Inferência Estática da Frequência Máxima de Instruções de Retorno para Detecção de Ataques ROP

Rubens Emílio, Mateus Tymburibá e Fernando Magno Quintão Pereira

[1]Universidade Federal de Minas Gerais

{rubens,mateustymbu,fernando}@dcc.ufmg.br

Abstract. *A program subject to a Return Oriented Programming (ROP) attack usually presents an execution trace with a high frequency of return instructions. From this observation, several research groups have proposed to monitor the density of returns to detect ROP attacks. These techniques use universal thresholds: the density of return operations that characterizes an attack is considered to be the same for every application. This paper shows that universal thresholds are easy to circumvent. As an alternative, we introduce a static code analysis that estimates the maximum density of return instructions possible for a program. This analysis determines detection thresholds for each application; thus, making it more difficult for hackers to compromise programs via ROPs.*

Resumo. *Programas que sofrem ataques baseados em Programação Orientada a Retorno (ROP) tendem a apresentar traços de execução com alta densidade de operações de retorno. A partir dessa observação, diversos pesquisadores propuseram formas de detectar ataques baseadas na monitoração da frequência de execução de instruções de retorno. Essas soluções utilizam limiares universais: a mesma densidade de retornos caracteriza ataques em diferentes aplicações. Este artigo mostra que limiares universais são fáceis de evadir. Como alternativa, apresenta-se um algoritmo que estima estaticamente a maior densidade de instruções de retorno possível durante a execução de um programa. Essa análise de código encontra limiares de detecção específicos para cada aplicação, dificultando assim, a realização de ataques baseados em ROP.*

1. Introdução

Ataques do tipo ROP (do inglês *Return-Oriented Programming*) estão entre os mais difíceis de detectar e prevenir. Investidas desse tipo ocorrem quando o invasor consegue encadear a execução de pequenos blocos de instrução, popularmente chamados *gadgets*, que terminam em uma instrução de retorno ou de desvio indireto [Shacham 2007]. Ataques ROP são efetivos porque eles são capazes de contornar mecanismos de proteção típicos de sistemas operacionais modernos, como $W \oplus X$ [PaX 2003a]. Testemunho desse sucesso é o fato de alguns *malware* famosos, como Stuxnet e Duqu [Callas 2011], serem baseados no modelo de ataque ROP.

Existe um grande esforço, tanto na academia quanto na indústria, para desenvolver mecanismos que protejam programas contra ataques do tipo ROP. Por exemplo, os primeiros três lugares do prêmio BlueHat 2012, patrocinado pela Microsoft Research Foundation, foram destinados a métodos de prevenção de ataques

 ©2015 SBC — Soc. Bras. de Computação

ROP[1]. Uma técnica de proteção bastante utilizada atualmente consiste em monitorar a frequência de desvios indiretos executados pelo programa [Chen et al. 2009, Davi et al. 2009, Han et al. 2013, Jiang et al. 2011, Pappas et al. 2013, Yuan et al. 2011, Cheng et al. 2014, Tymburibá et al. 2014]. Essa técnica baseia-se em um princípio muito simples: *gadgets* usadas em ataques ROP possuem poucas instruções. Portanto, uma alta densidade de desvios indiretos durante a execução de um processo é um forte indício de ataque. Assim, quando tal densidade ultrapassa um certo *limiar*, um ataque é reportado.

Em geral, as proteções que monitoram a frequência de instruções estabelecem um limiar único para qualquer aplicação a ser protegida. Esse fato abre brechas para que atacantes consigam contornar defesas baseadas na monitoração de frequência de desvios indiretos [Carlini and Wagner 2014, Göktas et al. 2014]. O principal mecanismo de evasão, nesse caso, consiste na utilização de *gadgets* mais longos. Encontrar tais *gadgets* não é fácil, porém é possível, conforme discutiremos na Seção 4. O desafio de estabelecer limiares específicos para a frequência de instruções executadas por aplicações, impulsionado pelas descobertas de Carline e Goktas, é a principal motivação deste trabalho, e nos leva a defender as seguintes teses:

> (i) A monitoração da frequência de instruções deve utilizar limiares de segurança baseados na natureza da aplicação, em vez de usar valores fixos.
> (ii) É possível utilizar técnicas de análise estática de código para encontrar a maior densidade de desvios indiretos (T_{rop}) que pode ser observada durante a execução de uma aplicação.
> (iii) O valor (T_{rop}) pode ser usado para melhorar a detecção de ataques ROP via mecanismos baseados em monitoração da frequência de desvios indiretos.

Com o intuito de comprovar essas teses, foi desenvolvido neste trabalho um algoritmo capaz de identificar estaticamente a frequência máxima de instruções de retorno executadas por uma aplicação, à partir da análise do seu código-fonte. A principal contribuição deste artigo é, então, suprir uma lacuna existente entre as técnicas de defesa contra ROP baseadas em monitoração de frequência de instruções: mesmo quando a proteção baseada na densidade de desvios oferece suporte à definição de limiares variáveis, não se conhece um mecanismo preciso e eficiente que encontre esses limiares. Este artigo descreve, na Seção 4, o primeiro algoritmo que estima, de forma totalmente estática, e com baixo custo computacional, a maior densidade de instruções de retorno possível para uma certa aplicação. Ressalta-se que o algoritmo descrito é prático: sua análise de complexidade, apresentada na Seção 4.4, demonstra que ele executa em tempo $O(IW)$, sendo I o número de instruções *assembly* no programa e W um parâmetro pré-definido que indica o tamanho da maior sequência de instruções em que desvios indiretos serão procurados.

O algoritmo elaborado neste trabalho foi implementado na infra-estrutura de compilação LLVM [Lattner and Adve 2004]. Essa implementação foi usada para estimar máximas densidades de instruções de retorno para todos os programas da coleção SPEC CPU 2006. Esses experimentos, descritos na Seção 5, indicam que nossa técnica é eficiente, consistente e precisa. Ela é eficiente porque mesmo para aplicações grandes, compostas por até 700 mil instruções *assembly*, os limiares são estimados em menos

[1] `http://www.microsoft.com/security/bluehatprize/`

 ©2015 SBC — Soc. Bras. de Computação

de 2.500 segundos. Uma abordagem dinâmica para resolver o mesmo problema gasta 3.600 segundos. O algoritmo proposto é consistente porque falsos negativos não foram observados em testes sobre traços com mais de sete trilhões de instruções assembly. E ele é preciso porque encontrou valores exatos para um conjunto de benchmarks pequeno o suficiente a ponto de possibilitar que resultados fossem verificados manualmente. A acurácia e o baixo custo computacional do algoritmo proposto nos permitem afirmar que essa técnica será uma aliada importante dos mecanismos de proteção contra ataques ROP baseados no controle da densidade de instruções.

2. Visão Geral

Existem diversas abordagens diferentes para monitorar a frequência de instruções de desvio indireto a fim de detectar ataques ROP. Neste trabalho, adotaremos o recente método da *Janela Deslizante* [Tymburibá et al. 2014]. Utilizaremos essa técnica porque ela é menos sucetível aos ataques descritos por Carlini [Carlini and Wagner 2014] e Goktas [Göktas et al. 2014]. Além disso, ela admite uma implementação em *hardware* de custo computacional zero, enquanto as demais proteções impõem um tempo de execução adicional médio que varia entre 1% e 530%, segundo os próprios autores. Ressaltamos que a técnica de estimativa da densidade máxima de instruções que descrevemos neste artigo não depende do mecanismo de monitoração adotado. Em outras palavras, mediante pequenos ajustes, a solução descrita na Seção 4 deste trabalho pode ser empregada sobre outras abordagens de monitoração da frequência de instruções.

A proteção baseada no controle da frequência de instruções de retorno através de uma janela deslizante estabelece que cada instrução executada pela aplicação deve ser monitorada. O monitoramento de cada instrução é efetuado pela marcação de valores 0 (zero) ou 1 (um) em uma janela de bits de tamanho fixo, que registra o tipo das últimas instruções executadas. O valor 1 (um) é escrito na posição corrente da janela sempre que uma instrução RET é executada. Para todas as outras instruções, o valor 0 (zero) é escrito. A cada atualização da janela, conta-se a quantidade de bits ligados (valor 1). Se essa contagem exceder um limiar preestabelecido, dispara-se um alerta de ataque ROP. A Figura 1 ilustra o funcionamento da janela deslizante para um exemplo de aplicação.

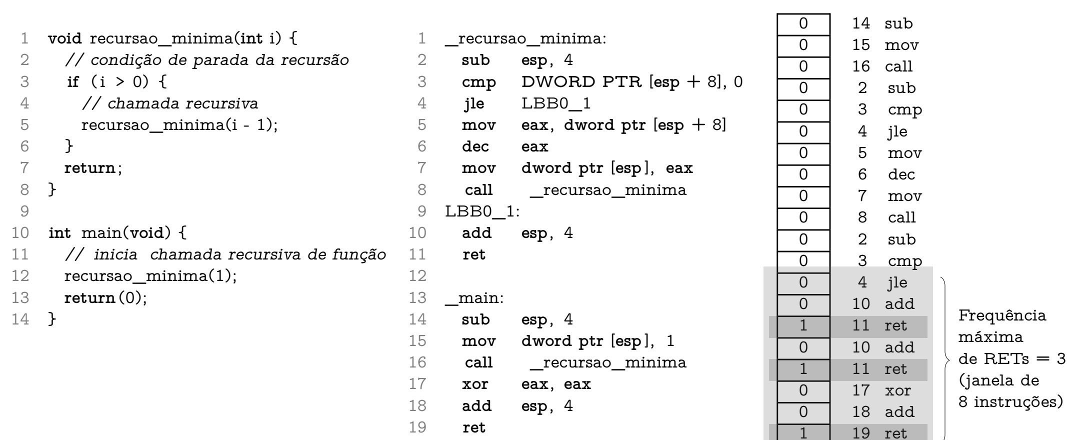

Figura 1. Funcionamento da janela que registra a frequência de RETs

O código-fonte exibido à esquerda na Figura 1, escrito na linguagem C, corresponde à chamada de uma função recursiva que executa apenas a checagem da condição

 ©2015 SBC — Soc. Bras. de Computação

de fim da recursão. Note que, nesse exemplo, a função recursiva executará apenas duas vezes, já que ela é chamada inicialmente com um parâmetro de valor 1 (linha 12). À direita do código-fonte, está indicado um código *assembly* equivalente. Cada instrução desse código *assembly* representa uma única instrução de máquina. A ordem em que essas instruções são executadas determina o estado da janela deslizante de instruções ao longo da execução do programa. Na Figura 1, estão ilustrados, à direita do código *assembly*, os valores a serem armazenados em cada posição da janela ao executar o programa. Ao lado de cada valor binário anotado na janela, está indicada a linha correspondente à instrução no código *assembly*, bem como o tipo da instrução. Note que o valor 1 (um) é anotado na janela somente quando uma instrução de retorno (RET) é executada. No exemplo representado na Figura 1, se considerarmos uma janela deslizante com capacidade para armazenar 8 (oito) instruções, a frequência máxima de RETs observada será 3 (três). Essa frequência máxima é atingida quando a janela deslizante alcança a última instrução do programa, conforme região sombreada na Figura 1.

Nesse cenário de proteção contra ataques ROP, é imprescindível estimar com exatidão a frequência máxima de RETs que pode ser atingida durante a execução de um programa. Na Seção 4 deste trabalho é apresentado um algoritmo capaz de fazer isso através da análise do código-fonte da aplicação e da simulação de todos os caminhos de execução que o programa pode percorrer. Dessa forma, pode-se estabelecer individualmente, para cada aplicação que se queira proteger, os respectivos limiares de densidade máxima de RETs.

3. Como Evadir Detecções Baseadas em um Limiar Universal

A primeira tese que este artigo defende, citada na Seção 1, baseia-se na premissa de que mecanismos de detecção de ataques ROP baseados em limiares universais são pouco seguros. Essa baixa segurança é a motivação para a principal contribuição deste trabalho: a identificação de limiares específicos por aplicação. O objetivo desta Seção é demonstrar essa baixa segurança. Para tanto, foram executados experimentos com ataques ROP reais.

A fim de burlar o monitor de densidade de desvios indiretos, decidiu-se usar a seguinte estratégia: interpor, entre os *gadgets* usados no ataque ROP original (deste momento em diante chamados *gadgets originais*), novos exemplares que possuem seis ou mais instruções. Esse número (seis instruções) é um valor normalmente usado na literatura para reconhecer um ataque ROP, conforme descrito por Cheng *et al.* [Cheng et al. 2014]. Chamaremos tais *gadgets* de *inócuos*, e diremos que um *gadget* é inócuo se: (i) ele consiste em uma sequência de instruções que termina com uma operação de retorno; (ii) ele não modifica os registradores de uso geral da arquitetura x86; (iii) ele não causa violações de acesso à memória, seja via acesso inválido, seja via uso de instrução privilegiada; e (iv) ele não escreve sobre a pilha de execução do programa.

Para encontrar esses *gadgets inócuos*, nós utilizamos a ferramenta Mona [Eeckhoutte 2014], um buscador automático de *gadgets*. Restringimos nossa busca a códigos que não pertencem ao sistema operacional e cujo endereço de carga não é randomizável pela proteção ASLR [2], para facilitar a construção do mecanismo de evasão. Ao analisar 10 casos de ataque ROP disponíveis em um banco de dados público de códigos maliciosos (`www.exploit-db.com/`), encontramos 3075 *gadgets inócuos*.

[2] Para maiores informações sobre *Address Space Layout Randomization*, see [PaX 2003b]

 ©2015 SBC — Soc. Bras. de Computação

Aplicação	Núm.	Máx.	Aplicação	Núm.	Máx.
AoA Audio Extractor	21	15	D.R. Software Audio Converter	312	52
Firefox	27	7	Zinf Audio Player	461	17
Free CD to MP3 Converter	52	10	DVD X Player	776	52
PHP	106	52	ASX to MP3 Conv.	1.161	44
Wireshark	159	9			

Figura 2. (Núm) Número de *gadgets inócuos* encontrados no espaço de endereços executáveis das aplicações. (Máx) Tamanho do maior *gadget inócuo*.

A figura 2 relata o resultado dessa investigação. O menor número de *gadgets inócuos*, 21, foi visto em `AoA Audio Extractor`. O maior número, 1.161, foi encontrado em `ASX to MP3 Converter`. Para verificar quais *gadgets inócuos* funcionam na prática, foram testados todos os candidatos em `AoA Audio Extractor` e em `Free CD to MP3 Converter`. Essas aplicações foram escolhidas por pragmatismo: um dos autores é familiar com ambos os códigos. As demais aplicações não foram testadas porque esse processo é lento e tedioso. Para se ter uma medida, testar os 21 + 52 *gadgets inócuos* consumiu cerca de 70 horas de trabalho. 6 *gadgets inócuos* foram bem sucedidos em ataques contra `Free CD to MP3 Converter`. Nesses ataques, os *gadgets inócuos* foram interpostos entre cada par de *gadgets úteis*. Essa técnica reduziu a densidade de desvios indiretos a um patamar capaz de iludir as proteções que estabelecem limiares universais, contudo sem comprometer a eficácia dos ataques. Dessa investigação, conclui-se que mecanismos de detecção de ataques ROP que disparam exceções ao observar instruções de retorno separadas por cinco ou menos operações [Chen et al. 2009, Davi et al. 2009, Pappas et al. 2013] podem ser contornados com relativa facilidade.

4. RopDeducer

Esta seção descreve RopDeducer, um algoritmo que estima – estaticamente – o valor δ-RET, o qual definimos abaixo:

Definição 1. *Dado um programa P, e um inteiro K, define-se δ-RET como um inteiro* nRETs*, que representa o maior número de instruções de retorno que pode ser observado em uma sequência de K instruções durante a execução de P.*

Descobrir δ-RET estaticamente é um problema indecidível; portante, utilizaremos uma heurística para estimar esse valor. Considere o programa de exemplo apresentado na figura 1(Esquerda). A sequência de instruções emitidas durante a execução do programa é exibida na figura 1(Direita). Nossa abordagem para estimar a densidade máxima de instruções de retorno (RETs) trabalha com uma janela deslizante de tamanho parametrizável (o inteiro K na definição 1). No exemplo, escolhemos $k = 8$ para o tamanho da janela de instruções, e a posicionamos sobre a região do registro de instruções em que ocorre o maior número de RETs. Para o exemplo, o valor de δ-RET é de 3 RETs em 8 instruções. Sem perda de generalidade, devemos avaliar todas as sequências de instruções possíveis sobre o programa, para então determinar aquela de densidade máxima.

O algoritmo RopDeducer determina o valor δ-RET de uma aplicação em dois passos. Primeiro, analisamos cada função f do programa separadamente, estimando valores locais δ-RET$_f$. Para tanto, usa-se uma versão modificada do grafo de fluxo de controle

(CFG) da função, denominada δ-CFG. A partir do δ-CFG de cada função, encontramos a sequência de instruções que produz o maior número de RETs, estimando assim a densidade máxima de RETs da função, ou δ-RET$_f$. No segundo passo, agregamos a estimativa de função em um valor único para toda a aplicação.

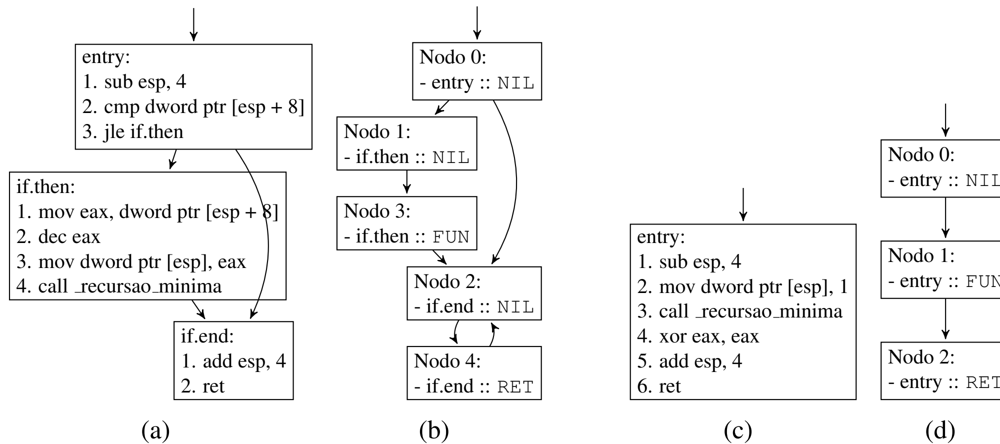

Figura 3. Grafos de fluxo de controle (a, c) e δ-CFGs (b, d) para funções 'recursao_minima(int)' e 'main', respectivamente. Os CFGs são obtidos diretamente do código fonte, e são usados para construir os δ-CFGs, como mostra o algoritmo 1.

4.1. δ-CFG: Representação Intermediária de Funções

Funções podem ser descritas como grafos de fluxo de controle, ou CFGs. Esta abstração é bastante utilizada na área de análise e otimização de código, por servir de base para diversos algoritmos. CFGs são criados por compiladores, durante o processo de otimização de código. Os nós de um CFG são denominados blocos básicos, e são sequências de instruções delimitadas por uma instrução de desvio (condicionais e instruções de retorno). A figura 3(a) mostra um exemplo de grafo de fluxo de controle para a função 'recursao_minima'. A estrutura do CFG é importante, pois nos permite analisar todos os fluxos de controle possíveis em uma aplicação sem executá-la. Uma vez que o CFG é um conceito padrão em compilação [Aho et al. 2006, Cap. 9], não explicaremos como derivá-lo a partir do código fonte de um programa. A partir do CFG, nós construímos uma nova estrutura de dados, denominada δ-CFGs. O algoritmo 1 mostra como construir δ-CFGs a partir de CFGs. As figuras 3(b) e 3(d) mostram os δ-CFGs obtidos a partir dos CFGs vistos nas figuras 3(a) e 3(c).

O δ-CFGs possui três tipos de nós: `NIL`, `FUN` e `RET`. Nós `FUN` indicam chamadas de função e nós `RET` indicam operação de retorno. Nós `NIL` marcam o início de blocos básicos, e garantem equivalência entre o fluxo de execução do δ-CFG e do CFG de origem. A cada aresta do grafo é associado um peso, que indica a quantidade de instruções entre dois nós do δ-CFG. Assim, ir de um nó u a um vizinho v corresponde a emitir, em ordem: a instrução representada por u; as instruções da aresta (u, v); e a instrução do nó v. Dada a tipagem dos nós e a adição de peso às arestas, transformamos o problema de determinar o valor δ-RET$_f$ em um problema de grafos. Desejamos encontrar o caminho no δ-CFG de uma função com o maior número de nós de tipo `RET` ou `FUN`. Mas assim como CFGs, δ-CFGs podem conter loops, e determinar o valor δ-RET$_f$ de

 ©2015 SBC — Soc. Bras. de Computação

Algoritmo 1: Construção de δ-CFG $G(V, E)$ a partir de CFG

```
1  u ← cria raíz do δ-CFG G(V, E) com nó de tipo NIL;
2  para cada bloco básico bb do CFG faça
3      para cada instrução inst de bb faça
4          if inst é chamada de função then
5              v ← cria nó de tipo FUN para instrução inst;
6              E ← E ∪ {(u, v, u_peso)}; u ← v;
7          else if inst é instrução de retorno then
8              v ← cria nó de tipo RET para instrução inst;
9              E ← E ∪ {(u, v, u_peso)}; u ← v;
10         else incrementa custo de saída u_peso;
11     para cada bloco básico bb-suc sucessor de bb faça
12         v ← inicializa nó NIL para bloco básico bb-suc;
13         E ← E ∪ {(u, v, u_peso)};
14 V_RET ← {v ∈ V | v é do tipo RET};
15 V_rec ← {v ∈ V | v representa chamada recursiva};
16 E ← E ∪ {(v, u', u_peso) | v ∈ V_RET ∧ u ∈ V_rec ∧ (u, u', u_peso) ∈ E};
17 return G(V, E);
```

uma função equivale a encontrar o caminho máximo em um grafo, que é um problema NP-difícil [Schrijver 2003].

4.2. δ-CFG e Estimativas δ-RET$_f$

Para mitigar o caráter não polinomial do problema, utilizamos o conceito de janela deslizante: buscamos o caminho de maior número de nós RET ou FUN no δ-CFG, mas cuja quantidade de instruções emitidas não exceda k (tamanho da janela deslizante). O tamanho da janela indica o número máximo de instruções a serem avaliadas no cálculo da densidade δ-RET$_f$. Seja n o número de nós do δ-CFG de uma função. A complexidade de tempo de execução da nossa abordagem é $\Omega(n^k)$, pois podemos escolher até n nós a cada instrução que adicionamos à janela. Contudo, na seção 5, mostramos que RopDeducer é eficiente e escalável na prática, pois em geral grafos δ-CFGtendem a ser esparsos.

Deseja-se maximizar o número de RETs dentro de uma janela de até k instruções. Sabe-se que fluxos de controle válidos em uma função são caminhos sobre o seu δ-CFG. Assim, procura-se um caminho de até k instruções no δ-CFG, que possua a maior quantidade de instruções de retorno. Dada uma função arbitrária f e seu δ-CFG $G(V, E)$, de vértices V e arestas E, apresentam-se algumas definições abaixo:

Definição 2. *Um caminho C do grafo é uma sequência de arestas* $\{e_1 e_2 \dots e_n \mid e_i(u, v), e_{i+1}(v, w) \in E, \forall\, i < n\}$.

Definição 3. *O número C_r de instruções de retorno de um caminho C corresponde à quantidade de nós* RET *e* FUN *em C.*

Definição 4. *O total de instruções de um caminho C é dado por* $C_t = C_r + \sum_{e \in C} e_{peso}$.

A densidade de RETs de um caminho C no δ-CFG, depende da quantidade C_r de RETs que o caminho contém, mas também do total C_t de instruções de C. O valor de C_t envolve os pesos das arestas e o número C_r de nós do tipo RET e FUN em C. C_t

não é acrescido da quantidade de nós de tipo `NIL`, pois esses não representam qualquer instrução – apenas marcam o início de blocos básicos. Precisamos agora determinar qual o caminho de maior C_r em todo o grafo que respeite a restrição da janela deslizante: o máximo de instruções percorridas deve ser k.

Definição 5. *Considere o δ-CFG $G(V, E)$ de uma função, e uma constante k correspondente ao tamanho da janela deslizante. Considere também o conjunto $\mathbb{C}_k$ de caminhos de G, tal que $C_t \leq k, \forall\ C \in \mathbb{C}_k$. A densidade máxima de RETs da função, denominada δ-RET$_f$, é dada por $\max_{C \in \mathbb{C}_k} C_r$.*

Instruções de retorno ocorrem em dois tipos de nó: nós do tipo `RET` e `FUN`. No primeiro caso, conhecemos a instrução de retorno que deu origem ao nó `RET`, e tal instrução contribui em 1 para a contagem de RETs, e em apenas 1 para o total de instruções. Já no segundo caso, sabemos que uma função será executada, e que ao menos um RET será emitido (para retorno da função). Mas sabemos também que ao menos duas instruções serão executadas ao todo, devido à instrução `pop`, que antecede retornos de função. Por isso, para nós do tipo `FUN`, estimamos o valor inicial de um RET sobre duas instruções.

Apesar da estimativa inicial para nós do tipo `FUN` ser válida, no sentido de não subestimarmos a densidade de RETs, uma função pode emitir várias instruções antes de um retorno; e tais instruções deveriam ser contabilizadas durante o cálculo da densidade de RETs. No entanto, ao determinar a densidade de RETs de uma função f, precisaríamos da densidade de todas as funções chamadas por f. Simplificamos o cálculo das densidades δ-RET$_f$ dividindo a tarefa em dois estágios: primeiro geramos estimativas por função, e em seguida agregamos o resultado em um único valor δ-RET para toda a aplicação. Essa estratégia resolve o problema de não conhecermos as densidades de todas as funções ao mesmo tempo e é segura: ela pode apenas aumentar a densidade final δ-RET. O algoritmo 2 mostra como estimar o valor de δ-RET$_f$ de uma função.

Algoritmo 2: δRET$_f$(δ-CFG *raiz*, *nRETs*, *nInsts*) $\rightarrow \langle$*nRETs*, *nInsts*$\rangle$

1 **se** *raiz* é do tipo `RET` **então** *nRETs*', *nInsts*' $\leftarrow$ *nRETs* + 1, *nInsts* + 1;
2 **se** *raiz* é do tipo `FUN` **então** *nRETs*', *nInsts*' $\leftarrow$ *nRETs* + 1, *nInsts* + 2;
3 **se** *nInsts*' $>$ *tamJanela* **então retorna** $\langle$*nRETs*, *nInsts*$\rangle$;
4 **para cada** *nó* filho de *raiz* **faça**
5     $\langle$*nRETs*', *nInsts*'$\rangle \leftarrow \delta$`RET`$_f$ (*nó*, *nRETs*, *nInsts*) ;
6   **se** (*nRETs*' $>$ *nRETs*) ou (*nRETs*' $=$ *nRETs* e *nInsts*' $<$ *nInsts*) **então**
7         $\langle$*nRETs*, *nInsts*$\rangle \leftarrow$ *nRETs*', *nInsts*';
8 **retorna** $\langle$*nRETs*, *nInsts*$\rangle$

4.3. Grafo de Chamadas de Função e Estimativa δ-RET

Ao final das estimativas geradas por função, precisamos agregar os valores obtidos, e determinar o valor de δ-RET para a aplicação como um todo. Para tanto, utilizamos o grafo de chamadas de função do programa. Neste grafo, nós são funções invocadas durante a aplicação, e arestas existem de um nó u para outro v sempre que a função representada por u chama a função do nó v. A partir de um caminhamento em profundidade sobre este grafo, determinamos uma ordem para agregar as estimativas δ-RET$_f$ em um único valor, resolvendo assim eventuais interdependências (*loops*) sobre o grafo de chamadas.

 ©2015 SBC — Soc. Bras. de Computação

Algoritmo 3: δRET(GrafoChamadas *raiz*, Tabela δRET_f) $\rightarrow \langle nRETs, nInsts \rangle$

1 **se** *raiz* não foi visitada **então retorna** $\delta\texttt{RET}_f$ [*raiz*];
2 marca *raiz* como visitada;
3 *estimativas* $\leftarrow \{\}$;
4 **para cada** *nó* filho de *raiz* **faça**
5   *estimativas* $\leftarrow$ *estimativas* $\cup$ $\{\delta\texttt{RET}(\textit{nó}, \delta\texttt{RET}_f)\}$;
6 $\delta\texttt{RET}_f$ [*raiz*] $\leftarrow$ melhor combinação entre $\delta\texttt{RET}_f$ [*raiz*] e *estimativas*;
7 **retorna** $\delta\texttt{RET}_f$ [*raiz*];

Utilizando os valores δ-RET$_f$ calculados para cada função, a densidade máxima δ-RET de instruções de retorno de uma aplicação é calculada de acordo com o algoritmo 3. Como apenas percorremos em profundidade o grafo de chamadas de função do programa, a complexidade de tempo de execução de gerar a estimativa global δ-RET é linear sobre o tamanho do grafo de chamadas. É importante ressaltar que apenas combinações válidas são geradas durante a escolha pela maior densidade de retornos de uma função (linha 6 do algoritmo 3), ou seja, todas as estimativas respeitam a restrição do tamanho da janela deslizante.

4.4. Corretude

Dado um programa P, cada caminho que o Algoritmo 2 explora é um caminho possível sobre o CFG de P. Caso P não fizesse chamadas de função, seria trivial provar a corretude do Algoritmo 2, pois sua resposta é o menor caminho do início de P até uma instrução de retorno. Havendo chamadas de função, o Algoritmo 2 primeiro aproxima cada uma delas como um par $\langle 1, 2 \rangle$, ou seja, uma sequência contendo uma instrução de retorno a cada duas instruções. Essa estimativa pode ser alargada pelo Algoritmo 3. Um argumento indutivo sobre esse algoritmo provê intuição sobre a corretude dessa última rotina: Ao analisar uma função f, assumindo que o Algoritmo 3 estima conservadoramente o fator δ-RET para cada função f' que f invoca, então a estimativa para f também é conservadora, pois ela assume sempre a mais conservadora estimativa para qualquer f'.

Todos os algoritmos apresentados neste artigo terminam. A construção do grafo δ-CFGs (Algoritmo 1) termina após cada nó do CFG de um programa ter sido visitado. O algoritmo 2 termina após uma sequência de no máximo k nós ter sido visitada a partir de todo nó de um δ-CFGs. Finalmente, o Algoritmo 3 termina, novamente segundo um argumento indutivo: assumindo que ele termina para cada função f' chamada a partir de uma função f, então ele termina para f. Note que a guarda na linha 1 garante que uma função não pode ser visitada duas vezes.

5. Avaliação Experimental

A fim de validar a precisão do RopDeducer, as estimativas δ-RET obtidas através do algoritmo introduzido neste artigo foram comparadas a valores observados durante a execução das aplicações. A análise estática, efetuada pelo RopDeducer, foi agregada à infraestrutura de compilação LLVM [Lattner and Adve 2004]. Esse *framework* possui uma vasta quantidade de APIs que facilitam a manipulação do CFG de aplicações. A análise dinâmica, por outro lado, foi implementada com o auxílio do instrumentador binário dinâmico denominado Pin. Essa ferramenta permite a instrumentação de

 ©2015 SBC — Soc. Bras. de Computação

aplicações durante a execução. O Pin provê APIs que permitem o rastreamento de cada instrução executada por um programa. Explorando essa funcionalidade, criou-se um código de instrumentação que implementa uma janela deslizante para registrar a densidade máxima de RETs executados por uma aplicação. Em nossos experimentos, utilizou-se uma janela de tamanho 32, uma vez que já foi demonstrado que esse valor permite detectar a alta frequência de instruções de desvio indireto inerente a ataques ROP [Tymburibá et al. 2014]. Também foram conduzidos experimentos com o intuito de avaliar os tempos de execução alcançados pelo RopDeducer.

Tanto nos experimentos dedicados à análise de precisão quanto nos testes voltados à verificação do desempenho do RopDeducer, foram utilizados os *benchmarks* da suíte SPEC CPU2006 compiláveis através do LLVM. Ou seja, todos os códigos dessa suíte que estão implementados nas linguagens de programação C e/ou C++. O equipamento utilizado nos experimentos é dotado de um processador Intel Xeon E5-2620 2.00GHz, Hexa-Core (cache de 15360 KB), 16GB de memória, sistema operacional Linux Ubuntu 12.04 x86_64 kernel 3.2.0-76-generic, LLVM 3.4.2 e Pin 2.13 (kit 62728).

5.1. Análise Estática vs. Análise Dinâmica

A Figura 4 compara as estimativas obtidas pelo RopDeducer com os valores de densidade máxima de RETs observados durante a execução dos *benchmarks*. Nota-se que, para todos os programas, os valores estimados estaticamente são superiores àqueles observados dinamicamente. Esses resultados confirmam a hipótese de que a execução de aplicações pode não explorar todos os possíveis caminhos do fluxo de execução, dependendo da abrangência da massa de testes utilizada. Em nossos experimentos, utilizamos os dados de entrada do tipo *reference* (referência) da suíte SPEC, que corresponde à maior massa de dados disponível. Ainda assim, todos os valores obtidos estaticamente pelo RopDeducer superaram os respectivos dados registrados dinamicamente através do Pin.

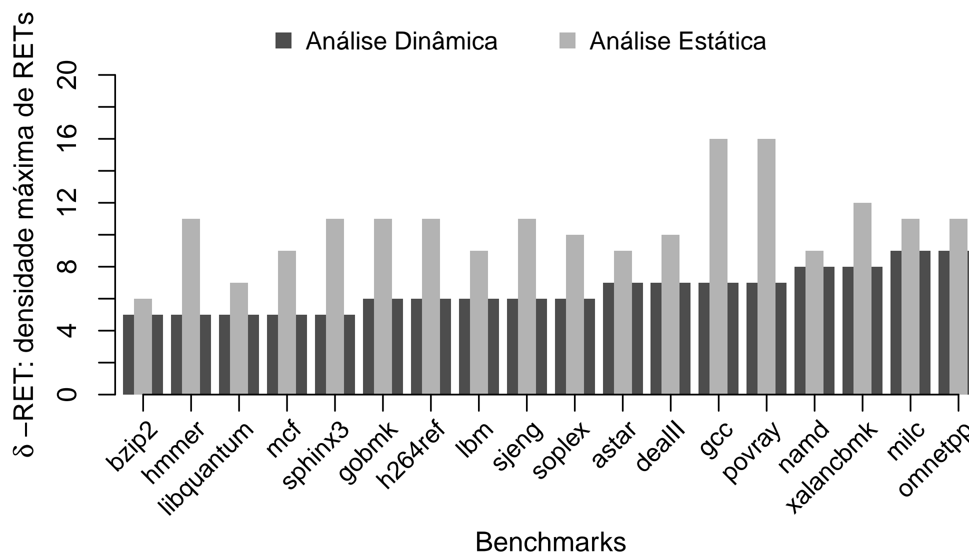

Figura 4. Densidade máxima de RETs estimada estaticamente pelo RopDeducer e registrada dinamicamente com o Pin

O fato de a análise dinâmica representar um limite inferior para os possíveis valores de frequência máxima de RETs reforça as contribuições oferecidas pelo RopDe-

 ©2015 SBC — Soc. Bras. de Computação

ducer. A incapacidade de explorar todos os caminhos do fluxo de execução, observada na análise dinâmica, pode acarretar na ocorrência de falsos positivos. Se o trecho inexplorado do CFG corresponder ao ponto de frequência máxima de RETs, a aplicação será classificada indevidamente como um ataque. Por outro lado, ao garantir a análise completa do código da aplicação, o RopDeducer evita esse tipo de ocorrência.

Outra característica que se destaca na Figura 4 é o elevado δ-RET apresentado pelo *benchmark* "povray". Esse valor decorre da existência de uma recursão caudal no código dessa aplicação, semelhante ao padrão apresentado na Figura 1. Note que, nesses casos, a densidade máxima de RETs será de 16 em uma janela de 32 instruções, em função da execução alternada das instruções "add" e "ret" (linhas 10 e 11 do código *assembly*). A fim de reforçar a exatidão das estimativas obtidas pelo RopDeducer, foram codificados 3 *microbenchmarks* para os quais o δ-RET é conhecido. Um desses *microbenchmarks* equivale ao código ilustrado na Figura 1. O código-fonte dos demais *microbenchmarks*, onde constam também as frequências máximas de RETs esperadas, está disponível para download [3]. Em todos os casos, o RopDeducer identificou corretamente os valores do δ-RET.

5.2. Desempenho

Além de possibilitar uma análise mais precisa do que aquela oferecida por mecanismos dinâmicos, o RopDeducer consegue aliar essa vantagem a um tempo de execução consideravelmente menor. Quando executado sobre os programas de SPEC CPU 2006, RopDeducer apresentou um tempo de execução menor em todos os experimentos. O tempo médio de execução registrado para o RopDeducer foi de 987 segundos. Em contrapartida, o tempo médio de execução alcançado pela análise dinâmica atingiu 2797 segundos. Essa discrepância existe porque, durante a instrumentação dinâmica do código, 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 isso ocorre durante a execução da aplicação instrumentada, o que impacta significativamente no custo computacional da solução. De fato, estudos apontam que, para efetuar uma tarefa simples de contagem do número de blocos de instruções executadas, ferramentas diversas de instrumentação dinâmica acarretam em *overheads* que variam de 230% a 750% [Luk et al. 2005, Guha et al. 2007].

O algoritmo proposto neste artigo possui um pior caso exponential, conforme discutido na seção 4. Contudo, na prática, esse algoritmo mostra forte tendência a comportamento linear. Para fundamentar essa afirmação, a figura 5 compara o tempo de execução do algoritmo com o número de instruções em cada programa de SPEC CPU 2006. O coeficiente de determinação para esse conjunto de amostras é 0.98. Quanto mais próximo de 1.0, mais linear é o comportamento do algoritmo.

6. Trabalhos Relacionados

Existem várias estratégias de detecção de ataques ROP baseadas na frequência de execução de instruções de retorno ou desvios indiretos. Por outro lado, até o presente momento, não existe uma forma de determinar um limiar de frequência seguro por aplicação. A determinação de limiares específicos por aplicação é uma contribuição original deste

[3] `http://removido.devido.revisao.cega`

 ©2015 SBC — Soc. Bras. de Computação

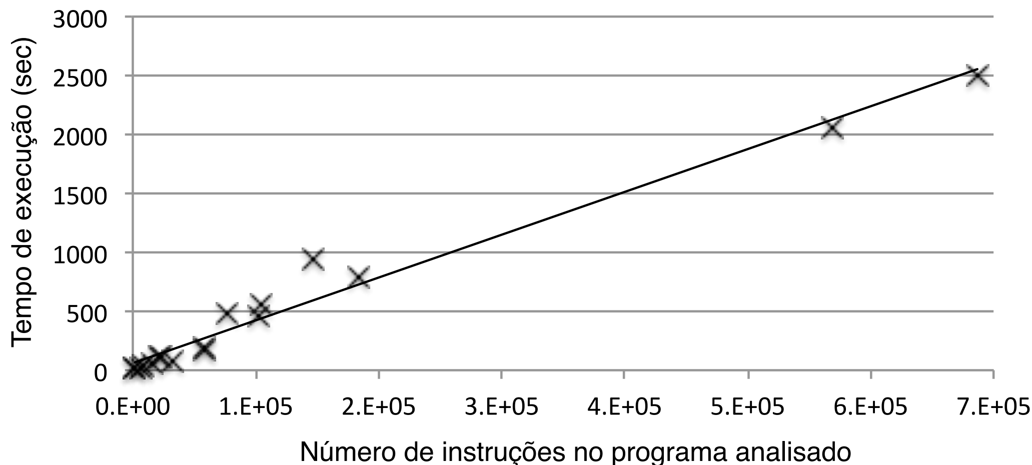

Figura 5. Relação entre tamanho e tempo de execução. O coeficiente de determinação (R^2) foi 0.98, indicando forte tendência linear. Cada ponto representa um benchmark de SPEC CPU 2006. Benchmarks estão ordenados por quantidade de instruções.

trabalho, pois todas as técnicas já descritas na literatura usam limiares universais. Nesta seção iremos rever tais técnicas, dedicando especial atenção aos trabalhos de Chen *et al.* [Chen et al. 2009], Davi *et al.* [Davi et al. 2009], Pappas *et al.* [Pappas et al. 2013], e Cheng *et al.* [Cheng et al. 2014]. Todos esses trabalhos compartilham um *modus operandi* comum: eles monitoram a sequência de instruções produzida durante a execução de um programa, e caso detectem uma densidade alta de operações de retorno, disparam uma exceção. A título de exemplo, Chen *et al.* levanta uma exceção se forem observadas sequências de três instruções do tipo RET separadas por até cinco outras operações. Por não utilizar limiares específicos, essa abordagem tende a gerar mais falsos positivos: Chen *et al.* detectaram um aviso falso em uma coleção relativamente pequena de testes. Davi *et al.* discutem outras quatro situações que podem levar a técnica de Chen *et al.* a emitir falsos positivos.

Pappas *et al* [Pappas et al. 2013] e Cheng *et al.* [Cheng et al. 2014] usam os registradores LBR (*Last Branch Record*) para detectar cadeias de gadgets ROP. Esses registradores foram inicialmente concebidos para aumentar a precisão de técnicas de predição de *branches*. Eles estão presentes em processadores Intel Core 2, Xeon e Atom. Alguns processadores ARM possuem lógica similar. Tanto no trabalho de Pappas quanto no trabalho de Cheng, a infra-estrutura LBR é usada para contar a distância entre terminadores de funções. Essa contagem acontece antes que o sistema operacional permita a invocação de alguma função que faça parte de seu núcleo. Novamente, frequências altas de terminadores são vistos como indicadores de ataques ROP. Carline *et al.* [Carlini and Wagner 2014] e Goktas *et al.* [Göktas et al. 2014] demonstraram que é possível contornar esse tipo de defesa. Ambos os grupos usaram, para tanto, gadgets longas interpostas entre grupos de 10 gadgets pequenas. A técnica de evasão que utilizamos na seção 3 deste artigo é mais efetiva: interpomos gadgets inócuas entre cada gadget válida, em vez de entre grupos de 10 gadgets válidas. Acreditamos que o uso de limiares específicos por aplicação é uma forma de dificultar tanto a abordagem de Carline e Goktas, quanto a nossa.

Finalmente, Yuan *et al.* [Yuan et al. 2011] propuseram contar a frequência de

 ©2015 SBC — Soc. Bras. de Computação

execução de instruções que ficam localizadas próximas a terminadores. Com o intuito de identificar padrões ROP, eles atribuiram a cada instrução uma quantidade de "suspeita". Quanto mais próxima de um terminador, mais suspeita e a instrução. Assim que um limiar de suspeita é atingido, um aviso de ataque é disparado. Yuan *et al.* [Yuan et al. 2011] demonstraram que é possível usar contadores de desempenho para manter o custo dessa técnica abaixo de 5% do tempo de execução original da aplicação. É provável que esse tipo de metodologia produza muitos falsos positivos, pois Yuan *et al.* [Yuan et al. 2011] reportaram um aviso falso em uma quantidade pequena de testes.

7. Conclusão

Este trabalho apresentou uma técnica para encontrar a maior frequência de instruções de retorno possível de ser observada durante a execução de um programa dentro de uma janela de K instruções. Tal estimativa foi denominada δ-RET. O algoritmo apresentado determina δ-RET de forma estática, isto é, sem demandar a execução do programa. A estimativa é consistente, porém conservadora. Isto quer dizer que, durante sua execução, um programa nunca apresentará uma frequência maior de operações de retorno que δ-RET, porém, esse valor pode nunca ser alcançado na prática. A principal serventia de δ-RET é permitir que técnicas de detecção baseadas na monitoração da frequência de instruções de retorno possam usar limiares específicos para cada aplicação, em vez de limiares universais. Esses últimos, o artigo mostrou que podem ser contornados com relativa facilidade.

Trabalhos Futuros. Apesar do controle da frequência de qualquer instrução de desvio indireto ser viável, este trabalho foca apenas nas instruções de retorno (RET). É possível construir ataques ROP que utilizam instruções indiretas de chamada de função (CALL) ou desvios incondicionais (JMP) para interligar os *gadgets* que compõem um ataque ROP [Checkoway et al. 2010]. Esse tipo de ataque, baseado em instruções de desvio indireto, é mais difícil de construir. Ainda assim, uma sequência natural deste trabalho é a detecção de densidades máximas de desvios indiretos em geral.

Agradecimentos. Este projeto é financiado pela Companhia Intel de Semicondutores – *Projeto eCoSoC*, CNPq, CAPES e FAPEMIG. Os autores são extremamente gratos a *George Cox* por todo o suporte e inspiração dados ao longo do projeto eCoSoC.

Referências

[Aho et al. 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

[Callas 2011] Callas, J. (2011). Smelling a RAT on duqu. On-line.

[Carlini and Wagner 2014] Carlini, N. and Wagner, D. (2014). ROP is still dangerous: Breaking modern defenses. In *Security Symposium*, pages 385–399. USENIX.

[Checkoway et al. 2010] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *CCS*, pages 1–14. ACM.

[Chen et al. 2009] Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). DROP: Detecting return-oriented programming malicious code. In *ISS*, pages 163–177. IEEE.

 ©2015 SBC — Soc. Bras. de Computação

[Cheng et al. 2014] Cheng, Y., Zhou, Z., and Yu, M. (2014). ROPecker: A generic and practical approach for defending against ROP attacks. *NDS*.

[Davi et al. 2009] Davi, L., Sadeghi, A., and Winandy, M. (2009). Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *WSTC*, pages 49–54.

[Eeckhoutte 2014] Eeckhoutte, P. V. (2014). Analyzing heap objects with mona.py. https://www.corelan.be/.

[Göktas et al. 2014] Göktas, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Security Symposium*, pages 417–432. USENIX.

[Guha et al. 2007] Guha, A., Hiser, J. D., Kumar, N., Yang, J., Zhao, M., Zhou, A., Childers, B. R., Davidson, J. W., Hazelwood, K., and Soffa, M. L. (2007). Virtual execution environments: Support and tools. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6. IEEE.

[Han et al. 2013] Han, Y. H., Park, D. S., Jia, W., and Yeo, S. S. (2013). Detecting return oriented programming by examining positions of saved return addresses. In *LNEE*, pages 3:1–3:18. Springer.

[Jiang et al. 2011] Jiang, J., Jia, X., Feng, D., Zhang, S., and Liu, P. (2011). HyperCrop: a hypervisor-based countermeasure for return oriented programming. In *LNCS*, pages 46–62. Springer.

[Lattner and Adve 2004] Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE.

[Luk et al. 2005] 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. ACM.

[Pappas et al. 2013] Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent rop exploit mitigation using indirect branch tracing. In *SEC*, pages 447–462. USENIX.

[PaX 2003a] PaX, T. (2003a). Non-executable pages design & implementation.

[PaX 2003b] PaX, T. (2003b). PaX address space layout randomization (ASLR).

[Schrijver 2003] Schrijver, A. (2003). *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Science & Business Media.

[Shacham 2007] 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. ACM.

[Tymburibá et al. 2014] Tymburibá, M., Filho, A., and Feitosa, E. (2014). Controlando a Frequência de Desvios Indiretos para Bloquear Ataques ROP. In *SBSeg*, pages 223–236. SBC.

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

 ©2015 SBC — Soc. Bras. de Computação