

Figura 2.4: Estrutura da pilha e fluxo entre *gadgets* do *exploit* apresentado na Listagem 2.1.

O valor de ESP anotado na pilha corresponde à região da pilha exatamente anterior aos valores anotados pela instrução PUSHAD. Esse valor é aproveitado como endereço de retorno a ser passado para a função "SetProcessDEPPolicy", forçando-a a retornar o fluxo de execução direto para o *shellcode* (precedido por NOPs). Os registradores EBP, ESI e EDI são sobrescritos por *gadgets* compostos por instruções do tipo "POP", que carregam valores da pilha direto nesses

registradores. EBP é usado para armazenar o endereço da função a ser chamada (0x7C862144). Os registradores ESI e EDI guardam o endereço de um *gadget* composto apenas pela instrução RETN. Esse *gadget* é usado duas vezes para deslocar o ponteiro para o topo da pilha em 8 bytes (2 deslocamentos de 4 bytes). Após esses deslocamentos, o apontador para o topo da pilha estará apontando para o endereço da função “SetProcessDEPPolicy”, que finalmente é chamada ao executar a instrução RETN contida no penúltimo *gadget*.

Ao terminar sua execução, a função “SetProcessDEPPolicy” usa o endereço de retorno armazenado na pilha pela instrução PUSHAD (correspondente ao valor contido em ESP no momento em que a instrução PUSHAD executa), o que desvia o fluxo de execução para o *shellcode*. Como nesse momento o DEP já foi desativado pela função “SetProcessDEPPolicy”, o *shellcode* executa normalmente, mesmo estando localizado na pilha.

Na linha 35 do *exploit*, é criada uma variável de nome “junk”, que tem a função apenas de completar o tamanho do *shellcode* para garantir que o arquivo malicioso criado pelo *exploit* tenha 2000 bytes. Da linha 36 à 46, o arquivo de saída é escrito e são impressas na tela mensagens indicando a criação do arquivo denominado “exploit.m3u”. Esse arquivo simula uma lista de músicas a serem reproduzidas pelo software. Ao ler os dados desse arquivo, o software “My MP3 Player” desencadeia o ataque, que executa a calculadora do Windows (*shellcode*).

Além do Windows, o ROP é utilizado para desenvolver *exploits* direcionados a outros sistemas operacionais que implementam proteções baseadas no bit NX/XD. Normalmente, assim como no Windows, a técnica ROP é empregada em uma etapa inicial dos ataques, necessária para organizar os parâmetros a serem passados para as funções. A diferença, portanto, reside nas funções disponíveis para a conclusão dos ataques e em como essas funções recebem os parâmetros [54].

O uso do ROP para executar códigos arbitrários foi inicialmente demonstrado para a arquitetura x86, executando o sistema operacional Linux e utilizando instruções de retorno (RET) para interligar os *gadgets*. Posteriormente, seu emprego foi demonstrado em ataques a outros ambientes, tais como: Atmel AVR [55], PowerPC [56], Z80 e Sequoia AVC Advantage [10], ARM [11] e iPhone [57], [58]. Outros ataques de reúso de código foram propostos, tais como “JIT *Spraying*” [59] e “DCG (*Dynamic Code Generation*) *Spraying*” [60]. No entanto, essas estratégias não guardam uma relação direta com a técnica ROP e, por isso, não são abordadas neste trabalho.

3 Trabalhos Relacionados

Confirmando a tradicional “queda de braço” travada entre atacantes e defensores nos diversos assuntos que compõem a área de segurança de sistemas computacionais, desde que a técnica de ataque ROP foi introduzida, uma grande quantidade de trabalhos têm sido desenvolvidos para propor estratégias de mitigação contra essas investidas. Na Seção 3.2, as estratégias gerais de proteção contra ataques ROP que vêm sendo empregadas pelos trabalhos já publicados são discutidas e classificadas segundo uma nova taxonomia.

Além de classificar as principais estratégias, também introduzimos na Seção 3.3 uma proposta de classificação das soluções segundo a abordagem utilizada por elas para implementar as estratégias escolhidas. Ambas as classificações agrupam as proteções de acordo com características semelhantes, o que facilita a compreensão dos fatores que influenciam a qualidade dos esquemas de proteção. Para contrapor as classes de proteções, bem como os trabalhos específicos pertencentes a cada uma das classes, foram definidas métricas que visam refletir a qualidade das soluções propostas (Seção 3.1).

3.1 Métricas de Avaliação

Uma vez que quando este trabalho foi iniciado não existia um conjunto bem definido de métricas destinadas a exprimir os principais fatores a serem considerados durante a avaliação de proteções contra ataques ROP, buscou-se estabelecer esse conjunto. As métricas aqui propostas foram utilizadas durante a análise das características, virtudes e deficiências inerentes a cada um dos trabalhos avaliados. Da mesma forma, acredita-se que elas constituem valiosa ferramenta no processo de análise dos benefícios e limitações inerentes a novas proteções contra ataques ROP. As seguintes métricas foram consideradas relevantes:

- Tipos de ataques bloqueados.
- *Overhead* médio.
- Exceções.
- Viabilidade prática do protótipo.

A seguir, cada uma dessas métricas é detalhada.

Tipos de ataques bloqueados

Inicialmente, os ataques ROP foram concebidos através da interligação de *gadgets* finalizados pela instrução RET. A capacidade de controlar o conteúdo da pilha em ataques de injeção de código arbitrário, aliada à vasta disponibilidade dessa instrução nos arquivos binários, faziam dessa operação a ferramenta ideal para interligar o fluxo de execução entre os trechos de código que compõem um ataque ROP [8]. Como todos eles guardavam essa característica em comum, as primeiras propostas de proteções contra esses códigos maliciosos destinavam-se a analisar aspectos específicos relacionados ao comportamento da instrução de retorno [61]–[64].

Posteriormente, novos trabalhos demonstraram a possibilidade de utilização de instruções do tipo "jump indireto" (JMP) para encadear os *gadgets*, ataque que foi batizado de JOP (*Jump-Oriented Programming*) [12], [65]–[69]. Apesar de não ter sido demonstrado que é possível executar computações arbitrárias (*Turing complete computation*) interligando apenas *gadgets* terminados com a instrução do tipo "chamada de função" (CALL), essas instruções de desvio indireto também apresentam a capacidade de interligar *gadgets* e podem, portanto, ser utilizadas em ataques ROP [65].

Em decorrência da existência dessa variedade de instruções úteis na construção de *exploits* ROP, este trabalho propõe a distinção dos tipos de ataques bloqueáveis por cada proteção como a primeira métrica de avaliação das soluções. Ao analisar o funcionamento de um mecanismo de defesa, sugere-se indicar se a proteção é capaz de identificar cadeias de *gadgets* interligadas pelas instruções RET, JMP ou CALL. Essa distinção permite separar com clareza os tipos de códigos maliciosos bloqueados, fator preponderante na definição do grau de segurança oferecido por cada solução.

***Overhead* médio**

Um dos principais fatores impactados pela implementação de mecanismos de segurança em sistemas computacionais é a elevação da carga de trabalho (*overhead*) e a consequente redução do desempenho do sistema. Naturalmente, esse é um aspecto que também atinge as proteções contra ataques ROP e que, portanto, deve ser considerado. Por isso, propõe-se estabelecer a perda de desempenho que a solução impõem às aplicações protegidas como a segunda métrica de avaliação a ser considerada.

Como existe um consenso de que a degradação do desempenho é um fator importante para a viabilidade prática de soluções de segurança em sistemas computacionais, a maioria dos

trabalhos que propõem proteções contra ataques ROP apresentam estudos do impacto dessas proteções na performance dos sistemas. Em todos os trabalhos avaliados neste estudo, o *overhead* computacional é expresso através da porcentagem média de tempo de execução adicional imposto pela solução. Ou seja, se uma aplicação executava em média em 10 segundos e, após a implementação da proteção contra ataques ROP, passa a executar em 11 segundos, o *overhead* médio imposto por essa solução é de 10%. Essa é a forma de representar o *overhead* adotada ao longo deste trabalho.

Exceções

Muitas proteções contra ataques ROP baseiam-se em convenções de codificação ou padrões de compilação para aplicar suas estratégias de detecção de explorações. No entanto, é comum ocorrerem situações que fogem do padrão de execução estabelecido para uma arquitetura ou sistema operacional. Em geral, esses casos decorrem de códigos personalizados escritos diretamente em linguagem de montagem (*assembly*). A opção por desenvolver códigos em *assembly* costuma estar relacionada a uma necessidade de otimização de desempenho ou à exigência de adaptação às funcionalidades de um hardware específico.

Infelizmente, já foi observado que, em uma vasta quantidade de ocasiões, as convenções nas quais algumas proteções contra ataques ROP se baseiam não são respeitadas. Essa disparidade entre o padrão esperado por uma solução e o formato diferenciado adotado por alguns códigos acarreta em situações em que códigos autênticos são classificados equivocadamente. Em outras palavras, trechos de instruções que deveriam executar normalmente podem ser bloqueados ou corrompidos, resultando em falsos positivos.

Apesar de algumas proteções tratarem as exceções encontradas nos ambientes onde seus protótipos foram testados, é difícil prever quais exceções podem surgir em ambientes com outros sistemas operacionais e aplicações. Por isso, em geral quanto maior o número de exceções catalogadas para uma determinada solução, menor é o seu grau de confiabilidade. Pensando nisso, estabeleceu-se o número de exceções (falsos positivos) de cada solução como a terceira métrica de avaliação.

Enquanto alguns trabalhos apresentam exemplos de situações que acarretaram em exceções durante os testes de seus protótipos, outros estudos, apesar de claramente sujeitos às mesmas situações, não indicam essa peculiaridade. Por isso, registrar apenas os casos de falsos positivos explicitados nas respectivas publicações não resultaria em uma estimativa precisa. Por

conta disso, adotou-se neste trabalho uma metodologia alternativa para classificação das proteções quanto ao número de exceções. Inicialmente, os casos de falsos positivos relatados na literatura foram catalogados e relacionados a cada uma das estratégias de proteção (discutidas na Seção 3.2 deste texto). Em seguida, analisou-se o funcionamento de cada mecanismo de proteção para identificar-se, entre as exceções catalogadas, quais delas afetam cada uma das soluções analisadas.

É importante ressaltar que as exceções identificadas não contabilizam os casos que representam possíveis brechas de segurança deixadas pelas soluções propostas. Por se tratar de observações que não foram testadas, essas situações possivelmente exploráveis por atacantes para superar algumas das proteções avaliadas não são contabilizadas na contagem de exceções. Apesar disso, elas são discutidas ao longo do texto na oportunidade em que as proteções são descritas.

Viabilidade prática do protótipo

Alguns mecanismos de defesa avaliados apresentam uma boa qualidade de proteção quando analisados sob a ótica das três primeiras métricas estabelecidas, mas pecam no quesito viabilidade prática da solução para proteção de aplicações reais. Por outro lado, existem soluções que, apesar de apresentar deficiências em algumas das métricas anteriormente definidas, são capazes de oferecer uma proteção imediata para sistemas em ambientes de produção. Diante disso, constatou-se a necessidade de incluir no modelo de avaliação a métrica de viabilidade prática do protótipo, que reflete a possibilidade de emprego imediato da solução para proteção de softwares em pleno funcionamento. Essa pode ser uma característica importante em situações na qual o avaliador deseja apontar a melhor solução dentre aquelas que permitem um emprego imediato. Esse cenário pode ocorrer, por exemplo, no caso da necessidade de proteger aplicações que contenham vulnerabilidades para as quais ainda não tenham sido disponibilizadas atualizações de segurança.

Entre as soluções avaliadas neste trabalho, algumas apresentam apenas proteções teóricas e não implementam protótipos. Por isso, foram consideradas inviáveis na prática. As proteções que exigem a disponibilidade da tabela de símbolos dos executáveis também foram consideradas inviáveis na prática, já que a maioria dos executáveis não guarda essa informação. Além delas, proteções que exigem adaptações na estrutura do hardware também foram consideradas inviáveis, já que não podem ser empregadas de imediato.

3.2 Estratégias de Proteção

Neste trabalho observou-se que as proteções contra ROP têm utilizado estratégias semelhantes de detecção do ataque. Em função disso, propõe-se o agrupamento das soluções em classes, a fim de facilitar a compreensão de aspectos comuns às proteções pertencentes a um mesmo segmento. A taxonomia sugerida é composta pelos seguintes elementos:

- Randomização.
- Construção de uma pilha-sombra.
- Checagem da instrução anterior ao endereço de retorno.
- Checagem dos endereços autênticos para desvios.
- Checagem da posição de entrada nas funções.
- Controle da frequência de instruções de retorno.
- Checagem do apontador para o topo da pilha.

Cada uma dessas classes, bem como as proteções pertencentes a elas, são detalhadas nas subseções a seguir.

3.2.1 Randomização

A estratégia de randomização baseia-se na ideia de impossibilitar o atacante de prever o endereço para o qual o fluxo do programa deve ser transferido, através do embaralhamento dos endereços de memória atribuídos às instruções do programa. Se os endereços de memória forem distribuídos aleatoriamente entre as instruções, um atacante não saberá para qual endereço de memória desviar. Essa estratégia é, na verdade, um aprimoramento da técnica ASLR, apresentada na Seção 2.2.2 [70].

O ASLR é implementado na maioria dos sistemas operacionais modernos. Sua estratégia funciona muito bem quando devidamente implementada e integrada às aplicações. No entanto, ela pode oferecer as seguintes brechas:

- Algumas bibliotecas podem não ser compatíveis com o ASLR ou podem ser compiladas sem essa opção. Nesses cenários, já foi demonstrado que é possível usar códigos binários desse tipo de biblioteca para inferir a localização de outras bibliotecas úteis para um ataque, tal qual a biblioteca *libc* [71];

- Na maioria dos binários de aplicações, o endereço das instruções é fixo, variando apenas os endereços das bibliotecas carregadas (exceto para códigos de posição independente - *Position Independent Code* – PIC). Assim, os atacantes podem usar a alternativa de redirecionar o fluxo de controle para *gadgets* construídos com instruções da própria aplicação [13];
- Quando o atacante consegue recuperar algum endereço do processo durante um ataque (*information leakage attacks*) [36], [72], [73], ele obtém informações sobre a disposição da memória suficientes para dar prosseguimento a um ataque ROP;
- Como o ASLR embaralha apenas os endereços base dos segmentos de memória de cada módulo (o Windows 7 de 32 bits embaralha apenas os bits 17 a 24), um esquema de força bruta pode ser utilizado para testar todos os possíveis endereços onde o *shellcode* possa estar localizado [74];
- Outro problema decorrente do embaralhamento apenas do endereço base é a possibilidade de escrita parcial do endereço de retorno durante a exploração de uma vulnerabilidade de transbordamento de *buffer* na pilha. Nas arquiteturas de 32 bits, por exemplo, o ASLR altera apenas os 16 bits mais significativos do endereço. Por isso, em situações específicas, os atacantes podem calcular o tamanho da entrada maliciosa de forma a sobrescrever apenas a fração menos significativa do endereço de retorno [75], [76], fazendo-o apontar para uma cadeia de *gadgets* escolhida.
- Existe ainda outra vertente de técnicas de superação do ASLR, denominadas genericamente de *Spraying* (pulverização) [59], [60], [77], que baseiam-se em forçar a alocação sequencial de vários grandes pedaços de dados precedidos por instruções NOP. Como, por razões de desempenho, os sistemas operacionais garantem que novos blocos sempre são alinhados durante alocações, é possível prever a posição relativa de um endereço dentro de um bloco alocado.

Em função dessas fragilidades identificadas nas implementações do ASLR, diversas propostas de proteções contra ataques ROP retomam a ideia de inserir aleatoriedade aos endereços das instruções, procurando corrigir as deficiências apontadas no ASLR [63], [78]–[84].

Quando bem implementada, a estratégia de randomização eleva consideravelmente a complexidade de desenvolvimento de ataques ROP. No entanto, existem deficiências inerentes a essa estratégia, relacionadas à possibilidade de exploração através de ataques de vazamento de

informação ou ataques de força bruta. A maioria das soluções que aplica a randomização procura fazê-la no nível de blocos de instruções ou instruções individuais, justamente para elevar o grau de embaralhamento (entropia) e, consequentemente, aumentar o número de tentativas a serem testadas em ataques de força bruta. O problema é que essa randomização de pequenos blocos de instruções pode corromper códigos assinados, blocos de instruções que executam verificações sobre si mesmos – como checagens de somas (*checksums*) –, ou códigos que se modificam durante a execução – como trechos ofuscados (*enconded*). Além disso, a estratégia de randomização é incapaz de proteger códigos compilados sob demanda (*JIT-compiled*), comum em aplicações que dão suporte a ambientes com linguagens interpretadas, como Java, JavaScript, Flash, .Net e SilverLight [60]. Finalmente, a randomização não é considerada uma solução de prevenção completa, porque ela oferece uma proteção probabilística e, portanto, não provê garantias de proteção [81].

3.2.2 Construção de uma pilha-sombra

Muito antes do surgimento dos ataques ROP, a estratégia de construção de uma pilha-sombra para proteção contra ataques de execução de código arbitrário já havia sido idealizada. Inicialmente, essa solução foi proposta para evitar a consolidação de ataques de injeção de código no segmento de pilha [85]–[87]. Porém, nos últimos anos, o uso de pilha-sombra tem sido amplamente empregado na construção de soluções de bloqueio contra ataques ROP. Em algumas propostas, a pilha adicional é implementada via software [88]–[94]. Em outras, ela é construída diretamente no hardware [95], [96].

Essa estratégia consiste na criação de uma cópia dos endereços de retorno anotados na pilha, para comparação no momento em que uma instrução de retorno de função (RET) estiver sendo executada. Conforme ilustrado na Figura 2.5, sempre que uma instrução de chamada de uma função (CALL) é executada, além de ser anotado na tradicional pilha do processo, o endereço de retorno é também escrito em uma estrutura de pilha adicional (conhecida como pilha-sombra), protegida contra a escrita por instruções do processo. Essa proteção é necessária para evitar que um atacante altere os endereços de retorno nas duas pilhas, fazendo-os coincidir e, consequentemente, superando a proteção. No momento em que instruções de retorno são executadas (RET), checa-se a coincidência dos endereços entra as duas pilhas. Assim, se o endereço de retorno armazenado na pilha do processo for alterado, o ataque será identificado.

Apesar de apresentar a vantagem de também ser capaz de impedir a consolidação de ataques do tipo RILC, além do ROP, essa estratégia apresenta o inconveniente de classificar alguns

códigos autênticos como ataques, incorrendo em falsos positivos. Normalmente, esses códigos classificados equivocadamente estão relacionados a práticas de programação não convencionais usadas para lidar com tarefas de baixo nível ou para otimizar manualmente códigos de montagem [97]. Conforme ressaltado por Davi et al. [94], esses falsos positivos podem ocorrer em três situações:

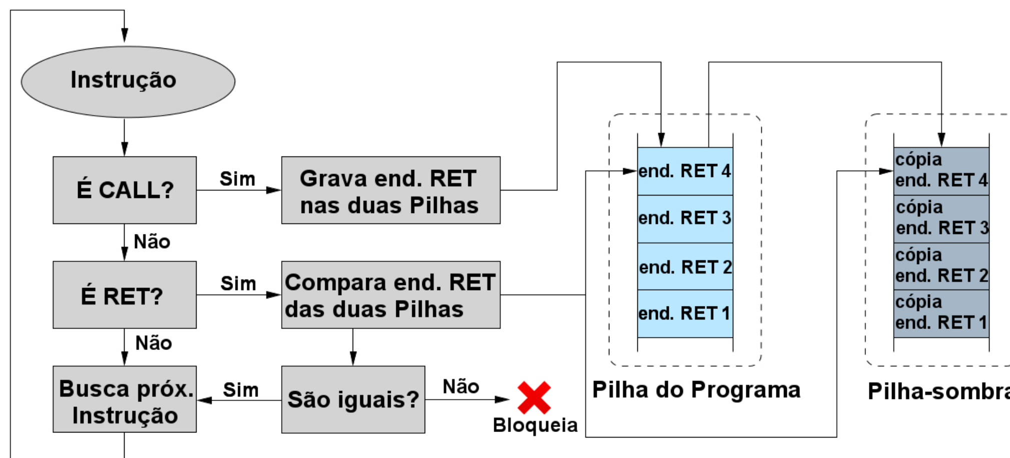

Figura 2.5: Estratégia de proteção baseada na construção de uma pilha-sombra [94].

- Quando uma função não retorna ou faz isso utilizando uma instrução de desvio diferente da instrução RET.
- Quando uma função é chamada sem o uso da instrução CALL.
- Quando o endereço de retorno é sobrescrito pela própria aplicação.

Essas situações fogem do padrão de codificação estabelecido para a arquitetura x86, uma vez que essa arquitetura é projetada para otimizar o processo de chamada e retorno de funções utilizando as tradicionais instruções CALL e RET. Infelizmente, existe uma quantidade considerável de códigos que não seguem essa convenção. Os seguintes exemplos de exceções já foram catalogados por trabalhos que implementam a estratégia de construção de uma pilha-sombra:

- Códigos que utilizam as chamadas de sistema setjmp/longjmp [85], [88], [94]–[96];
- Códigos independentes de posição (*Position Independent Code* – PIC) [90];
- O código de tratamento de sinais nos sistemas derivados do UNIX [94];
- O código que implementa a ligação tardia (*lazy binding*) de bibliotecas dinâmicas nos sistemas derivados do UNIX, onde esse é o procedimento padrão [94];
- O código de tratamento de exceções da linguagem C++ inserido pelo compilador GNU C++ [85], [94];

- *Threads* criadas na extremidade de uma sequência de chamadas de funções em profundidade e que começam a retornar logo após sua criação [85].

Apesar de algumas proteções baseadas no uso da pilha-sombra tratarem as exceções encontradas nos ambientes onde seus protótipos foram testados, é difícil prever quais exceções podem surgir em ambientes com outros sistemas operacionais e aplicações, reduzindo a confiabilidade dessa estratégia para emprego em ambientes de produção.

3.2.3 Checagem da instrução anterior ao endereço de retorno

Em função da convenção utilizada para efetuar-se chamadas de funções na arquitetura x86, normalmente, quando uma instrução de retorno é executada, o fluxo de execução é transferido para a função chamadora na posição imediatamente subsequente à instrução que efetuou a chamada (CALL). Baseado nessa constatação, alguns trabalhos propuseram a estratégia de verificar se existe uma instrução CALL na posição imediatamente anterior ao endereço de destino das instruções de retorno [63], [98]–[102]. Em outras palavras, se o endereço de retorno indicar uma posição que não seja precedida por uma instrução de CALL, o mecanismo de proteção assume que o processo está sofrendo um ataque ROP.

Assim como a estratégia de construção de uma pilha-sombra, a checagem da instrução anterior ao endereço de retorno pode acarretar na classificação incorreta de códigos autênticos, ocasionando falsos positivos. Isso ocorrerá nas situações em que a instrução de retorno (RET) for utilizada como uma instrução genérica de desvio de fluxo, sem estar relacionada a uma instrução de chamada de função (CALL). Dois exemplos de exceções que afetam a estratégia de construção de uma pilha-sombra também desencadeiam a ocorrência de falsos positivos ao utilizar-se a estratégia de checagem da instrução anterior ao endereço de retorno:

- O código de tratamento de sinais nos sistemas derivados do UNIX.
- O código que implementa a ligação tardia (*lazy binding*) de bibliotecas dinâmicas nos sistemas derivados do UNIX, onde esse é o procedimento padrão.

Além de possibilitar a ocorrência de falsos positivos, a estratégia de checagem da instrução anterior ao endereço de retorno deixa brechas para que, em situações específicas, um atacante consiga construir um ataque ROP. Um adversário pode, por exemplo, usar *gadgets* desalinhados (não intencionais) que iniciem em uma posição na qual os bytes anteriores correspondam a uma instrução de CALL. Outra possibilidade pode ser o uso de *gadgets*

posicionados imediatamente após uma instrução de CALL autêntica, mas que não chamou a função que está retornando. Os autores que utilizam a estratégia de checagem da instrução anterior ao endereço de retorno sustentam-se no fato de que o número de *gadgets* que satisfazem esses pré-requisitos é reduzido, o que torna a construção de uma cadeia de *gadgets* completa para emprego em ataques ROP impraticável, na maioria das situações. Contudo, assim como afirmado em relação à estratégia de randomização, é importante ressaltar que essa solução não é considerada uma prevenção completa, porque ela oferece uma proteção probabilística e, portanto, não provê garantias de proteção.

3.2.4 Checagem dos endereços autênticos para desvios

Uma estratégia que vem sendo estudada há vários anos na tentativa de conter diversos tipos de ataques de execução de código arbitrário consiste em identificar os endereços autênticos de desvio executados pela aplicação, para que esses endereços sejam checados no momento em que os desvios de fluxo forem executados [103]. Como o objetivo dessa estratégia é restringir os desvios do fluxo de execução a apenas aqueles que foram originalmente planejados durante o desenvolvimento do programa, os ataques ROP tornaram-se também alvo desse tipo de proteção.

Para registrar os destinos válidos de desvios do fluxo de execução, as soluções estabelecem conjuntos compostos por esses endereços autênticos. Esses conjuntos podem ser construídos durante a compilação [102], [104], extraindo-se as informações diretamente do arquivo binário em uma etapa posterior à compilação [91], [92], ou através do monitoramento dos desvios efetuados pelo programa durante sua execução – em uma etapa de treinamento da solução [99].

A necessidade de identificar os desvios válidos do fluxo de execução durante uma etapa de treinamento pode acarretar na ocorrência de falsos positivos, no momento em que a aplicação estiver executando em definitivo. Essa possibilidade decorre do fato de que é improvável que todos os caminhos do fluxo de execução de aplicações grandes e complexas sejam disparados pela massa de testes utilizada durante o treinamento [1]. Normalmente, os dados de entrada gerados para testar a aplicação, especialmente quando não se tem acesso ao código-fonte do programa, serão capazes de induzir os caminhos de execução mais frequentes durante o uso da aplicação. No entanto, algumas combinações específicas de entradas que não tenham sido testadas durante o treinamento podem ser submetidas para o processo, ocasionando o desvio do fluxo de execução para um caminho não catalogado e, consequentemente, acarretando na classificação equivocada de fluxos autênticos do software como tentativas de ataques ROP.

As implicações de soluções aplicadas durante a compilação de programas são discutidas na Seção 3.3.1. Soluções baseadas em instrumentação binária estática (detalhada na Seção 3.3.2), mesmo se as tabelas de símbolos dos binários estiverem disponíveis – o que normalmente não acontece em softwares comerciais –, não são capazes de identificar corretamente todos os destinos de desvios indiretos [99]. Além disso, inserir instruções no código binário para checagem de condições antes das instruções de desvio [91] , [92] e [104], só funciona contra ataques ROP em arquiteturas onde o tamanho das instruções é fixo, porque apenas nesses casos o atacante é incapaz de acessar a memória em posições desalinhadas para forçar a execução de instruções não intencionais. Para possibilitar o uso dessa estratégia na arquitetura x86, onde o tamanho das instruções é variável, Bletsch et al. [104] evita a exploração de instruções desalinhadas através de um mecanismo que força o alinhamento.

Uma característica indesejável comum às soluções que utilizam a estratégia de checagem dos endereços autênticos para desvios é o *overhead* computacional elevado que elas impõem [38]. Essa deficiência de desempenho decorre da necessidade de efetuar muitas comparações, em função da quantidade de possibilidades e da complexidade de implementação dos mecanismos de checagem. Apesar disso, pesquisadores apontam a estratégia de checagem dos endereços autênticos para desvios como um esquema promissor para proteção contra ataques ROP [8].

3.2.5 Checagem da posição de entrada nas funções

A estratégia de checagem da posição de entrada nas funções consiste em verificar se, ao transferir o fluxo de execução de uma função para outra, o destino do desvio corresponde ao ponto de entrada da função alvo. Para isso, as soluções propostas efetuam checagens sempre que uma instrução de desvio é executada. A maioria das soluções realiza essa verificação garantindo que o destino do desvio a ser executado corresponda a uma instrução pertencente à própria função atualmente em execução ou ao ponto de entrada de uma outra função [93], [95], [96], [99]. Huang et al. [105] optou por checar se o fluxo de execução passou pelo ponto de entrada da função corrente, sempre que uma instrução de desvio é executada.

Apesar de nenhum trabalho que utiliza a estratégia de checagem da posição de entrada nas funções indicar a existência de falsos positivos em decorrência do uso dessa estratégia, na prática, é provável que existam códigos customizados que utilizam instruções de desvio para saltar para uma outra função em destinos que não correspondam ao ponto de entrada da função alvo. Nesses casos,

a estratégia de checagem da posição de entrada nas funções apontará a ocorrência de ataque ROP durante a execução de códigos autênticos.

3.2.6 Controle da frequência de instruções de retorno

Normalmente, as sequências de instruções que compõem cada *gadget* usado em um ataque ROP são extremamente curtas, dificilmente contendo mais do que cinco instruções. Essa é uma característica inerente aos ataques ROP, porque quanto maior a sequência de instruções, maior a probabilidade de existir entre essas instruções uma operação que altere o status da memória ou de um registrador de forma a comprometer o ataque. Essa alteração de status é comumente chamada por atacantes de "efeito colateral" de um *gadget*. A fim de evitar esses efeitos colaterais, quase sempre os *gadgets* escolhidos pelos atacantes são extremamente curtos.

Diante dessa constatação, diversos autores investiram esforços em uma estratégia de controle da frequência de instruções de retorno como forma de detectar a execução de cadeias de *gadgets*. Foram propostos trabalhos que verificam o pico na frequência de instruções de retorno através do monitoramento em tempo real de cada instrução de retorno executada [61], [62]. Outras soluções adotaram a postura de contabilizar a frequência de instruções de retorno previamente executadas, através da análise de um *buffer* de desvios disponível em hardware (*Branch Trace Store buffer*) [106], ou da verificação da distância entre os endereços de retorno armazenados na área mais recentemente desocupada da pilha (endereços imediatamente menores do que o endereço que aponta para o topo da pilha, armazenado no registrador ESP) [107], [108].

Apesar de eficaz no bloqueio aos ataques ROP reais identificados até o momento, da forma como vem sendo empregada, a estratégia de controle da frequência de instruções de retorno apresenta as seguintes deficiências:

- Uma sequência de retornos de funções próximos, situação típica em funções com recursão em cauda[6], pode induzir proteções baseadas na estratégia de controle da frequência de instruções de retorno a bloquear equivocadamente a execução de códigos autênticos.
- No caso das soluções que analisam a frequência de instruções de retorno percorrendo a pilha, o atacante pode forjar pilhas estruturadas com valores quaisquer entre os endereços de retorno a fim de superar essa proteção. Basta que os *gadgets* possuam alguma instrução que incremente o registrador ESP (ponteiro de topo de pilha), por exemplo.

[6] Funções recursivas em cauda são aquelas nas quais a chamada recursiva é a última instrução a ser executada.

- Nesse mesmo cenário (análise da proximidade de endereços de retorno na pilha), funções que contenham poucas variáveis e parâmetros podem apresentar endereços de retorno próximos, levando à ocorrência de falsos positivos. Essa possibilidade pode ainda aumentar caso tenha sido utilizada a otimização de compilação que emprega o registrador EBP (*base pointer*) como um registrador de uso geral, pois isso força a liberação dos espaços na pilha reservados para armazenar os apontadores de *frames* de funções (*base pointers* ou *frame pointers* – vide as Figuras 2.1 e 2.2).

3.2.7 Checagem do apontador para o topo da pilha

Nos ataques de execução arbitrária de código baseados em ROP, o adversário precisa ser capaz de controlar os conteúdos armazenados no contador de programa (registrador EIP) e no ponteiro para o topo da pilha (registrador ESP) [109], [110]. O controle do contador de programa permite que o atacante desvie o fluxo de execução para o primeiro *gadget*. O domínio do ponteiro para o topo da pilha, por sua vez, possibilita que os demais *gadgets* sejam encadeados através dos endereços de retorno escolhidos pelo invasor.

Dependendo da vulnerabilidade explorada, ao invés de ser gravada na pilha, a entrada maliciosa inserida pelo atacante é armazenada em outro segmento da área de memória do programa. Nesses casos, o ponteiro para o topo da pilha precisar ser ajustado durante o ataque ROP, fazendo-o apontar para a posição exata da entrada maliciosa, posicionada fora dos limites de endereços reservados para a pilha. Essa operação é conhecida pelo termo "*stack pivoting*" (reposicionamento da pilha).

Tendo em vista que o ponteiro para o topo da pilha jamais deve armazenar um endereço que não pertença à faixa de endereços reservados para a pilha, foi proposta a estratégia de checagem do apontador para o topo da pilha [63], [101]. Caso o registrador ESP contenha um endereço que não pertença ao segmento de pilha, o ataque é identificado.

Apesar de efetiva no combate aos ataques que corrompem outros segmentos de memória, a estratégia de checagem do apontador para o topo da pilha não é útil em ataques que afetam a pilha (ex: transbordamento de *buffer* na pilha). Isso porque os dados de entrada do atacante já estarão posicionados na região da pilha e, assim, não será necessário deslocar o apontador de topo de pilha para outro segmento. Além disso, conforme mencionado na Seção 1.1, uma implementação dessa estratégia foi incorporada ao Windows 8 Developer Preview, em 2011, e superada poucos dias após seu lançamento [22]–[24]. Como a checagem é efetuada no momento em que uma API do Windows

é chamada, antes de executar qualquer chamada de API, o atacante pode executar *gadgets* que copiam a entrada maliciosa para uma área pertencente à pilha. Portanto, se a verificação do conteúdo do registrador ESP não for efetuada regularmente, essa estratégia pode tornar-se sem efeito, pois o atacante poderá reposicionar o código malicioso na pilha antes que a checagem seja realizada.

3.3 Abordagens de Implementação

As diversas estratégias de proteção discutidas nas Seções 3.2.1 a 3.2.7 podem ser aplicadas aos sistemas que se pretende defender através de várias abordagens. Essas abordagens de implementação impactam diretamente na qualidade da solução proposta e apresentam características peculiares. Diante dessa percepção, foi proposta a seguinte taxonomia para classificação das proteções segundo as abordagens escolhidas para aplicar as medidas de prevenção contra ataques ROP:

- Compilação.
- Instrumentação Binária Estática.
- Instrumentação Binária Dinâmica.
- Adaptação do hardware.
- Virtualização.
- Emulação de código.

Deve-se ressaltar que algumas soluções utilizam uma combinação dessas classes de abordagens para viabilizar a implementação de suas estratégias de proteção contra ataques ROP.

3.3.1 Compilação

Como o próprio nome indica, a abordagem de compilação é utilizada pelas soluções que efetuam modificações no código binário de uma aplicação no momento em que o seu código-fonte é compilado. A principal desvantagem desse tipo de abordagem decorre da necessidade de obtenção do código-fonte que, para a maioria dos softwares comerciais, não é disponibilizado. Além disso, para serem consideradas efetivas, proteções baseadas na abordagem de compilação exigem que todos os desenvolvedores de sistemas optem por empregar a solução. Caso contrário, uma única biblioteca que não tenha a proteção habilitada poderá ser explorada pelo atacante.

Infelizmente, a história recente tem mostrado que muitos fabricantes de software priorizam o desempenho em detrimento da segurança. Essa constatação pode ser comprovada pela evolução de algumas proteções incorporadas pelo compilador Visual Studio. A fim de coibir os ataques que exploram os Tratadores de Exceção Estruturados (*Structured Exception Handlers* – SEHs), a Microsoft lançou um dispositivo denominado "SafeSEH" (*Safe Structured Exception Handler*), que agregou novos mecanismos de proteção da pilha, com o intuito de resguardar os registros para tratamento de exceção utilizados pelos seus sistemas operacionais [111]. Para isso, o *flag* de ligação /SafeSEH está disponível no compilador Visual Studio, desde a versão ".Net 2003". No entanto, uma estratégia frequentemente empregada por atacantes para superar essa proteção consiste simplesmente em utilizar alguma biblioteca carregada pelo processo que não tenha sido compilada com a opção /SafeSEH, já que esse é um cenário recorrente em muitas aplicações vulneráveis.

A exploração de aplicações e bibliotecas compiladas sem a opção /SafeSEH tornou-se tão comum que, em 2009, foi lançada uma extensão para a segurança dos SEHs denominada *Structured Exception Handler Overwrite Protection* (SEHOP), que opera através de checagens efetuadas unicamente durante a execução de um processo [112]. Assim, esse mecanismo independe das escolhas durante a etapa de compilação dos softwares. Uma evolução similar aconteceu com as proteções ASLR (descrita na Seção 2.2.2) e DEP (descrita na Seção 2.2.3), que originalmente dependiam do uso dos *flags* de compilação /DYNAMICBASE e /NXCOMPACT, respectivamente, para ativar as proteções. Ao constatar que muitos softwares vulneráveis continuavam a ser explorados por não aderirem às soluções de compilação, a Microsoft incorporou em seus sistemas operacionais a possibilidade de forçar o funcionamento do ASLR [113] e do DEP [48] para todos os processos, a despeito de eventuais problemas de compatibilidade.

As estratégias de proteção contra ataques ROP empregadas nas soluções que utilizam a abordagem de compilação variam bastante. Todavia, uma delas aplica-se apenas a esse tipo de abordagem. Dois trabalhos [64], [97] adotam a mesma estratégia, viável apenas através da abordagem de compilação, pois ambas as soluções ajustam o código binário a ser gerado para evitar que ele apresente instruções de desvio úteis para o encadeamento de *gadgets*. No entanto, enquanto um trabalho previne apenas a existência de instruções de retorno (RET) indesejadas [64], o outro evita o abuso de qualquer tipo de desvio indireto [97].

3.3.2 Instrumentação binária estática

O conceito de Instrumentação Binária Estática é usualmente utilizado para designar operações de manipulação de instruções contidas em arquivos binários já compilados [114]. Normalmente, essas modificações são realizadas com o auxílio de ferramentas especializadas nesse tipo de tarefa, tais como DynInst [115] , Vulcan [116] e DTrace [117]. Esses softwares substituem instruções originais do arquivo executável por instruções conhecidas como trampolins, que desviam o fluxo para um novo código indicado pelo usuário.

Neste trabalho, o significado do termo Instrumentação Binária Estática é expandido para englobar todas as soluções que analisam ou modificam os códigos binários executáveis após a sua compilação e antes que esses códigos iniciem sua execução. Dessa forma, estão inseridas no grupo de soluções que utilizam a abordagem de Instrumentação Binária Estática as propostas que:

- Extraem informações de arquivos binários através de ferramentas de desmontagem (*disassemblers*);
- Efetuam alterações no código, ou em sua estrutura, no momento de seu carregamento para a memória;
- Aplicam *patches* (remendos) em bibliotecas do sistema;
- Inserem novas instruções em arquivos binários executáveis.

As soluções pertencentes à abordagem de Instrumentação Binária Estática diferenciam-se dos trabalhos que compõem a classe Instrumentação Binária Dinâmica (detalhada na Seção 3.2.3) apenas pelo fato de que, na última, as manipulações do código binário ocorrem em tempo real. Ou seja, nos trabalhos que utilizam a abordagem de Instrumentação Binária Dinâmica, o código binário é analisado, e possivelmente modificado, durante a execução do programa.

De maneira geral, as estratégias implementadas através de abordagens estáticas oferecem um *overhead* menor para as aplicações protegidas do que as abordagens dinâmicas [114]. Isso acontece porque a Instrumentação Binária Estática introduz apenas os códigos usados para checar as condições previstas pela estratégia de proteção escolhida. Em contrapartida, na Instrumentação Binária Dinâmica, é inserido um *overhead* adicional para que a ferramenta de instrumentação execute trocas de contexto com a aplicação instrumentada, além de efetuar a desmontagem de códigos e a geração de instruções, tudo durante a execução da aplicação instrumentada.

Por outro lado, as soluções de Instrumentação Binária Estática que exigem uma desmontagem precisa do código binário para que possam modificá-lo só funcionam caso informações de depuração, como a tabela de símbolos, estejam disponíveis [81], [95], [96], [105]. Entretanto, por questões de economia de espaço e proteção de propriedade intelectual, essas informações de depuração normalmente são removidas dos softwares destinados a ambientes de produção.

Outra deficiência relacionada a proteções que adotam a abordagem de Instrumentação Binária Estática recai sobre as implementações que utilizam a inserção de instruções no código executável original [78], [81], [84], [91], [92], [95], [96], [104]. Esse procedimento sofre dos mesmos problemas relacionados à estratégia de randomização. Pode-se corromper códigos assinados, blocos de instruções que executam verificações sobre si mesmos – como checagens de somas (*checksums*) –, ou códigos que se modificam durante a execução – como trechos ofuscados (*enconded*). Além disso, a inserção estática de instruções também é incapaz de proteger códigos compilados sob demanda (*JIT-compiled*), comum em aplicações que dão suporte a ambientes com linguagens interpretadas, como Java, JavaScript, Flash, .Net e SilverLight [60]. É importante reforçar também que inserir instruções para checagem de condições antes de instruções de desvio, como realizado em dois trabalhos [91], [92], só funciona como proteção contra ataques ROP em arquiteturas onde o tamanho das instruções é fixo, ou caso algum esquema que force o alinhamento de instruções seja imposto, conforme estabelecido por Bletsch et al. [104].

3.3.3 Instrumentação binária dinâmica

Conforme mencionado na Seção anterior, as soluções classificadas na categoria Instrumentação Binária Dinâmica diferenciam-se das propostas incluídas na classe Instrumentação Binária Estática por efetuarem manipulações nos códigos binários após o início de suas execuções. Para isso, tipicamente são empregadas ferramentas de instrumentação binária do tipo JIT (*Just-in-time*), que permitem a análise e eventual modificação do código à medida que ele é executado. Pin [118], Valgrind [119], Strata [120] e DynamoRIO [121] são exemplos de sistemas que dão suporte a essa funcionalidade. Para isso, antes que uma instrução seja executada pelo processador, esses *frameworks* interceptam a instrução, geram e executam novos códigos, e garantem que o *framework* retomará o controle do processador após a execução da instrução.

A grande desvantagem da abordagem de Instrumentação Binária Dinâmica decorre do custo computacional para efetuar as trocas de contexto entra a ferramenta de instrumentação e a

aplicação monitorada. Estudos apontam que, para efetuar uma tarefa simples de contagem do número de blocos de instruções executadas, os *frameworks* Pin, Strata, DynamoRIO e Valgrind apresentam, respectivamente, um *overhead* médio de 230%, 260%, 490% e 750% [118], [122]. Por outro lado, soluções que utilizam a abordagem de Instrumentação Binária Dinâmica não necessitam de informações adicionais sobre o código binário, como tabelas de símbolos, e tampouco requerem o código-fonte das aplicações. Além disso, esses sistemas permitem a análise de programas que geram códigos compilados sob demanda (*JIT-compiled*).

3.3.4 Adaptação do hardware

Diante das desvantagens inerentes às abordagens já apresentadas, alguns autores propuseram adaptações nas estruturas de hardware dos processadores [95], [96], [102], a fim de permitir que estratégias de proteção contra ataques ROP sejam implementadas de maneira eficiente e independente da disponibilidade de dados relacionados ao código das aplicações (tabelas de símbolos ou códigos-fonte). Outros trabalhos [99], [100], [106] apresentam esquemas que reutilizam estruturas de hardware e instruções especiais, concebidas para outras finalidades, com o intuito de viabilizar a implementação de proteções contra ataques ROP. Para efeitos de classificação, ambos os conjuntos de propostas são considerados pertencentes à abordagem de Adaptação do hardware. Contudo, deve-se ressaltar que, na prática, nenhuma solução que emprega a abordagem de Adaptação do hardware a utiliza de maneira isolada.

O principal argumento de esquemas que adotam a adaptação do hardware reside na redução do *overhead* imposto pela proteção ao utilizar-se essa abordagem. No entanto, deve-se considerar também o impacto resultante da incorporação de mudanças à arquitetura dos processadores modernos. Modificações complexas podem requerer a inclusão de muitos circuitos, o que pode prejudicar o desempenho geral do processador. Além disso, os resultados apresentados por protótipos podem não ser confiáveis, porque é muito difícil simular todos os aspectos relevantes para o desempenho final da arquitetura adaptada. Em geral, os mecanismos utilizados para simular o funcionamento dos protótipos de proteções desprezam algumas características consideradas menos impactantes. Porém, a importância dessas características pode ser subestimada até que uma simulação completa seja executada.

3.3.5 Virtualização

O uso de máquinas virtuais também foi adotado como mecanismo para possibilitar a implementação de estratégias de proteção contra ataques ROP. Dois trabalhos [89], [108] exploram funcionalidades específicas oferecidas por virtualizadores para detectar ataques em tempo real. Essas soluções utilizam a assistência à virtualização, incorporada em alguns processadores modernos, para disparar a troca de contexto entre a aplicação e o virtualizador, permitindo a execução de procedimentos de checagem da área de memória pertencente ao processo monitorado. Em ambientes com assistência em hardware à virtualização, se o sistema operacional virtualizado (*guest*) tentar executar alguma operação privilegiada, o processador chama um código de tratamento previamente cadastrado pelo sistema de virtualização. As soluções baseadas em virtualização [89], [108] forçam a execução de operações privilegiadas em momentos estratégicos da execução da aplicação monitorada, a fim de garantir a transferência do fluxo para os códigos de checagem, que são previamente inseridos no tratador cadastrado pelo virtualizador.

A abordagem de virtualização limita-se a arquiteturas nas quais existem estruturas específicas para auxiliar as tarefas de virtualização, como x86. Nas arquiteturas onde não existe esse suporte em hardware, o desempenho desse tipo de abordagem via software torna-se impraticável. Na verdade, mesmo com o suporte do hardware, é de se esperar que soluções que adotem essa abordagem apresentem um *overhead* comparável às soluções que utilizam a abordagem de Instrumentação Binária Dinâmica, já que as máquinas virtuais também efetuam constantes trocas de contexto entre a aplicação monitorada e o código do virtualizador. Os trabalhos que utilizam virtualização para implementar proteções contra ataques ROP apresentam um *overhead* médio melhor do que as soluções que utilizam Instrumentação Binária Dinâmica (10% em [89] e aproximadamente 15% em [108]). No entanto, é necessário que se realizem testes adicionais para confirmar esses *overheads*, já que, em ambos os trabalhos, os testes foram realizados com um conjunto limitado de aplicações e com a proteção aplicada a um pequeno grupo de bibliotecas.

3.3.6 Emulação de código

ROPGuard [101] foi a primeira solução a sugerir o uso de uma abordagem baseada na emulação de código. Entre várias estratégias de detecção de ataques ROP, essa proteção adota um esquema de simulação das instruções a serem executadas após o retorno de funções críticas. São consideradas funções críticas aquelas que o atacante usa para desabilitar a proteção dos bits NX/XD ou, de alguma outra forma, permitir a execução do *shellcode*. No ROPGuard a emulação de código é

utilizada para verificar se os endereços de retorno subsequentes são executáveis e se esses endereços estão posicionados imediatamente após uma instrução de chamada de função (estratégia de checagem da instrução anterior ao endereço de retorno).

Para isso, ROPGuard simula algumas instruções comumente usadas em *gadgets* para alterar o valor do apontador de topo da pilha: PUSH; POP; ADD ESP, *valor*; SUB ESP, *valor*; RETN. As demais instruções são simplesmente passadas à diante. Quando uma instrução de retorno é encontrada, a proteção realiza as verificações mencionadas. A simulação é encerrada quando um limite de instruções, estabelecido em um arquivo de configuração, é atingido, ou no momento em que uma outra instrução que desvia o fluxo de execução é encontrada. Diante disso, é fácil perceber que o ROPGuard foi planejado para bloquear apenas ataques ROP construídos com *gadgets* encadeados por instruções de retorno. No entanto, o autor salienta a possibilidade de, em um trabalho futuro, estender-se a emulação de código para simular outras instruções, incluindo JMPs e CALLs indiretos.

Apesar de também se limitar a detectar os ataques ROP tradicionais, nos quais a interligação dos *gadgets* é garantida unicamente por instruções de retorno (RET), ROPscan [123] expandiu a abordagem de emulação de código adotando-a para detectar ataques ROP em qualquer tipo de entrada para programas. Essa proteção utiliza a ferramenta de emulação de código "Nemu" [124] para verificar se as entradas do programa correspondem a endereços válidos de *gadgets* que possam ser encadeados. A fim de limitar o número de instruções a serem simuladas, é estabelecido um tamanho limite para um *gadget* e para o total de instruções executadas.

Como as ferramentas de emulação de código já impõem um elevado custo computacional [125], independente das estratégias adotadas, espera-se que proteções que utilizam a abordagem de emulação de código acarretem em um *overhead* significativo. No caso específico do ROPGuard, o *overhead* reportado corresponde a apenas 0,48%. No entanto, além de executar simulações extremamente simples, essa proteção é aplicada apenas às APIs do Windows consideradas críticas, o que limita o impacto da emulação de código. Os autores de ROPscan, por sua vez, não informam o *overhead* decorrente de sua solução, porque seus testes utilizam uma ferramenta de detecção de *shellcode* para analisar as entradas, ao invés de interferir na execução do processo. Por isso, a métrica utilizada para medir a qualidade da solução foi o *throughput* médio, que atingiu 120Mbit/s na implementação do ROPscan. Essa é uma métrica comumente utilizada na avaliação de sistemas de detecção de *shellcode*, já que normalmente essas soluções são empregadas na análise de tráfegos de rede.

Portanto, apesar de aparentemente não ser útil para a proteção em tempo real do espaço de memória das aplicações, a abordagem de emulação de código parece apresentar serventia em outras situações. Um cenário viável para aplicação dessa abordagem pode ser a verificação de arquivos de dados descarregados pelo usuário. A emulação de código poderia ser usada para escanear os dados antes de submetê-los efetivamente para a aplicação de destino, evitando infecções oriundas de arquivos de dados maliciosos.

3.4 Discussão

A Tabela 3.1 apresenta, de forma resumida, uma comparação entre as proteções contra ataques ROP, ilustrando as estratégias, abordagens e métricas relacionadas a cada uma. Os valores de *overhead* indicados na Tabela 3.1 correspondem às taxas apresentadas pelos próprios trabalhos. Uma comparação precisa das soluções exigiria que todas as proteções fossem implementadas em um mesmo ambiente e submetidas a testes com o mesmo conjunto de aplicações.

Apesar disso, é possível ratificar relações mais gerais, estabelecidas entre as abordagens de implementação das proteções. É possível constatar, por exemplo, que os trabalhos que empregam a abordagem de Instrumentação Binária Dinâmica acarretam *overheads* maiores do que os trabalhos que utilizam as demais abordagens. Em contrapartida, as soluções que empregam Instrumentação Binária Estática e Adaptações do Hardware tendem a apresentar um *overhead* menor.

Uma observação interessante reside no fato de que apenas 5 soluções parecem não apresentar problemas de compatibilidade com algum tipo de aplicação. Todas as demais soluções esbarram em algum tipo de exceção, seja por identificar equivocadamente softwares autênticos como maliciosos, seja por corromper códigos que efetuam algum tipo de manipulação sobre si mesmos, como decodificações ou verificações de assinaturas. É importante reforçar que a coluna de exceções da tabela não ilustra os casos que representam possíveis brechas de segurança deixadas pelas soluções propostas. Em outras palavras, é possível imaginar situações exploráveis por atacantes para superar algumas das proteções listadas, mas essas possibilidades não estão contabilizadas entre as exceções.

Tendo em vista que ainda não há uma proteção eficaz contra ataques ROP implementada em nenhum sistema operacional moderno, espera-se que esse assunto continue atraindo a atenção de pesquisadores em busca de uma solução viável. Para isso, os trabalhos futuros deverão aliar eficácia no bloqueio de todas as variantes de ataques ROP, reduzido *overhead* computacional e baixo índice de ocorrência de exceções.

Tabela 3.1: Comparação das proteções contra ataques ROP

Proteção	Abordagens	Estratégias	Ataques bloqueados	*Overhead* *(%)*	Exceções	Viabilidade prática
G-Free [97]	1	8	R, J e C	3,0	0	Sim
Return-less Kernels [64]	1	8	R	9,7	0	Sim
CFL [104]	1 e 2	4	R, J e C	5,6	3	Sim
Security mitigations for ROP [63]	1 e 2	1, 3, 7 e 8	R	*	4	Não
/ROP [102]	1 e 4	3 e 4	R	3,3	2	Não
Hardware Virtualization [89]	1 e 5	2	R	10,0	4	Sim
Poster [91] / MoCFI [92]	2	2 e 4	**	1,2	7	Não
Marlin [80]	2	1	R, J e C	*	0	Sim
STIR [78]	2	1	R, J e C	1,6	3	Sim
In-place randomization [81], [84]	2	1	R, J e C	0	3	Sim
ILR [82]	2 e 3	1	R, J e C	16,0	6	Sim
ROP Monitor [90]	2 e 3	2 e 8	R, J e C	350,0	6	Não
ROPDetector [105]	2 e 3	5	R, J e C	350,0	0	Sim
DynIMA [62]	2 e 3	6	R	*	4	Não
BR [95], [96]	2 e 4	2 e 5	R, J e C	2,1	6	Não
kBouncer [100]	2 e 4	3	R	4,0	3	Sim
CFIMon [99]	2 e 4	3, 4 e 5	R, J e C	6,1	6	Sim
Eunomia [106]	2 e 4	6	R	4,7	1	Sim
ROPGuard [101]	2 e 6	3,7 e 8	R	0,5	3	Sim
Code Shredding [79]	3	1	R, J e C	365,0	3	Sim
TRUSS [88]	3	2	R	53,4	4	Sim
ROPdefender [94]	3	2	R	217,0	4	Sim
ROPgrind [105]	3	2 e 5	**	900,0	4	Sim
DROP [61]	3	6	R	530,0	1	Sim
CFLC [126]	3	8	R, J e C	374,0	0	Sim
HyperCrop [108]	5	6	R	15,0	1	Sim
ROPscan [123]	6	8	R, J e C	*	1	Sim

Abordagens: 1-Compilação ; 2-Instrumentação Binária Estática ; 3-Instrumentação Binária Dinâmica ; 4-Adaptação do hardware ; 5-Virtualização ; 6-Emulação de código.

Estratégias: 1-Randomização ; 2-Construção de uma pilha-sombra ; 3-Checagem da instrução anterior ao endereço de retorno ; 4-Checagem dos endereços autênticos para desvios ; 5-Checagem da posição de entrada nas funções ; 6-Controle da frequência de instruções de retorno ; 7-Checagem do apontador para o topo da pilha ; 8-Outras.

Ataques bloqueados: R-encadeamento via RET; J-encadeamento via JMP; C-encadeamento via CALL.

*O *overhead* não é informado na publicação.

**Na arquitetura ARM, as instruções de desvio são diferentes. Essa solução trata todos os desvios.

A análise das proteções elaboradas até o presente momento permite, inclusive, que se formule afirmações mais gerais quanto ao futuro das proteções contra ataques ROP. Todas as soluções baseadas em verificações efetuadas em pontos específicos da execução de uma aplicação, como as ocasiões em que chamadas de sistema são efetuadas, podem eventualmente ser superadas por estratégias que, após preparar o ataque, forcem a execução de códigos sem efeito simplesmente para iludir as checagens futuras. Como as verificações limitam-se às últimas N instruções executadas (já que uma verificação completa é inviável por questões de desempenho e espaço de armazenamento), se o código sem efeito executar N ou mais instruções, o histórico do código malicioso será apagado e o ataque não será detectado. Além disso, soluções que verificam a ocorrência do ataque em um único momento do período de execução do processo, como nas chamadas de APIs, são incapazes de verificar áreas de código com permissão de escrita, usadas em

códigos compilados sob demanda (*JIT compiled*), por exemplo, porque os dados a serem checados podem já ter sido alterados legitimamente desde a execução dos desvios que se pretende verificar.

Por tudo isso, acredita-se que soluções efetivas contra o ROP devem checar o status do fluxo de execução durante toda a execução do processo. Isso pode ser alcançado com as ferramentas de instrumentação binária dinâmica, ou através do aparelhamento do hardware com estruturas e lógica suficiente para bloquear os ataques sem onerar em excesso o desempenho das aplicações. As soluções de instrumentação binária dinâmica podem ser muito úteis para os processos de prototipação de soluções. Além disso, em ambientes de produção, elas aparecem como uma ferramenta importante no bloqueio a novos tipos de ataques e investidas do tipo *zero-day* (aquelas para as quais ainda não existem atualizações de segurança disponíveis). Nesses casos, pode ser vantajoso arcar com o *overhead* imposto por uma ferramenta de instrumentação binária dinâmica para garantir a segurança do sistema até que uma atualização de segurança esteja disponível.

4 Controle da frequência de desvios indiretos

Diante da constatação de que, para evitar "efeitos colaterais" (discutidos na Seção 3.2.6), os ataques ROP obrigatoriamente apresentam uma elevada concentração de instruções de desvio indireto (RETs, JMPs indiretos ou CALLs indiretos) em um curto espaço de tempo, optou-se por adaptar a estratégia de controle da frequência de instruções de retorno (Seção 3.2.6) com o intuito de detectar as três variantes desse tipo de ataque. Ao invés de medir a frequência apenas das instruções de retorno, esse novo esquema supervisiona a frequência de qualquer tipo de desvio indireto, incluindo aqueles efetuados através de instruções CALL indireto ou JMP indireto. Dessa forma, é possível evitar os três tipos de ataques ROP. Além de incluir a contabilidade dos demais desvios indiretos (JMP e CALL), fato inédito entre as soluções baseadas no controle da frequência de instruções, a solução apresentada neste trabalho propõe um mecanismo para evitar a ocorrência dos casos de falsos positivos que acometem as soluções baseadas no controle da frequência de instruções de retorno (RET), como os códigos de funções com recursão em cauda (vide Seção 4.1.2.2).

O esquema proposto neste trabalho consiste em checar se a contagem do número de instruções de desvio indireto (RET, JMP, CALL) em uma determinada "janela de instruções" é maior do que um determinado limiar. Para definir o valor ideal desse limiar, é possível tanto estabelecer um valor padrão, com base na análise de um conjunto de aplicações (*benchmark*[7]), quanto efetuar uma etapa de treinamento com cada software que se pretende proteger, a fim de estabelecer o limiar máximo atingido por aquela aplicação durante a sua execução normal.

O tamanho da janela de instruções foi definido através de testes de validação (detalhados na Seção 4.1.2), optando-se, ao final, pelo comprimento de janela para o qual a diferença foi maior na contagem de desvios entre execuções normais e sob ataque. A Figura 3.1 ilustra dois gráficos, (a) e (b), resultantes de dois conjuntos de testes hipotéticos. No gráfico (a), são indicadas as frequências máximas de desvios indiretos supostamente observadas ao executar 10 aplicações em modo normal e sob ataque ROP, considerando-se uma janela de 16 instruções. O gráfico (b) apresenta as mesmas informações para uma janela de 32 instruções. Deve ser escolhido o tamanho de janela que oferecer a maior diferença entre a maior densidade apresentada nas execuções normais e a menor densidade

---

[7] *Benchmark*: conjunto de programas usados para testar e comparar sistemas de hardware e/ou software.

apresentada por uma cadeia de *gadgets*. Nos exemplos hipotéticos criados na Figura 3.1, seria escolhida a janela de 32 instruções.

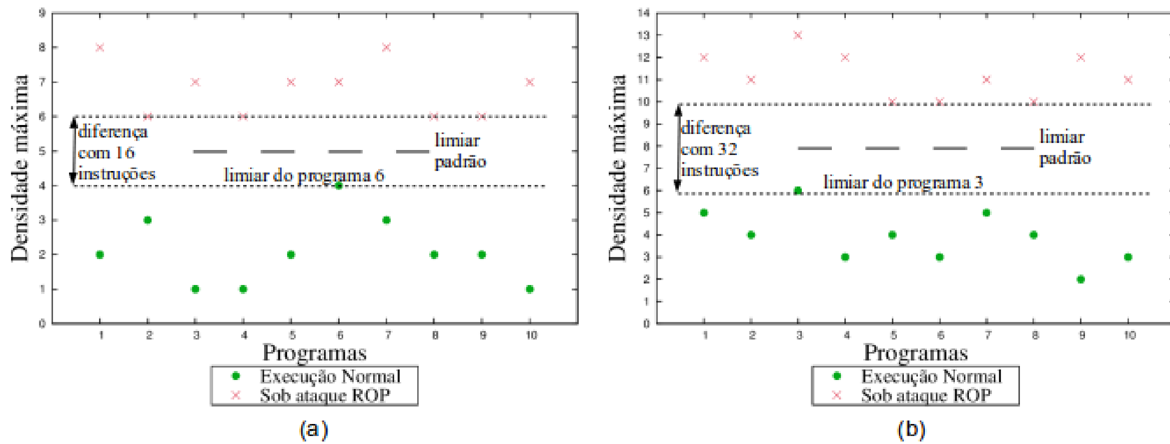

Figura 3.1: Gráficos hipotéticos de densidade máxima de desvios indiretos para 10 aplicações

Para o tamanho de janela escolhido, o valor mediano entre a maior densidade apresentada nas execuções normais e a menor densidade apresentada em um ataque ROP deve ser escolhido como limiar padrão. No exemplo hipotético ilustrado pela Figura 3.1 (b), o limiar padrão seria 8. A Figura 3.1 (a) ilustra também qual seria o limiar específico para o programa 6, ao utilizar-se uma janela de 16 instruções, assim como a Figura 3.1 (b) representa o limiar específico para a aplicação 3, caso opte-se por uma janela de 32 instruções. É importante ressaltar que a adoção de um limiar específico é melhor do que o uso de um limiar padrão, porque ela impõe restrições mais severas para a construção de uma cadeia de *gadgets* capaz de iludir a proteção.

Na sequência deste capítulo são apresentadas a metodologia utilizada durante este trabalho (Seção 4.1), a descrição das decisões de implementação adotadas durante o desenvolvimento do protótipo (Seção 4.2) e uma discussão dos principais desafios relacionados à solução proposta (Seção 4.3).

4.1 Metodologia

Para comprovar a eficácia da proteção desenvolvida neste trabalho, elaborou-se uma metodologia que contempla a análise do comportamento dos desvios indiretos tanto em aplicações normais quanto em ataques ROP. Através dessa avaliação, foi possível confirmar que a diferença de comportamento entre essas duas classes é suficiente para ratificar a hipótese de que o controle da frequência de desvios indiretos é uma estratégia eficaz na detecção de ataques ROP (vide Seção 5.3). A análise da frequência de desvios indiretos foi dividida em duas etapas:

- Avaliação teórica: foram estudados os cenários em que o comportamento de aplicações autênticas mais se aproxima do padrão apresentado por ataques ROP.
- Avaliação empírica: foram registrados os valores de frequência máxima de desvios indiretos observados durante a execução dos *benchmarks* que compõem a suíte SPEC CPU2006 (descrita na Seção 4.1.1.1), comparando-os com as frequências máximas de desvios indiretos observadas durante a execução de *exploits* ROP reais disponíveis no repositório *Exploit Database* (descrito na Seção 4.1.1.2).

Além de demonstrar a eficácia da proteção no bloqueio a ataques ROP, a metodologia utilizada neste trabalho inclui procedimentos para a avaliação do impacto da solução no tempo de execução (tempo de CPU) das aplicações protegidas. Para isso, comparou-se o desempenho dos *benchmarks* que compõem o SPEC CPU2006 (Seção 4.1.1.1) com o tempo de execução observado quando esses mesmos aplicativos são protegidos pela solução de referência denominada ROPdefender [94] e pelo protótipo desenvolvido neste trabalho (Seção 4.2).

Nas subseções a seguir são detalhadas as bases de dados utilizadas durante os experimentos (Seção 4.1.1) e as três etapas componentes da metodologia (Avaliação teórica – Seção 4.1.2, Avaliação empírica – Seção 4.1.3 e Avaliação de desempenho – Seção 4.1.4).

4.1.1 Bases de dados

As etapas de avaliação empírica e avaliação de desempenho da metodologia utilizada neste trabalho exigiram a seleção de um conjunto de aplicações e de um conjunto de *exploits* ROP para a execução dos experimentos. As subseções 4.1.1.1 e 4.1.1.2, apresentadas a seguir, detalham as duas bases de dados utilizadas para compôr esses conjuntos.

4.1.1.1 SPEC CPU2006

Quase todas as proteções contra ataques ROP estudadas durante o desenvolvimento deste trabalho e discutidas no Capítulo 3 utilizam a suíte de *benchmarks* denominada SPEC CPU2006 para medir o impacto das soluções no desempenho das aplicações protegidas. Além disso, muitas delas empregam esse conjunto de aplicações para avaliar o seu funcionamento e detectar falsos positivos. De fato, como esse conjunto de *benchmarks* é composto por uma vasta gama de códigos pertencentes a aplicações reais usadas frequentemente por usuários convencionais, ele pode ser considerado uma base de dados representativa do que se espera de um ambiente computacional padrão.

O CPU2006 é uma suíte de *benchmarks* desenvolvida pela *Standard Performance Evaluation Corporation* (SPEC) e aceita como padrão por indústrias do setor de computação que se destina a testar exaustivamente o processador, sistema de memória e compilador de um sistema [127]. Trata-se de uma carga de trabalho cuidadosamente selecionada a partir de aplicações de usuário reais para prover uma medida comparativa do desempenho computacional intensivo de um amplo espectro de hardwares. Os conjuntos de testes que compõem o SPEC CPU2006 são disponibilizados na forma de código-fonte, o que permite a avaliação dos códigos executáveis em diversos ambientes computacionais. Conforme detalhado nas Seções 4.1.3 (Avaliação empírica) e 4.1.4 (Avaliação de desempenho), neste trabalho os códigos do SPEC CPU2006 foram compilados utilizando-se os compiladores mais populares das linguagens C e C++ nos sistemas operacionais Linux e Windows: GCC e Visual C/C++ [128], respectivamente. Essa escolha objetivou adequar a cobertura de possibilidades avaliadas neste estudo aos ambientes mais utilizados, reduzindo a probabilidade de ocorrência e o impacto dos casos não tratados.

4.1.1.2 Exploit Database

O *Exploit Database* é o maior repositório de *exploits* públicos e softwares vulneráveis correspondentes, desenvolvido para uso em testes de invasão e pesquisas sobre vulnerabilidades. Segundo seus mantenedores [129], seu objetivo é servir como a mais completa coleção de *exploits* coletados de submissões diretas, listas de e-mails e outras fontes públicas. Ele está disponível gratuitamente e permite a execução de consultas por palavras em seu banco de dados, o que facilitou a seleção de *exploits* que utilizam a técnica ROP.

Outra funcionalidade interessante oferecida nesse repositório é o link para download das versões vulneráveis das aplicações correspondentes aos *exploits* disponibilizados. Apesar de não estar disponível para todos os exemplares de *exploits* publicados, quando presente, essa opção facilita o processo de teste dos códigos maliciosos, já que na maioria dos casos os softwares vulneráveis são descontinuados e removidos dos repositórios de download oficiais dos desenvolvedores.

Finalmente, a verificação dos *exploits* catalogados, realizada por voluntários, indica se os códigos estão funcionando corretamente. Essa informação é importante porque muitas vezes os *exploits* disponibilizados necessitam de ajustes para funcionar em ambientes diferentes daqueles onde foram desenvolvidos, o que pode consumir muito tempo e esforço. Assim, durante a seleção

dos *exploits* ROP utilizados neste trabalho, pôde-se priorizar aqueles que já haviam sido verificados e que possuíam links para as aplicações vulneráveis correspondentes.

Deve-se salientar que muitos trabalhos utilizam *exploits* ROP desenvolvidos pelos próprios autores das proteções [61], [89], [90], [93], [105], o que pode dar margem a vícios e erros de concepção, já que os autores seguem uma mesma linha de raciocínio durante a construção dos artefatos. Ao utilizar *malwares* públicos variados, este trabalho oferece uma visão mais realista do comportamento da solução em ambientes de produção, adequando-se a diferentes estratégias de desenvolvimento de *exploits*.

4.1.2 Avaliação teórica

Conforme discutido na Seção 3.2.6, a quantidade de instruções executadas por cada *gadget* que compõe um ataque ROP é extremamente pequena. Em geral, os *gadgets* possuem de 1 a 3 instruções apenas [106]. Assim, em um esquema de proteção com uma janela de instruções capaz de registrar, por exemplo, as últimas 32 instruções executadas, um ataque ROP alcançaria uma densidade de, no mínimo, 10 desvios indiretos.

Inicialmente, pode-se imaginar que existirão muitos casos de aplicações cujas execuções normais apresentem uma elevada densidade de desvios indiretos, em função da execução de laços (*loops*) para repetição de instruções. Contudo, é importante lembrar que – fora as instruções de retorno – as instruções de desvio indireto são raramente utilizadas, restringindo-se a situações muito específicas, como chamadas de funções virtuais (em linguagens orientadas a objetos), estruturas de controle do tipo "*switch-case*", chamadas para ponteiros de funções e chamadas de funções pertencentes a bibliotecas ligadas dinamicamente [98], [130].

Para aplicações desenvolvidas em linguagens orientadas a objetos, onde o índice de desvios indiretos é maior, em média, uma instrução de desvio indireto (incluindo instruções de retorno) é executada a cada 32 instruções [131]. No caso de aplicações escritas em C, esse índice sobe para um desvio indireto a cada 96 instruções. Em ambos os casos, essa frequência de desvios é muito menor do que a frequência de saltos observada em cadeias de *gadgets*. A título de comparação, a Tabela 4.1 apresenta a frequência de instruções de desvio indireto presente nos *exploits* ROP catalogados neste trabalho. Nela, constata-se que os códigos maliciosos executam, em média, aproximadamente uma instrução de desvio indireto a cada 2 instruções.

Tabela 4.1: Proporção de instruções de desvio indireto nos *exploits* catalogados

Exploit	Nº total de instruções	Nº de desvios indiretos	Proporção
Windows			
Wireshark * [139]	49	24	2,04
DVD X Player * [140]	62	28	2,21
Zinf Audio Player * [141]	95	38	2,50
D.R. Audio Converter * [142]	69	36	1,92
Firefox – use after free * [143]	51	21	2,43
Firefox – integer overflow * [144]	36	18	2,00
PHP [145]	53	21	2,52
AoA Audio Extractor [132]	212	81	2,62
ASX to MP3 Converter [133]	150	59	2,54
Free CD to MP3 Converter [134]	47	19	2,47
Média	**82**	**35**	**2,33**
Linux			
ProFTPD Debian * [135]	66	24	2,75
ProFTPD Ubuntu * [135]	45	19	2,37
PHP [136]	81	27	3,00
Wireshark [137]	69	30	2,30
NetSupport [138]	45	14	3,21
Média	**61**	**23**	**2,73**
MÉDIA GERAL	**75**	**31**	**2,46**

* *Exploits* que utilizam também as instruções indiretas JMP ou CALL para interligar os *gadgets*.

A seguir, são analisados os três cenários de aplicações autênticas cujas frequências de desvios indiretos teoricamente mais se aproximam da frequência tipicamente registrada durante ataques ROP:

- Códigos de laços de repetição mínimos com uma estrutura de controle interna (Seção 4.1.2.1).
- Códigos de funções recursivas (Seção 4.1.2.2).
- Códigos de laços de repetição mínimos com uma chamada de função interna (Seção 4.1.2.3).

4.1.2.1 Laços de repetição com estrutura de controle

Nos tradicionais laços de execução, são as instruções de desvios condicionais que garantem a repetição do corpo do laço, dependendo das condições de parada. Na arquitetura x86, todas as instruções de salto condicionais possuem um endereço imediato, registrado na própria instrução [146]. Desvios indiretos são inseridos dentro de laços de repetição apenas quando existem estruturas de controle, como “if/else”, ou chamadas a procedimentos. O impacto desses dois casos na frequência de desvios indiretos foi analisado isoladamente.

Quando uma estrutura de controle, como “if/else”, exige a execução de um salto para uma posição cuja distância em relação ao contador de programa (*Program Counter* - PC) é maior do que

é possível representar na instrução, é utilizada uma instrução de salto indireto. Como na arquitetura x86 os valores imediatos podem ser de 8, 16 ou 32 bits, é possível efetuar um desvio direto para uma distância entre o destino do salto e a instrução de desvio de até 2 GB. No entanto, apesar de pouco usual por fugir do padrão, podem existir códigos customizados em que o corpo de um laço de repetição inclua uma instrução de desvio indireto.

Para avaliar o impacto desse tipo de situação na frequência de desvios indiretos, foi analisado um exemplo de código que efetua um "laço mínimo". Esse laço é considerado mínimo porque possui apenas a estrutura necessária para analisar a condição de repetição do laço, além da estrutura de controle "if/else", responsável pela inserção de um desvio indireto (JMP). A Listagem 4.1 apresenta, à esquerda, um exemplo de código de um laço mínimo escrito na linguagem C e, à direita, o código *assembly* equivalente. O código *assembly* está representado segundo a sintaxe adotada pela Intel [147].

Listagem 4.1. Exemplo de código de laço mínimo, em C e *assembly*.

```c
int main(){
   int i=0;
   // executa loop que não faz nada
   do {
      // "if/else" força a inserção de um JMP
      if(i < 1000){i++;}
      else{i++;}
   } while (i < 1000);
   return(0);
}
```

```asm
main:
  push ebp
  mov  ebp, esp
  sub  esp, 4
  mov  DWORD PTR [ebp-4], 0
.L4:
  cmp  DWORD PTR [ebp-4], 999
  jg   .L2
  add  DWORD PTR [ebp-4], 1
  jmp  .L3
.L2:
  add  DWORD PTR [ebp-4], 1
.L3:
  cmp  DWORD PTR [ebp-4], 999
  jle  .L4
  mov  eax, 0
  leave
  ret
```

Ao analisar o código *assembly*, constata-se que a instrução de desvio incondicional (JMP) será executada a cada 6 instruções, frequência quase três vezes menor do que a média observada nas cadeias de *gadgets* presentes nos *exploits* ROP catalogados neste trabalho (Tabela 4.1). De fato, um laço mínimo com uma estrutura de seleção interna necessitará, no mínimo, das seguintes instruções:

- Dois conjuntos de instruções "comparação – desvio condicional" (CMP - JC): um para o teste do laço e outro para o teste de seleção interno "if" (4 instruções).
- Uma instrução de JMP para saltar o código do "else", caso o código do "if" seja executado. No exemplo ilustrado na Listagem 4.1, utilizou-se um JMP direto, já que esse é o comportamento padrão, mas a estrutura do laço seria a mesma caso essa instrução fosse substituída por uma instrução de salto indireto.

- Uma instrução a ser executada dentro do "if" ou do "else" (daí o termo "laço mínimo").

Na prática, para tornar o laço de repetição útil, seria incluída, pelo menos, mais uma instrução de máquina, já que o laço mínimo apresentado apenas incrementa a variável que controla a condição de repetição. Isso reduziria ainda mais a frequência de desvios indiretos, o que permite concluir que a eventual execução de instruções de desvio indireto dentro de laços de repetição não acarreta em situações de falso positivo com a solução proposta.

4.1.2.2 Funções recursivas

Outra situação que pode gerar uma alta densidade de desvios indiretos é a frequente chamada a procedimentos, seja dentro de um laço, seja em chamadas recursivas. Funções com recursividade no início podem gerar uma alta densidade de desvios do tipo CALL, enquanto aquelas com recursividade em cauda podem acarretar em uma elevada frequência de instruções de retorno. Novamente, apesar de não ser comum a existência de funções recursivas que efetuam a chamada recursiva através de uma instrução de CALL indireto, códigos desenvolvidos manualmente em linguagem de montagem podem, eventualmente, fugir do padrão. Em função disso, assim como na análise de um laço de repetição mínimo, foi desenvolvido um exemplo de função recursiva mínima, que executa apenas a checagem da condição de fim da recursão. Os códigos que representam essa função recursiva mínima nas linguagens C e *assembly* estão indicados na Listagem 4.2.

Listagem 4.2. Exemplo de código de função recursiva mínima, em C e *assembly*.

```c
void recursao_minima(int i){
   // condição de parada da recursão
   if(i>0){
      // chamada recursiva
      recursao_minima(i-1);
   }
   return;
}

int main(){
  //inicia chamada recursiva de função
   recursao_minima(1000);
   return(0);
}
```

```asm
recursao_minima:
   sub  esp, 4
   cmp  DWORD PTR [esp+8], 0
   jle  .L1
   mov  eax, DWORD PTR [esp+8]
   sub  eax, 1
   mov  DWORD PTR [esp], eax
   call recursao_minima
.L1:
   add  esp, 4
   ret

main:
   sub  esp, 4
   mov  DWORD PTR [esp], 1000
   call recursao_minima
   mov  eax, 0
   add  esp, 4
   ret
```

Para simular o cenário com a maior frequência de desvios possível, foi utilizada a otimização de compilação que omite o ponteiro de *frame*, liberando o registrador EBP para uso geral [148]. O uso desse tipo de otimização implica no descarte das instruções "PUSH EBP" e "MOV EBP, ESP", que tradicionalmente aparecem no início do código de uma função. Em

chamadas sucessivas a essas funções, a remoção dessas instruções pode impactar significativamente na frequência de instruções de desvio indireto. A omissão do ponteiro de *frame* acarreta também na substituição da instrução "LEAVE" pela instrução "ADD ESP, *valor*", mas essa alteração não repercute em mudanças na densidade de instruções de salto indireto.

Ao analisar o código *assembly* apresentado na Listagem 4.2, constata-se que a instrução de chamada de procedimento (CALL) será executada a cada 7 instruções. Se a otimização de compilação que omite o ponteiro de *frame* não for utilizada, essa relação passa para uma instrução de chamada de procedimento a cada 9 instruções. Na prática, assim como no exemplo do laço mínimo, existirão outras instruções dentro da função para torná-la útil. Portanto, constata-se que a execução sucessiva de instruções de chamada de procedimento (CALL) no início de funções recursivas também não acarreta em falsos positivos.

O mesmo não pode ser dito em relação à execução sucessiva de instruções de retorno (RET) em funções com recursividade em cauda, como aquela apresentada na Listagem 4.2. Esse é um caso em que a frequência de desvios pode atingir um salto a cada 2 instruções executadas, caso a omissão do ponteiro de *frame* seja empregada. Mesmo se o ponteiro de *frame* estiver em uso, a densidade de desvios será de 1 para 3, o que pode acarretar em falsos positivos. Para evitar esse tipo de erro, pode-se utilizar uma mecanismo para identificar a ocorrência de recursões em cauda, conforme descrito no esquema alternativo apresentado na Seção 4.3.

4.1.2.3 Laços de repetição com chamada de função

A fim de verificar a possibilidade de ocorrência de falsos positivos, analisou-se ainda um terceiro exemplo de código. Trata-se de um laço de repetição com uma chamada interna para um procedimento com poucas instruções. A listagem 4.3 apresenta exemplos de código para essa situação na linguagem C e em *assembly*, considerando-se a omissão do ponteiro de *frame*. No exemplo ilustrado, a função chamada durante a execução do laço de repetição executa uma única operação, responsável por incrementar uma variável global.

Nesse cenário, a frequência de execução das instruções de desvio, em relação ao total de instruções executadas, atinge uma relação de 2 para 8. Esse é o caso que mais se aproxima da densidade média de desvios observada em ataques ROP. Mesmo assim, a frequência de saltos ainda é quase duas vezes menor do que a média de desvios observada nas cadeias de *gadgets* catalogadas (Tabela 4.1). Se a otimização de omissão do ponteiro de *frame* não for empregada, essa densidade cai para 2 desvios a cada 11 instruções.

Listagem 4.3. Exemplo de código de laço de repetição com chamada interna de procedimento, em C e *assembly*.

```
int cont=0; // variável global

// apenas incrementa a variável global
void contador(){
   cont++;
   return;
}

int main(){
  int i=0;
  // laço de repetição
   do {
      contador();
      i++;
   } while (i < 1000);
  return(0);
}
```

```
cont:
   .zero
   .globl

contador:
   mov  eax, DWORD PTR cont
   add  eax, 1
   mov  DWORD PTR cont, eax
   ret

main:
   sub  esp, 4
   mov  DWORD PTR [esp], 0
.L3:
   call contador
   add  DWORD PTR [esp], 1
   cmp  DWORD PTR [esp], 999
   jle  .L3
   mov  eax, 0
   add  esp, 4
   ret
```

Na prática, a frequência de desvios quase sempre é menor do que os piores casos apresentados, porque não se constrói uma função apenas para incrementar uma variável, tampouco se cria uma função recursiva que apenas chama a si mesma. No caso de códigos compilados, que correspondem à imensa maioria das instruções de máquina executadas, os próprios compiladores eliminam estruturas desnecessárias como aquelas incluídas nos exemplos de códigos analisados (*function inlining optimization*), a fim de otimizar o código gerado [149]. Além disso, conforme já mencionado, a ocorrência de desvios do tipo indireto é pouco comum, independente da situação específica do seu uso [131]. Nos testes efetuados com aplicações reais, por exemplo, nenhum caso de falso positivo foi constatado, conforme discutido na Seção 5.3. Ainda assim, caso experimentos futuros identifiquem a ocorrência de falsos positivos durante a análise de executáveis, pode-se verificar se as chamadas ou retornos sequenciais de funções utilizam o mesmo endereço. Essa checagem permite distinguir, mediante um pequeno custo computacional adicional, tanto laços de repetição com chamada a procedimentos pequenos quanto funções recursivas com poucas instruções.

4.1.3 Avaliação empírica

A etapa de avaliação empírica destinou-se a comprovar duas conjecturas primordiais para este trabalho:

- Confirmar, na prática, que o controle da frequência de instruções de desvio indireto permite a detecção acurada de ataques ROP.

- Demonstrar o correto funcionamento do protótipo desenvolvido no bloqueio de *exploits* ROP reais.

Para efetuar a análise da primeira conjectura, foram contabilizadas as frequências máximas de desvios indiretos alcançadas pelos *benchmarks* da suíte SPEC CPU2006 e por 15 *exploits* ROP (10 para Windows e 5 para Linux – vide Tabela 4.1). Através dessa contagem, buscou-se determinar o tamanho de janela ideal para a detecção de ataques ROP. Ou seja, foi identificado o tamanho de janela que maximiza a diferença da frequência máxima de desvios indiretos registrada por cadeias de *gadgets* ROP e por aplicações autênticas (representadas pelos *benchmarks* da suíte SPEC CPU2006), conforme ilustrado na Figura 3.1.

Foram registradas as frequências máximas de instruções de desvio indireto alcançadas com janelas dos seguintes tamanhos: 32, 64, 96 e 128. A razão para utilizar esses tamanhos de janelas está ligada a questões de otimização de desempenho, detalhadas nas Seções 4.2.3.1 e 4.2.3.2.

Para executar a contagem dos desvios indiretos, foram utilizados dois métodos distintos: um para os *benchmarks* e outro para os *exploits* ROP. No caso dos *benchmarks*, monitorou-se em tempo real a sua execução através do *framework* de instrumentação binária Pin (vide Seção 4.2.1). Por outro lado, como o Pin impõe um elevado tempo de execução a algumas das aplicações atacadas, optou-se por adotar um procedimento alternativo para monitorar os desvios indiretos executados pelos *exploits* ROP.

No caso dos *exploits* ROP, utilizou-se um *script* desenvolvido na linguagem de programação Perl. Esse script lê um fluxo de zeros (0) e uns (1) de um arquivo de entrada, contabiliza a densidade máxima e imprime o resultado. Cada valor (0 ou 1), anotado em uma linha diferente do arquivo de entrada, denota uma instrução executada pelo *exploit*. As instruções de desvio indireto são representadas pelo valor 1 (um) e as demais instruções são identificadas com o valor 0 (zero). De acordo com o tamanho de janela passado como parâmetro para o *script* pela linha de comandos, ele imprime a maior contagem de valores 1 (um) que aparece no arquivo de entrada dentro do tamanho de janela escolhido. Para criar esses arquivos de entrada com os valores zero (0) e um (1), que representam o fluxo de execução dos *exploits*, recorreu-se aos *debuggers* Ollydbg [150] – no Windows – e GDB [151] – no Linux. Ao recriar os ataques ROP em um ambiente de testes composto por máquinas virtuais, à medida que cada instrução do *exploit* era executada pelo *debugger*, o valor correspondente (0 ou 1) era escrito manualmente em um arquivo de texto.

Posteriormente, esse arquivo de texto foi usado como entrada para o *script* que contabiliza a densidade máxima de desvios indiretos.

A descrição completa dos ambientes utilizados durante os experimentos destinados a confirmar a primeira conjectura, bem como os detalhes de configuração e execução dos testes, são apresentados nas Seções 5.1 e 5.2. Os resultados são discutidos na Seção 5.3.

A comprovação da segunda conjectura foi realizada através da execução direta dos mesmos quinze (15) *exploits* ROP contra aplicações vulneráveis protegidas pelo protótipo desenvolvido. Para isso, foram usadas dez (10) aplicações vulneráveis para o Windows e cinco (5) para o Linux. O número de *exploits* para os dois sistemas operacionais não coincide porque a disponibilidade de *exploits* públicos para Linux é muito menor, especialmente quando a aplicação vulnerável correspondente também deve ser encontrada para instalação e teste.

Primeiramente, o correto funcionamento dos *exploits* foi testado contra as aplicações vulneráveis desprotegidas. Após constatar a consolidação dos ataques, esses mesmos *exploits* foram executados novamente contra as aplicações vulneráveis correspondentes, porém dessa vez protegidas pela nossa solução. A descrição completa dos ambientes utilizados durante esses experimentos, bem como os detalhes de configuração e execução dos testes, estão descritos nas Seções 5.1 e 5.2. Os resultados são apresentados na Seção 5.4.

4.1.4 Avaliação de desempenho

Conforme discutido na Seção 3.1, o impacto das proteções contra ataques ROP no desempenho das aplicações protegidas é uma fator primordial a ser estudado. Para medir a elevação no tempo de CPU (*overhead*) imposta pela solução desenvolvida neste trabalho, executou-se os aplicativos que compõem o *benchmark* SPEC CPU2006 nos seguintes cenários:

1. Executados diretamente pelas máquinas de testes.
2. Executados sob a supervisão do *framework* de instrumentação binária (Pin – vide Seção 4.2.1), mas sem qualquer código de instrumentação.
3. Executados sob a supervisão do Pin com o código de instrumentação que implementa a proteção desenvolvida neste trabalho (RIP-ROP – vide Seção 4.2).
4. Executados sob a supervisão do Pin com um código de instrumentação que implementa a proteção denominada ROPdefender [94] (Seção 3.2.2), usada como modelo de referência.

5. Executados sob a supervisão do Pin com um código simples de instrumentação que apenas conta o número de blocos de código (*Basic Blocks* ou BBLs – vide Seção 4.2.1) executados.

O primeiro cenário corresponde ao referencial mínimo para a comparação de desempenho, pois é usado apenas para registrar o tempo de execução consumido pelos *benchmarks* em situação normal. Em outras palavras, ele indica o desempenho dos executáveis quando nenhuma proteção contra ataques ROP está ativa.

O segundo cenário foi utilizado para estimar a parcela de *overhead* mínima a que toda solução baseada no Pin está sujeita. Como o Pin é reconhecido como um dos *frameworks* de instrumentação binária dinâmica mais eficientes (vide Seção 4.2.1), essa estimativa pode ser generalizada para todos os instrumentadores binários dinâmicos. Esse cenário serve também para distinguir a parcela mínima de *overhead* incorporada ao RIP-ROP (avaliado no terceiro cenário) que decorre exclusivamente da abordagem de implementação adotada no desenvolvimento do protótipo.

A estratégia de proteção contra ataques ROP mais frequente entre as soluções estudadas é a construção de uma pilha-sombra (vide Tabela 3.1). O ROPdefender [94], por sua vez, corresponde à proteção mais recente que utiliza a abordagem de instrumentação binária dinâmica para implementar essa estratégia. Por isso, essa solução foi adotada no quarto cenário como modelo de referência para a avaliação de desempenho.

Além de reproduzir o ROPdefender, foi desenvolvido também um código de instrumentação cuja única função é contar o número de BBLs executados pelo programa. O desempenho desse código de instrumentação é avaliado com o intuito de balizar o desempenho do RIP-ROP, já que o principal fator de influência no desempenho do protótipo é justamente a necessidade de instrumentar todos os BBLs de um programa (vide Seção 4.2.2). Ao comparar o RIP-ROP com um código simples que também instrumenta todos os BBLs de uma aplicação, pode-se estimar o grau de otimização do protótipo.

A descrição completa dos ambientes utilizados durante esses experimentos, bem como os detalhes de configuração e execução dos testes, são apresentados nas Seções 5.1 e 5.2. Os resultados são discutidos na Seção 5.4.

4.2 RIP-ROP

O protótipo implementado para materializar a proteção contra ataques ROP desenvolvida neste trabalho, permitindo o emprego imediato dessa solução em ambientes de produção, recebeu o nome

RIP-ROP. Trata-se do mnemônico para (*Reuse of Instructions Prevention against Return-Oriented Programmimg*). O termo RIP também faz alusão ao verbo em inglês de mesma escrita, que significa rasgar, romper, dilacerar. Dessa forma, acredita-se que esse nome reflete bem o objetivo da proteção: prevenir o reúso de instruções de desvio indireto, inerente aos ataques ROP, frustando a intenção dos atacantes de executar códigos arbitrários nos sistemas alvejados.

Para viabilizar o desenvolvimento da proteção através da instrumentação binária dinâmica de código, necessária para a análise em tempo real das instruções executadas, utilizou-se o instrumentador binário Pin, descrito na Seção 4.2.1. Em seguida, são detalhados os aspectos de implementação da estratégia de controle da frequência de instruções de desvio indireto (Seção 4.2.2). A Seção 4.2.3 é destinada à explicação das principais decisões de implementação motivadas pela necessidade de otimizar o desempenho do protótipo. Finalmente, na Seção 4.3 são discutidos os desafios relacionados à proteção implementada.

4.2.1 Pin

O Pin é um *framework* de instrumentação binária dinâmica desenvolvido pela Intel para as arquiteturas IA-32 e x86-64. As ferramentas criadas utilizando-se o Pin, chamadas de *Pintools*, podem ser usadas para a análise de programas pertencentes ao espaço de aplicações do usuário nos sistemas operacionais Android, Linux, OSX e Windows. Como trata-se de uma ferramenta de instrumentação binária dinâmica, a instrumentação é realizada na etapa de execução de arquivos binários previamente compilados. Portanto, o Pin não requer a recompilação de códigos-fontes e permite a análise de programas que geram códigos dinamicamente. Para isso, o *framework* salva e restabelece automaticamente o conteúdo dos registradores que são sobrescritos pelos códigos injetados, garantindo que a aplicação possa continuar a executar [152], [153].

Por apresentar o melhor desempenho entre as aplicações de instrumentação binária dinâmica [118], [119], [122], além de possuir uma comunidade de desenvolvimento e suporte ao usuário bastante ativa, o Pin foi escolhido como a base para o desenvolvimento do RIP-ROP. Além dessas vantagens, ele provê uma rica API que facilita o acesso a informações de contexto, como o conteúdo de registradores ou o endereço de instruções.

O Pin é uma ferramenta de instrumentação binária do tipo JIT (*Just-in-time*), que permite a análise e eventual modificação do código à medida que ele é executado. Para isso, antes que uma instrução seja executada pelo processador, esse *framework* intercepta a instrução, gera e executa

novos códigos, e garante que o *framework* retomará o controle do processador após a execução da instrução [118].

A Figura 4.1 ilustra a arquitetura geral do Pin. Quando uma aplicação qualquer é instrumentada via Pin, três programas binários diferentes compartilham o mesmo espaço de endereços: o Pin, a *Pintool* e a aplicação. O Pin é o código que controla e instrumenta a aplicação. A *Pintool* corresponde ao código das rotinas de instrumentação e análise criadas pelo usuário do *framework*. As *Pintools* podem ser escritas nas linguagens de programação C ou C++ e correspondem ao componente onde as soluções do usuário devem ser implementadas, incluindo o RIP-ROP. O código da *Pintool* é ligado a uma biblioteca de APIs específicas do Pin, que permitem a comunicação entre o código do usuário e o Pin.

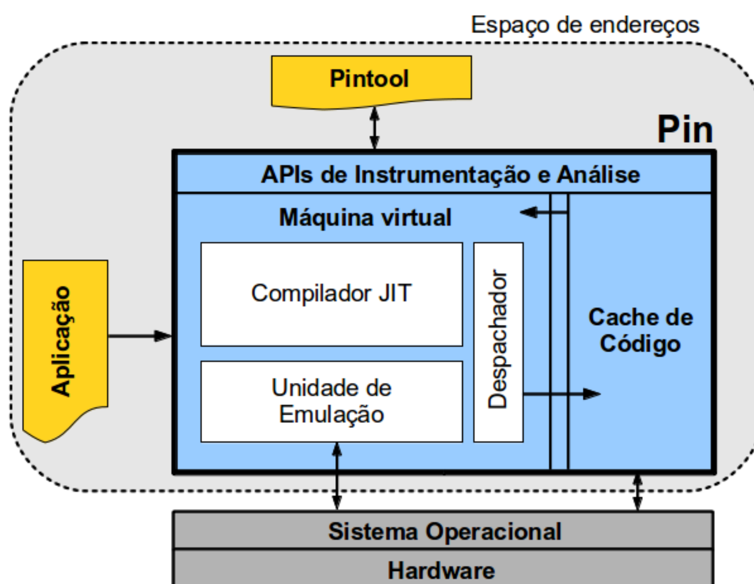

Figura 4.1: Arquitetura do Pin [118].

Conceitualmente, cada *Pintool* é constituída por dois componentes: um mecanismo que decide onde e qual código deve ser inserido, denominado código de instrumentação; e o código a ser enxertado nos pontos de inserção, denominado código de análise [153]. Ambos os componentes residem em uma única *Pintool*. Por conta disso, as *Pintools* podem ser encaradas como *plugins* capazes de guiar o processo de geração de código controlado pelo Pin. Para fazer isso, as *Pintools* registram junto ao Pin rotinas de instrumentação que são chamadas pelo *framework* sempre que um novo bloco de código precisa ser gerado. Essas rotinas de instrumentação, que correspondem ao componente de instrumentação mencionado, inspecionam o código a ser gerado, investigam suas propriedades estáticas e decidem onde inserir chamadas para funções de análise. Por sua vez, as rotinas de análise reúnem dados sobre a aplicação, que podem ser recebidos do Pin na forma de parâmetros, e executam as ações desejadas pelo usuário. Uma *Pintool* pode ainda registrar junto ao

Pin rotinas para serem chamadas no caso da ocorrência de alguns eventos específicos, como a criação de novas *threads* ou o lançamento de exceções.

Apesar de compartilharem o mesmo espaço de endereços, o Pin, a *Pintool* e a aplicação não compartilham nenhuma biblioteca. Dessa forma, evita-se qualquer tipo interação não desejada entre esses três binários. Teoricamente, isso garante também que um atacante não seja capaz de corromper as áreas de memória do Pin ou da *Pintool* com o intuito de burlar uma proteção desenvolvida no Pin.

Como podemos verificar na Figura 4.1, o Pin é constituído por uma máquina virtual, uma área de cache de código e um conjunto de APIs invocáveis pelas *Pintools*. A máquina virtual, por sua vez, é composta por um compilador JIT, uma unidade de emulação e um despachador. Assim que o Pin assume o controle de uma aplicação, a máquina virtual coordena os seus componentes para executar a aplicação, respeitando as diretivas estabelecidas na *Pintool*.

O compilador JIT compila a aplicação, adicionando os códigos de instrumentação nas devidas posições. O código compilado é então armazenado no cache de código, que tem a função de reduzir a penalidade de desempenho caso um trecho de código seja invocado múltiplas vezes. Se uma sequência de instruções for repetida, o código pode ser obtido diretamente do cache de código, evitando-se o esforço de recompilação. Na prática, o código original da aplicação nunca é executado diretamente e apenas as instruções armazenadas no cache são encaminhadas para o processador.

Depois que o código a ser executado está posicionado no cache, o despachador transfere o fluxo para a aplicação. A troca de contexto entre o código da máquina virtual e o código da aplicação, armazenado no cache, envolve o salvamento e a recuperação do conteúdo dos registradores. Essa troca de contexto é um dos principais responsáveis pelo elevado *overhead* imposto pelas ferramentas de instrumentação binária dinâmica.

A unidade de emulação é necessária para interpretar instruções que não podem ser executadas diretamente, tais como instruções privilegiadas e chamadas de sistema específicas. Como o Pin trabalha acima do sistema operacional, ele é capaz de capturar apenas instruções no nível do usuário.

Os blocos de instruções compilados a cada passo pelo Pin são denominados *traces* (rastros) [153]. Um *trace* é uma sequência contínua de instruções que normalmente inicia no alvo de um desvio executado e termina com uma instrução de salto incondicional, incluindo CALLs e

RETs. O Pin garante que cada *trace* possui um único ponto de entrada, mas um único *trace* pode possuir vários pontos de saída, dependendo da existência de desvios condicionais. Se um desvio que aponta para o meio de um *trace* é executado, o Pin constrói um novo *trace* que começa com a instrução alvo do salto.

Aumentando a granularidade da divisão de blocos de instruções, o Pin divide cada *trace* em blocos básicos, ou BBLs (*Basic Blocks*). Um BBL corresponde a uma sequência de instruções com um único ponto de entrada e um único ponto de saída. De maneira análoga ao que foi dito em relação aos *traces*, desvios para o meio de um BBL geram um novo *trace* e, consequentemente, um novo BBL. Em muitos casos é possível registrar uma função de análise para BBLs ao invés de uma rotina para cada instrução. Isso permite reduzir o número de chamadas a funções de análise, tornando a instrumentação mais eficiente.

Quando uma aplicação é iniciada pelo Pin, ele intercepta o primeiro *trace* de instruções e o compilador JIT gera novas instruções para incorporar o código de instrumentação. As aplicações são compiladas um *trace* de cada vez. A cada saída de um *trace*, o fluxo de execução é inicialmente desviado para uma área predeterminada, que redireciona o fluxo para a máquina virtual. Então, a máquina virtual reconhece o endereço de destino – que pode não ser conhecido previamente, no caso de desvios indiretos – gera um novo *trace* que inicie com a instrução alvo do desvio (somente caso o *trace* já não esteja armazenado no cache) e retoma a execução da aplicação a partir do início do novo *trace*.

4.2.2 Módulo de controle da frequência de desvios indiretos

O código-fonte do RIP-ROP corresponde à *Pintool* utilizada pelo Pin para definir os códigos de instrumentação e, por isso, foi implementado em um módulo na linguagem C++. A proteção elaborada requer a criação de uma estrutura de armazenamento, aqui designada pelo termo “janela”, para registrar as últimas instruções executadas. Assumindo-se que a janela possui um tamanho N, pode-se dizer que a função da janela é permitir a contagem do número de desvios indiretos executados nas últimas N instruções. Nessa janela, as posições correspondentes às instruções de desvio indireto são anotadas com um bit 1 e as demais instruções são representadas pelo bit 0.

A Figura 4.2 ilustra a lógica de verificações utilizada para controlar a execução de instruções de desvio indireto. Ao executar qualquer instrução, a janela precisa ser atualizada. Para evitar um *overhead* excessivo decorrente da análise de todas as instruções de um programa, na abordagem de instrumentação binária dinâmica provida pelo Pin é possível explorar o conceito de

bloco básico (*Basic Block*, ou BBL – vide Seção 4.2.1) [153]. Assim, insere-se um código de análise para um BBL, ao invés de avaliar cada instrução do programa, tornando a instrumentação mais eficiente.

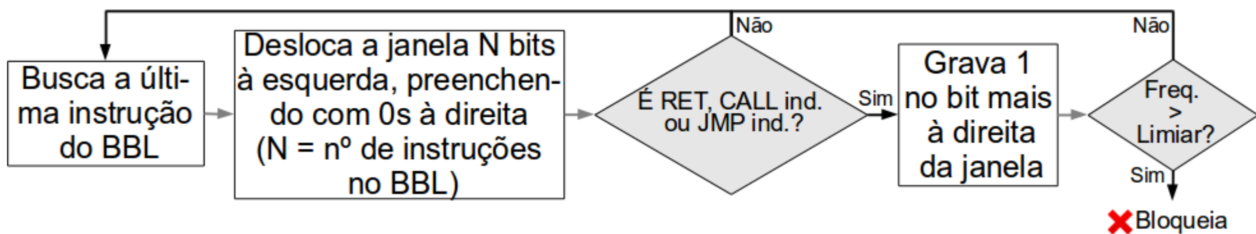

Figura 4.2: Lógica de controle das instruções de desvio indireto.

No esquema proposto, utiliza-se uma API do Pin (BBL_InsTail) para buscar a última instrução do BBL que está sendo instrumentado. Como, por definição, um BBL é um bloco de instruções com um único ponto de entrada e um único ponto de saída, sabe-se que a única instrução desse bloco que eventualmente poderá corresponder a um desvio indireto será a última instrução do bloco.

Depois de capturar a última instrução do BBL, o RIP-ROP desloca a janela de instruções à esquerda, preenchendo os bits deslocados à direita com o bit zero (0). O número de bits deslocados corresponde à quantidade de instruções existentes no BBL em análise. Essa operação de deslocamento da janela obrigatoriamente deve ser realizada para todos os BBLs executados pela aplicação, independente de eles possuírem alguma instrução de desvio indireto ou já terem sido executados anteriormente. Esse é um dos principais fatores que pesa negativamente no desempenho da proteção, já que acarreta em uma mudança de contexto entre o *framework* e a aplicação instrumentada a cada execução de um BBL (vide Seção 5.5). Considerando que boa parte dos laços de repetição são mapeados para BBLs, é fácil constatar o impacto dessa característica no desempenho de aplicações que possuem muitos laços de repetição curtos. Infelizmente, em função da forma como o Pin foi concebido, não foi possível evitar esse *overhead* no funcionamento do RIP-ROP.

Após deslocar a janela de instruções, checa-se a última instrução do BBL. Caso ela não corresponda a um desvio indireto, nada mais é preciso ser feito e a execução da aplicação prossegue até que um novo BBL seja buscado. Por outro lado, se a última instrução do BBL corresponder a um desvio indireto, o RIP-ROP grava o valor um (1) no bit mais à direita da janela e calcula a quantidade de desvios indiretos registrados na janela. Caso o valor calculado ultrapasse o limiar estabelecido para a aplicação, o RIP-ROP sinaliza no log a ocorrência de um ataque ROP e encerra a execução da aplicação.

4.2.3 Otimizações

Uma vez que as ferramentas de instrumentação binária dinâmica, por si só, já impõem uma sobrecarga computacional elevada às aplicações instrumentadas (vide Seção 3.3.3), é de suma importância otimizar ao máximo o código de instrumentação. Pensando nisso, foram adotados mecanismos com esse propósito específico no código do RIP-ROP. Os seguintes aspectos foram avaliados com foco na melhoria do desempenho do protótipo:

- Uso da instrução de máquina POPCNT para contar os bits habilitados na janela.
- Tamanhos de janelas de instruções múltiplos do tamanho dos registradores.
- Alinhamento dos dados de cada *Thread* às linhas de Cache da CPU.
- Ajuste na ordem dos testes condicionais do código de acordo com a quantidade mais frequente de bits deslocados.

Cada um desses aspectos está descrito nas Seções 4.2.3.1 a 4.2.3.4.

4.2.3.1 A instrução POPCNT

Considerando-se que na arquitetura x86 existem instruções de hardware que permitem deslocar os bits de um registrador e contar a quantidades de bits ativos [146], a execução das operações de atualização das janelas não acarretou em um *overhead* tão elevado quanto se fossem utilizados laços de repetição. A instrução SHL (deslocamento lógico para a esquerda, em direção aos bits mais significativos), por exemplo, usa um operando para indicar a quantidades de bits a serem deslocados de um registrador. Os bits menos significativos deslocados pela instrução são preenchidos com o valor zero (0). Isso evita a necessidade de executar um laço iterativo para deslocar a estrutura que representa a janela de instruções, o que exigiria um esforço computacional maior.

Para contar a quantidade de bits ativos em um operando, foi utilizada a instrução POPCNT (Contagem de população). Trata-se de uma instrução introduzida em 2007 pela AMD em sua microarquitetura denominada Barcelona [154]. Os processadores da família Intel Core introduziram a instrução POPCNT junto com a extensão do conjunto de instruções SSE4.2, em 2008 [155]. Desde então, os processadores produzidos por esses e outros fabricantes contam com uma instrução equivalente. Assim como no deslocamento da janela de instruções, o uso dessa instrução de hardware evita o elevado custo computacional de percorrer a janela para contar os bits ativos, garantindo uma considerável melhoria de desempenho.

4.2.3.2 Tamanho da janela

Naturalmente, para usufruir dos benefícios de desempenho oferecidos ao utilizar-se as instruções SHL e POPCNT, discutidas na Seção anterior (4.2.3.1), somente puderam ser definidas janelas de tamanhos compatíveis. As instruções SHL e POPCNT trabalham com operandos de 32 bits ou 64 bits (restrito às arquiteturas de 64 bits). Além disso, quase sempre as cadeias de *gadgets* usadas nos *exploits* ROP são curtas, com menos do que 100 instruções, já que elas destinam-se apenas a uma etapa inicial dos ataques responsável por superar a proteção do bit de execução (vide a Seção 2.3.2). Assim, a fim de estabelecer o melhor balanceamento entre desempenho e capacidade de bloqueio de ataques, foram criadas versões do RIP-ROP com janelas dos seguintes tamanhos: 32, 64, 96 e 128.

4.2.3.3 Linhas de Cache da CPU

Uma questão estrutural inerente à arquitetura dos processadores que costuma ocasionar a degradação do desempenho de aplicações que executam várias *threads* é o problema do falso compartilhamento (*false sharing*). Essa situação ocorre quando múltiplas *threads* acessam diferentes partes de uma mesma linha de cache da CPU e ao menos um desses acessos é uma escrita [153]. Para manter a coerência dos dados em memória, o computador copia os dados da cache de uma CPU para a outra, mesmo que as *threads* concorrentes não estejam efetivamente compartilhando nenhum dado. Normalmente, problemas de falso compartilhamento entre *threads* podem ser evitados ajustando-se o tamanho das estruturas de dados para que cada estrutura ocupe uma linha de cache diferente.

A implementação do RIP-ROP trata aplicações que lançam múltiplas *threads* através da criação de uma janela de instruções independente para cada *thread*. No entanto, isso não impede que mais de uma janela ocupe a mesma linha de cache da CPU, o que pode eventualmente acarretar no problema de falso compartilhamento. Para evitar essa situação indesejada, o RIP-ROP força a alocação de uma estrutura de dados inútil junto com a estrutura que representa a janela de instruções. O tamanho dessa estrutura de dados inerte é calculado para que a soma da área de armazenamento ocupada pela janela com o espaço ocupado por essa estrutura inútil corresponda ao tamanho exato de uma linha de cache da CPU. Assim, o protótipo força o armazenamento dos dados de cada *thread* em uma linha de cache diferente, evitando o problema de falso compartilhamento.

Para garantir o correto funcionamento desse mecanismo de prevenção do problema de falso compartilhamento, deve-se registrar no código-fonte do RIP-ROP o tamanho da linha de cache usada pelo processador. Em ambientes Linux, uma forma de identificar o tamanho da linha de cache

(em bytes) usada pelo processador do equipamento onde se pretende executar o RIP-ROP é executar o comando "getconf LEVEL1_DCACHE_LINESIZE" [156].

4.2.3.4 Número de bits deslocados

Para permitir o uso das instruções de hardware SHL e POPCNT no processo de deslocamento da janela de instruções, garantindo um melhor desempenho para essa operação (vide as Seções 4.2.3.1 e 4.2.3.2), a estrutura de dados usada no RIP-ROP para representar a janela de instruções corresponde a um arranjo de inteiros sem sinal de 32 ou 64 bits, dependendo da arquitetura. Para uma janela de 128 instruções em uma arquitetura de 32 bits, por exemplo, é utilizado um arranjo com 4 inteiros sem sinal de 32 bits.

Como a janela é composta por frações de 32 ou 64 bits, o processo de deslocamento dos bits por toda a janela utilizando a instrução de máquina SHL exige o uso de uma lógica de programação que reconheça os diversos casos que podem ocorrer. Tomando-se o mesmo cenário anterior como exemplo (janela de 128 instruções em uma arquitetura de 32 bits), se o número de bits a serem deslocados for maior do que 96 e menor do que 128, o conteúdo da posição mais à direita do arranjo de inteiros que representa a janela deve ser copiado para a posição mais à esquerda do arranjo. Em seguida, a instrução SHL deve operar apenas na posição mais à esquerda do arranjo e todas as demais posições do arranjo devem ser zeradas. Por outro lado, se o número de bits a serem deslocados na janela for maior do que 64 e menor do que 96, o conteúdo das duas posições mais à direita do arranjo deve ser copiado para as duas posições mais à esquerda do arranjo. Na sequência, a instrução SHL é usada para deslocar os bits das duas posições mais à esquerda do arranjo, tomando-se o devido cuidado para transportar os bits excedentes de uma posição para a outra. As demais posições do arranjo são zeradas.

Da análise desses dois exemplos fica fácil perceber que o código do RIP-ROP precisa possuir uma sequência de testes lógicos do tipo "if/else" para distinguir os casos e aplicar as operações corretas de deslocamento da janela de acordo com cada cenário. Esses testes lógicos são excludentes, ou seja, assim que um determinado caso é identificado, não há a necessidade de executar os testes "if/else" usados para identificar os demais casos. Portanto, uma maneira de otimizar esse trecho de código é garantir que os casos mais frequentes sejam testados primeiro, reduzindo o número de testes do tipo "if/else" executados na maior parte das ocasiões.

Para identificar os casos de deslocamento da janela de instruções mais comuns em cada cenário (tamanho de janela e arquitetura do processador), foi incluído no código da *Pintool* que

contabiliza a frequência de desvios indiretos um contador do número de ocorrências de cada caso. Ao término da execução da *Pintool*, a porcentagem de ocorrência dos casos é impressa em um arquivo de log, permitindo a rearrumação do código-fonte de modo que os casos mais frequentes sejam testados primeiro. A execução dos *benchmarks* que compõem a suíte SPEC CPU2006 mostrou que, em todos os testes, mais de 99% dos casos correspondem aos menores deslocamentos da janela de instruções. Isso ocorre porque, em geral, os BBLs possuem menos do que 32 ou 64 instruções. Assim, os testes "if/else" foram ordenados do menor para o maior, garantindo que, para os casos mais frequentes, uma quantidade menor de testes é executada.

4.3 Discussão

Apesar de nenhum caso desse tipo ter sido encontrado durante a realização da etapa de avaliação empírica, é possível que, conforme identificado na avaliação teórica (Seção 4.1.2.2), funções com recursividade em cauda registrem uma frequência de desvios indiretos acima do limiar, acarretando em falsos positivos. A fim de tratar essa possibilidade caso ela se concretize durante testes com outras aplicações ou ambientes, projetou-se uma solução alternativa que evita a classificação incorreta de códigos com funções recursivas em cauda e que pode ser codificada em cima do protótipo original do RIP-ROP mediante poucas alterações.

Essa solução requer a criação de três janelas para registrar as últimas instruções executadas. Na primeira janela, as posições correspondentes às instruções de desvio indireto são anotadas com um bit 1 (um) e as demais instruções são representadas pelo bit 0 (zero). A segunda janela é usada para manter-se um registro específico das instruções de retorno. Nessa estrutura de armazenamento, as instruções de retorno (RET) são marcados com o bit 1 (um), enquanto as demais instruções recebem o bit 0 (zero). A terceira janela é usada para controle do número de instruções de retorno executadas que não apontam para um endereço imediatamente posterior a uma instrução de chamada de função (CALL) (vide Seção 3.2.3).

A janela 1 funciona da mesma forma que a janela descrita para os casos gerais na Seção 4.2.2. Por sua vez, a janela 2 é usada para distinguir os casos de funções com recursividade em cauda, que podem apresentar uma frequência de instruções de retorno próxima àquela observada em ataques ROP. Para esses casos, a solução proposta usa a janela 3 como forma de registrar as instruções de retorno mal comportadas e, assim, impedir os ataques.

A Figura 4.3 ilustra a lógica de verificações utilizada para controlar a execução de instruções de desvio indireto. Ao executar qualquer instrução, as janelas 1, 2 e 3 precisam ser

atualizadas. No esquema proposto, quando uma instrução de retorno é identificada, a primeira providência tomada é verificar se o endereço de retorno corresponde a uma instrução precedida por uma instrução de chamada de procedimento (CALL). Se esse comportamento padrão for identificado, os demais passos da análise seguem o procedimento adotado para os demais desvios indiretos (CALLs e JMPs). Por outro lado, se a instrução de retorno em análise não atender ao padrão mencionado, ela é considerada uma forte indicação da existência de um ataque ROP.

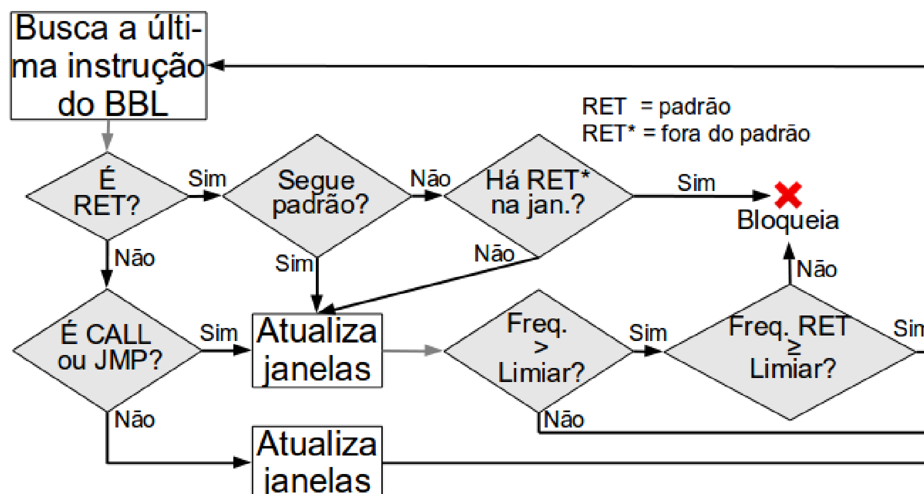

Figura 4.3: Solução para falsos positivos ocasionados por funções com recursividade em cauda.

A fim de evitar os falsos positivos relacionados à estratégia de checagem da instrução anterior ao endereço de retorno (Seção 3.2.3), o esquema proposto admite a ocorrência de uma única instrução de retorno fora do padrão entre as instruções registradas na janela 3. Portanto, se ao identificar uma instrução de retorno mal comportada, for constatado que já existe na janela 3 o registro de outra instrução com essa característica, a incidência de um ataque ROP é sinalizada. O afrouxamento dessa restrição evita a ocorrência dos falsos positivos relacionados à estratégia de checagem da instrução anterior ao endereço de retorno.

Sempre que uma instrução de desvio indireto é executada, as três janelas são atualizadas e o número de bits ativos na janela 1 é contabilizado. Se a quantidade de saltos indiretos registrados nas últimas instruções for maior do que o limiar estabelecido para a aplicação, esse comportamento é considerado um indício de ataque ROP. Caso contrário, a execução do programa segue normalmente, buscando-se a próxima instrução a ser executada.

Quando a frequência de instruções de desvio indireto é maior do que o limiar da aplicação, antes de decretar a existência de um ataque, é necessário verificar se não se está diante de uma função com recursividade em cauda. Para isso, utiliza-se os registros armazenados na janela 2. Como a segunda janela guarda exclusivamente o histórico das instruções de retorno, é possível

identificar se a superação do limiar decorre da sucessão de instruções de retorno bem comportadas. Se a quantidade de instruções de retorno anotadas na janela 2 for maior ou igual ao limiar da aplicação, constata-se que a superação do limiar foi ocasionada pela execução de um código autêntico. Por outro lado, se o limiar de frequência de desvios indiretos da aplicação for superado e, ao mesmo tempo, o número de instruções de retorno entre as últimas instruções executadas for inferior a esse limiar, constata-se a incidência de um ataque ROP.

Conforme indicado na Seção 3.2.3, existem várias soluções que utilizam apenas a estratégia de checagem da instrução anterior ao endereço de retorno como forma de reduzir a quantidade de *gadgets* úteis para a construção de um ataque ROP. O trabalho apresentado em [100], por exemplo, venceu o concurso BlueHat Prize valendo-se principalmente dessa estratégia. Como a solução desenvolvida neste projeto combina essa estratégia com o controle da frequência de instruções de desvio indireto, essa solução alternativa para o RIP-ROP alcança um nível maior de segurança contra ataques ROP baseados na interligação de *gadgets* pela instrução RET. Ademais, o esquema de controle da frequência proposto possibilita o bloqueio das demais variantes de ataques ROP, fato inédito entre as soluções que utilizam uma estratégia de controle da frequência de instruções de desvio.

É importante ressaltar que o desempenho dessa solução alternativa será pior do que aquele apresentado pela versão do RIP-ROP que não incluiu o tratamento de funções com recursividade em cauda. Isso acontece porque na solução alternativa a operação de deslocamento da janela de instruções deve ser executada três vezes, em contraponto a uma úncia execução para a versão original do RIP-ROP. Além disso, a checagem da instrução posicionada antes do endereço de retorno não é realizada na versão primária do protótipo. A inclusão dessa operação também impõe à solução alternativa um custo computacional adicional.

5 Resultados

Este capítulo apresenta e discute os resultados dos experimentos realizados nas etapas de avaliação empírica (Seção 4.1.3) e avaliação de desempenho (Seção 4.1.4). Antes disso, porém, a Seção 5.1 discorre sobre o protocolo experimental adotado, detalhando os seguintes aspectos relacionados ao planejamento dos experimentos:

- Número de repetições dos experimentos.
- Incompatibilidades entre *benchmarks* e o compilador Visual C/C++.
- Configurações e parâmetros do SPEC CPU2006.
- Execuções paralelas e sequenciais.

Na Seção 5.2, são descritas as características dos equipamentos de hardware e as versões dos sistemas operacionais, compiladores e *framework* de instrumentação binária utilizados nos ambientes de experimentação. A Seção 5.3 discute os resultados do processo de validação da estratégia de controle da frequência de instruções de desvio indireto, comprovando a viabilidade dessa solução e apresentando o tamanho de janela ideal para a sua implementação através do Pin. A Seção 5.4 detalha os experimentos realizados para comprovar a eficácia do RIP-ROP no bloqueio de ataques reais. Finalmente, a Seção 5.5 apresenta uma comparação entre o desempenho do protótipo desenvolvido neste trabalho e o modelo de referência escolhido (ROPdefender), investigando os aspectos que podem interferir nesse quesito e contrapondo com os resultados reportados por outras soluções correlatas.

5.1 Protocolo experimental

Todos os experimentos realizados durante este projeto foram executados em equipamentos dedicados exclusivamente aos testes deste trabalho, a fim de evitar oscilações no tempo de execução decorrentes da divisão da capacidade de computação com outras tarefas ou usuários.

Número de repetições dos experimentos

Cada experimento foi executado uma única vez. Essa decisão não oferece restrições aos resultados obtidos na validação da estratégia de proteção, já que a frequência máxima de desvios indiretos

apresentada por uma aplicação não se altera de uma execução para outra, a menos que os parâmetros de entrada se modifiquem ou algum trecho do fluxo de execução seja afetado por alguma condição de paralelismo – fatores inexistentes nos experimentos realizados. Por outro lado, idealmente as medições do tempo de execução deveriam ser repetidas algumas vezes para conferir maior acurácia aos valores obtidos. A análise da variância em torno de um valor médio, por exemplo, pode ser um bom indicador de quão próximas são as medidas e, por conseguinte, quão exato é o valor médio obtido. Contudo, dois fatores foram determinantes para a decisão de limitar o número de repetições de cada experimento a uma única execução:

- No ambiente de testes disponível, o elevado tempo de execução consumido pelos experimentos (da ordem de milhares de segundos para cada teste) iria requerer mais tempo do que o disponível para a conclusão do projeto.
- A pequena variação do tempo de execução observada durante a repetição dos experimentos iniciais (entre frações de segundo e poucos segundos), quando comparada ao tempo de execução total de cada experimento (da ordem de milhares de segundos), mostrou-se pouco significativa.

Incompatibilidades entre *benchmarks* e o compilador Visual C/C++

Todos os *benchmarks* da suíte SPEC CPU2006, tanto inteiros quanto de ponto flutuante, foram utilizados nos ambientes de testes Linux. Por outro lado, no ambiente Windows, os seguintes *benchmarks* não foram executados por incompatibilidade com o compilador Visual C/C++[157]:

- libquantum: o Visual C++ não suporta algumas opções da versão C99 usadas por esse *benchmark*.
- CactusADM, gromacs, zeusmp, leslie3d, GemsFDTD, calculix, bwaves, gamess, wrf, tonto: o Visual Studio não suporta a linguagem Fortran, usada por esses *benchmarks*.

Configurações e parâmetros do SPEC CPU2006

Para a execução dos *benchmarks* disponíveis no SPEC CPU2006, é necessário estabelecer algumas configurações e parâmetros de execução. A fim de garantir coerência entre os experimentos realizados neste trabalho e aqueles executados na maioria das publicações que versam sobre proteções contra ataques ROP, as seguintes opções foram adotadas:

- suítes: CINT2006 (mede o desempenho na computação de instruções inteiras) e CFP2006 (mede o desempenho na computação de instruções de ponto flutuante)
- método de compilação (*tune*): base (opções de compilação consistentes entre todos os programas de uma determinada linguagem)
- métrica: speed (tempo para completar um conjunto de tarefas individuais)
- ação: run (execução)
- tamanho: ref (entrada padrão dos *benchmarks*)
- iterações: 1 (uma)

Os *flags* e opções de compilação dos *benchmarks* foram utilizados conforme os valores padrão estabelecidos nos modelos de arquivos de configuração do SPEC disponibilizados junto com a suíte. Para o Windows, foram utilizados os arquivos de exemplo denominados "Example-windows-amd64-visualstudio.cfg" e "Example-windows-ia32-visualstudio.cfg". Para o Linux, foram utilizados como modelos os arquivos "Example-linux64-ia32-gcc43+.cfg" e "Example-linux64-amd64-gcc43+.cfg". Em todos os casos, apenas a diretiva "use_submit_for_speed=yes" foi adicionada e a opção "submit" foi alterada dentro de cada um desses arquivos para refletir os experimentos executados. Todos os demais parâmetros e configurações foram mantidos conforme definidos nos referidos arquivos de exemplo.

No caso do Pin, além dos parâmetros tradicionais que indicam a localização da Pintool, as entradas específicas da Pintool e a localização do arquivo executável a ser instrumentado, nenhum outro modificador foi utilizado em todos os experimentos.

Execuções paralelas e sequenciais

Como a execução dos experimentos mostrou-se demorada, procurou-se agilizar a conclusão dos testes de validação da estratégia de proteção através da exploração de execuções paralelas. Uma vez que o SPEC não possui mecanismos próprios para distribuir o processamento dos *benchmarks* entre múltiplos núcleos de um processador, utilizou-se a estratégia de lançar vários *benchmarks* ao mesmo tempo, deixando que o sistema operacional escalonasse os processos entre os núcleos. Para o ambiente de testes com um processador de 06 (seis) núcleos (vide Seção 5.2), por exemplo, eram lançados simultaneamente 06 (seis) *scripts* com listas sequenciais de *benchmarks* a serem

executados. Esse procedimento garantia que a todo instante havia 06 (seis) *benchmarks* distintos sendo executados naquela máquina.

Exceções ocorriam apenas quando algum dos *scripts* se encerrava antes dos demais. Mesmo sabendo que a existência dessas exceções e de eventuais interrupções a critério do escalonador de processos do sistema operacional podem impactar nos tempos de execução, conforme discutido na Seção 5.3, o objetivo dos experimentos de validação da estratégia de proteção foi catalogar as frequências máximas de desvios indiretos. A execução dos experimentos de desempenho, por sua vez, foi executada de forma completamente sequencial.

5.2 Ambientes de experimentação

Tanto os experimentos de validação da estratégia de proteção quanto os testes de desempenho executados no ambiente Windows foram realizados no seguinte equipamento:

- Processador: Pentium E5800 3.20GHz, Dual-Core, cache size: 2048 KB
- Memória: 6GB
- Sistema Operacional: Windows 7 Ultimate, x64-based
- Compilador: Visual Studio 8
- Versão do Pin: Pin 2.13 kit 62728

Para o ambiente Linux, os experimentos foram divididos em 02 (dois) equipamentos. Os testes de desempenho foram executados em uma máquina com capacidade de processamento análoga ao servidor usado para o Windows:

- Processador: Pentium E5800 3.20GHz, Dual-Core, cache size: 2048 KB
- Memória: 6GB
- Sistema Operacional: Linux Ubuntu 12.04 x86_64 kernel 3.8.0-38-generic
- Compilador: GCC 4.6.3
- Versão do Pin: Pin 2.13 kit 62728

O experimentos de validação da estratégia de proteção no Linux, por sua vez, foram executados no seguinte equipamento:

- Processador: Intel Xeon E5-2630 2.30GHz, Hexa-Core, cache size: 15360 KB
- Memória: 32GB
- Sistema Operacional: Linux Ubuntu 12.04 x86_64 kernel 3.8.0-38-generic
- Compilador: GCC 4.6.3
- Versão do Pin: Pin 2.13 kit 62728

5.3 Validação da estratégia de proteção

A estratégia de proteção contra ataques ROP proposta neste trabalho, baseada no controle da frequência de instruções de desvio indireto, foi validada através da análise comparativa entre a frequência de desvios indiretos observada em *exploits* ROP e em aplicações autênticas, representadas pelos *benchmarks* da suíte SPEC CPU2006. As Figuras 5.1, 5.2, 5.3 e 5.4 ilustram, respectivamente, os resultados obtidos nos experimentos com o Linux para janelas de 32, 64, 96 e 128 instruções. Os resultados dos experimentos realizados no Windows com janelas de mesmo tamanho estão expressos nas Figuras 5.5, 5.6, 5.7 e 5.8.

Nesses gráficos, as marcações posicionadas mais à esquerda, representadas com um "X" vermelho, correspondem às frequências máximas de desvios indiretos registradas pelos *exploits* ROP catalogados. Cada um desses *exploits* é designado no eixo horizontal pelo nome de sua aplicação vulnerável correspondente. Em contraponto, marcações posicionadas mais à direita, representadas com um disco verde, correspondem às frequências máximas de desvios indiretos registradas pelos *benchmarks* da suíte SPEC CPU2006. Os *benchmarks* são designados no eixo das abscissas pelo seu nome. Note que nas Figuras 5.2, 5.4, 5.6 e 5.8, onde são exibidos os gráficos referentes às janelas de 64 e 128 instruções, cada *benchmark* possui duas marcações. Isso acontece porque, nesses casos, cada *benchmark* foi compilado para duas arquiteturas: 32 bits e 64 bits. Para diferenciá-las nos gráficos, os termos "32" e "64" foram agregados à identificação dos *benchmarks* no eixo horizontal, de acordo com cada caso.

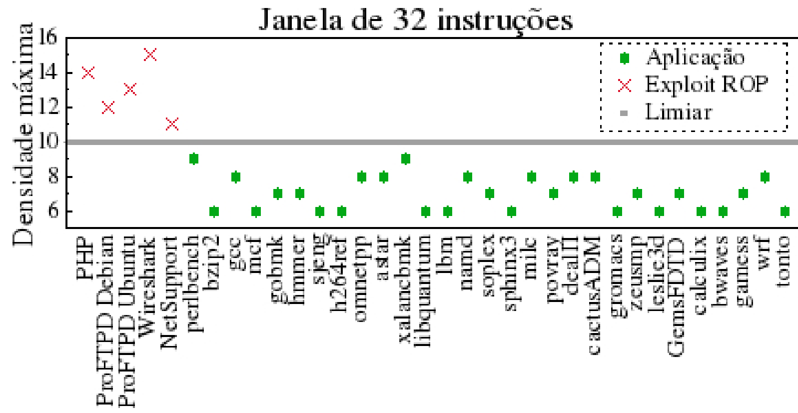

Figura 5.1: Frequência máxima de desvios indiretos com janela de 32 instruções no Linux.

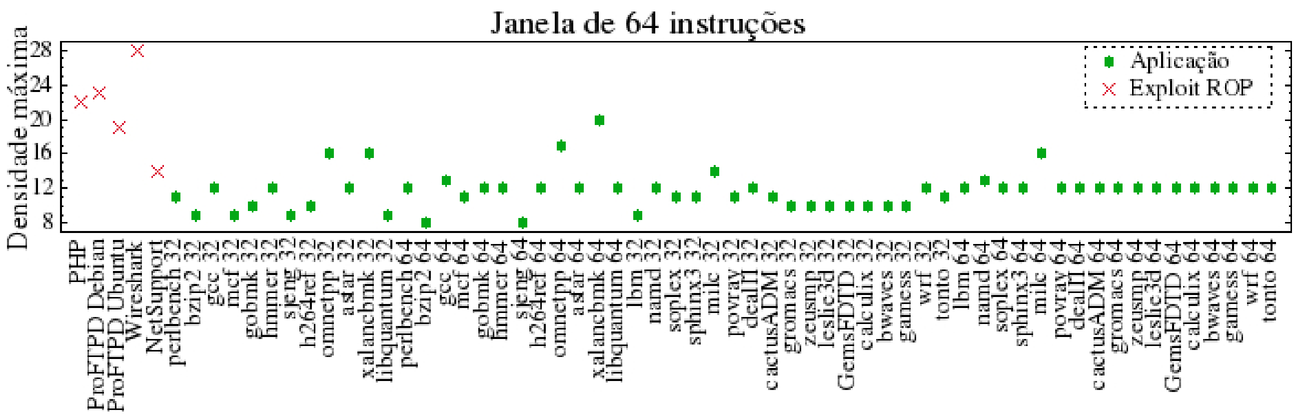

Figura 5.2: Frequência máxima de desvios indiretos com janela de 64 instruções no Linux.

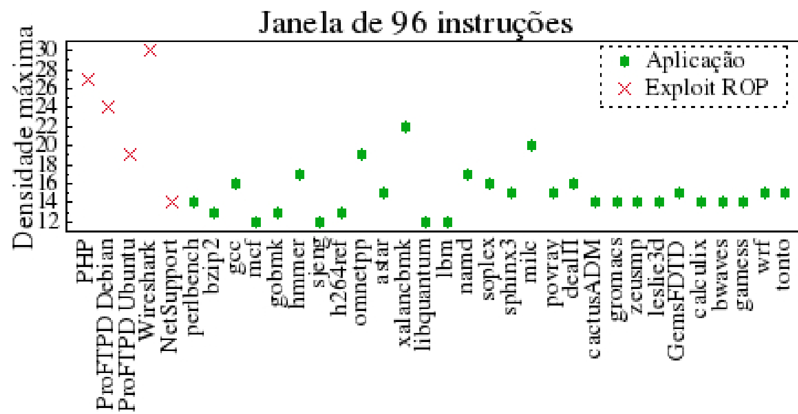

Figura 5.3: Frequência máxima de desvios indiretos com janela de 96 instruções no Linux.

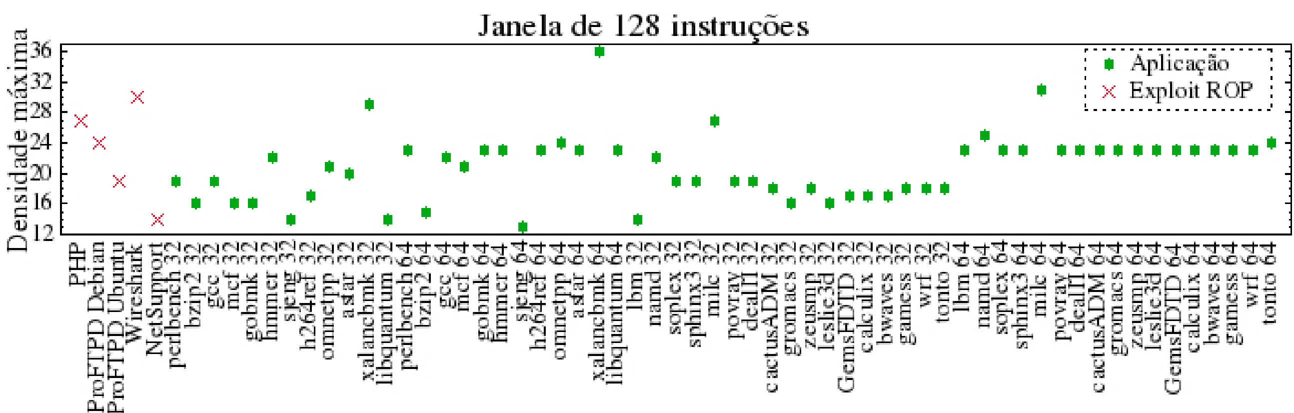

Figura 5.4: Frequência máxima de desvios indiretos com janela de 128 instruções no Linux.

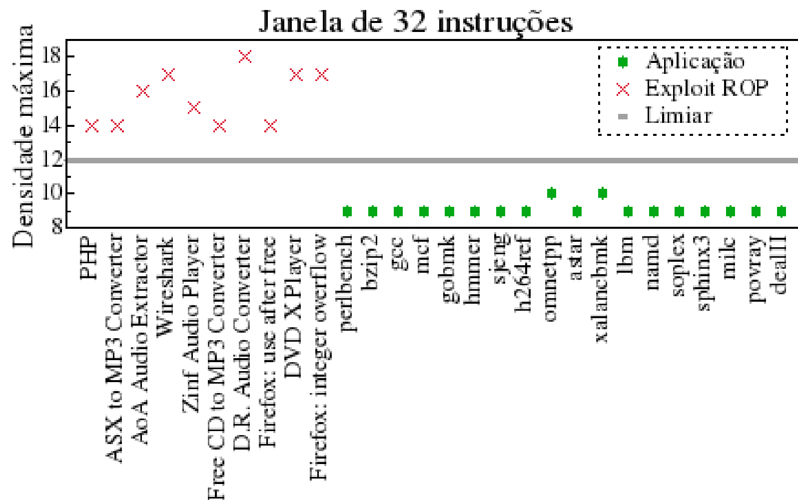

Figura 5.5: Frequência máxima de desvios indiretos com janela de 32 instruções no Windows.

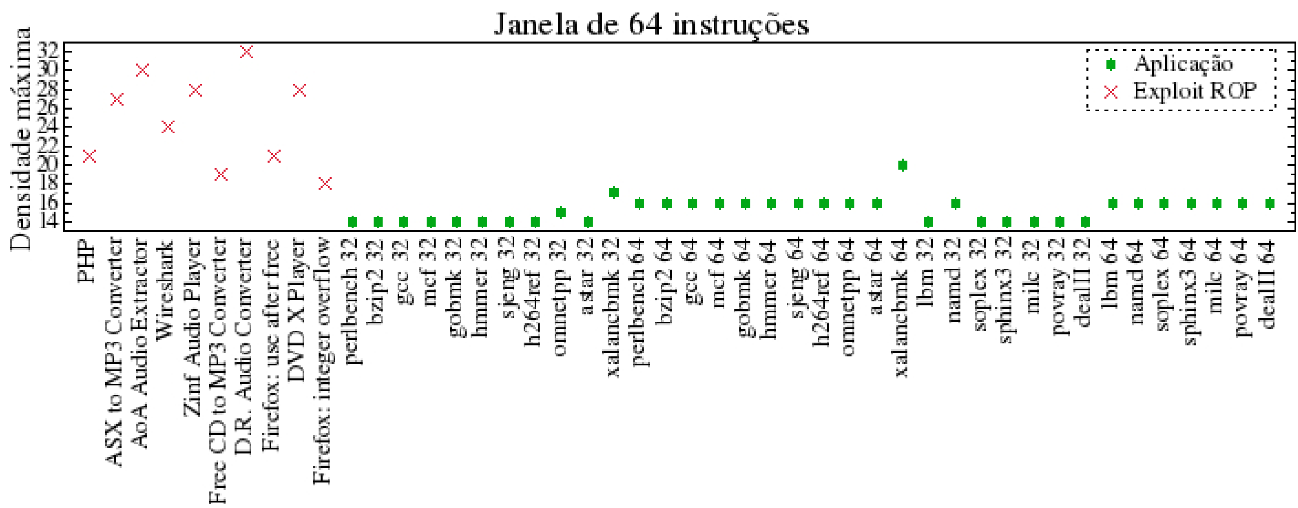

Figura 5.6: Frequência máxima de desvios indiretos com janela de 64 instruções no Windows.

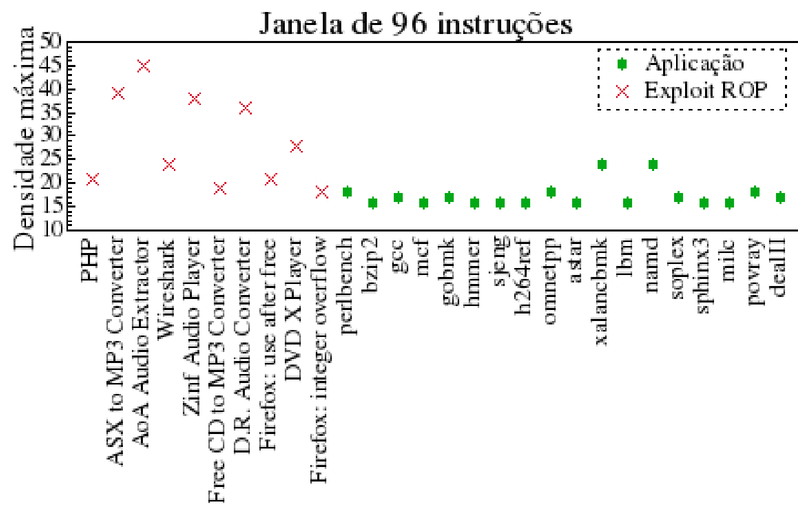

Figura 5.7: Frequência máxima de desvios indiretos com janela de 96 instruções no Windows.

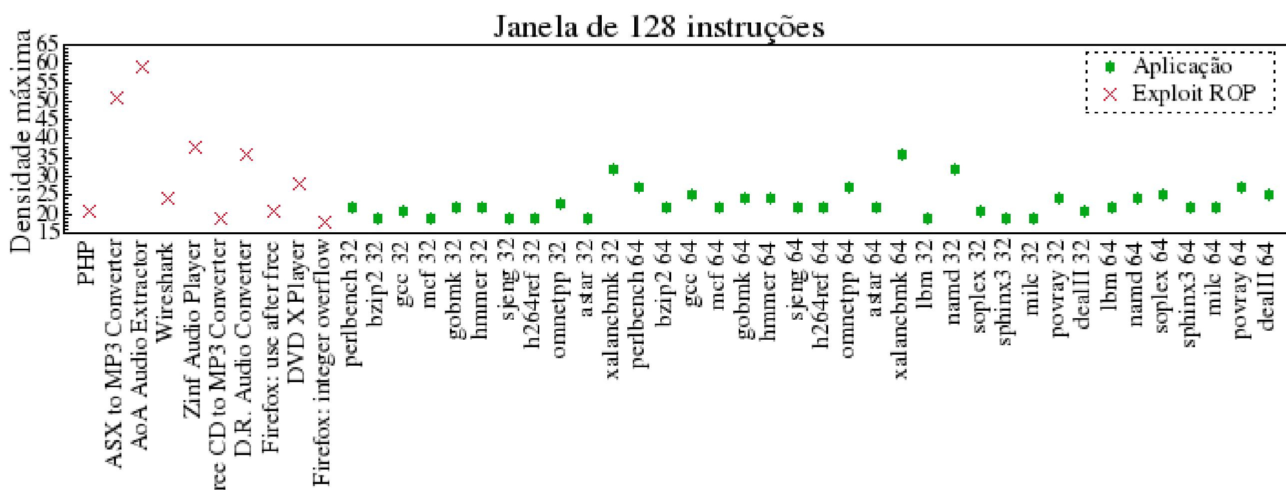

Figura 5.8: Frequência máxima de desvios indiretos com janela de 128 instruções no Windows.

Ao analisar as figuras é possível constatar que, tanto no Linux quanto no Windows, a densidade máxima de instruções de desvio indireto tende a ser maior nos *exploits* ROP do que em aplicações autênticas. No entanto, quanto maior o tamanho da janela, menor é a discrepância dos resultados entre as duas classes de executáveis analisadas (*exploits* ROP versus aplicações autênticas). Tanto que, no caso da janela de 128 instruções, cujos resultados são apresentados nas Figuras 5.4 e 5.8, não é possível distinguir os *exploits* das aplicações.

Em contrapartida, à medida que o tamanho da janela diminui para 96 (Figuras 5.3 e 5.7), 64 (Figuras 5.2 e 5.6) ou 32 (Figuras 5.1 e 5.5) instruções, essa distinção torna-se nítida. Apesar disso, o único tamanho de janela testado para o qual foi possível estabelecer um limiar "universal", teoricamente capaz de distinguir qualquer aplicação de qualquer *exploit* ROP, foi o de 32 instruções. Em outras palavras, para a janela de 32 instruções, pode-se traçar uma linha (conforme indicado nas Figura 5.1 e 5.5) que separa os dois padrões de frequência máxima de desvios indiretos, já que os valores registrados para *exploits* ROP variaram de 11 a 15 no Linux e de 14 a 18 no Windows, enquanto nos *benchmarks* esses valores se espalharam entre 6 a 9 no Linux e 09 a 10 no Windows. Portanto, pelos resultados obtidos, limiares de 10 e de 11 a 13 desvios indiretos são capazes, respectivamente, de distinguir aplicações autênticas de *exploits* ROP no Linux e no Windows, quando empregados junto com uma janela de 32 instruções.

Apesar de os resultados indicarem a possibilidade de se estabelecer um valor padrão para separar a frequência máxima de desvios indiretos apresentada por um *exploit* ROP daquela atingida por aplicações autênticas, a definição de um limiar universal pode não ser totalmente confiável, em função da proximidade entre as fronteiras. Uma solução mais robusta é usar como limiar a

frequência máxima de desvios indiretos específica de cada aplicação, porque isso permite a redução do limiar para a maioria das aplicações e, consequentemente, amplia a diferença em relação aos *exploits* ROP. Para ficar mais claro como essa abordagem pode beneficiar um mecanismo de proteção contra ataques ROP baseado no controle da frequência de desvios indiretos, basta observar-se que poucas aplicações apresentam uma frequência máxima de desvios indiretos próxima do limiar geral. Isso pode ser depreendido, por exemplo, da Tabela 5.1, que apresenta a estatística de distribuição das frequências máximas de desvios indiretos entre os *benchmarks* monitorados com a janela de 32 instruções.

Tabela 5.1: Distribuição das frequências máximas de desvios indiretos com a janela de 32 instruções

Linux		Windows	
Frequência máxima de desvios indiretos	**Porcentagem dos *benchmarks***	**Frequência máxima de desvios indiretos**	**Porcentagem dos *benchmarks***
9	6,9%	10	11,1%
8	27.6%	9	88,9%
7	24,1%		
6	41,4%		

No Linux, apenas 6,9% dos experimentos alcançaram uma frequência máxima de desvios indiretos igual a 9 (nove), que corresponde ao valor mais próximo do limiar universal. No Windows, o valor mais próximo dos possíveis limiares foi observado somente em 11,1% dos casos. Além disso, nos dois ambientes, a frequência máxima de desvios indiretos mais comum é justamente a mais distante do limiar universal, abarcando 41,4% das ocasiões no Linux e 88,9% no Windows. Essa distribuição estatística permite inferir que, ao ampliar-se o escopo de experimentos para uma gama ainda maior de aplicações, a tendência é de que poucos casos se aproximem do limiar universal. Portanto, a utilização de um limiar específico para cada aplicação permite aumentar ainda mais a confiabilidade da estratégia de controle da frequência máxima de desvios indiretos.

Para desenvolver um *exploit* ROP, um atacante deve superar diversas limitações que, em muitos cenários, impedem a construção de um código malicioso efetivo (vide Seção 2.3.2). A imposição de um limiar para a frequência de instruções de desvio indireto acrescenta um fator significativo de dificuldade a essa tarefa, impossibilitando-a em todos os casos testados. Uma redução do limiar restringe ainda mais as opções de montagem de *exploits* ROP efetivos, tornando ainda mais árdua essa tarefa. Portanto, o uso de limiares específicos para as aplicações é uma conduta recomendável. Para estabelecer o limiar específico de um determinado software, deve-se executar uma etapa prévia de experimentação com o sistema, explorando todas as suas

funcionalidades e registrando a frequência máxima de desvios indiretos. Ao final dos experimentos, a frequência máxima de desvios indiretos observada corresponderá ao limiar específico da aplicação. O objetivo desse teste é forçar o fluxo de execução do software a percorrer todos os seus caminhos, garantindo que a proteção não venha a classificar equivocadamente trechos de código autênticos.

Os gráficos expressos nas Figuras 5.1 a 5.8 indicam que, ao contabilizar a frequência máxima de instruções de desvio indireto, o uso de janelas de instruções maiores tende a aproximar os resultados aferidos para *exploits* ROP dos valores observados para aplicações convencionais. De fato, esse é o comportamento esperado, uma vez que, em geral, as cadeias de *gadgets* incorporadas aos *malwares* ROP possuem poucas instruções. Nos códigos ROP catalogados neste trabalho, o número médio de instruções apresentado pelas cadeias de *gadgets* é de 61 instruções, nos *malwares* destinados a ambientes Linux, e de 83 instruções nos *exploits* para Windows. Porém, conforme pode ser observado na Tabela 4.1, foram encontrados códigos que executam apenas 36 instruções dentro da cadeia de *gadgets*. Nesses casos, se for utilizada uma janela de instruções grande, o código malicioso ocupará apenas uma pequena fração do total de instruções consideradas para o cálculo. Assim, diluída entre instruções autênticas, a cadeia de *gadgets* acaba gerando uma frequência máxima de instruções de desvio indireto similar àquela observada com códigos autênticos.

Avaliando essa questão da relação entre o comprimento das cadeias ROP e o tamanho das janelas de instruções sob uma ótica oposta, é de se esperar que janelas menores do que 32 instruções possam melhorar o fator de distinção entre *malwares* ROP e softwares autênticos, especialmente no caso de códigos maliciosos com menos do que 32 instruções. No entanto, esse cenário não foi reproduzido porque não seria possível utilizar a instrução de máquina POPCNT, essencial para a otimização de desempenho do protótipo (vide Seção 4.2.3.1). Ao invés disso, seria necessário efetuar a contagem dos bits ativos na janela de instruções através de um laço de repetição, o que acarretaria em um *overhead* muito elevado, pelo menos em implementações via instrumentadores binários dinâmicos. Além disso, existe um compromisso entre o tamanho da janela e a capacidade de evitar falsos positivos. Janelas muito pequenas podem ocasionar o bloqueio equivocado de aplicações autênticas que possuem um pequeno trecho do fluxo de execução com um alto índice de instruções de desvio indireto. Finalmente, não foram encontrados *exploits* ROP públicos com menos do que 32 instruções em suas cadeias de *gadgets*.

Outro ponto importante a ser discutido sobre os resultados reside no fato de que 15 (quinze) *exploits* é um volume pequeno de exemplares para se estabelecer um limiar dito "universal" por abarcar todos os casos e possibilidades. Porém, a quantidade de códigos maliciosos do tipo ROP é muito pequena quando comparada ao volume total de *malwares* disponíveis publicamente na Internet, especialmente no caso de códigos maliciosos destinados ao sistema operacional Linux. Na base de *exploits* do *Exploit Database* [129], por exemplo, uma consulta realizada em 02 de julho de 2014 pelos termos ROP e *gadget* retornou apenas 255 (duzentas e cinquenta e cinco) entradas, a maioria para sistemas Windows, de um total de 30.115 (trinta mil cento e quinze) exemplares de *exploits* catalogados. Essa reduzida quantidade de *exploits* ROP disponíveis publicamente decorre das características restritivas impostas pela própria técnica ROP para o desenvolvimento de códigos maliciosos. Em muitos casos, não existem *gadgets* suficientes na área de memória executável do processo para a consolidação de um ataque. Nessas situações, *exploits* ROP não são disponibilizados simplesmente porque não é possível criá-los, apesar de existirem diversos exemplares de *malwares* que exploram a mesma vulnerabilidade, porém em outros cenários (exemplo: quando a aplicação vulnerável não é executada em ambientes protegidos pelo bit de execução NX/XD).

O mesmo raciocínio se aplica à quantidade de variações possíveis em relação a aplicações e outros ambientes de teste, tais como arquiteturas, sistemas operacionais e compiladores. Não se tem a pretensão neste trabalho de exaurir todas as possibilidades de códigos autênticos e ambientes computacionais existentes ou que possam vir a ser desenvolvidos. Ao invés disso, buscou-se avaliar a estratégia de proteção desenvolvida neste projeto em um contexto que abranja grande parte dos usuários de sistemas computacionais. De qualquer modo, o principal objetivo da avaliação aqui realizada não é estabelecer um limiar definitivo, mas sim demonstrar que a estratégia de controle da frequência de instruções de desvio indireto é eficaz na detecção de ataques ROP.

5.4 Eficácia no bloqueio de *exploits* reais

O protótipo foi testado na proteção de 15 (quinze) aplicações para as quais existem *exploits* ROP publicamente disponíveis no repositório *Exploit Database*. Foram utilizados os mesmos exemplares de códigos maliciosos empregados na validação da estratégia de proteção (vide Tabela 5.2). Os experimentos foram executados em duas etapas. Na primeira, o correto funcionamento dos *exploits* foi confirmado através da reprodução dos ataques contra máquinas virtuais onde as aplicações

vulneráveis foram instaladas. Na sequência, executou-se essas aplicações sob o controle do protótipo e repetiu-se os ataques. Dessa forma, pôde-se observar a eficácia da estratégia de controle da frequência de desvios indiretos na proteção contra ataques ROP reais. Em todos os casos, o RIP-ROP foi capaz de detectar o ataque ROP e impedir a sua consolidação.

Conforme mencionado na Seção 4.1.1.2, ao contrário de algumas publicações sobre proteções contra ataques ROP [61], [89], [90], [93], [105], neste trabalho primou-se por utilizar códigos maliciosos reais, desenvolvidos por terceiros. Isso garante uma maior diversidade no conjunto de testes do que se fossem utilizados *exploits* desenvolvidos pelos próprios autores. Acredita-se que a conduta de empregar *malwares* construídos exclusivamente para os experimentos tende a limitar a diversidade dos exemplares, uma vez que desenvolvedores tendem a repetir as mesmas estratégias e técnicas durante a elaboração dos *exploits* ROP. Por isso, ao utilizar códigos maliciosos públicos, escritos por diferentes autores, o universo de experimentação empregado neste trabalho alcança uma melhor representatividade, o que impacta diretamente na qualidade e confiabilidade dos resultados observados. Em contrapartida, a quantidade de exemplares se limita ao número de *exploits* públicos cujos ataques podem ser reproduzidos com sucesso, já que muitas aplicações rapidamente tornam-se indisponíveis após a divulgação de suas vulnerabilidades.

5.5 Desempenho

Esta Seção é dedicada à apresentação e discussão dos resultados da avaliação de desempenho descrita na Seção 4.1.6. As Figuras 5.9 e 5.10 ilustram, respectivamente, os gráficos de desempenho dos *benchmarks* inteiros e de ponto flutuante no Windows de 64 bits. Os resultados referentes ao Linux de 64 bits estão expressos nas Figuras 5.11 e 5.12. Afim de evitar repetições, os resultados de desempenho observados com a arquitetura de 32 bits foram omitidos, uma vez que a relação de desempenho entre as soluções avaliadas nessa arquitetura se assemelha àquela obtida com a arquitetura de 64 bits. Apesar disso, vale destacar que essa semelhança reforça a confiabilidade dos resultados.

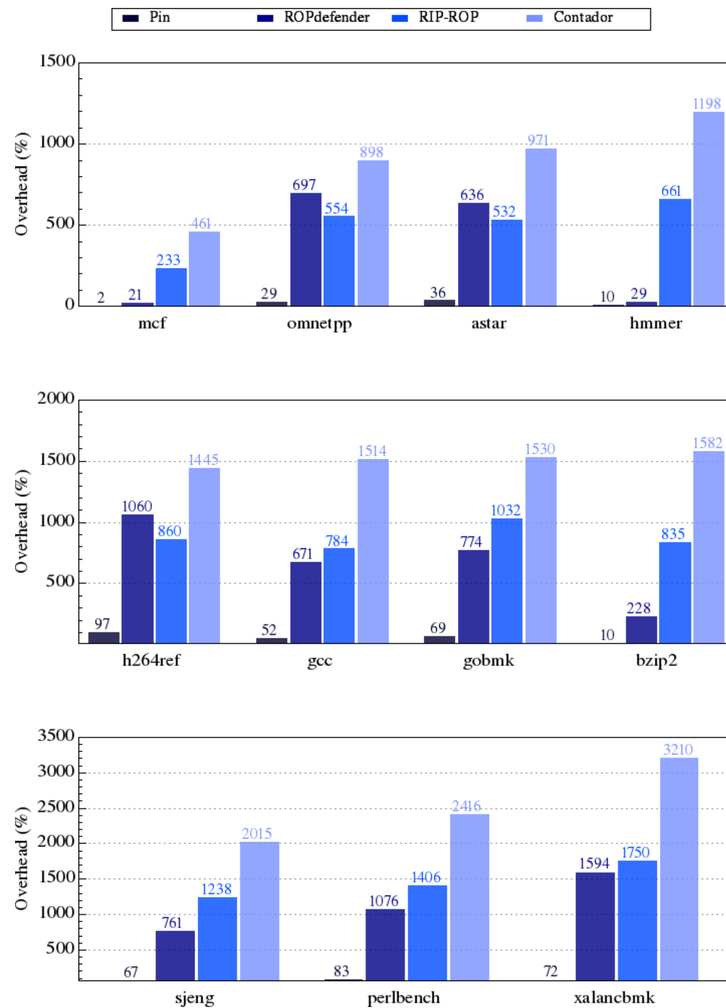

Figura 5.9: O*verhead* imposto aos *benchmarks* inteiros no Windows de 64 bits.

Apesar de variações no resultado específico de determinados *benchmarks*, nota-se que a relação geral de desempenho entre as soluções avaliadas é a mesma no Windows e no Linux. O resultado mais consistente, que se destaca em todos os gráficos, recai sobre o desempenho ruim alcançado pela Pintool que conta o número de BBLs executados pelo programa. Essa solução atingiu o maior *overhead* entre as soluções analisadas em praticamente todos os casos (a única exceção ocorreu com o benchmark “povray” no Windows – vide Figura 5.10). De fato, conforme discutido na Seção 4.2.2, a necessidade de executar códigos de análise para todos os BBLs de uma aplicação impõe uma severa penalidade ao tempo de CPU consumido pelo processo nesse caso.

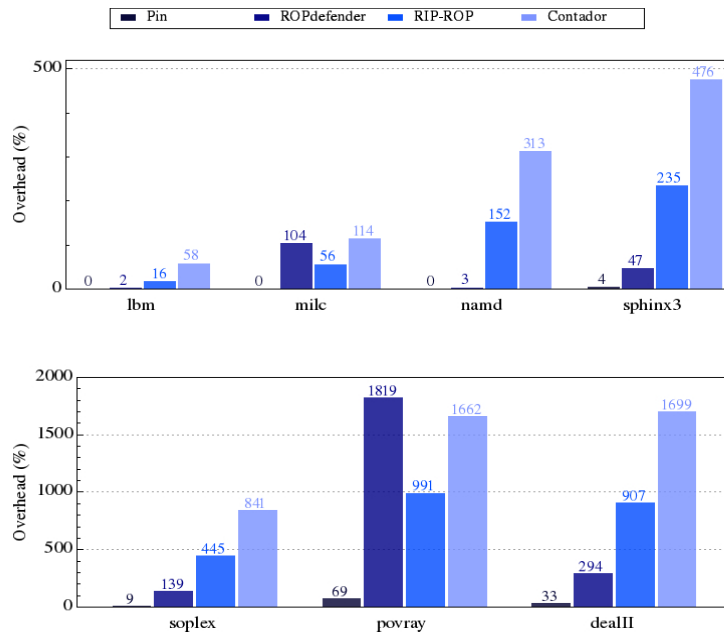

Figura 5.10: O*verhead* imposto aos *benchmarks* de ponto flutuante no Windows de 64 bits.

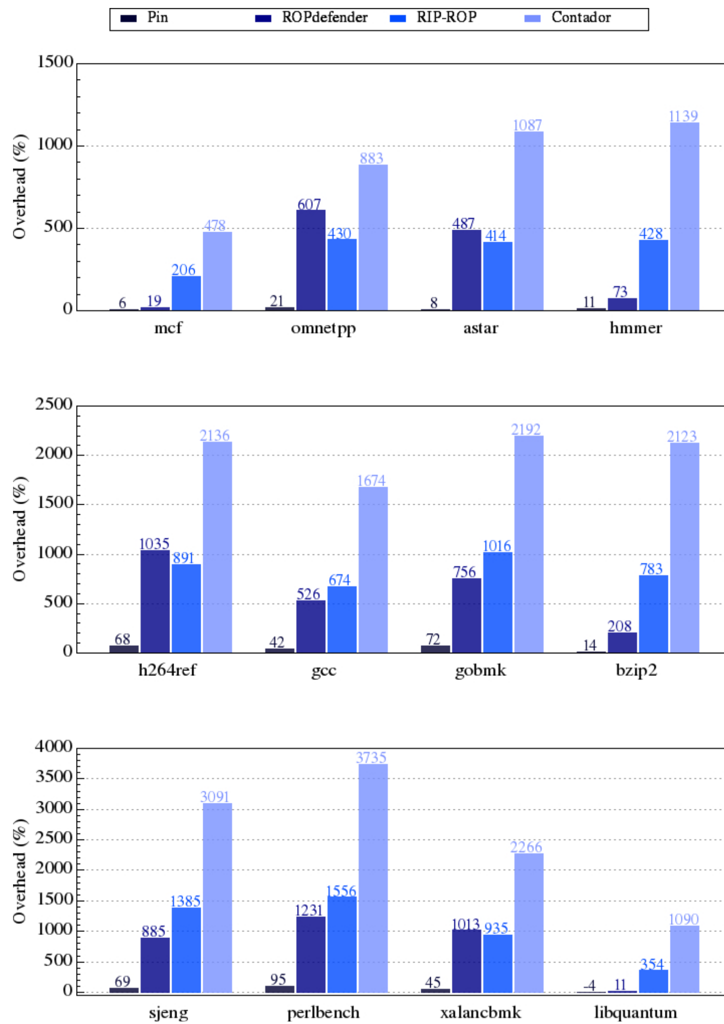

Figura 5.11: O*verhead* imposto aos *benchmarks* inteiros no Linux de 64 bits.

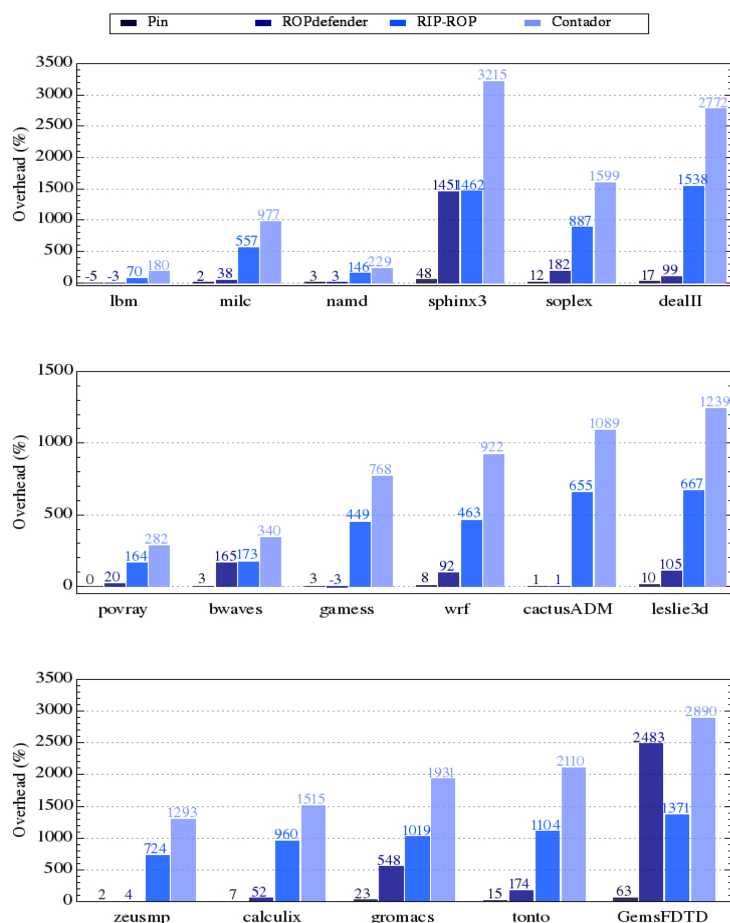

Figura 5.12: O*verhead* imposto aos *benchmarks* de ponto flutuante no Linux de 64 bits.

Essa constatação ajuda a confirmar a eficácia das otimizações de código adotadas no desenvolvimento do RIP-ROP (vide Seção 4.2.3). Apesar de também exigir a execução de uma função de análise para todos os BBLs executados pela aplicação, o RIP-ROP acarreta em um overhead muito menor do que a Pintool que conta os BBLs. Isso se deve ao fato de que os constantes acessos à memória, necessários no Contador para incrementar a variável global que armazena o número de blocos executados, dominam o tempo de execução dessa Pintool. Em contrapartida, apesar de efetuar mais operações a cada execução da função de análise, o RIP-ROP consegue concentrá-las em um contexto com localidade de referência mais próximo, o que garante maior velocidade de acesso às estruturas de armazenamento (exemplos: registradores e caches). Além disso, como as operações efetuadas pelo RIP-ROP durante a função de análise foram projetadas para aproveitar instruções de máquina como POPCNT e SHL (vide Seção 4.2.3.1), reduziu-se a demanda por ciclos de *clock* do processador, o que compensou a necessidade de efetuar uma quantidade maior de operações.

Outro ponto que se destaca nos gráficos é a proximidade do desempenho alcançado pelo RIP-ROP com a performance obtida pelo ROPdefender, mesmo sofrendo com as penalidades impostas pela necessidade de instrumentar cada BBL da aplicação alvo. Além de apresentar um desempenho similar ao ROPdefender para a maioria dos casos, em 10 cenários dos 47 estudados, o RIP-ROP supera a performance do modelo de referência.

Em relação ao *overhead* imposto pelo Pin, apesar de pequeno se comparado à sobrecarga decorrente das Pintools implementadas, pode-se afirmar que o *framework* de instrumentação binária dinâmica acarreta em um *overhead* elevado, já que nenhuma operação efetiva é realizada. Na Tabela 5.2, que resume os resultados gerais obtidos, verifica-se que o *overhead* médio do Pin sem a adição de qualquer Pintool atingiu 28%. Esse resultado confirma a expectativa de um significativo custo computacional imposto por instrumentadores binários dinâmicos [118]. Da mesma forma, o fato de o Pin acarretar em um *overhead* maior com os *benchmarks* inteiros confirma os resultados reportados pelos autores dessa ferramenta. Segundo eles, isso se deve à maior quantidade de desvios indiretos e retornos de funções existente nesses testes. Essa característica também é ratificada pelos resultados obtidos com as demais soluções avaliadas.

Tabela 5.2: *Overhead* médio ao executar *benchmarks*

Benchmark	***Overhead* (%)**			
	Pin	**ROPdefender**	**RIP-ROP**	**Contador**
SPEC CPU2006 FP 64	14	326	634	1188
SPEC CPU2006 INT 64	42	626	824	1702
Média	**28**	**473**	**727**	**1439**

A Tabela 5.3 exibe um extrato da Tabela 3.1 para comparar o RIP-ROP com proteções contra ataques ROP que utilizam instrumentadores binários dinâmicos para implementar soluções baseadas na estratégia de controle da frequência de instruções de retorno.

Tabela 5.3: Proteções que controlam a frequência de instruções de retorno via instrumentação binária dinâmica

Proteção	**Abordagens**	**Estratégias**	**Ataques bloqueados**	***Overhead* *(%)***	**Exceções**	**Viabilidade prática**
DynIMA [62]	2 e 3	6	R	*	4	Não
DROP [61]	3	6	R	530,0	1	Sim
RIP-ROP	**3**	**8**	**R, J e C**	**727**	**0**	**Sim**

Abordagens: 2-Instrumentação Binária Estática ; 3-Instrumentação Binária Dinâmica.
Estratégias: 6-Controle da frequência de instruções de retorno ; 8-Outras.
Ataques bloqueados: R-encadeamento via RET; J-encadeamento via JMP; C-encadeamento via CALL.
*O *overhead* não é informado na publicação.

É importante ressaltar que estão expressos na Tabela 5.3 os *overheads* médios reportados pelos autores de cada solução. Isso significa que os resultados remontam a diferentes conjuntos de teste e ambientes de experimentação. Apesar disso, pode-se dizer que o RIP-ROP apresenta um

custo computacional comparável à solução DROP, que também utiliza o Pin, mas mediu o desempenho através da execução de uma seleção de aplicações, ao invés da suíte de *benchmarks* SPEC. As únicas aplicações utilizadas tanto nos testes executados com o DROP quanto nos experimentos realizados com o RIP-ROP (bzip2 e gcc), que podem oferecer uma comparação um pouco mais realista, indicam que o protótipo desenvolvido neste trabalho impõe um *overhead* menor. Nos experimentos com o bzip2, o DROP acarretou em um custo computacional de 1.540%, consideravelmente superior aos 809% registrados pelo RIP-ROP. Nos testes com o gcc, o DROP impôs um *overhead* de 960%, enquanto o RIP-ROP elevou o tempo de CPU em 729%.

Outro fator de comparação entre as proteções recai sobre os tipos de ataques ROP bloqueados. Nesse caso, conforme indicado na Tabela 5.3, apenas a solução desenvolvida neste trabalho oferece uma proteção contra todos os tipos de *exploits* ROP. Essa capacidade está diretamente relacionada à mudança na estratégia de detecção dos ataques adotada neste projeto, que amplia o escopo de monitoramento para abarcar todas as instruções de desvio indireto.

6 Conclusões

Neste capítulo, a dissertação é concluída reiterando-se as contribuições oferecidas por este trabalho e apontando-se trabalhos futuros que podem dar prosseguimento a este projeto de pesquisa. Na Seção 6.1, as contribuições são revistas com foco nos objetivos que nortearam o desenvolvimento deste trabalho. Em seguida, a Seção 6.2 discorre sobre as oportunidades de melhorias identificadas no transcorrer do projeto, que podem ensejar novos trabalhos de pesquisa.

6.1 Contribuições

A significativa representatividade de ataques ROP nos atuais incidentes de segurança tem atraído a atenção de muitos pesquisadores na tentativa de criar mecanismos para contê-los. Levando-se em conta a grande quantidade de proteções publicadas e a complexidade do processo de comparação dessas soluções, este trabalho contribui com a pesquisa nessa área ao apresentar uma visão geral dos trabalhos que propõem proteções contra ataques ROP, estruturar as soluções em duas classificações – segundo as estratégias de detecção e as abordagens de implementação existentes – e propor métricas para a análise e comparação de proteções. Apesar de procurar realizar a análise mais completa possível dos mecanismos de defesa disponíveis contra ataques ROP, este trabalho não é capaz de exaurir o tema, tendo em vista que esse é um campo de pesquisa em plena atividade. No entanto, acredita-se que ele contribui de forma relevante ao oferecer uma análise agrupada e estruturada dos trabalhos, além de sugerir métricas que poderão ser aplicadas futuramente na análise e classificação de novas soluções.

Conforme apresentado, para desenvolver um *exploit* ROP efetivo, um atacante deve superar várias restrições, incluindo limitações quanto aos *gadgets* disponíveis na área de memória executável do processo atacado, tais como: quantidade e tipo de *gadgets* disponíveis, efeitos colaterais inerentes aos *gadgets*, endereços de memória inválidos e *layouts* de memória variáveis. Ainda que não represente uma proteção comprovadamente insuperável, a imposição de um limite para o uso de instruções de desvio indireto impõe severas limitações à capacidade de criação de um *exploit* ROP efetivo. Essa abordagem de dificultar a criação de códigos maliciosos efetivos, ao invés de garantir uma proteção impenetrável, é um expediente comum entre soluções consagradas

contra códigos maliciosos. Entre soluções amplamente difundidas nos sistemas computacionais modernos, pode-se citar o nx-stack (Seção 2.2.1), o ASLR (Seção 2.2.2) e o NX/XD (Seção 2.2.3) como exemplos de proteções que buscam reduzir a capacidade de ação do desenvolvedor de *exploits*. Este trabalho demonstrou que a imposição de um limite para o uso de instruções de desvio indireto acarreta em severas limitações à capacidade de criação de um *exploit* ROP efetivo, impossibilitando-a em todos os casos testados. Além disso, a estratégia de controle da frequência de instruções de desvio indireto introduzida neste trabalho possibilita o bloqueio das demais variantes de ataques ROP, fato inédito entre as soluções que utilizam uma estratégia de controle da frequência de instruções.

Também foi desenvolvido neste trabalho um protótipo, denominado RIP-ROP, que pode ser facilmente adotado em ambientes de produção que executem os sistemas operacionais Linux, Windows, Android ou OSX. Testes com *exploits* ROP disponíveis em repositórios públicos de códigos maliciosos comprovaram a plena capacidade do protótipo em bloquear esse tipo de ataque. Além disso, a análise de desempenho indicou que as contribuições mencionadas são obtidas a um custo computacional comparável à performance de soluções correlatas, superando-as em alguns casos. O RIP-ROP é especialmente atraente no caso de ambientes de produção para os quais já se conhecem vulnerabilidades não mitigáveis. Por ser uma ferramenta pronta para o uso imediato, ele pode, por exemplo, garantir a segurança do sistema até que uma atualização de segurança esteja disponível.

6.2 Trabalhos Futuros

Tendo em vista que novas estratégias de proteção e abordagens de implementação de soluções contra ataques ROP surgem a todo momento, no futuro pode ser conveniente atualizar as classificações introduzidas neste trabalho com o intuito de agrupar novos trabalhos que não se adequem às classes já existentes. Da mesma forma, a definição de novas métricas de avaliação pode ser necessária para refletir a qualidade das proteções em relação a outros aspectos que venham a se tornar relevantes no futuro.

Além disso, algumas limitações estão ligadas ao protótipo desenvolvido e podem ensejar trabalhos futuros. Uma oportunidade de melhoria que se destaca é a necessidade de reduzir o *overhead* imposto pelo RIP-ROP. Apesar de competitivo quando comparado a outras soluções

baseadas em *frameworks* de instrumentação binária dinâmica, para algumas aplicações, o protótipo desenvolvido pode acarretar em um custo computacional impraticável. Portanto, uma oportunidade de trabalho futuro consiste em utilizar abordagens alternativas para implementar a estratégia de controle da frequência de desvios indiretos que permitam reduzir o *overhead* computacional.

Entre essas abordagens alternativas de implementação, uma opção é adaptar estruturas de hardware disponíveis nos processadores atuais e originalmente desenvolvidas para outras finalidades, como o *Return Stack Buffer* (RSB), o *Last Branch Recording* (LBR) e o *Branch Trace Store* (BTS), com o intuito de implementar a estratégia introduzida por este trabalho.

Outra possibilidade é modificar o código de algum *framework* de instrumentação binária dinâmica para especializá-lo nas tarefas a serem desempenhadas pela estratégia de proteção. Um exemplo bem sucedido disso é a solução denominada ILR [82]. Ao manipular as estruturas internas do *framework* Strata, adaptando-o exclusivamente a uma estratégia específica de proteção de sistemas, pesquisadores conseguiram reduzir drasticamente o *overhead* imposto à solução, alcançando valores inéditos para uma proteção baseada em uma ferramenta de instrumentação binária dinâmica. Portanto, uma abordagem análoga pode ser adotada para implementar-se a estratégia de controle da frequência de instruções de desvio indireto.

Uma terceira alternativa de trabalho futuro que pode reduzir o *overhead* observado no RIP-ROP é efetuar a análise da janela de instruções apenas nas chamadas de sistema úteis em ataques. No entanto, nesse caso é preciso estudar se um atacante não será capaz de, depois de preparar os argumentos para a chamada de sistema, executar instruções que não afetem os argumentos e que tenham uma frequência de desvios menor do que o limiar, iludindo a estratégia de proteção. Deve-se investigar, por exemplo, se um atacante pode ser capaz de intercalar chamadas a funções sem efeito prático (inertes) entre os *gadgets* como forma de diluir a frequência de desvios indiretos executados.

Na tentativa de minimizar o custo computacional imposto pelo RIP-ROP, pode-se também combinar a proteção desenvolvida com mecanismos já existentes, como o ASLR. Em implementações de instrumentação binária dinâmica, durante a fase de instrumentação, se for detectado que todas a bibliotecas carregadas pelo processo utilizam ASLR, não é preciso inserir códigos de análise ou mesmo códigos adicionais de instrumentação. Além disso, a instrumentação e análise de código pode ser efetuada apenas para as regiões de código para as quais o ASLR não

esteja ativado. Isso reduziria consideravelmente a quantidade de trocas de contexto entre o instrumentador e a aplicação, além de diminuir a quantidade de código adicional executado. O risco de se confiar no ASLR é que, em muitos casos, o endereço de algum módulo pode ser descoberto através de alguma vulnerabilidade de vazamento de informação (*information leakage*). Por isso, a instrumentação de bibliotecas com o ASLR ativo pode ser uma opção oferecida ao usuário da proteção. Dessa forma, torna-se possível elevar o nível de proteção de aplicações críticas, em contrapartida a uma queda no desempenho desses aplicativos.

Outro projeto que pode dar prosseguimento a este trabalho é a avaliação do protótipo em outros ambientes. Uma vez que o Pin também oferece suporte para os sistemas operacionais Android e OSX, a condução de experimentos com o RIP-ROP nesses ambientes exigiria poucos ajustes e poderia iluminar questões que não foram identificadas nos ambientes de experimentação utilizados neste trabalho. O mesmo pode ser dito em relação à utilização de outros compiladores, *benchmarks* e *exploits*. Além disso, a implementação da estratégia de controle da frequência de instruções de desvio indireto em outros *frameworks* de instrumentação binária pode descortinar oportunidades inexploradas.

Finalmente, dependendo dos resultados observados em outros ambientes de testes, no futuro pode haver a necessidade de se implementar e testar a solução alternativa sugerida na Seção 4.3 para o caso de ocorrência de falsos positivos em decorrência de funções com recursividade em cauda. Nesse caso, em implementações baseadas em modificações do hardware, as checagens para instruções de retorno previstas nessa solução podem ser evitadas caso o endereço de retorno coincida com o endereço no topo do RSB (*Return Stack Buffer*), o que ocorre em mais de 95% dos casos [158]. Esse procedimento reduz o número de operações de checagem executadas, o que melhora o desempenho, através do reaproveitamento de um dispositivo de hardware já existente. Em soluções baseadas em adaptações do hardware, o aproveitamento de estruturas previamente existentes é desejável, sempre que possível, para reduzir o *overhead* inerente a qualquer expansão do hardware.

Referências

[1] G. Hoglund and G. McGraw, *Exploiting Software: How to break code*. Pearson Education, 2004.

[2] NIST, “National Vulnerability Database (NVD) CVE Statistics,” 2013. [Online]. Available: http://web.nvd.nist.gov/view/vuln/statistics.

[3] Microsoft, “Security Bulletin Severity Rating System,” 2012. [Online]. Available: http://technet.microsoft.com/en-us/security/gg309177.aspx.

[4] CVE, “Vulnerability distribution of cve security vulnerabilities by types,” 2013. [Online]. Available: http://www.cvedetails.com/vulnerabilities-by-types.php.

[5] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell, *The Shellcoder’s Handbook: discovering and exploiting security holes*, 2nd ed. Wiley, 2007.

[6] W. Shi, J. B. Fryman, G. Gu, H.-H. Lee, Y. Zhang, and J. Yang, “InfoShield: A security architecture for protecting information usage in memory,” in *International Symposium on High-Performance Computer Architecture*, 2006, pp. 222–231.

[7] R. Bojanc and B. Jerman-Blažič, “An economic modelling approach to information security risk management,” *Int. J. Inf. Manage.*, vol. 28, no. 5, pp. 413–422, 2008.

[8] R. Roemer and E. Buchanan, “Return-oriented programming: Systems, languages, and applications,” *ACM Trans. Inf. Syst. Secur. - Spec. Issue Comput. Commun. Secur.*, vol. V, pp. 1–36, 2012.

[9] J. Callas, “Smelling a RAT on Duqu,” 2011. [Online]. Available: http://blogs.entrust.com/enterprise-authentication/?p=236.

[10] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, “Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage,” in *Proceedings of the 2009 conference on Electronic voting technology/workshop on trustworthy elections*, 2009, p. 6.

[11] T. Kornau, “Return oriented programming for the ARM architecture,” Ruhr-Universitat Bochum, 2010.

[12] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin, “Automatic construction of jump-oriented programming shellcode (on the x86),” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011, p. 20.

[13] E. Schwartz, T. Avgerinos, and D. Brumley, “Q: Exploit hardening made easy,” *Proc. 20th USENIX Secur. Symp.*, 2011.

[14] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: Generalizing Return-Oriented Programming to RISC,” in *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*, 2008, p. 27.

[15] R. Hund, T. Holz, and F. C. Freiling, “Return-oriented rootkits: bypassing kernel code integrity protection mechanisms,” in *Proceedings of the 18th conference on USENIX security symposium*, 2009, pp. 383–398.

[16] R. Roemer, “Finding the bad in good code: Automated return-oriented programming exploit discovery,” University of California, 2009.

[17] T. Dullien, T. Kornau, and R.-P. Weinmann, “A framework for automated architecture-independent gadget search,” in *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*, 2010, no. Reil Vm.

[18] Corelan Team, “Overview - mona - Corelan Team,” 2011. [Online]. Available: http://redmine.corelan.be/projects/mona.

[19] Rapid7, “Metasploit: Defeat the Hard and Strong with the Soft and Gentle Metasploit RopDB,” 2012. [Online]. Available: https://community.rapid7.com/community/metasploit/blog/2012/10/03/defeat-the-hard-and-strong-with-the-soft-and-gentle-metasploit-ropdb.

[20] P. Solé, “Hanging on a ROPe,” 2010. [Online]. Available: http://immunitysec.com/downloads/DEPLIB20_ekoparty.pdf.

[21] J. A. P. Marpaung, M. Sain, and H. Lee, “Survey on malware evasion techniques: state of the art and challenges,” in *International Conference on Advanced Communication Technology (ICACT)*, 2012, pp. 744–749.

[22] D. Rosenberg, “Defeating Windows 8 ROP Mitigation,” 2011. [Online]. Available: http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/.

[23] N. H. Son, “ROP chain for Windows 8,” 2011. [Online]. Available: http://blog.bkav.com/en/rop-chain-for-windows-8/.

[24] L. M. Tung, “Advanced Generic ROP chain for Windows 8,” 2011. [Online]. Available: http://blog.bkav.com/en/advanced-generic-rop-chain-for-windows-8/.

[25] Microsoft, “The Enhanced Mitigation Experience Toolkit,” 2013. [Online]. Available: http://support.microsoft.com/kb/2458544.

[26] Microsoft, “Microsoft Security Toolkit Delivers New BlueHat Prize Defensive Technology,” 2012. [Online]. Available: http://www.microsoft.com/en-us/news/Press/2012/Jul12/07-25BlueHatPrizePR.aspx.

[27] Snake, “Bypassing EMET 3.5′s ROP Mitigations,” 2012. [Online]. Available: https://repret.wordpress.com/2012/08/08/bypassing-emet-3-5s-rop-mitigations/.

[28] Microsoft, “The BlueHat Prize Winners Announced,” 2012. [Online]. Available: http://www.microsoft.com/security/bluehatprize/.

[29] A. One, “Smashing the stack for fun and profit,” *Phrack Mag.*, vol. 7, no. 49, pp. 14–16, 1996.

[30] H. Orman, “The Morris worm: a fifteen-year perspective,” *Secur. Privacy, IEEE*, vol. 1, no. 5, pp. 35–43, 2003.

[31] T. Werthmann, “Survey on Buffer Overflow Attacks and Countermeasures,” in *Proceedings of SEMINAR SS*, 2006, pp. 1–19.

[32] P. I. T. A. Committee, “Cyber Security: a crisis of prioritization,” National Coordination Office for Information Technology Research and Development, 2005.

[33] Secunia, “Half Year Report 2011,” 2011.

[34] M. Ferreira, G. Martins, T. Rocha, E. Feitosa, and E. Souto, “Análise de Vulnerabilidades em Sistemas Computacionais Modernos: Conceitos, Exploits e Proteções.,” in in *Minicursos XII Simpósio em Segurança da Informação e de Sistemas Computacionais*, 1st ed., Porto Alegre: Sociedade Brasileira de Computação, 2012, pp. 1–50.

[35] A. Cugliari and M. Graziano, “Smashing the stack in 2010,” 2010.

[36] A. Sotirov and M. Dowd, “Bypassing Browser Memory Protections: Setting back browser security by 10 years,” in *Proceedings of BlackHat*, 2008, pp. 1–53.

[37] S. Designer, “Getting around non-executable stack (and fix),” *Bugtraq mailing list*, 1997. [Online]. Available: http://seclists.org/bugtraq/1997/Aug/63.

[38] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, “On the Expressiveness of Return-into-libc Attacks,” *Lect. Notes Comput. Sci.*, 2011.

[39] J. McDonald, “Defeating Solaris/SPARC Non-Executable Stack Protection,” *Bugtraq mailing list*, 1999. [Online]. Available: http://seclists.org/bugtraq/1999/Mar/4.

[40] T. Newsham, “non-exec stack,” *Bugtraq mailing list*, 2000. [Online]. Available: http://seclists.org/bugtraq/2000/May/90.

[41] R. N. Wojtczuk, “The advanced return-into-lib (c) exploits: PaX case study,” *Phrack Mag.*, vol. 0x0b, no. 0x3a, 2001.

[42] T. Newsham, “Re: Smashing the stack: prevention? Bugtraq, Apr. 1997,” *Online: http://seclists. org/bugtraq/1997/Apr/129*, 1997.

[43] D. Spyrit and A. B. Jack, “Win32 buffer overflows (location, exploitation, and prevention),” *Phrack Mag.*, vol. 9, no. 55, 1999.

[44] G. Richarte, “Re: Future of buffer overflows,” *Bugtraq, Oct*, 2000.

[45] G. Richarte, “Insecure programming by example: Esoteric# 2,” *Online http//community. corest. com/~gera/InsecureProgramming/e2. html, July*, 2001.

[46] S. Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique,” *Online: http://packetstorm.igor.onlinedirect.bg/papers/bypass/no-nx.pdf*, pp. 1–20, 2005.

[47] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” *Proc. 14th ACM Conf. Comput. Commun. Secur.*, pp. 1–30, 2007.

[48] Microsoft, “A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003,” 2006. [Online]. Available: http://support.microsoft.com/kb/875352.

[49] Chris Jackson, “Understanding Application Compatibility,” 2013. [Online]. Available: http://technet.microsoft.com/en-us/windows/jj863248.aspx. [Accessed: 06-Mar-2013].

[50] M. Miller and K. Johnson, “Bypassing Windows Hardware-enforced Data Execution Prevention,” 2005. [Online]. Available: http://www.uninformed.org/?v=2&a=4.

[51] B. Damele, “DEP bypass with SetProcessDEPPolicy(),” 2009. [Online]. Available: http://bernardodamele.blogspot.com.br/2009/12/dep-bypass-with-setprocessdeppolicy.html.

[52] P. Van Eeckhoutte, “Exploit writing tutorial part 10?: Chaining DEP with ROP – the Rubik’s[TM] Cube,” 2010. [Online]. Available: https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/.

[53] Blake, “MY MP3 Player 3.0 m3u Exploit DEP Bypass,” 2011. [Online]. Available: http://www.exploit-db.com/exploits/17854/.

[54] L. Le, “Payload already inside: data reuse for ROP exploits,” in *Black Hat USA*, 2010.

[55] A. Francillon, C. Castelluccia, S. I. Cedex, and I. Rhône-alpes, “Code injection attacks on harvard-architecture devices,” in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 15–26.

[56] F. Lidner, “Developments in cisco ios forensics. CONFidence 2.0.” 2009.

[57] C. Miller and V. Iozzo, “Fun and games with Mac OS X and iPhone payloads,” *BlackHat Eur.*, 2009.

[58] R. Naraine, “Pwn2Own 2010: iPhone hacked, SMS database hijacked,” *Online: http://blogs.zdnet.com/security*, 2010.

[59] D. Blazakis, “Interpreter exploitation,” in *Proceedings of the USENIX Workshop on Offensive Technologies*, 2010.

[60] T. Wei, T. Wang, L. Duan, and J. Luo, “Secure dynamic code generation against spraying,” in *Proceedings of the 17th ACM conference on Computer and communications security (CCS)*, 2010, p. 738.

[61] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, “DROP: Detecting Return-Oriented Programming Malicious Code,” in *Proceedings of the 5th International Conference on Information Systems Security*, 2009, vol. 5905, pp. 163–177.

[62] L. Davi, A. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, 2009, pp. 49–54.

[63] P. Bania, "Security mitigations for return-oriented programming attacks," 2010.

[64] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with 'Return-Less' kernels," *Proc. 5th Eur. Conf. Comput. Syst.*, p. 195, 2010.

[65] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," *Proc. 17th ACM Conf. Comput. Commun. Secur.*, p. 559, 2010.

[66] L. Davi, A. Dmitrienko, A. Sadeghi, and G. Horst, "Return-Oriented Programming without Returns on ARM," 2010.

[67] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," *Proc. 6th ACM Symp. Information, Comput. Commun. Secur.*, 2011.

[68] B. Murgante, O. Gervasi, S. Misra, N. Nedjah, A. M. A. C. Rocha, D. Taniar, and B. O. Apduhan, Eds., *Jump Oriented Programming on Windows Platform (on the x86)*, vol. 7335. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[69] M. Payer and T. R. Gross, "String Oriented Programming: When ASLR is not Enough," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, 2013, pp. 1–9.

[70] P. Team, "PaX address space layout randomization (ASLR)," 2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt.

[71] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib (c)," in *Annual Computer Security Applications Conference (ACSAC)*, 2009, pp. 60–69.

[72] P. Vreugdenhil, "Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit," 2010. [Online]. Available: http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf.

[73] The H Security, "Pwn2Own 2009: Safari, IE 8 and Firefox exploited," 2009. [Online]. Available: http://www.h-online.com/security/news/item/Pwn2Own-2009-Safari-IE-8-and-Firefox-exploited-740663.html.

[74] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security (CCS)*, 2004, pp. 298–307.

[75] T. Durden, "Bypassing pax aslr protection," *Phrack Mag.*, vol. 59, no. 9, 2002.

[76] Rapid7, "Exploiting the ANI vulnerability on Vista," 2007. [Online]. Available: https://community.rapid7.com/community/metasploit/blog/2007/04/01/exploiting-the-ani-vulnerability-on-vista.

[77] P. Van Eeckhoutte, "Exploit writing tutorial part 11?: Heap Spraying Demystified," 2011. [Online]. Available: https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/.

[78] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*, 2012, pp. 157–168.

[79] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu, "Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks," *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, pp. 309–318, 2012.

[80] A. Gupta, S. Kerr, M. Kirkpatrick, and E. Bertino, "Marlin: making it harder to fish for gadgets," in *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*, 2012, pp. 1016–1018.

[81] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.

[82] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd My Gadgets Go?," *2012 IEEE Symp. Secur. Priv.*, pp. 571–585, May 2012.

[83] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Diversifying the Software Stack Using Randomized NOP Insertion," in in *Moving Target Defense II*, 2013, pp. 151–173.

[84] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Practical Software Diversification Using In-Place Code Randomization," in in *Moving Target Defense II*, vol. 100, S. Jajodia, A. K. Ghosh, V. S. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, Eds. New York, NY: Springer New York, 2013.

[85] M. and G. Payer, "StackGhost: Hardware facilitated stack protection," in *Proceedings of the 10th conference on USENIX Security Symposium*, 2001, p. 5.

[86] Vendicator, "Stack Shield: A 'stack smashing' technique protection tool for Linux," 2000. [Online]. Available: http://www.angelfire.com/sk/stackshield/.

[87] T. Chiueh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *International Conference on Distributed Computing Systems*, 2001, pp. 409–417.

[88] S. Sinnadurai and Q. Zhao, "Transparent Runtime Shadow Stack: Protection against malicious return address modifications," 2008.

[89] T. Shuo, H. Yeping, and D. Baozeng, "Prevent Kernel Return-Oriented Programming Attacks Using Hardware Virtualization," in in *Lecture Notes in Computer Science*, vol. 7232, M. D. Ryan, B. Smyth, and G. Wang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 289–300.

[90] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie, "Efficient detection of the return-oriented programming malicious code," in in *Proceedings of the 6th international conference on Information systems security (ICISS)*, vol. 6503, S. Jha and A. Mathuria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 140–155.

[91] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "Poster: control-flow integrity for smartphones," in *Proceedings of the 18th ACM conference on Computer and communications security (CCS)*, 2011, p. 749.

[92] L. Davi, A. Dmitrienko, and M. Egele, "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones," *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012.

[93] Z. J. Huang, T. Zheng, and J. Liu, "A dynamic detective method against ROP attack on ARM platform," in *2012 Second International Workshop on Software Engineering for Embedded Systems*, 2012, pp. 51–57.

[94] L. Davi, A. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011, pp. 1–21.

[95] M. Kayaalp and M. Ozsoy, "Efficiently Securing Systems from Code Reuse Attacks," *IEEE Trans. Comput.*, pp. 1–14, 2012.

[96] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," *2012 39th Annu. Int. Symp. Comput. Archit.*, pp. 94–105, Jun. 2012.

[97] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[98] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure Execution via Program Shepherding," in *Proceedings of the 11th USENIX Security Symposium*, 2002, pp. 191–206.

[99] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1–12.

[100] V. Pappas, "kBouncer: Efficient and Transparent ROP Mitigation," 2012.

[101] I. Fratric, "Runtime Prevention of Return-Oriented Programming Attacks," 2012.

[102] J. Demott, "/ROP - BlueHat Prize Submission," 2012.

[103] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," *Proc. 12th ACM Conf. Comput. Commun. Secur.*, p. 340, 2005.

[104] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference on (ACSAC)*, 2011, p. 353.

[105] Z. Huang, T. Zheng, Y. Shi, and A. Li, "A dynamic detection method against ROP and JOP," *2012 Int. Conf. Syst. Informatics*, no. Icsai, pp. 1072–1077, May 2012.

[106] L. Yuan, W. Xing, H. Chen, and B. Zang, "Security breaches as PMU deviation," in *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)*, 2011, p. 1.

[107] Y.-H. Han, D.-S. Park, W. Jia, and S.-S. Yeo, Eds., "Detecting Return Oriented Programming by Examining Positions of Saved Return Addresses," in in *Lecture Notes in Electrical Engineering*, vol. 214, Dordrecht: Springer Netherlands, 2013.

[108] J. Jiang, X. Jia, D. Feng, S. Zhang, and P. Liu, "HyperCrop: a hypervisor-based countermeasure for return oriented programming," in in *Lecture Notes in Computer Science*, 2011.

[109] Ú. Erlingsson, "Low-level software security: Attacks and defenses," in in *Foundations of security analysis and design IV*, Springer, 2007, pp. 92–134.

[110] D. Dai Zovi, "Practical return-oriented programming," in *SOURCE Boston*, 2010.

[111] Microsoft, "/SAFESEH (Image has Safe Exception Handlers)," 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/9a89h429(v=vs.110).aspx. [Accessed: 06-Mar-2013].

[112] M. Howard, M. Miller, J. Lambert, and M. Thomlinson, "Windows ISV Software Security Defenses," 2010. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb430720.aspx.

[113] Microsoft, "An update is available for the ASLR feature in Windows 7 or in Windows Server 2008 R2," 2013. [Online]. Available: http://support.microsoft.com/kb/2639308.

[114] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for Linux," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010, pp. 175–183.

[115] B. Buck, "An API for Runtime Code Patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, Nov. 2000.

[116] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary transformation in a distributed environment," 2001.

[117] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2004, p. 14.

[118] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Not.*, vol. 40, no. 6, p. 190, Jun. 2005.

[119] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007, vol. 42, no. 6, p. 89.

[120] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, “Retargetable and reconfigurable software dynamic translation,” in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2003, pp. 36–47.

[121] D. L. Bruening, “Efficient, transparent, and comprehensive runtime code manipulation,” Massachusetts Institute of Technology, 2004.

[122] A. Guha, J. D. Hiser, N. Kumar, J. Yang, M. Zhao, S. Zhou, B. R. Childers, J. W. Davidson, K. Hazelwood, and M. Lou Soffa, “Virtual Execution Environments: Support and Tools,” in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–6.

[123] M. Polychronakis and A. D. Keromytis, “ROP payload detection using speculative code execution,” in *2011 6th International Conference on Malicious and Unwanted Software*, 2011, pp. 58–65.

[124] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, “Detection of Intrusions and Malware & Vulnerability Assessment,” in *Proceedings of the Third international conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2006, vol. 4064, pp. 54–73.

[125] F. Bellard, “QEMU, a fast and portable dynamic translator,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005, p. 41.

[126] L. Chen, J. Jiang, and D. Zhang, “Code Reuse Prevention through Control Flow Lazily Check,” in *2012 IEEE 18th Pacific Rim International Symposium on Dependable Computing*, 2012, pp. 51–60.

[127] Standard Performance Evaluation Corporation, “SPEC CPU2006,” 2014. [Online]. Available: https://www.spec.org/cpu2006/.

[128] Stack Overflow, “Which is the best C++ compiler?,” 2014. [Online]. Available: http://stackoverflow.com/questions/1114860/which-is-the-best-c-compiler.

[129] Offensive Security, “Exploit Database,” 2014. [Online]. Available: http://www.exploit-db.com/about/.

[130] M. Tan, X. Liu, J. Zhang, D. Tong, and X. Cheng, “Compiler-Assisted Value Correlation for Indirect Branch Prediction,” *Chinese J. Electron.*, vol. 21, no. 3, 2012.

[131] T. Li, R. Bhargava, and L. Kurian John, “Rehashable BTB: an adaptive branch target buffer to improve the target predictability of Java code,” *High Perform. Comput. 2002*, pp. 597–608, 2002.

[132] H. Samir and Mr_me, “AoA Audio Extractor 2.x - ActiveX ROP exploit,” *Exploit Database*, 2010. [Online]. Available: http://www.exploit-db.com/exploits/15235/.

[133] Node, “ASX to MP3 Converter 3.1.2.1 - SEH Exploit (Multiple OS, DEP and ASLR Bypass),” *Exploit Database*, 2010. [Online]. Available: http://www.exploit-db.com/exploits/14352/.

[134] C. G0M3S, “Free CD to MP3 Converter 3.1 Universal DEP Bypass Exploit.” [Online]. Available: http://www.exploit-db.com/exploits/17634/.

[135] Metasploit Framework, “ProFTPD 1.3.2rc3 - 1.3.3b Telnet IAC Buffer Overflow (Linux),” *Exploit Database*, 2010. [Online]. Available: http://www.exploit-db.com/exploits/16851/.

[136] J. Salwan, “PHP 5.3.6 - Buffer Overflow PoC (ROP),” *Exploit Database*, 2011. [Online]. Available: http://www.exploit-db.com/exploits/17486/.

[137] Ipv, “Wireshark <= 1.4.4 , DECT Dissector Remote Buffer Overflow,” *Exploit Database*, 2011. [Online]. Available: http://www.exploit-db.com/exploits/18145/.

[138] Metasploit Framework, “NetSupport Manager Agent Remote Buffer Overflow,” *Exploit Database*, 2011. [Online]. Available: http://www.exploit-db.com/exploits/16838/.

[139] Metasploit Framework, “Wireshark <= 1.4.4 packet-dect.c Stack Buffer Overflow,” *Exploit Database*, 2011. [Online]. Available: http://www.exploit-db.com/exploits/17195/.

[140] Rew, “DVD X Player 5.5 Pro (SEH DEP + ASLR Bypass) Exploit,” *Exploit Database*, 2011. [Online]. Available: http://www.exploit-db.com/exploits/17803/.

[141] C4SS!0 and H1ch4m, "Zinf Audio Player 2.2.1 - (.pls) Buffer Overflow Vulnerability (DEP BYPASS)," *Exploit Database*, 2011. [Online]. Available: http://www.exploit-db.com/exploits/17600/.

[142] C. G0M3S, "D.R. Software Audio Converter 8.1 - DEP Bypass Exploit," *Exploit Database*, 2011. [Online]. Available: http://www.exploit-db.com/exploits/17665/.

[143] Mr_me, "Mozilla Firefox 3.6.16 mChannel Object Use After Free Exploit (Win7)," *Exploit Database*, 2011. [Online]. Available: http://www.exploit-db.com/exploits/17672/.

[144] M. Memelli, "Mozilla Firefox Array.reduceRight() Integer Overflow Exploit," *Exploit Database*, 2011. [Online]. Available: http://www.exploit-db.com/exploits/17974/.

[145] M. Memelli, "PHP 6.0 Dev str_transliterate() Buffer overflow - NX + ASLR Bypass," *Exploit Database*, 2010. [Online]. Available: http://www.exploit-db.com/exploits/12189/.

[146] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z," 2013. [Online]. Available: http://download.intel.com/products/processor/manual/325383.pdf.

[147] Syddansk Universitet, "Intel and AT&T Syntax," 2014. [Online]. Available: http://www.imada.sdu.dk/Courses/DM18/Litteratur/IntelnATT.htm.

[148] Intel, "fomit-frame-pointer, Oy," 2013. [Online]. Available: http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/GUID-9ECCD2E7-EBFE-44DC-B07D-415DE59F32CA.htm.

[149] Nullstone Corporation, "Function Inlining," 2014. [Online]. Available: http://www.compileroptimizations.com/category/function_inlining.htm.

[150] O. Yuschuk, "OllyDbg," 2014. [Online]. Available: http://www.ollydbg.de/.

[151] Free Software Foundation, "GDB: The GNU Project Debugger," 2014. [Online]. Available: http://www.gnu.org/software/gdb/.

[152] Intel, "Pin - A Dynamic Binary Instrumentation Tool," 2014. [Online]. Available: http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[153] Intel, "Pin 2.13 User Guide," 2014. [Online]. Available: http://software.intel.com/sites/landingpage/pintool/docs/62732/Pin/html/.

[154] AMD, "'Barcelona' Processor Feature: Advanced Bit Manipulation (ABM)," 2014. [Online]. Available: http://developer.amd.com/community/blog/2007/09/26/barcelona-processor-feature-advanced-bit-manipulation-abm/.

[155] Intel, "Intel® SSE4 Programming Reference," 2007. [Online]. Available: http://software.intel.com/sites/default/files/m/d/4/1/d/8/d9156103.pdf.

[156] Stack Overflow, "Programmatically get the cache line size?," 2014. [Online]. Available: http://stackoverflow.com/questions/794632/programmatically-get-the-cache-line-size.

[157] Standard Performance Evaluation Corporation, "CPU2006 Frequently asked questions," 2014. [Online]. Available: http://www.spec.org/cpu2006/Docs/faq.html.

[158] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques," *IEEE Trans. Comput.*, vol. 48, no. 11, 1999.