Controlando a Frequência de Desvios Indiretos para Bloquear Ataques ROP

Mateus Tymburibá Ferreira, Ailton Santos Filho, Eduardo Feitosa

[1]IComp/UFAM, Manaus, Brasil

mateustymbu@gmail.com, ailton.santos07@gmail.com, efeitosa@icomp.ufam.edu.br

Abstract. *Because of its wide use in attacks against modern computing systems, protections against malicious codes based on the technique called Return-Oriented Programming (ROP) have been extensively studied. Nevertheless, it is not yet known a definitive solution. This article demonstrates that by controlling the frequency of indirect branch instructions it is possible to avoid the consolidation of ROP attacks. For this, we developed a prototype for Linux, Windows, OSX and Android environments. Experiments conducted with exploits confirmed the effectiveness of the proposed model at a comparable and, in some cases, lower computational cost than that achieved by related protections.*

Resumo. *Em função de seu vasto emprego em investidas contra sistemas computacionais modernos, proteções contra códigos maliciosos baseados na técnica denominada Return-Oriented Programming (ROP) têm sido extensamente estudadas. Apesar disso, ainda não se conhece uma solução definitiva. Este artigo demonstra que, através do controle da frequência de instruções de desvio indireto, é possível evitar a consolidação de ataques ROP. Para isso, foi desenvolvido um protótipo destinado a ambientes Linux, Windows, Android e OSX. Experimentos realizados com exploits confirmaram a eficácia do modelo proposto a um custo computacional comparável e, em alguns casos, inferior àquele alcançado por proteções correlatas.*

1. Introdução e Motivação

Nas últimas décadas, o software tornou-se o elo mais fraco da cadeia de componentes alvejados por atacantes em atos hostis contra sistemas computacionais [Hoglund and McGraw 2004]. Dentre as variadas formas de exploração para execução de código arbitrário, a técnica denominada *Return-Oriented Programming* (ROP) 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. Empregos da técnica ROP em ataques bem sucedidos podem ser vistos, por exemplo, nos *malwares* Stuxnet e Duqu [J. Callas 2011] e no código usado na violação de um tipo de urna de votação eletrônica empregada em diversas localidades [Checkoway et al. 2009].

Por ter se tornado uma das principais técnicas utilizadas por atacantes para desenvolver *exploits*[1], mitigações contra o ROP têm sido amplamente estudadas. Contudo, ainda não há uma solução definitiva. O Windows 8, por exemplo, agrega

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

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, pesquisadores apresentaram demonstrações de estratégias relativamente simples capazes de burlar essa defesa [D. Rosenberg 2011, N. H. Son 2011].

Diante desse contexto, este trabalho tem por objetivo apresentar uma nova estratégia para detecção e bloqueio de ataques ROP: o controle da frequência de instruções de desvio indireto. A eficácia dessa solução no bloqueio de *exploits* é demonstrada através da construção de um protótipo, testado com códigos maliciosos disponíveis no repositório público Exploit Database[2]. A análise de desempenho da solução também é avaliada e comparada com soluções correlatas.

As contribuições deste trabalho são duas: a elaboração e demonstração da eficácia do controle da frequência de desvios indiretos como estratégia para detecção de ataques ROP, incluindo suas variantes inexploradas pela maioria das soluções atuais; e o desenvolvimento de um protótipo de proteção contra ataques ROP destinado a ambientes Windows, Linux, OSX e Android em um *framework* de instrumentação binária dinâmica.

2. Return-Oriented Programming

Tão logo proteções de memória começaram a ser incorporadas aos sistemas computacionais (pilha não executável[3] e bit de execução[4], por exemplo), surgiram novas propostas de ataques alternativos baseados no reaproveitamento dos códigos originais das aplicações [S. Designer 1997].

Nos primeiros ataques de reúso de código, a biblioteca padrão de C (libc) foi o alvo dos desvios do fluxo de execução, os chamados ataques *return-into-libc* (RILC). Contudo, como 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, surgiram ideias para interligar funções [J. McDonald 1999, Wojtczuk 2001] ou trechos de código executáveis [T. Newsham 1997, S. Krahmer 2005]. Ao ser demonstrado que o encadeamento desses trechos de código permite a execução de computações arbitrárias (*Turing complete computation*) [Shacham 2007], essa técnica se popularizou entre atacantes e passou a ser referida por *Return-Oriented Programming* ou ROP.

Ao contrário da tradicional RILC, na qual o atacante desvia o fluxo de execução para o início de alguma função útil para o ataque, 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

[2]http://www.exploit-db.com/

[3]Pilha não executável (*non-executable stack* ou *nx-stack*) é um mecanismo que impede a inserção e execução de instruções (código malicioso) oriundas da área de pilha. Atualmente, esse tipo de proteção está presente por padrão na maioria dos sistemas operacionais.

[4]Bit de execução (NX/XD) é uma extensão natural do mecanismo de pilha não executável concebida para bloquear esse tipo de tentativa em outras áreas da memória. Essa estratégia baseia-se na utilização de um recurso incorporado aos processadores, em 2004, para marcar as páginas de memória com um bit de execução.

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 [Shacham 2007]. 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*) [Chen et al. 2011, Checkoway et al. 2010]. Apesar de não ter sido demonstrado que é possível executar computações arbitrárias (*Turing complete computations*) 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 [Checkoway et al. 2010].

A técnica de desenvolvimento de *exploits* ROP baseia-se no reúso de código para superar a proteção oferecida pelo bit de execução (NX/XD). Porém, a construção de *shellcodes*[5] inteiramente formados por *gadgets*, usando a técnica ROP, pode ser muito custosa e até inviável, dependendo da disponibilidade de *gadgets* na área de memória executável do processo atacado. Por isso, usualmente as cadeias de *gadgets* existentes nos *exploits* ROP limitam-se à função de preparar o ambiente para a posterior execução do *shellcode*. Esse encadeamento de códigos efetuado nos *malwares* 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.

Maiores explicações sobre o funcionamento de ataques ROP, incluindo exemplos, podem ser encontrados em [Ferreira et al. 2012].

3. Trabalhos Relacionados

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.

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 [Davi et al. 2009, Bania 2010]. 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*) [Yuan et al. 2011], ou da verificação da distância entre os endereços de retorno armazenados na área mais recentemente desocupada da pilha [Min et al. 2013, Jiang et al. 2011].

Apesar de eficaz no bloqueio aos ataques ROP identificados até o momento, da forma como vem sendo empregada, a estratégia de controle da frequência de

[5]Shellcode é um conjunto de instruções que, ao serem executadas pelo processador, efetuam alguma atividade maliciosa.

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.
- 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 como um registrador de uso geral, pois isso força a liberação dos espaços na pilha reservados para armazenar *frame pointers*.

4. Controle da Frequência de Desvios Indiretos

Diante da constatação de que 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, a solução proposta neste trabalho é focada no controle da frequência de instruções de desvio indireto com o intuito de detectar as três variantes desse tipo de ataque. Assim, 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 ou JMP indireto, o que possibilita evitar os três tipos de ataques ROP.

O esquema proposto consiste em checar se a contagem do número de instruções de desvio indireto 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 universal, com base na análise de um conjunto de aplicações, 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. 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.

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. A seguir, são analisados os três

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

cenários de aplicações autênticas cujas frequências de desvios indiretos teoricamente mais se aproximam da frequência tipicamente registrada durante ataques ROP.

4.1. 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 condicional possuem um endereço imediato, registrado na própria instrução. 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 Figura 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 [Universitet 2014].

```
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);
}
```

```
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
```

Figura 1. Código de laço mínimo

Ao analisar o código assembly, constata-se que a instrução JMP será executada a cada 6 instruções, frequência três vezes menor do que a média observada nas cadeias de *gadgets* presentes nos *exploits* ROP catalogados neste trabalho. 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.

4.2. Funções recursivas

Outra situação que pode gerar uma alta densidade de desvios indiretos é a frequente chamada a procedimentos. 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 Figura 2.

```
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);
}
```

```
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
```

Figura 2. Código de função recursiva mínima

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. 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 da Figura 2, constata-se que a instrução 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, existirão outras instruções dentro da função para torná-la útil. Portanto, constata-se que a execução sucessiva de instruções CALL no início de funções recursivas não acarreta em falsos positivos.

O mesmo não ocorre para funções com recursividade em cauda, uma vez que a frequência de desvios pode atingir um salto a cada 2 instruções executadas, caso a omissão do ponteiro de *frame* seja empregada, e entre 1 e 3 com o uso do ponteiro. Para evitar esse tipo de erro, pode-se utilizar um mecanismo para identificar a ocorrência de recursões em cauda. No entanto, uma vez que nenhuma ocorrência de falso positivo foi identificada durante os experimentos, esse mecanismo não está descrito neste artigo.

4.3. 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 Figura 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.

```
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
```

Figura 3. Código de laço de repetição com chamada interna de procedimento

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 duas vezes menor do que a média de desvios observada em cadeias de *gadgets*. Se a otimização de omissão do ponteiro de *frame* não for empregada, essa densidade cai para 2 desvios a cada 11 instruções.

Na prática, a frequência de desvios quase sempre é menor do que os piores casos apresentados. 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. Além disso, a ocorrência de desvios do tipo indireto é pouco comum, independente da situação específica do seu uso [Li et al. 2002]. 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.

5. Implementação

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. Em seguida, são detalhados os aspectos de implementação da estratégia proposta.

5.1. Pin

O Pin é um *framework* de instrumentação binária dinâmica do tipo JIT (*Just-in-time*), desenvolvido pela Intel para as arquiteturas IA-32 e x86-64, 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 [Luk et al. 2005]. 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.

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. Cada Pintool possui um mecanismo que decide onde e qual código deve ser inserido, denominado código de instrumentação, e um código a ser enxertado nos pontos de inserção, denominado código de análise [Intel 2014]. É importante ressaltar que o Pin, a Pintool e a aplicação não compartilham nenhuma biblioteca, o que evita qualquer tipo interação não desejada entre esses três binários.

O Pin foi escolhido como base para o protótipo por apresentar o melhor desempenho entre as aplicações de instrumentação binária dinâmica [Luk et al. 2005, Guha et al. 2007], além de fornecer uma API que facilita o acesso a informações de contexto, como o conteúdo de registradores ou o endereço de instruções.

5.2. Módulo de controle da frequência de desvios indiretos

A proteção elaborada requer a criação de uma estrutura de armazenamento, aqui designada 'janela', para registrar as últimas instruções executadas. Assumindo-se que a janela possui um tamanho N, pode-se dizer que sua função 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 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 (BBL) [Intel 2014]. Assim, insere-se um código de análise para um BBL, ao invés de avaliar cada instrução do programa, tornando a instrumentação mais eficiente.

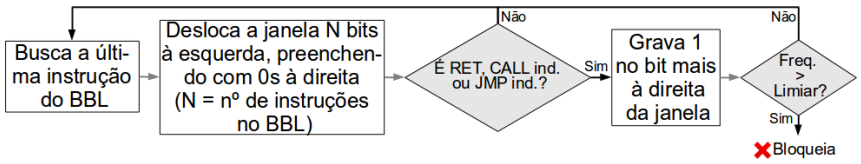

Figura 4. Lógica de controle das instruções de desvio indireto

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 protótipo 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. 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 protótipo.

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 protótipo 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 protótipo sinaliza a ocorrência de um ataque ROP e encerra a execução da aplicação.

6. Avaliação e Resultados Experimentais

Esta seção apresenta e discute os resultados dos experimentos realizados. Para tanto, inicialmente o ambiente de experimentação é detalhado. Em seguida, os resultados do processo de validação são mostrados, comprovando a viabilidade dessa solução. Também são discutidos os experimentos realizados para comprovar a eficácia do protótipo. Por fim, uma análise do desempenho do protótipo em comparação com soluções correlatas é apresentada.

6.1. Ambientes de experimentação

Os testes de desempenho foram executados em um computador com processador Pentium E5800 3.20GHz, Dual-Core (cache de 2048 KB), 6GB de memória, sistema operacional Linux Ubuntu 12.04 x86_64 kernel 3.8.0-38-generic, compilador GCC 4.6.3 e Pin versão 2.13 (kit 62728). Para a validação da estratégia de proteção foi usado um computador com processador Intel Xeon E5-2630 2.30GHz, Hexa-Core (cache de 15360 KB), 32 GB de memória, mesmo sistema operacional, compilador e versão do Pin.

6.2. 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 (a e b) e 6 (a e b) ilustram, respectivamente, os resultados obtidos nos experimentos com o Linux para janelas de 32, 96, 64 e 128 instruções. Note que na Figura 6, 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.

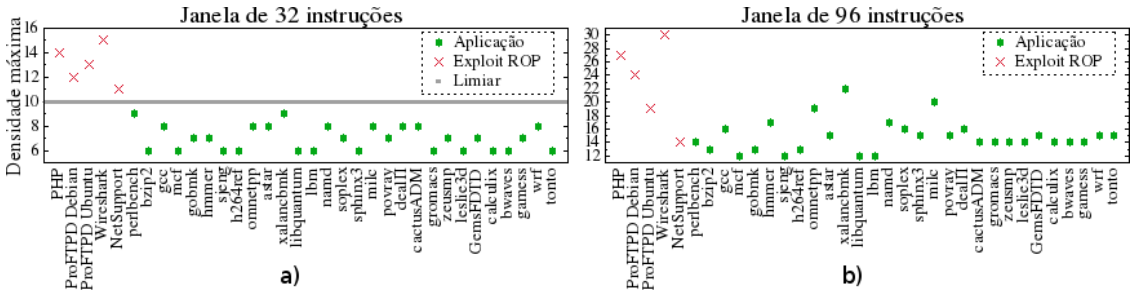

Figura 5. Frequência máxima de instruções de desvio indireto registrada com janelas de 32 e 96 instruções no Linux

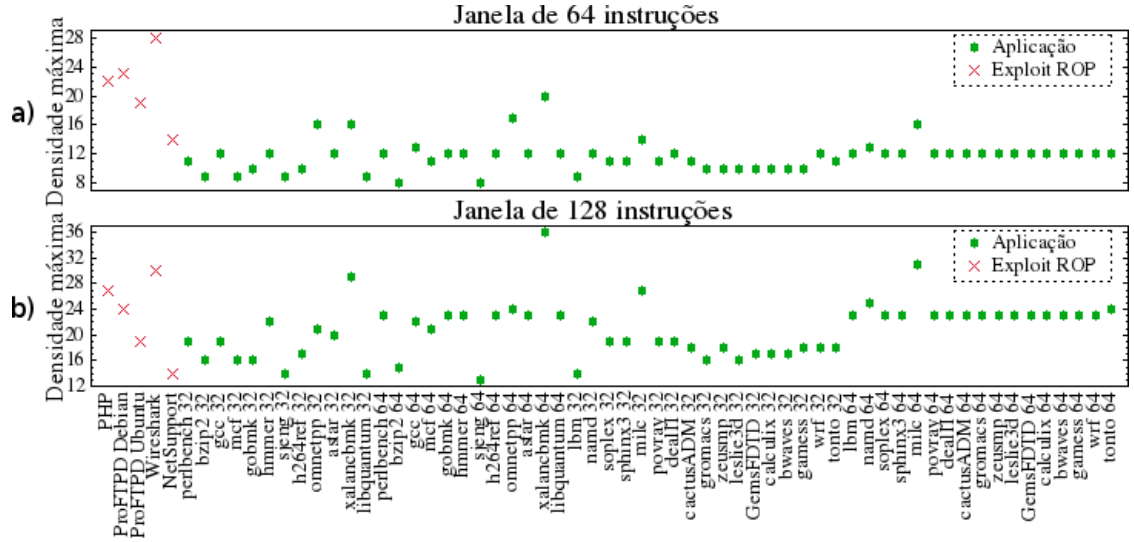

Figura 6. Frequência máxima de instruções de desvio indireto registrada com janelas de 64 e 128 instruções no Linux

Ao analisar as Figuras 5 e 6, é possível constatar que 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. Tanto que, no caso da janela de 128 instruções, cujos resultados são apresentados na Figura 6.b, não é possível distinguir os *exploits* das aplicações. Em contrapartida, à medida que o tamanho da janela diminui para 96 (Figura 5.b), 64 (Figura 6.a) ou 32 (Figura 5.a) 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 na Figura 5.a) que separa os dois padrões de frequência máxima de desvios indiretos, já que os valores registrados para *exploits* ROP variaram de 11 a 15, enquanto nos *benchmark* esses valores se espalharam entre 6 e 9. Portanto, pelos resultados obtidos, um limiar de 10 desvios indiretos é capaz de distinguir aplicações autênticas de *exploits* ROP, quando empregado 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.

A distribuição estatística das frequências máximas de desvios indiretos entre os *benchmarks* monitorados com a janela de 32 instruções mostra que 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 (10). Além disso, a frequência máxima de desvios indiretos mais comum é justamente a mais distante do limiar universal (6), abarcando 41,4% dos casos. As frequências 8 e 7 representam, respectivamente, 27,6% e 24,1% dos casos. 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.

6.3. Eficácia no bloqueio de *exploits* reais

O protótipo foi testado na proteção de 5 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. 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 protótipo foi capaz de detectar o ataque ROP e impedir a sua consolidação.

6.4. Desempenho

Os experimentos realizados para avaliar o desempenho do protótipo desenvolvido estão resumidos na Tabela 1. Nela, estão expressos os *overheads* impostos pelo Pin e pelo protótipo ao executar todos os *benchmarks* da suíte SPEC CPU2006 nas arquiteturas de 32 bits e de 64 bits. O Pin foi executado sem a adição de qualquer Pintool, com o intuito de registrar o custo computacional mínimo imposto pelo *framework*. Seu *overhead* médio atingiu 29%. Esse resultado confirma a expectativa de

um significativo custo computacional imposto por instrumentadores binários dinâmicos [Luk et al. 2005]. Da mesma forma, o fato do Pin acarretar em um *overhead* maior com os *benchmarks* inteiros confirma os resultados reportados pelos autores dessa ferramenta.

Tabela 1. *Overhead* médio ao executar *benchmarks*

	***Overhead* (%)**	
Benchmark	**Pin**	**Protótipo**
SPEC CPU2006 FP 32	10	1278
SPEC CPU2006 INT 32	56	801
SPEC CPU2006 FP 64	13	730
SPEC CPU2006 INT 64	37	756
Média Geral	**29**	**891**

Por outro lado, na arquitetura de 32 bits, o protótipo apresentou um *overhead* maior quando executou os *benchmarks* de ponto flutuante. Mesmo na arquitetura de 64 bits, os resultados obtidos para os testes inteiros e de ponto flutuante foram próximos. Esse distanciamento em relação ao padrão apresentado pelo Pin ocorreu porque o principal fator de influência no desempenho do protótipo é a quantidade de BBLs executados. Isso acontece porque a estratégia de controle da frequência de desvios indiretos exige que uma função de análise seja lançada ao executar qualquer BBL. Se observarmos que cada repetição de um laço corresponde a um BBL no Pin, fica fácil entender porque o desempenho do protótipo tende a ser pior com os *benchmarks* do tipo ponto flutuante, que executam inúmeras repetições de laços.

Outra constatação decorrente dos resultados reportados na Tabela 1 reside no fato de que para a arquitetura de 64 bits, tanto o Pin quanto o protótipo apresentaram um desempenho melhor. Isso ocorre em função das otimizações incorporadas ao processador utilizado nos experimentos, que beneficiam códigos de 64 bits.

A Tabela 2 exibe uma comparação do protótipo desenvolvido neste trabalho com outras proteções contra ataques ROP que utilizam instrumentadores binários dinâmicos para implementar soluções baseadas na estratégia de controle das instruções de retorno. Nessa tabela estão expressos os tipos de ataques bloqueados pelas proteções e o *overhead* médio reportado 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 protótipo desenvolvido apresenta um custo computacional comparável à solução DROP [Chen et al. 2009], 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 nosso protótipo (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 721% registrado pelo nosso protótipo. Nos testes com o gcc, o DROP impôs um *overhead* de 960%, enquanto o nosso protótipo elevou o tempo de CPU em 830%.

Outro fator de comparação entre as proteções recai sobre os tipos de ataques ROP bloqueados. Nesse caso, conforme indicado na Tabela 2, apenas a solução desenvolvida neste trabalho oferece uma proteção contra todos os tipos de *exploits*

Tabela 2. Comparação das proteções contra ataques ROP

Proteção	Ataques Bloqueados	*Overhead* (%)
DynIMA	R	Não informado na publicação
DROP	R	530,0
Protótipo	**R, J e C**	**891,0**

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

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.

7. Conclusões e Trabalhos Futuros

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 possibilita o bloqueio das demais variantes de ataques ROP, fato inédito entre as soluções que utilizam uma estratégia de controle da frequência de instruções. Finalmente, foi desenvolvido neste trabalho um protótipo que pode ser facilmente adotado em ambientes de produção que executem os sistemas operacionais Linux, Windows, Android ou OSX. A análise de desempenho do protótipo indicou que as contribuições são obtidas a um custo computacional comparável à de soluções correlatas, superando-as em alguns casos.

Entre os projetos futuros que podem dar prosseguimento a este trabalho, a avaliação do protótipo em outros ambientes já está em andamento. Além disso, está sendo elaborada uma solução para tratar eventuais casos de falsos positivos decorrentes de funções com recursividade em cauda. Outra possibilidade 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 elas, 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 descrita neste artigo.

Referências

Bania, P. (2010). Security mitigations for return-oriented programming attacks. *CoRR*, abs/1008.4099.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *Proceedings of the 17th ACM CCS*, pages 559–572. ACM.

Checkoway, S., Feldman, A. J., Kantor, B., Halderman, J. A., Felten, E. W., and Shacham, H. (2009). Can dres provide long-lasting security? In *Proceedings of the EVT/WOTE*, pages 6–6. USENIX Association.

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). Drop: Detecting return-oriented programming malicious code. In *Information Systems Security*, pages 163–177. Springer Berlin Heidelberg.

Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., and Yin, X. (2011). Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the ACM ASIACCS*, pages 20–29. ACM.

D. Rosenberg (2011). Defeating windows 8 rop mitigation. http://goo.gl/2Ae7aN.

Davi, L., Sadeghi, A.-R., and Winandy, M. (2009). Dynamic integrity measurement and attestation. In *Proceedings of the ACM STC*, pages 49–54. ACM.

Ferreira, M. T., Rocha, T., Martins, G., Feitosa, E., and Souto, E. (2012). Analise de vulnerabilidades em sistemas computacionais modernos: Conceitos, exploits e protecoes. In *Livro de Minicursos do XII SBSeg*, pages 2–51. SBC.

Guha, A., Hiser, J. D., Kumar, N., Yang, J., Zhao, M., Zhou, S., Childers, B. R., Davidson, J. W., Hazelwood, K., and Soffa, M. L. (2007). Virtual execution environments: Support and tools. In *NSF Next Generation Software Program Workshop*, Long Beach, CA.

Hoglund, G. and McGraw, G. (2004). *Exploiting Software: How to Break Code*. Pearson Higher Education.

Intel (2014). Pin 2.13 user guide. http://goo.gl/xvsW6l.

J. Callas (2011). Smelling a rat on duqu. http://goo.gl/FTM1Jn.

J. McDonald (1999). Defeating solaris/sparc non-executable stack protection. http://goo.gl/fglJTX.

Jiang, J., Jia, X., Feng, D., Zhang, S., and Liu, P. (2011). Hypercrop: A hypervisor-based countermeasure for return oriented programming. In *Proceedings of the 13th ICICS*, pages 360–373, Berlin, Heidelberg. Springer-Verlag.

Li, T., Bhargava, R., and John, L. K. (2002). Rehashable btb: An adaptive branch target buffer to improve the target predictability of java code. In *In The International Conference on High Performance Computing (HiPCP*.

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 *Proceedings of the ACM SIGPLAN PLDI*, pages 190–200. ACM.

Min, J.-W., Jung, S.-M., and Chung, T.-M. (2013). Detecting return oriented programming by examining positions of saved return addresses. In *Ubiquitous Information Technologies and Applications*, pages 791–798. Springer Netherlands.

N. H. Son (2011). Rop chain for windows 8. http://goo.gl/MAujbX.

S. Designer (1997). Getting around non-executable stack (and fix). http://goo.gl/XNEE7n.

S. Krahmer (2005). x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://goo.gl/5cN0Bm.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM CCS*, pages 552–561. ACM.

T. Newsham (1997). Re: Smashing the stack: prevention? http://goo.gl/fglJTX.

Universitet, S. (2014). Intel and att syntax. http://goo.gl/gkTvxL.

Wojtczuk, R. N. (2001). The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58).

Yuan, L., Xing, W., Chen, H., and Zang, B. (2011). Security breaches as pmu deviation: Detecting and identifying security attacks using performance counters. In *Proceedings of the Second APSys*, pages 6:1–6:5. ACM.

