Modelo Multicamadas para Detecção de Ataques ROP

Hugo Araújo de Sousa[1], **Mateus Tymburibá**[2], **Fernando Magno Quintão Pereira**[1]

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

[2]Departamento de Computação – CEFET-MG
Av. Amazonas, 7675 – 30510-000 – Belo Horizonte – MG – Brazil

{hugosousa,fernando}@dcc.ufmg.br, mateustymbu@decom.cefetmg.br

Abstract. *This paper presents a multi-layer protection system to guard programs again Return-Oriented Programming (ROP) attacks. Upper layers have low computational cost; thus, they do not compromise the runtime of legitimate program executions. Lower layers provide very strong enforcement guarantees; thus, they are difficult to circumvent. Our multilayer system combines several techniques already described in the literature; however, to be practical, we had to augment them with an extra verification, which consists in checking if the target address of call instructions lays in an executable program segment. We demonstrate the effectiveness of our multilayer protection in internet banking applications taken from two among the main Brazilian banks: Banco do Brasil and Bradesco. Our experiments on these two systems and on several well-known benchmarks indicate that we can protect programs with almost no overhead in practice.*

Resumo. *Esse artigo apresenta um sistema de proteção multicamadas para defesa de programas contra ataques de Return-Oriented Programming (ROP). Camadas superiores têm custo computacional baixo; assim, elas não comprometem o tempo de execução de programas que não estão sob ataque. Camadas inferiores proveem garantias muito fortes; assim, elas são difíceis de contornar. Nosso sistema multicamadas combina várias técnicas já descritas na literatura; entretanto, para serem práticas, nós tivemos que estendê-las com uma verificação extra, que consiste em checar se o endereço alvo de uma instrução de chamada de função (`call`) está em um segmento executável da memória. Nós demonstramos a eficiência da nossa proteção multicamadas em aplicações bancárias de dois dos principais bancos brasileiros: Banco do Brasil e Bradesco. Nossos experimentos nesses dois sistemas e em vários benchmarks populares indicam que podemos proteger programas em tempo de execução com custo quase nulo na prática.*

1. Introdução

Return-Oriented Programming (ROP) é uma forma de ataque contra *softwares* que consiste no encadeamento de sequências de instruções binárias de um programa, com o objetivo de forçar ações não originalmente desejadas pelos desenvolvedores da aplicação [Shacham 2007]. Se o código binário é suficientemente grande, ele pode conter sequências de instruções que formam uma linguagem Turing-completa. Assim,

tais sequências de instruções, os *gadgets*, quando combinados, dão ao atacante controle total sobre a aplicação afetada. Ataques ROP têm sido particularmente prejudiciais. Exemplos famosos de tais *malwares* incluem Conficker [F-Secure Response Labs 2013], Duqu [Estes 2017] e o ataque a máquina de votação americana [Checkoway et al. 2009].

Muito tem sido feito para dificultar o sucesso de ataques ROP. Técnicas populares que combatem esses ataques incluem a randomização do espaço de endereçamento (ASLR) [Shacham et al. 2004] e o controle da integridade do fluxo de execução (CFI) [Abadi et al. 2005]. A adoção dessas técnicas tem dificultado significativamente o sucesso de técnicas ROP. Mesmo assim, trabalhos recentes mostraram que esses ataques ainda são possíveis, mesmo contra as defesas mais modernas existentes atualmente [Szekeres et al. 2013, Carlini et al. 2015, van der Veen et al. 2017]. Ataques recentes contam com uma observação simples: os mecanismos de proteção mais eficazes são computacionalmente muito caros para serem práticos [Szekeres et al. 2013].

Neste artigo, nós afirmamos que, para serem práticas, proteções contra ataques de reuso de código devem aliar eficiência e eficácia. Para isso, nós apresentamos argumentos a favor de um mecanismo de proteção multicamadas, descrito na seção 3. Esse mecanismo multicamadas não somente permite o uso escalável de proteções custosas (em relação ao aumento no tempo de execução das aplicações protegidas), como também a combinação de várias dessas defesas. O resultado é um sistema de proteção mais robusto com *overhead*, isto é, custo computacional, aceitável. As proteções em camadas superiores devem ser capazes de verificar a idoneidade da maior parte das instruções de um programa, a um baixo custo computacional. Proteções em camadas inferiores são responsáveis por lidar com situações menos usuais, que poderiam levar nosso sistema a incorretamente indicar ataques que não existem, ou desconsiderar ataques reais, conforme descrito na seção 3.3. Por isso, elas executam verificações mais complexas, que se tornam mais caras do ponto de vista computacional. Contudo, devido a esses testes mais elaborados, elas são mais difíceis de serem contornadas. Como a porcentagem de casos tratados nas camadas inferiores do modelo é muito pequena, o impacto desse elevado custo computacional torna-se irrisório.

Como prova de conceito, apresentamos um sistema de defesa formado por quatro camadas contra ataques de reuso de código que abusam de instruções de retorno para tomar controle do fluxo de execução de um programa. A primeira camada, explicada na seção 3.1, utiliza preditores de desvios em hardware para filtrar ataques ROP. Se o destino de uma instrução de retorno é previsto corretamente, ela pode ser considerada parte do fluxo de execução válido. Essa conclusão vem da correspondência entre chamadas de funções e instruções de retorno. A verificação dessa correspondência é uma estratégia clássica usada por várias proteções para garantir a autenticidade de instruções de retorno. A segunda camada de nosso sistema, assunto da seção 3.2, é um pouco mais complexa. Nós a implementamos como uma extensão do conceito bem conhecido de verificação da existência de uma instrução de chamada de função imediatamente antes do endereço destino de instruções de retorno [Kiriansky et al. 2002]. Ao invés de somente assegurar que um endereço de retorno é precedido por uma instrução `call`, também verificamos se o `call` que precede o endereço de retorno é válido. Essa etapa reduz o número de *gadgets* disponíveis para a construção de um ataque ROP, como mostrado na seção 4.1. Nossa terceira camada, apresentada na seção 3.3, lida com falsos positivos associados

à verificação de endereço de retorno precedido por `calls`. Finalmente, nossa quarta camada consiste em uma verificação mais completa e, portanto, mais cara (seção 3.4).

Para validar essas ideias, nós as implementamos em um protótipo de pesquisa baseado no arcabouço Pin, da Intel [Luk et al. 2005]. Nós testamos nossas técnicas em duas aplicações bancárias, do Banco Bradesco e do Banco do Brasil [Banco Central do Brasil 2018]. Dado o grande volume de clientes que utilizam essas aplicações, elas são alvos atrativos para ataques de *software*. Nós mostramos empiricamente que podemos reduzir o número de *gadgets* disponíveis para $0,09\%$ do número total de *gadgets* coletados com a ferramenta Mona [Corelan Team 2011]. Além disso, mesmo se um atacante conseguir encontrar *gadgets* válidos nesse cenário restrito, ele terá que ultrapassar os demais níveis de proteção do nosso modelo. O cerne de nosso trabalho é que o último nível de proteção, embora extremamente custoso computacionalmente, só é ativado para $0,0161\%$ das instruções de retorno dos benchmarks que usamos. Nós mostramos na seção 4.2 que tais eventos são raros: observamos que acontecem uma vez a cada $1,8$ bilhões de instruções, na média.

2. *Return-Oriented Programming*

Ataques ROP são construídos através de cadeias de *gadgets*. Um *gadget* é uma sequência de instruções terminada com uma instrução de desvio indireto, geralmente uma instrução de retorno. Esses ataques são usados após a exploração de algum tipo de vulnerabilidade, como estouro de *buffer*, estouro de inteiros e uso depois de *free*, entre outras [Szekeres et al. 2013]. Após explorar uma vulnerabilidade, um atacante passa a controlar o fluxo de execução da aplicação alvo. Sobrescrevendo a pilha de execução com os endereços de vários *gadgets*, consegue-se garantir que eles serão executados em sequência. Depois que um *gadget* termina sua execução, sua última instrução de desvio indireto leva o fluxo de execução do programa para o próximo *gadget*, cujo endereço foi empilhado anteriormente pelo atacante.

Esses *gadgets* podem ser encadeados para burlar proteções típicas de nível de sistema operacional, como a *Write*$\oplus$*Execute*. Uma vez desabilitada, os atacantes podem executar código de páginas de memória marcadas como dados. Por exemplo, o sistema operacional Windows, dependendo de suas configurações, permite que um processo desabilite a proteção *Data Execution Prevention* (DEP) através da rotina `SetProcessDEPPolicy`. Para invocar essa função, o atacante deve construir seus argumentos. Simplesmente dispor esses parâmetros na pilha, após a exploração de um estouro de *buffer*, pode não ser suficiente para configurar o ataque. Endereços de código desconhecidos ou a presença de caracteres de parada (o *byte* '$\backslash 0$'que termina *strings*, por exemplo) na pilha requerem uma escolha cuidadosa de *gadgets* para construir o ataque.

O maior desafio que atacantes encontram ao executar ataques baseados em ROP é encontrar *gadgets* livres de efeitos colaterais de forma a evitar estragar valores previamente empilhados pelos próprios atacantes. Para isso, existem ferramentas que encontram *gadgets* no código binário de programas. Como exemplo, Q [Schwartz et al. 2011], Mayhem [Cha et al. 2012] e Mona [Corelan Team 2011] dão ao atacante meios de descobrir automaticamente *gadgets* em um arquivo binário. Mona fornece até mesmo sugestões de *gadgets* que têm maior chance de serem úteis em um ataque real.

A necessidade de defesas multicamadas: Apesar de anos de pesquisa por técnicas para

proteger programas de ataques ROP, essa forma de exploração de *software* permanece uma ameaça. Essa constatação é confirmada por resultados que surgiram nos últimos três anos [Carlini et al. 2015, van der Veen et al. 2017]. Todas essas técnicas encontram formas para burlar proteções ROP individuais. Elas contam com um fato bem conhecido: os mecanismos de proteção mais eficazes impõem às aplicações um *overhead* alto suficiente para torná-las impraticáveis. Nós enfatizamos que defesas de baixo *overhead*, tais como ASLR, não são suficientes para proteger programas contra redirecionamento de fluxo de execução - e já foram há muito superadas pelos atacantes [Shacham et al. 2004]. Por outro lado, defesas como *Write Integrity Testing* (WIT) [Akritidis et al. 2008] e pilhas-sombra (*shadow stack*) [Szekeres et al. 2013] são muito custosas para serem utilizadas: seu uso na prática tornaria a aplicação protegida excessivamente lenta. Motivados por esses fatos, nós alegamos que é possível combinar diferentes mecanismos de defesa em camadas, de forma que camadas em níveis inferiores (custosas, mas eficazes) somente são utilizadas quando camadas em níveis superiores (baratas, porém mais fáceis de serem superadas) foram burladas pelos atacantes. A seguir, explicamos como podemos implementar essa abordagem.

3. Camadas de proteção

A Figura 1 mostra uma visão abstrata do mecanismo de proteção que defendemos. A ideia chave é certificar a validez de desvios indiretos - o pilar de ataques ROP modernos - em camadas. Os alvos da vasta maioria desses desvios serão certificados em nossa primeira camada, descrita na seção 3.1. Os poucos alvos que não podemos validar nesse nível passarão por uma etapa de verificação adicional em nossa segunda camada, explicada na seção 3.2. Uma terceira camada, baseada em trabalhos anteriores [Xia et al. 2012], filtrará esse fluxo de desvios ainda mais, usando a metodologia descrita na seção 3.3. Finalmente, se não é possível verificar um dado alvo nessas três camadas, nós recorreremos ao uso de caixas de areia (*sandbox*), em nível de sistema operacional, uma ação que discutimos na seção 3.4. Nossa ideia é filtrar a maioria dos alvos de desvios indiretos com custo computacional baixo. Para os poucos restantes, podemos pagar o preço alto de uma proteção mais forte. Como mostramos na seção 4.2, a proporção de alvos que escapam de uma camada para a próxima é minúscula.

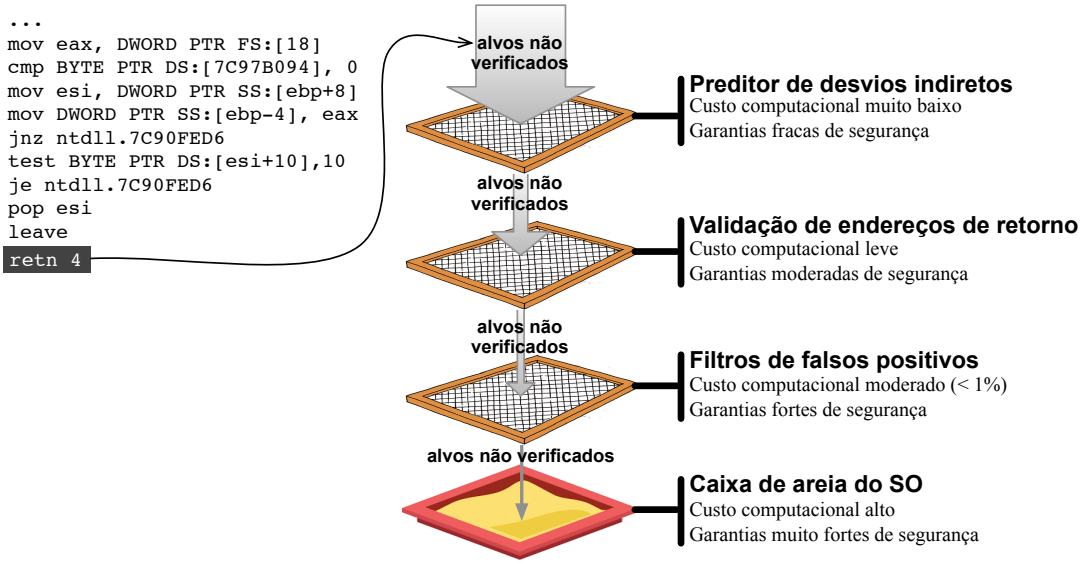

Figura 1. Visão geral abstrata de nosso mecanismo de proteção multicamadas.

3.1. Primeiro nível: preditores de desvios

A primeira camada de proteção depende de preditores de desvios para validar os alvos de desvios indiretos. Arquiteturas de computadores modernas tentam prever o alvo de instruções de desvios indiretos para evitar paradas no *pipeline*. Por exemplo, implementações atuais da arquitetura x86 usam a chamada *Return Address Stack* (RAS) para prever os endereços alvos de tais instruções. Sempre que uma função é invocada através de uma instrução `call`, o endereço imediatamente posterior a essa invocação é empilhado na RAS. Quando uma operação de retorno é executada, o preditor tenta adivinhar seu alvo como o endereço no topo da RAS. Previsões feitas corretamente indicam fluxos legítimos do programa. Assim, esses alvos de desvios não precisam de verificação adicional. Esse *modus operandi* é simples, porém, muito eficaz e barato.

O uso que fazemos da RAS é similar a uma técnica conhecida como pilha-sombra [Vendicator 2000]. Uma pilha-sombra faz exatamente o que uma RAS faz: realiza a correspondência entre alvos de instruções `return` e os endereços posteriores a instruções `call`. Entretanto, pilhas sombra foram concebidas antes que a RAS se tornasse parte de processadores x86 atuais; dessa forma, elas não podiam se beneficiar desse *hardware*. Hoje em dia, nós alegamos que essa ideia, a pilha-sombra, pode ser implementada com *overhead* nulo, no nível do *hardware*, tirando proveito da infraestrutura da RAS, que já está implementada. Apesar de não tratarmos os outros tipos de desvios indiretos neste trabalho, enfatizamos que seus preditores possuem as mesmas características de eficácia e eficiência apresentadas pela RAS. Isso os torna candidatos perfeitos para compor a primeira camada de proteção contra o abuso de desvios indiretos (`calls` e `jmps`). Na seção 4.2.1, nós mostramos a taxa de acerto desses preditores, além da taxa de acerto alcançada pela RAS.

3.2. Segundo nível: `calls` válidos

Nem todo endereço de retorno previsto incorretamente é um sintoma de ataque ROP. Previsões incorretas podem ser causadas por estouros no *buffer* da RAS, um incidente que acontece devido a sequências muito longas de invocações de funções, explicadas tanto por padrões complexos de programação, quanto chamadas recursivas.

Se o alvo de uma instrução de retorno é previsto incorretamente, então devemos prosseguir com validações adicionais. O nosso próximo passo de verificação, a segunda camada do nosso sistema, conta com uma ideia simples: nós checamos se o alvo de uma instrução `return` é precedida por uma instrução `call` *e* se o alvo da instrução `call` que precede o endereço de retorno está presente em um segmento executável da memória. A Figura 2 fornece uma visão geral de todas essas interações. No restante desta seção, descrevemos essas duas checagens, usando a Figura 2 para guiar a discussão.

A restrição de precedência por `calls` válidos Se o alvo de uma instrução de retorno não é um endereço que sucede uma instrução `call`, então é possível que tal fluxo de execução foi criado artificialmente por um atacante. Essa observação já foi extensivamente explorada na literatura [Kiriansky et al. 2002]. Assim, na ausência dessa condição, nós imediatamente levamos a aplicação à terceira camada de verificação do nosso sistema (passo 1 na Figura 2), onde aplicamos checagens mais fortes para certificar a legitimidade do fluxo de execução do programa. Contudo, mesmo quando essa condição é satisfeita, nós ainda podemos estar diante de um ataque ROP. Por exemplo, Goktas *et al.* demonstraram como

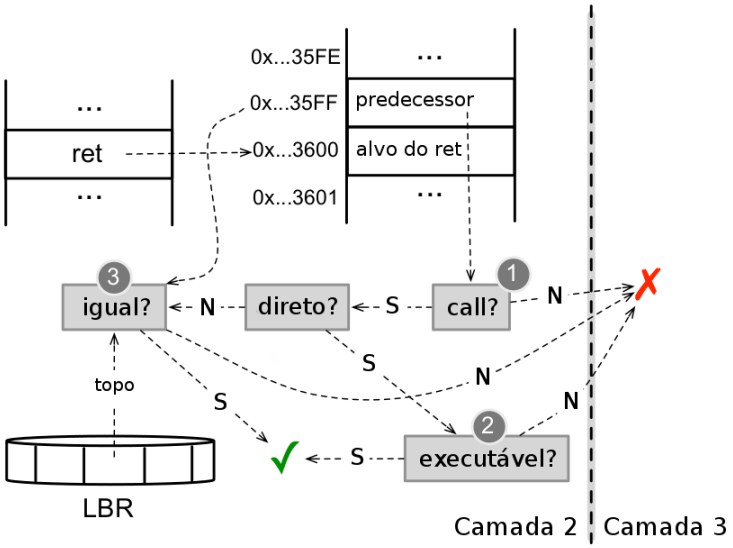

Figura 2. A segunda camada de proteção.

construir ataques ROP sob a restrição de precedência por `calls` [Göktas et al. 2014]. Portanto, para prevenir ataques como o realizado por Goktas *et al.*, nós aumentamos essa restrição com uma checagem extra, que reivindicamos como uma contribuição original deste trabalho: a restrição do *alvo executável*.

A restrição do alvo executável *Gadgets* cuja primeira instrução é precedida por instruções `call` podem emergir por acaso em um código binário grande, devido a *instruções não intencionais*. Essas são instruções formadas por sequências não alinhadas de *bits* dentro do código binário. Nós podemos mostrar que a vasta maioria de tais *gadgets* são inválidos através de um recurso simples: nós verificamos se a instrução `call` tem como alvo um segmento de memória executável (passo 2 na Figura 2). Dado que a área executável no espaço de endereçamento de um programa é limitada ao seu segmento de "texto", a chance de uma instrução `call` desalinhada ter como alvo esse segmento é muito pequena, principalmente em arquiteturas 64-*bit*. Como exemplo, a versão Linux do navegador Chrome 59.0.3071.115 64-*bit* possui uma grande quantidade de código, resultando em um segmento de texto igualmente longo (163 MB). Ainda assim, a chance de que um endereço aleatório corresponda a um endereço executável é somente 1 em 10^{11}. Essa restrição pode ser facilmente imposta em *hardware* geral. Por exemplo, máquinas Intel/AMD atuais possuem uma extensão chamada *eXecute Disable* (XD, de "execução desabilitada", para AMD; e NX, de "não executável", para Intel). Esse mecanismo marca páginas de memória virtual com um *bit* que denota memória executável. Esse *bit* é consultado com *overhead* quase nulo em *hardware*.

Impondo a restrição em `calls` indiretos A verificação do endereço alvo de `calls` indiretos não é trivial, porque quando uma operação de retorno executa, o endereço usado pelo `call` que deu origem a ela pode não estar mais disponível tanto em registradores quanto em memória. Para lidar com esse aspecto, nós recorremos a uma estrutura de *hardware* chamada *Last Branch Record* (LBR). Essa estrutura, disponível em processadores Intel, permite que a CPU registre os endereços de origem e destino de cada instrução de desvio em registradores específicos. Esses registradores formam um *buffer* circular, que é sobrescrito continuamente. Assim, ele armazena somente os desvios mais recentes. Nós registramos somente dados sobre `calls` no LBR. Além disso, usamos essa estrutura de *hardware* em seu modo de pilha. Dessa forma, sempre que um `call` é executado, seus

dados são empilhados no LBR. Quando um `return` é executado, os dados mais recentes empilhados são removidos do LBR. A principal diferença entre a pilha do LBR e a RAS é que o LBR garante a restauração de dados em trocas de contexto [Kleen 2016]. Isso permite que o LBR trate alguns casos não cobertos pela RAS. Diferentes modelos de processadores possuem diferentes tamanhos de LBR: Netburst e Merom possuem *buffers* com 4 células; modelos Nehalem até Haswell têm *buffers* com 16; Skylake possui 32 posições e Atom apenas 8. Quando encontramos um `call` indireto precedendo um endereço de retorno, nós examinamos a última entrada escrita do LBR, procurando por uma correspondência (passo 3 na Figura 2). Note que somente endereços de retorno previstos incorretamente passarão por essa verificação, porque instruções previstas corretamente já foram filtradas na primeira camada do nosso sistema.

3.3. Terceiro nível: filtragem de falsos positivos

É possível que endereços de retorno legítimos passem não verificados através de todas as nossas camadas anteriores, porque alguns fluxos de execução contêm instruções de retorno com endereços alvo não precedidos por operações `call` válidas. As situações típicas onde esse evento acontece foram catalogadas por Xia *et al.* [Xia et al. 2012]: ligação tardia de bibliotecas compartilhadas dinamicamente, tratamento de exceções e uso de instruções do tipo `long-jump`, por exemplo. Para lidar com instruções de retorno que têm como alvo endereços não precedidos por operações `call` válidas, nós recorremos à metodologia usada em CFIMon [Xia et al. 2012]. Para este fim, aplicamos as regras disponíveis na descrição original desse Sistema [Xia et al. 2012, Figura 1]. Essas regras podem ser avaliadas com um custo moderado. Xia *et al.* observaram um *overhead* menor que 1% para executar todas suas checagens, incluindo tratamento de falsos positivos, quando analisando todos os alvos em cargas de trabalho típicas de arquiteturas x86. No nosso caso, esse *overhead* será imposto sobre um número minúsculo de desvios indiretos, como explicamos na seção 4.2.

3.4. Último nível: caixa de areia

Se o alvo de um `return` passa pelas três primeiras camadas, então, com grande probabilidade, o fluxo de execução do programa pode ter sido alterado por um usuário malicioso. Para impedir o atacante de ser bem sucedido, nós prosseguimos com a execução do programa em uma caixa de areia. Caixas de areia são geralmente implementadas através de ambientes de execução emulados ou virtualizados [Egele et al. 2012]. Eles frequentemente detectam comportamento malicioso através do monitoramento de chamadas de sistema. Neste artigo, não implementamos essa última camada de proteção do nosso sistema e não podemos fornecer números sobre seu *overhead*. Entretanto, relatórios anteriores indicam que o custo computacional de emular um programa está entre 2 e 40%, dependendo de escolhas de implementação e da quantidade de chamadas de sistema executadas pela aplicação monitorada [Kolbitsch et al. 2009]. Apesar disso, esperamos que em nosso sistema multicamadas, o uso de caixas de areia acontecerá raramente com fluxos de execução legítimos.

4. Resultados

O objetivo dessa seção é responder três questões de pesquisa, que enumeramos abaixo:

- **QP1** Quão eficazes são as diferentes camadas em reduzir o número de *gadgets* disponíveis para um ataque?
- **QP2** Qual é a proporção de alvos de desvios indiretos filtrados em cada camada de proteção?
- **QP3** Qual é o *overhead* de cada camada da nossa abordagem?

Para responder a essas questões, nós implementamos um protótipo, usando o *framework* Pin, da Intel, que emula o comportamento de um *hardware* protegido com nosso sistema multicamadas.

4.1. QP1 – Redução de *gadgets*

Como Carlini *et al.* explicam [Carlini et al. 2015], uma métrica muito utilizada na literatura para medir a eficácia de abordagens que previnem ataques ROP é o número de *gadgets* que ela deixa ainda disponíveis para um atacante. No nosso caso, atacantes podem construir ataques ROP bem-sucedidos usando *gadgets* que possuam qualquer uma das seguintes propriedades:

1. O *gadget* não causa uma predição incorreta no preditor de desvios;
2. O *gadget* é precedido por uma instrução `call` direta que tem como alvo uma região executável de memória ou uma instrução `call` indireta que está no topo do LBR;
3. O *gadget* passa nas checagens de falsos positivos realizadas pelo CFIMon.

Medir (1) diretamente é difícil, porque não temos exemplos de ataques com essa propriedade. Nós avaliamos 15 ataques baseados em ROP, disponíveis no *Exploit Data-Base* (`www.exploit-db.com/`), e nenhum deles possuía essa propriedade. Nós também não fomos capazes de encontrar *gadgets* que reabilitassem esses ataques para o contexto da nossa proteção. Portanto, presumimos que ataques necessariamente causarão predições incorretas. Essa é uma constatação razoável, pois se um atacante utilizar um *gadget* que coincide com a previsão de desvios, ele estará executando um desvio válido da aplicação e, portanto, seguindo seu caminho de execução regular. O item (3) tem como objetivo filtrar casos autênticos específicos, de forma que eles não sejam classificados incorretamente. Usar *gadgets* com essas características é praticamente impossível, porque essas são operações que involvem chamadas ao sistema operacional. Assim, só sobram aos atacantes *gadgets* que satisfaçam a propriedade (2). No restante desta seção, nós apresentaremos sua contagem.

Para realizar tal contagem, nós utilizamos Mona para gerar uma lista de *gadgets* disponíveis em quatro aplicações vulneráveis para as quais existem ataques ROP conhecidos: Free CD to MP3 Converter 3.1, Firefox 3.6.16, PHP 6.0 e Sygate Personal Firewall 5.6 (*build* 2808). Além de contar o número de *gadgets* encontrados com Mona, nós adicionamos um comando ao *script* para gerar uma lista com todos os *bytes* correspondentes a um *opcode* de instrução `call`. Nós utilizamos essa lista para identificar quais *gadgets* são precedidos por um `call`, mesmo quando essa instrução é resultado de um acesso desalinhado à memória. Finalmente, usamos Pin para simular o mecanismo de validação de `calls` da seção 3.2. O Pin nos permite inspecionar cada instrução no momento de sua execução; assim, nós o utilizamos para (i) verificar que o endereço destino de um `call` que precede um *gadget* candidato é executável e (ii) simular as operações do LBR. A Tabela 1 mostra os resultados desse experimento.

Aplicação	# *Gadgets*	*Gadgets* precedidos por CALLs	*Gadgets* precedidos por CALLs válidos
Free CD to MP3 Converter 3.1	346415	2.63%	0.09%
Firefox 3.6.16	1444959	2.61%	0.10%
PHP 6.0	269318	2.77%	0.04%
Sygate Personal Firewall 5.6	624638	2.64%	0.14%

Tabela 1. Redução de *gadgets*.

Utilizando essa configuração experimental, nós observamos que a *restrição do alvo executável* reduz, em média, para $0,09\%$ o número de *gadgets* disponíveis. Para dar ao leitor uma perspectiva sobre esse número, se compararmos essa redução com a abordagem mais tradicional de somente impor a *restrição de precedência por* `calls`, então reduziríamos o número de *gadgets* disponíveis em $2839,89\%$. Portanto, nossa nova restrição, o alvo executável, produz uma redução extra de quase 30 vezes. Na prática, essa redução torna muito difícil a construção de ataques ROP funcionais. Como exemplo, Q [Schwartz et al. 2011] geralmente requer binários com pelo menos 100KB para conseguir executar chamadas para qualquer função na *libc*, ou binários com pelo menos 20KB para invocar funções dinâmicas e armazenar dados arbitrários em posições arbitrárias [Schwartz et al. 2011]. As aplicações que analisamos possuem binários de $1,08$MB, 890KB, 29KB e $2,45$MB. Aplicando as reduções na disponibilidade de *gadgets* indicada na Tabela 1, seus tamanhos seriam equivalentes a programas de 1KB, $0,8$KB, $0,03$KB e $2,3$KB. Nesse cenário, Q provavelmente seria incapaz de gerar uma corrente de *gadgets* úteis para um ataque.

4.2. QP2 – Estatísticas de filtragem

A abordagem multicamadas que defendemos neste artigo só é interessante se um grande volume de desvios indiretos é filtrado através de camadas sucessivas. Nesta seção, nós mostramos que o fluxo de desvios indiretos a ser validado é drasticamente reduzido após passar pelas camadas 1 e 2 do nosso modelo. Não apresentamos estatísticas para a terceira camada, porque seu propósito é unicamente tratar da ocorrência dos raros casos de falsos positivos descritos na seção 3.3. Nossos *benchmarks* para essa seção são as aplicações bancárias para Windows 10 dos bancos Bradesco e Banco do Brasil, além dos programas presentes na coleção de benchmarks do compilador LLVM [Lattner and Adve 2004], um compilador de código aberto. As aplicações bancárias nos permitem discutir nossos resultados em um contexto mais próximo do mundo real e o arcabouço de testes de LLVM nos fornece uma grande quantidade de dados para análise.

4.2.1. Predição de desvios indiretos

Para analisar a eficiência dos preditores de desvios indiretos (primeira camada do nosso modelo), nós contamos o número de acertos nas previsões desses desvios. Para isso, utilizamos contadores de performance de *hardware*(HPCs). Esses registradores contam as ocorrências de eventos micro-arquiteturais, como falhas em vários níveis de *cache* e predições incorretas de desvios. Nós conduzimos experimentos em um ambiente equipado com processador Intel Core i5-3230M, utilizando o utilitário "perf"do Linux, que fornece acesso aos contadores do i5. Nesses experimentos, monitoramos a quantidade de desvios indiretos previstos corretamente durante a execução de programas do conjunto de testes de LLVM. Uma vez que as estratégias de previsão são distintas para diferentes

tipos de desvios, nós capturamos as contagens de performance para cada tipo de desvio indireto (`return`, `call` e `jmp`). Nós não medimos o *overhead* desses preditores de desvios, porque eles não afetam o tempo de execução das aplicações.

A figura 3 mostra quão preciso é o preditor de desvios, quando aplicado aos programas do conjunto de testes de LLVM. Essa coleção de programas contém 211 aplicações escritas em C, que nos dão mais de um milhão de linhas de código para analisar. Na média, o preditor dinâmico foi bem sucedido em $97,21\%$ dos desvios indiretos executados nos quatro programas que contêm vulnerabilidades do tipo ROP que usamos. No caso específico de endereços de retorno, o RAS atingiu a acurácia de $99,93\%$. A predição de alvos de `jmps` e `calls` indiretos atingiu, respectivamente, taxas de acerto de $91,89\%$ e $91,23\%$. Esses valores confirmam a hipótese de que preditores dinâmicos de desvios indiretos são uma forma eficaz de filtrar os casos a serem monitorados.

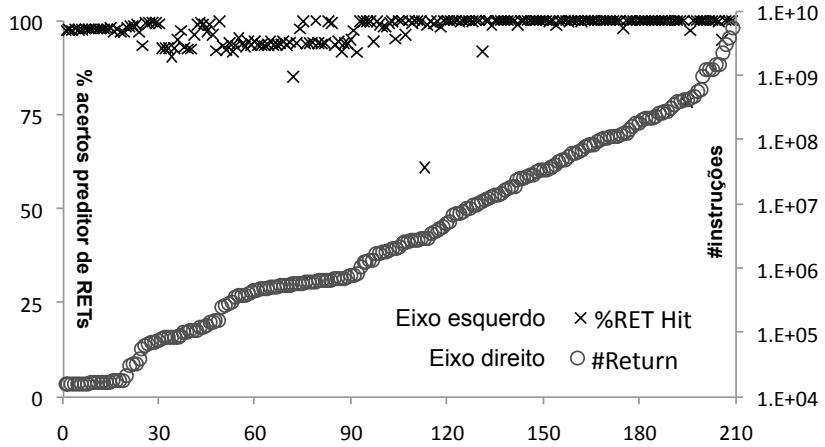

Figura 3. Taxa de acertos do preditor de desvios. Cada ponto do eixo X é um benchmark diferente, ordenado pelo número de instruções executadas.

4.2.2. Validação de endereços de retorno

O *framework* Pin, que utilizamos para implementar o mecanismo de validação de `calls`, também foi usado para obter as estatísticas de filtragem da segunda camada do nosso modelo. Os resultados são exibidos na Tabela 2. Os testes foram executados com os clientes Bradesco e Banco do Brasil para Windows 10.

Média ponderada (sobre o # de instruções)			
Instruções de retorno executadas		746446	
Endereços de retorno precedidos por CALLs válidos	Diretos	70.37%	**99.77%**
	Indiretos	29.40%	
Endereços de retorno precedidos por CALLs inválidos	Diretos	0.17%	**0.23%**
	Indiretos	0.04%	
Endereços de retorno não precedidos por CALLs		0.02%	

Tabela 2. Validações que passam da camada 2 para a camada 3.

As linhas superiores na Tabela 1 mostram casos onde os endereços de retorno são considerados válidos (`ret` precedidos por `call` válido). Linhas inferiores indicam a

Aplicação	*Overhead*
bzip2 1.0.2	76,55%
gzip 1.6	76,61%
oggenc de vorbis-tools 1.4.0	79,87%
gcc 4.8.4	0,62%

Tabela 3. *Overhead* da segunda camada de proteção.

porcentagem de casos enviados para uma nova análise na camada subsequente. Como na predição de endereços de retorno, a validação desses endereços apresenta uma porcentagem muito alta de sucesso ($99,77\%$). Note que a checagem dos `calls` que precedem os endereços de retorno adicionam poucos casos para a camada subsequente ($0,21\%$). Portanto, considerando o benefício fornecido por essa estratégia com a redução no número de *gadgets*, a validação de endereços de retorno proposta neste trabalho mostra-se vantajosa.

O tamanho do *buffer* do LBR utilizado não influenciou nossos resultados. Nós executamos os experimentos nesta seção para LBRs com 4, 8, 16 e 32 entradas. Os valores foram praticamente os mesmos em todos os casos. Este comportamento acontece porque as aplicações bancárias que testamos não contêm código que encadeia muitas chamadas de função, nem possuem funções recursivas. Esses são os cenários que geralmente extrapolam o tamanho do LBR e causam falhas no pareamento de dados.

4.3. QP3 – *Overhead*

Nossa primeira camada de proteção pode ser implementada com *overhead* nulo, porque ela depende de *hardware* que já faz parte de arquiteturas de computadores modernas. Além disso, como discutido na seção 4.2, a proporção de alvos de desvios indiretos que atingem nossa terceira camada é muito pequena para ter um impacto significante em termos de desempenho. Portanto, nesta seção focaremos no *overhead* de nossa segunda camada de proteção. Examinaremos esse custo sobre a execução de quatro *benchmarks*: `bzip2`, `gzip`, `oggenc` e `gcc`. Como nosso protótipo é implementado sobre Pin que, por si só já impõe um alto *overhead* no código que emula, para estimar o *overhead* da nossa abordagem, quando executada em nível de *hardware*, precisamos descontar de nossas medições o custo de execução do Pin. Assim, também executamos nossas aplicações com Pin sem nossa abordagem multicamadas. Nós consideramos que nosso *overhead* é a diferença em tempo de execução entre essa base de referência e nosso protótipo que implementa a proteção multicamadas. A Tabela 3 mostra o resultado dessa análise.

A Tabela 3 mostra um limite superior para o *overhead* esperado, pois estamos emulando em *software* todas as verificações que, na prática, seriam implementadas em *hardware*. Por exemplo, nossa *Pintool* produz sequências com várias operações para checar se a instrução que precede o alvo de um retorno é um `call`. Ainda mais instruções são usadas para verificar se o `call` tem como alvo uma área executável de memória. Em *hardware*, essas verificações aconteceriam em paralelo com o *pipeline* de instruções. Assim, especulamos que seu custo seria ordens de magnitude mais baixo.

5. Trabalhos relacionados

A ideia de combinar diferentes mecanismos de defesa em camadas, de forma que proteções de baixo *overhead* são utilizadas para filtrar gradualmente fluxos de execução seguros, é uma contribuição original deste artigo. Porém, exceto pela verificação dos alvos executáveis de `calls`, discutida na seção 3.2, nossas camadas de defesa reusam

técnicas apresentadas em trabalhos anteriores. Nossa lógica ao escolher qual técnica aplicar em cada camada foi baseada em dois princípios: (i) camadas superiores – que são aplicadas primeiro – devem possuir baixo custo computacional; e (ii) o trabalho feito em camadas superiores não será repetido em camadas inferiores.

O uso do preditor de desvios para detectar ataques do tipo ROP foi proposto originalmente por Shi e Lee [Shi and Lee 2007], logo após Hovav Shacham [Shacham 2007] introduzir o conceito de *Return-Oriented Programming*. A ideia de checar se o endereço anterior ao alvo de uma instrução `return` é uma instrução `call` tem sido descrita independentemente em vários trabalhos prévios. Carlini *et al.* [Carlini et al. 2015] fornecem uma visão geral completa da literatura relacionada. O fato de que Carlini *et al.* foram capazes de contornar esse tipo de defesa nos levou a conceber a *restrição do alvo executável de* `calls`, que descrevemos na seção 3.2. Nós escolhemos essa restrição, em particular, porque ela pode ser facilmente implementada a um baixo custo, uma vez que o mecanismo de validação do bit de execução (XD/NX) já existe.

As técnicas que utilizamos para filtrar falsos positivos, discutidas na seção 3.3, foram introduzidas por Xia *et al.* [Xia et al. 2012]. Nós adotamos as regras definidas nesse trabalho porque elas estão entre as filtragens de falsos positivos mais completas que conhecemos. Entretanto, se elas fossem utilizadas sem nossas camadas superiores, elas seriam custosas demais para serem práticas. Finalmente, o uso de caixas de areia, a abordagem descrita na seção 3.4, para proteger uma aplicação, quando todas as camadas superiores falham em certificar um desvio indireto, é muito conhecida na comunidade de sistemas. Nós defendemos o seu uso, apesar de seu alto custo computacional, porque, primeiro, os experimentos na seção 4.2 indicam que a última camada do nosso sistema de proteção é pouco provável de ser ativada durante a execução normal de um programa e, segundo, porque a caixa de areia já teve sua eficácia comprovada contra praticamente qualquer tipo de ataque de subversão do fluxo de execução [Egele et al. 2012].

Recentemente, a Intel anunciou o projeto de uma extensão de *hardware* que fornece recursos para defesa contra ataques de reúso de código, como ROP. *Control-Flow Enforcement Technology* (CET) [Intel 2016] irá prover uma implementação de pilha-sombra e um sistema de marcação dos destinos válidos de desvios indiretos `call` e `jmp`, cujo objetivo é garantir que apenas fluxos legítimos de execução das aplicações serão seguidos. Para isso, serão adicionados um bit à tabela de páginas, para proteção da pilha-sombra, além de novas atribuições a algumas instruções. Nossa solução, ao contrário, foi projetada para evitar adições dessa natureza.

6. Conclusão

Este artigo apresentou uma abordagem multicamadas para proteger programas contra ataques *Return-Oriented Programming* e demonstrou como esse sistema atua em aplicações de usuário final de dois grandes bancos brasileiros: Banco do Brasil e Bradesco. Cada camada do sistema proposto existe para validar os alvos de desvios indiretos, isto é, provar que esse alvo pertence ao fluxo de execução legítimo do programa. Nossa ideia chave é combinar camadas, de forma que a camada C_i é executada com custo computacional menor que a camada C_{i+1}. Um desvio certificado em C_i não precisa ser checado em C_{i+1}. Porque C_i, por construção, recebe mais desvios que C_{i+1}, aplicamos imposições de garantias mais fortes somente para casos que são difíceis de verificar.

Desenvolvimentos recentes mostraram que não existe um sistema capaz de deter todo ataque ROP [Carlini et al. 2015]. Este artigo não é uma exceção a tal regra. Nosso sistema pode eventualmente encontrar tanto falsos positivos quanto falsos negativos. Falsos positivos acontecem se marcarmos um alvo de desvio legítimo como inseguro, i.e., parte de um ataque. Nesse caso, recorremos ao uso da caixa de areia com a aplicação; assim, ficando sujeita a um alto *overhead* de tempo de execução, o que pode impactar a experiência do usuário. Os experimentos vistos na seção 4.2 indicam que essa possibilidade é muito improvável de acontecer, já que possuímos uma camada dedicada a evitá-la. Falsos negativos acontecem se permitirmos ao atacante subverter o fluxo de execução da aplicação protegida. Nesse caso, o atacante deve ser capaz de encontrar um conjunto de *gadgets* precedidos por instruções `call` que têm alvos em segmentos executáveis do programa. Essa restrição deixa um número muito pequeno de *gadgets* disponíveis para a construção de um ataque e seria capaz de parar ataques de estado da arte, como os realizados por Carlini *et al.* [Carlini et al. 2015]. A construção de ataques que contornam nossa proteção é, portanto, um problema aberto que esperamos explorar no futuro.

Referências

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2005). Control-flow integrity. In *CCS*, pages 340–353, New York, NY, USA. ACM.

Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M. (2008). Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE.

Banco Central do Brasil (2018). Bancos e financeiras com mais de quatro milhões de clientes. `https://www3.bcb.gov.br/ranking/`.

Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. (2015). Control-flow bending: On the effectiveness of control-flow integrity. In *SEC*, pages 161–176, Berkeley, CA, USA. USENIX Association.

Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. (2012). Unleashing mayhem on binary code. In *S&P*, pages 380–394, Washington, DC, USA. IEEE.

Checkoway, S., Feldman, A. J., Kantor, B., Halderman, J. A., Felten, E. W., and Shacham, H. (2009). Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. In *EVT/WOTE*, pages 6–6, Berkeley, CA, USA. USENIX Association.

Corelan Team (2011). mona.py - the manual. `https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/`.

Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):6.

Estes, A. C. (2017). Um novo tipo de malware está invadindo bancos de todo o mundo, inclusive do brasil. `http://gizmodo.uol.com.br/malware-invadindo-bancos-mundo/`.

F-Secure Response Labs (2013). Threat report. `https://www.f-secure.com/documents/996508/1030743/Threat\_Report\_H1\_2013.pdf`.

Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. (2014). Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 575–589. IEEE.

Intel (2016). Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`.

Kiriansky, V., Bruening, D., Amarasinghe, S. P., et al. (2002). Secure execution via program shepherding. In *USENIX Security Symposium*, volume 92, page 84.

Kleen, A. (2016). Advanced usage of last branch records. `https://lwn.net/Articles/680996/`.

Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., and Wang, X. (2009). Effective and efficient malware detection at the end host. In *SSYM*, pages 351–366, Berkeley, CA, USA. USENIX Association.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–, Washington, USA. IEEE.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, New York, NY, USA. ACM.

Schwartz, E. J., Avgerinos, T., and Brumley, D. (2011). Q: Exploit hardening made easy. In *SEC*, pages 25–25, Berkeley, CA, USA. USENIX Association.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, pages 552–561, New York, USA. ACM.

Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *CSS*, pages 298–307, New York, NY, USA. ACM.

Shi, Y. and Lee, G. (2007). Augmenting branch predictor to secure program execution. In *DSN*, pages 10–19, Washington, DC, USA. IEEE Computer Society.

Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). Sok: Eternal war in memory. In *S&P*, pages 48–62, Washington, DC, USA. IEEE Computer Society.

van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., and Giuffrdia, C. (2017). The dynamics of innocent flesh on the bone: Code reuse ten years later. In *CCS*, pages 1675–1689, New York, NY, USA. ACM.

Vendicator (2000). Stack shield: A stack smashing technique protection tool for linux. `http://www.angelfire.com/sk/stackshield/`.

Xia, Y., Liu, Y., Chen, H., and Zang, B. (2012). CFIMon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE.