Capítulo 1

Análise Dinâmica de Programas Binários

Hugo Sousa e Mateus Tymburibá

Abstract

Dynamic analyses are techniques used to observe the behavior of programs during their execution. Tools that implement such techniques include not only the well known profilers, such as gprof and Shark, but also dynamic binary instrumentation frameworks, the focus of this course. Dynamic Binary Instrumentation (DBI) adds code to a program's execution flow in order to study the events that occur during its execution. Analysis code is executed as if it were part of the program's normal execution flow, without affecting it, and performing extra tasks - like measuring performance or identifying errors - in a transparent manner. Since they work with the actual input data of programs, dynamic analyses are capable of assessing, in an exact way, conditions that cannot be defined statically. Due to this advantage and the maturity achieved by DBI tools, this technique has been widely used in multiple scenarios, notably the systems security area. This mini course presents the fundamentals about DBI and discusses real examples in security of applications. Hence, it enables professionals and researchers to benefit from the many possibilities offered by DBI, besides allowing the attendants to get acquainted with some concepts related to software security.

Resumo

Análises dinâmicas são técnicas utilizadas para observar o comportamento de programas durante sua execução. Ferramentas que implementam tais técnicas incluem não somente os bem conhecidos perfiladores (profilers), como gprof e Shark, mas também instrumentadores binários, foco deste curso. A Instrumentação Dinâmica de Binários (IDB) adiciona código ao fluxo de execução de um programa a fim de estudar eventos que ocorrem durante sua execução. O código de análise é executado como se fizesse parte da execução normal do programa, sem perturbá-la, e fazendo seu trabalho extra - como medir o desempenho ou identificar erros - de forma transparente. Por trabalharem com valores de entrada reais, análises dinâmicas são capazes de avaliar de forma exata condições que não podem ser definidas estaticamente. Em função dessa vantagem e da maturidade

alcançada pelas ferramentas de IDB, ela tem sido amplamente utilizada em diversos cenários, com destaque para a área de segurança de sistemas. Este minicurso apresenta os fundamentos sobre IDB e discute exemplos reais voltados à segurança de aplicações. Dessa forma, habilita profissionais e pesquisadores a usufruírem das diversas possibilidades oferecidas pela IDB, além de familiarizar seus participantes com alguns conceitos relacionados à segurança de software.

1.1. Introdução

À medida que os sistemas computacionais se tornam mais complexos, com novos paradigmas e modelos de programação, com diferentes tipos de hardware executando concomitantemente em um mesmo computador e com aplicações cada vez maiores, torna-se também mais difícil a compreensão de todas as nuances de uma aplicação. Muitas vezes, para obter-se dados mais precisos, é necessário analisar como um software se comporta quando executado, ao contrário de simplesmente analisar seu código fonte [Reddi et al. 2004].

Foi pensando em atender a essas necessidades que o processo de instrumentação binária foi proposto. Ele consiste na adição de código ao arquivo executável de uma aplicação, de modo a observar ou alterar o comportamento do programa [Laurenzano et al. 2010]. Esse código adicional permite, então, que o programador obtenha informações importantes sobre a aplicação executada, que não estariam disponíveis de outra maneira. Dessa forma, o processo de instrumentação binária é extremamente útil para a análise do comportamento de uma aplicação, a avaliação de desempenho e a detecção de defeitos (*bugs*). Contudo, os usos da técnica de instrumentação binária há muito não se limitam a essas possibilidades. A instrumentação binária já foi utilizada na academia com finalidades diversas, como no auxílio ao projeto de sistemas, na verificação da corretude de programas, na otimização de softwares e na verificação da segurança de sistemas, entre outros [Uh et al. 2006]. Neste trabalho, estudaremos uma modalidade dessa técnica, conhecida como instrumentação dinâmica de binários, que aplica a instrumentação durante a execução de uma aplicação.

1.1.1. Contextualização

Como já mencionado, a maior motivação para a criação da técnica de instrumentação binária foi o processo de compreensão de software [Cornelissen et al. 2009]. Esse processo envolve toda a compreensão do funcionamento de um sistema e era inicialmente feito de forma manual pelos programadores. Como tal, consistia basicamente no estudo do código fonte de um programa e sua documentação (quando existente). Esses fatores faziam com que o procedimento se tornasse demorado e insuficiente para cobrir todos os aspectos de um programa.

O uso de análise dinâmica de software mostrou-se eficaz na abordagem desses problemas. Com esse tipo de ferramenta, dados coletados durante a execução de um programa podem ser analisados. Essa técnica traz maior precisão à compreensão de software, uma vez que permite observar o comportamento de um programa quando executado sobre dados reais. Entretanto, por mais que a análise dinâmica de um programa represente um avanço em relação à compreensão de software essencialmente manual, ela também

apresenta limitações. Dentre elas, talvez a mais importante seja o fato de que uma única execução de um programa provavelmente não exercerá todos os seus possíveis fluxos de execução. Na maioria dos programas escritos hoje em dia, o conjunto de dados utilizados como entrada da aplicação afetam fortemente seu fluxo de execução. Hoje em dia, porém, já existem técnicas para lidar com essas limitações, como o uso de heurísticas e abstrações para agrupar execuções de um programa que compartilham propriedades [Cornelissen et al. 2009].

Desde que a análise dinâmica começou a ser utilizada no contexto de compreensão de software, os métodos de funcionamento e os focos de análise das ferramentas que a utilizam se tornaram os mais variados. Até meados dos anos 2000, o principal foco dessas ferramentas foi a visualização de propriedades estruturais dos programas analisados, como, por exemplo, fluxos de execução e estruturas de dados. Após os anos 2000, além das já citadas ferramentas de visualização, nota-se também que as ferramentas de análise dinâmica passam a ter maior foco em planejamento de sistemas e aspectos comportamentais, como a interação entre *threads* em sistemas distribuídos.

Atualmente, existe uma grande variedade de ferramentas de análise dinâmica disponíveis. Dessa forma, o programador pode ficar em dúvida em relação a qual delas escolher para seus propósitos. Por exemplo, as ferramentas de visualização se mostram mais úteis para o programador que visa estudar uma aplicação e entender suas peculiaridades em um nível mais alto, sem maiores preocupações com os efeitos da execução do programa em termos de arquitetura.

Neste trabalho, o foco de nosso estudo será a análise dinâmica de uma aplicação através da técnica de Instrumentação Dinâmica de Binários (IDB). Essa escolha, em detrimento dos outros tipos de ferramentas de análise dinâmica, se deu principalmente pelo nível de detalhe alcançável com o uso da IDB. Além disso, ela permite ao programador definir quais são os dados específicos de seu interesse, utilizando uma estratégia orientada a objetivos.

1.1.2. Instrumentação Dinâmica de Binários

Nesta seção, a técnica de Instrumentação Dinâmica de Binários será analisada em detalhes. Em um primeiro momento (Seção 1.1.2.1), o conceito geral de instrumentação de binários será apresentado, juntamente com algumas de suas aplicações. Em seguida (Seção 1.1.2.2), será mostrada uma comparação entre instrumentação estática e dinâmica de binários, apresentando-se as vantagens e desvantagens de cada uma. Finalmente (Seção 1.1.2.3), uma perspectiva sobre a grande quantidade de ferramentas de instrumentação dinâmica de binários existentes no mercado será traçada. As ferramentas de maior popularidade atualmente serão destacadas e comparadas com a ferramenta escolhida como foco de estudo deste trabalho.

1.1.2.1. Definição e Aplicações

Como mencionado anteriormente, a técnica de instrumentação binária consiste em adicionar porções de código ao executável de uma aplicação. Primordialmente, quando pensamos nessa ideia, duas perguntas básicas devem ser respondidas. A primeira se refere ao

local do executável onde o código de instrumentação deve ser inserido. A segunda diz respeito a qual código deve ser enxertado. A primeira pergunta define a granularidade da instrumentação, isto é, o nível de abstração do programa ao qual o código de instrumentação será aplicado. Por exemplo, o programador pode estar interessado em analisar os valores dos registradores do processador à medida que as instruções da aplicação são executadas. Nesse caso, a instrumentação será feita com granularidade de instrução. Por outro lado, outro programador pode estar interessado no efeito de uma função do programa analisado na memória da aplicação. Para obter esses dados, ele não precisa adicionar código de instrumentação a cada instrução do programa. Para isso, basta que ele adicione código de instrumentação ao começo e/ou fim de uma função da aplicação. Em relação à segunda pergunta, o programador deve estar atento ao tipo de informação que deseja obter com a instrumentação. Além disso, é importante notar que o nível de detalhes dos dados obtidos com a instrumentação será limitado pela ferramenta utilizada.

Diante dos diversos usos e ferramentas disponíveis para instrumentação binária atualmente, é interessante observar como essas ferramentas são utilizadas. Uma das aplicações mais comuns da técnica de instrumentação binária é seu uso como ferramenta de detecção de erros. Isso se deve à possibilidade de inserir código que coleta informações sobre o estado (memória e registradores) da aplicação em posições estratégicas da execução do programa. Dessa forma, pode-se, por exemplo, monitorar todas as operações de memória realizadas por um programa. Isso inclui a verificação do conteúdo de bytes lidos ou escritos no espaço de endereçamento de um processo. Pode ser de interesse do programador, também, detectar acessos a regiões de memória não endereçáveis, obter informações sobre travas de memória para detectar situações de corrida em aplicações paralelas, ou até mesmo checar os tipos utilizados por um programa durante sua execução. De fato, é possível monitorar todas as operações realizadas com uma variável e impedir que operações inapropriadas (de acordo com a tipagem) sejam executadas. Na literatura, um exemplo do uso da instrumentação para fins de detecção de erros é a implementação do conceito de memória sombra, através do qual dados sobre todos os bytes usados por um programa são coletados [Nethercote and Seward 2007a].

A instrumentação binária também se mostra útil no auxílio ao projeto de sistemas. Métodos para a avaliação de protótipos, com foco no desempenho da memória, já foram utilizados na literatura com o apoio da técnica de instrumentação [Uhlig and Mudge 1997]. A abordagem de avaliação orientada à sequência de acessos à memória, por exemplo, simula o comportamento de um projeto de memória quando efetivamente sob uso. Nela, uma aplicação de interesse é escolhida e executada. Durante sua execução, ela realiza uma série de referências à memória. Essa série de referências é, então, coletada por uma ferramenta de instrumentação binária. Como essa sequência de referências é potencialmente muito grande, algum tipo de redução é realizada. Por fim, a sequência é passada para algum programa que simulará o comportamento do sistema de memória projetado quando submetido à sequência de referências à memória previamente coletada. Note que, nesse contexto, é importante utilizar uma ferramenta que consiga coletar dados em diversos cenários, levando em consideração a existência de aplicações *multithread*, de interferência do sistema operacional, de códigos compilados ou ligados dinamicamente e de gigantescas sequências de referências à memória.

Finalmente, destacamos o uso da técnica de instrumentação para a otimização

de programas. O sistema Etch [Romer et al. 1997], desenvolvido na Universidade de Washington, permite reescrever o arquivo binário executável de uma aplicação para alcançar três objetivos principais: entender como uma aplicação interage com a arquitetura de um computador; avaliar o desempenho de um programa em desenvolvimento; e otimizar o desempenho da aplicação analisada. Os dois primeiros objetivos já foram abordados anteriormente. A ferramenta alcança o terceiro objetivo da seguinte forma: ela executa a aplicação alvo e coleta informações de interesse, como a sequência de acessos à memória e o número de instruções de desvio condicional que falham. A partir dessas e de outras informações, a ferramenta é então capaz de reescrever o binário da aplicação para, por exemplo, melhorar a localidade de referência de memória ou reordenar as instruções para manter o pipeline do sistema cheio, entre outras possíveis otimizações.

1.1.2.2. Instrumentação Estática x Dinâmica

A técnica de instrumentação binária possui duas abordagens, estática e dinâmica, que se diferem pelo momento em que são aplicadas ao arquivo executável do programa alvo. A instrumentação estática de binários ocorre antes que o programa seja executado. Por outro lado, a instrumentação dinâmica de binários insere o código de instrumentação à medida que o programa é executado. [Nethercote 2004].

Existem vantagens e desvantagens no uso de ambas abordagens. Pensando em termos de custo computacional, a abordagem estática se mostra mais atraente, uma vez que a inserção de código de instrumentação é feita somente uma vez, antes da execução do programa, gerando assim um novo arquivo executável [Laurenzano et al. 2010]. Em contrapartida, na Instrumentação Dinâmica de Binários, é inserido um *overhead* adicional para que a ferramenta de instrumentação execute trocas de contexto com a aplicação instrumentada, além de efetuar a desmontagem de códigos e a geração de instruções, tudo durante a execução da aplicação instrumentada. Estudos apontam que, para efetuar uma tarefa simples de contagem do número de blocos de instruções executadas, os frameworks Pin, Strata, DynamoRIO e Valgrind apresentam *overheads* médios que variam entre 2,3 a 7,5 vezes [Luk et al. 2005].

Por outro lado, as soluções de instrumentação estática que exigem uma desmontagem precisa do código binário para que possam modificá-lo só funcionam caso informações de depuração, como a tabela de símbolos, estejam disponíveis. Entretanto, por questões de economia de espaço e proteção de propriedade intelectual, essas informações de depuração normalmente são removidas dos softwares destinados a ambientes de produção. Outra deficiência relacionada à abordagem estática recai sobre a possibilidade de corromper códigos assinados, blocos de instruções que executam verificações sobre si mesmos – como checagens de somas (*checksums*) –, ou códigos que se modificam durante a execução – como trechos ofuscados (*enconded*). Além disso, a inserção estática de instruções também é incapaz de atuar em códigos compilados sob demanda (*JIT-compiled*), comum em aplicações que dão suporte a ambientes com linguagens interpretadas, como Java, JavaScript, Flash, .Net e SilverLight.

A abordagem dinâmica provê muitas vantagens sobre a estática. Entre elas está a possibilidade de instrumentar bibliotecas compartilhadas sem qualquer trabalho extra.

Ou seja, com a instrumentação dinâmica, não é necessário instrumentar separadamente cada uma das bibliotecas compartilhadas que um programa usa. Além disso, ferramentas de instrumentação dinâmica são mais flexíveis, no sentido de que o código de instrumentação pode ser removido ou modificado durante a execução do programa. É importante ressaltar também que a abordagem de IDB não necessita de informações adicionais sobre o código binário, como tabelas de símbolos, e tampouco requer o código-fonte das aplicações. Finalmente, esses sistemas permitem a análise de programas que geram códigos compilados sob demanda (*JIT-compiled*). Por tudo isso, este trabalho foca na Instrumentação Dinâmica de Binários.

1.1.2.3. Ferramentas

Como já discutido na Seção 1.1.1, existe atualmente uma grande variedade de ferramentas de instrumentação dinâmica de binários. Hoje em dia, nota-se uma grande tendência de desenvolvimento de sistemas que permitem a criação e a personalização de ferramentas de instrumentação dinâmica. Esses *frameworks* proveem ao programador a possibilidade de criar suas próprias ferramentas, de acordo com as suas necessidades. Isso permite reduzir o custo da instrumentação, uma vez que a ferramenta só instrumentará partes de interesse no programa. Nessa seção, três dos *frameworks* mais populares para criação de ferramentas de instrumentação dinâmica serão discutidos. São eles: DynamoRIO [Garnett 2003], Valgrind [Nethercote and Seward 2007b] e Pin [Reddi et al. 2004]. As ferramentas de instrumentação criadas com os dois primeiros são escritas em C, ao passo em que o terceiro utiliza C++.

DynamoRIO é um *framework* de instrumentação dinâmica de binários cujo foco é a otimização dinâmica. Por esse motivo, seus criadores se preocuparam com a implementação de um sistema que tivesse um custo de execução baixo. Para isso, ele utiliza um sistema de cache que executa blocos de instruções do programa original à medida que os encontra. Dos três *frameworks* discutidos nesta Seção, DynamoRIO foi mostrado ser o mais eficiente em termos de tempo de execução [Rodríguez et al. 2014]. Atualmente, o *framework* tem suporte para quatro arquiteturas: IA-32, AMD64, ARM e AArch64. Ele pode ser executado em sistemas Windows, Linux e Android.

O segundo *framework* mencionado, Valgrind, tem seu foco na construção de ferramentas de análise "pesada", no sentido de que examinam grandes quantidades de dados, acessados e atualizados em padrões irregulares. Um dos mais populares exemplos dessas ferramentas criadas com Valgrind é a Memcheck. Essa ferramenta mantém um registro das posições de memória utilizadas por uma aplicação que são indefinidas (não inicializadas ou derivadas de outras posições indefinidas), a fim de detectar acessos não seguros à memória. Justamente por seu foco em ferramentas de análise "pesada", o *framework* apresenta os piores índices de eficiência em termos de aumento de tempo de execução reportados na literatura [Rodríguez et al. 2014]. Valgrind apresenta maior flexibilidade do que DynamoRIO em termos de suporte a sistemas e arquiteturas, podendo ser executado nas arquiteturas x86, AMD64, ARM, ARM64, PPC32/64/64LE, S390X e MIPS32/64, e nos sistemas operacionais Linux, Solaris, Android e Darwin.

Por fim, o *framework* Pin da Intel funciona de forma semelhante a seus concorren-

tes. Os usuários escrevem ferramentas de instrumentação dinâmica em C++, utilizando a API[1] do *framework*. Sua API é a mais rica dos três *frameworks* analisados e possui uma grande coleção de funções dedicadas a sistemas x86, o que fornece ao programador maior nível de detalhe nos dados coletados. Em termos de custo de tempo de execução, o Pin apresenta um desempenho próximo do concorrente mais veloz (DynamoRIO) [Rodríguez et al. 2014]. A fim de oferecer instrumentação eficiente, ele utiliza um compilador *just-in-time* para inserir código de instrumentação. Além disso, ao contrário dos seus concorrentes, não precisa da assistência do usuário para otimizar o desempenho de suas ferramentas. Outro fator que o diferencia dos outros dois *frameworks* aqui analisados é a possibilidade de anexá-lo a um processo sendo executado e, em seguida, desvinculá-lo [Luk et al. 2005]. Consideramos que, devido à sua extensa API e ao desempenho competitivo em relação a seus concorrentes, o Pin é um bom ponto de partida para um programador interessado em se iniciar na análise dinâmica de binários. Portanto, o escolhemos como ferramenta a ser utilizada no nosso estudo da técnica de IDB. A próxima seção examina esse *framework* e fornece um guia sobre sua utilização.

1.2. Pin

Nesta seção, o *framework* Pin é apresentado. Em um primeiro momento (Seção 1.2.1), as características da ferramenta serão mostradas para informar o leitor sobre as capacidades e limitações do *framework*. Também serão discutidos aspectos técnicos gerais, como linguagens de programação e suporte a *multithreading*. Em seguida, o curso abordará tópicos específicos sobre funcionamento, instalação, compilação de ferramentas e execução de códigos com o Pin.

1.2.1. Funcionamento

O Pin é um *framework* para criação de ferramentas de IDB. Portanto, o programador precisa utilizar a interface fornecida pelo *framework* para especificar qual será o código aplicado ao programa instrumentado e em quais locais do programa esse código será inserido. Para isso, o programador cria um programa escrito em C++, que será sua ferramenta de IDB. As ferramentas de IDB criadas com o Pin são chamadas de *Pintools*. Atualmente, o *framework* pode ser utilizado nas arquiteturas IA-32, Intel64 e MIC, e nos sistemas operacionais Windows, Linux, OSX e Android.

O Pin pode ser utilizado com diversos tipos de aplicações. O programador só precisa especificar o arquivo executável ao qual o *framework* será anexado. Atualmente, ele tem suporte para aplicações *multithread* e para tratamento de sinais e de exceções. Para conseguir tempo de execução competitivo, o Pin aplica uma série de otimizações de compilação ao código de instrumentação criado pelo programador. Com o Pin, é possível examinar qualquer instrução executada por uma aplicação, mesmo aquelas que pertencem a bibliotecas compartilhadas. Também é possível instrumentar qualquer chamada de função. Além disso, o programador pode escolher instrumentar, ao mesmo tempo, um processo e todos os outros na sua árvore (os processos criados direta e indiretamente por ele) [Devor 2013].

Quando o Pin é executado, são especificados, no mínimo, a ferramenta de IDB a

[1] *Application Programming Interface: rotinas fornecidas para a construção de algum software.*

ser utilizada e o programa a ser instrumentado. Abstraindo alguns detalhes, o *framework* executa como descrito nos passos a seguir:

1. As rotinas de inicialização e outros dados do *framework*, da ferramenta de IDB e do programa alvo são carregados em memória.
2. O Pin começa a execução da ferramenta de IDB, que executa suas próprias rotinas de inicialização.
3. A ferramenta de IDB lê a primeira sequência de instruções da aplicação alvo.
4. A ferramenta de IDB adiciona código de instrumentação à sequência de instruções lida no passo anterior, de acordo com a especificação do programador.
5. O código obtido (aplicação alvo e código de instrumentação) é levado à cache de código do Pin.
6. O código presente na cache é executado e, então, volta-se ao passo 3 para as próximas porções de código da aplicação alvo.

Além do funcionamento regular apresentado acima, o *framework* também trata as chamadas de sistema, os eventos de *threads* e os sinais enviados à aplicação. Caso um desses eventos aconteça, o Pin recorre ao sistema operacional. A Figura 1.1 ilustra o funcionamento do Pin em memória e sua interação com o sistema operacional. Analisando o esquema de execução descrito acima, percebe-se que o custo em tempo de execução causado pelas ferramentas de IDB vem, principalmente, do grande número de trocas de contexto realizadas entre o *framework* utilizado, o sistema operacional e as ferramentas em si.

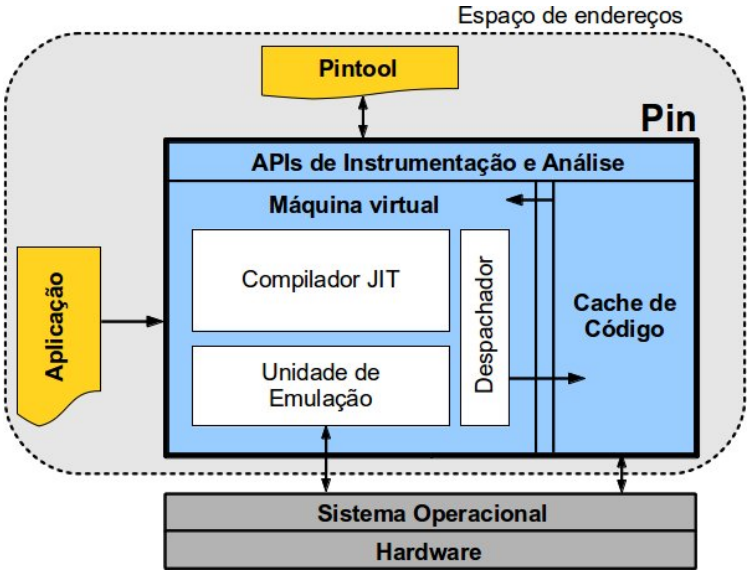

Figura 1.1. Esquema do *framework* Pin em memória. Na figura, a interação entre o Pin, a ferramenta de IDB, a aplicação instrumentada e o sistema operacional. [Ferreira et al. 2014].

1.2.1.1. JIT x Probe

Atualmente, o Pin provê dois métodos de instrumentação. O primeiro, sob demanda (*just-in-time* - JIT), cujo funcionamento foi descrito no passo a passo anterior, é o mais comum. Com ele, o Pin cria, incrementalmente, uma versão modificada da aplicação sendo instrumentada. Isso significa que, à medida que o *framework* encontra novas porções de código da aplicação alvo, ele insere porções de código de instrumentação, de forma que o código original da aplicação nunca é executado, apenas o código gerado a partir da instrumentação. Já o modo de sondagem (*probe*) só pode ser utilizado para instrumentar a aplicação alvo com granularidade de rotinas. Nesse caso, o *framework* modifica as instruções originais da aplicação, inserindo saltos para código de instrumentação, antes ou depois da execução das rotinas da aplicação alvo. Por exemplo, no caso de uma ferramenta que instrumenta as rotinas de uma aplicação antes de sua execução, este é o procedimento:

1. Imediatamente antes do início de uma função *foo*, Pin insere uma instrução de salto para a primeira instrução da função de instrumentação.
2. Ao fim do código da função de instrumentação, *foo* é chamada.
3. Ao fim do código de *foo*, uma instrução de salto redireciona a execução da aplicação para o ponto imediatamente após *foo*, no código do programa.

O modo de instrumentação sob demanda é mais comum e mais flexível, uma vez que algumas funções da API do Pin estão disponíveis somente para esse modo e ele funciona para todos os níveis de granularidade descritos na Seção 1.2.2. Entretanto, o modo de sondagem pode ser uma opção melhor para o programador que precisa de ferramentas com custo de tempo de execução reduzido. Isso se deve ao fato de que o modo de sondagem executa o código original da aplicação e reduz o número de trocas de contexto entre ela e o *framework*.

1.2.1.2. Análise x Instrumentação

Em relação ao processo de construção das ferramentas de IDB, Pin utiliza uma distinção importante entre rotinas de análise e rotinas de instrumentação. Como já mencionado, o programador precisa definir, ao criar sua *Pintool*, em quais regiões do código da aplicação alvo inserir o código de instrumentação e qual código inserir nessas regiões. Isso é feito através das rotinas de instrumentação e análise, respectivamente, e o *framework* fornece uma interface para que o programador defina essas rotinas. Para a criação de rotinas de análise, o programador deve identificar quais dados da execução do programa ele quer coletar e usar as funções apropriadas da API do *framework*. Entretanto, para a criação de rotinas de instrumentação, ele deve conhecer as opções de granularidade que Pin provê ao usuário. Essas opções são apresentadas na seção seguinte.

1.2.2. Granularidade

As opções de granularidade de instrumentação do Pin são apresentadas na Tabela 1.1.

Granularidade	Descrição
Instrução (INS)	Define uma instrução do arquivo executável do programa alvo. É a opção de granularidade que fornece o maior nível de detalhes, uma vez que é possível verificar como cada instrução afeta o estado da máquina. Por outro lado, é a opção que causa maior sobrecarga em tempo de execução.
Bloco básico (BBL)	Um bloco básico é uma sequência de instruções que tem somente um ponto de entrada e somente um ponto de saída. Dessa forma, um bloco básico é sempre atingido e abandonado através de uma instrução de desvio.
Traço (TRACE)	Sequência de instruções que têm somente um ponto de entrada, mas vários pontos de saída. Traços geralmente começam nos alvos de instruções de desvio e terminam em uma instrução de desvio incondicional, incluindo chamadas de função e instruções de retorno. Como o *framework* descobre o fluxo de execução da aplicação alvo à medida que ela é executada, ele constrói traços de forma incremental. Dessa forma, caso uma instrução de desvio seja encontrada no meio de um traço, ele é quebrado em dois blocos básicos.
Rotina (RTN)	Uma rotina se refere a uma função (ou procedimento) como gerada por um compilador para linguagens procedurais. Para que o *framework* consiga identificar essas funções, é necessário que ele tenha acesso às informações de tabela de símbolos criada pelo compilador da aplicação instrumentada. Para tornar essas informações disponíveis para o Pin, o programador deve chamar a função **PIN_InitSymbols()** antes que a aplicação comece a ser executada.
Imagem (IMG)	Uma imagem é o nível de granularidade que define a maior abstração em termos de tamanho da sequência de código instrumentada. Esse nível de granularidade se refere à instrumentação de todo o executável da aplicação alvo, além das bibliotecas que a aplicação utiliza. É importante ressaltar que os objetos do tipo **IMG** só são criados se a biblioteca compartilhada correspondente é carregada.
Seção (SEC)	Seções são as unidades que compõem uma imagem (IMG). Elas podem ser mapeadas ou não mapeadas. No primeiro caso, a seção ocupa um espaço de endereçamento dentro da imagem da qual faz parte. Essas seções foram modeladas no *framework* com base nas seções de arquivos de imagem *elf*.

Tabela 1.1. Opções de granularidade de instrumentação fornecidas pelo *framework* Pin.

O conceito de granularidade define o nível de abstração ou detalhe com que a

ferramenta de IDB instrumenta o programa alvo. Isto é, o programador, ao escrever sua ferramenta, precisa dizer ao *framework* quais são as porções de código que serão lidas do programa alvo, levadas à memória e instrumentadas a cada passo, como descrito na Seção 1.2.1. Esse processo é facilitado pela API do Pin, que define, ao todo, 6 opções de granularidade [Intel 2018b].

De maneira geral, o programador deve, além de determinar os níveis de granularidade que sua ferramenta utilizará, definir o ponto de inserção do código de instrumentação em relação à sequência de instruções instrumentada. No Pin, esse ponto de inserção pode ser antes da sequência de instruções, depois dela, dentro (em qualquer lugar) ou em seus desvios tomados (Seção 1.2.5). É importante notar que algumas dessas opções não estão disponíveis para todos os níveis de granularidade.

1.2.3. Instalação

Para começar o processo de criação de *Pintools*, é necessário instalar e configurar o ambiente de execução e compilação do Pin. Esta seção fornece um guia completo para esse processo, tendo como referência dois sistemas operacionais: Windows e Linux. Serão apresentadas informações sobre o download e a configuração necessária para executar o *framework* sem erros. Ao fim do passo a passo, o leitor poderá iniciar o processo de escrita de sua primeira *Pintool*.

O primeiro passo para instalar o Pin é identificar a versão apropriada para o sistema operacional e a arquitetura da máquina do programador. No *website* do *framework* [Intel 2012], a aba *Downloads* fornece várias opções para fazer o *download* do Pin. Após realizar o *download* e extração dos arquivos do Pin, as instruções são específicas para cada sistema operacional, como descrito nas seções seguintes.

1.2.3.1. Windows

O processo de instalação do Pin nas plataformas Windows é mais extenso do que nas plataformas Linux. Isso se deve ao fato de que muitos dos utilitários usados para execução e compilação das *Pintools* já são instalados por padrão nas distribuições Linux. O passo a passo completo é mostrado a seguir.

1. Download e instalação do ambiente de desenvolvimento Microsoft Visual Studio C++. Esse ambiente é necessário para a compilação das *Pintools*.

2. *Download* e instalação das ferramentas Cygwin. O pacote Cygwin é uma coleção de ferramentas GNU [GNU 2018] e outras de código aberto que fornecem, em plataformas Windows, grande parte da funcionalidade existente em distribuições Linux. Durante a instalação, o usuário será perguntado quais pacotes instalar. Nesse ponto, o único pacote necessário para o Pin é o pacote *Devel*, que deve ser marcado para instalação (opção *Install*). A Figura 1.2 ilustra esse passo.

3. Nas variáveis de ambiente do sistema operacional, adicione o diretório em que o Pin foi extraído na variável $PATH$. Apesar de não serobrigatório, esse passo permite que o programador execute o *framework* na linha de comandos a partir de qualquer

diretório, sem especificar o caminho completo para o diretório em que o Pin foi extraído.

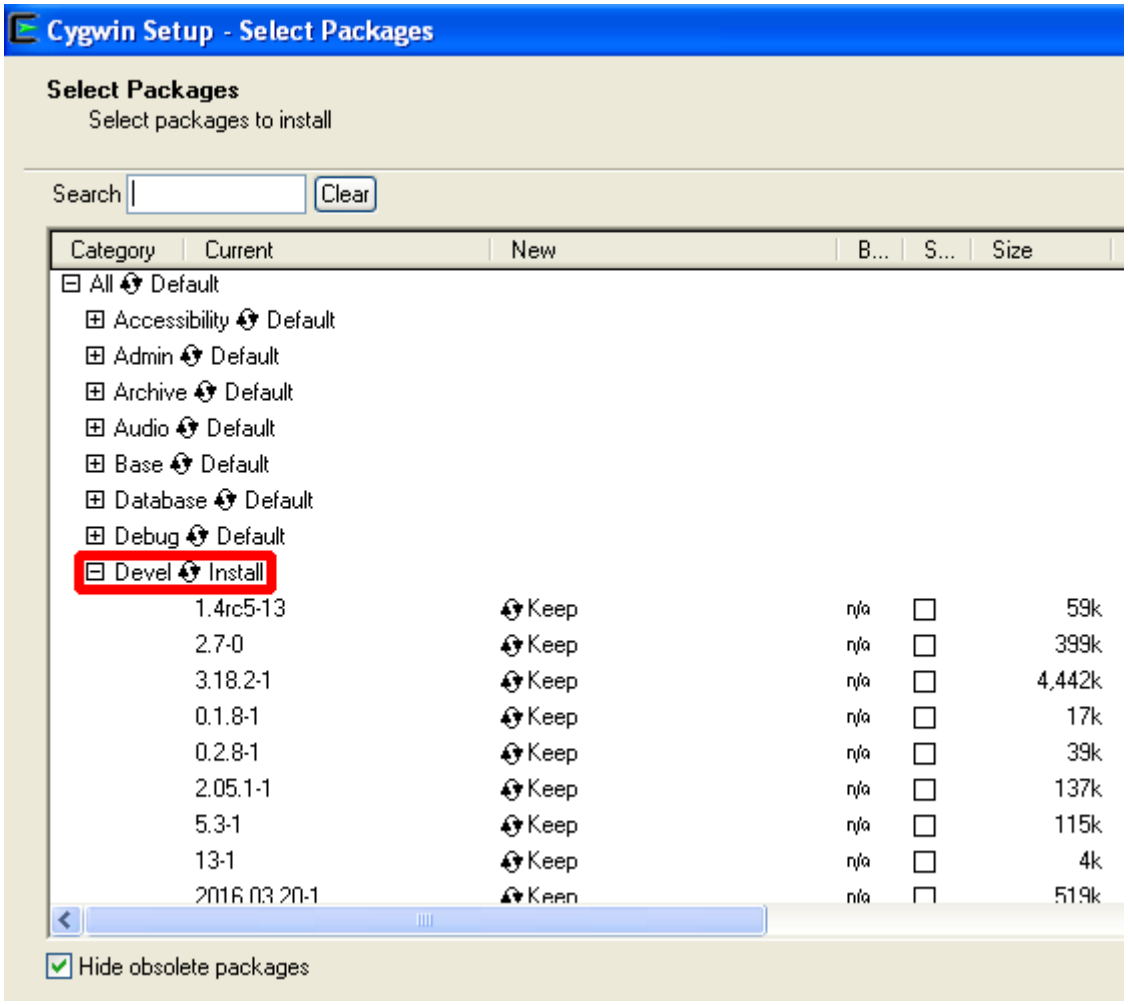

Figura 1.2. Instalação do pacote *Devel* através da coleção Cygwin.

Para compilar e executar o Pin em plataformas Windows, o usuário deve utilizar o terminal de comandos (*Command Prompt*) do Microsoft Visual Studio C++.

1.2.3.2. Linux

Como já foi dito, as distribuições Linux geralmente já possuem, por padrão, os utilitários que são requisitos para a compilação e a execução das *Pintools*. Usuários de plataformas Linux devem possuir em suas máquinas os seguintes programas instalados:

- Compilador *g++*. Necessário para compilação das *Pintools*, que são programas escritos em C++.
- Utilitário *make*. Necessário para automatizar o processo de compilação utilizado pelo Pin.

Em plataformas Ubuntu, o usuário pode instalar a versão mais recente de ambos com:

```
sudo apt-get update
sudo apt-get install g++
sudo apt-get install make
```

De forma similar ao processo de instalação do Pin em plataformas Windows, o usuário pode desejar executar o *framework* a partir de qualquer diretório, sem especificar o diretório em que ele foi extraído. Para isso, é necessário incluir o diretório de extração à variável PATH do sistema. O que pode ser feito da seguinte forma em sistemas Ubuntu:

1. Edição do arquivo **environment** utilizando qualquer editor de texto simples. Aqui, usaremos *vim*.

   ```
   sudo vim /etc/environment
   ```

2. Adição do diretório do Pin à variável PATH. Para isso, inserir o caminho completo do diretório onde o Pin foi extraído dentro das aspas duplas da definição da variável, após dois pontos. O comando de definição da variável será semelhante ao mostrado abaixo, onde o diretório de extração do *framework* foi "/home/hugo/pin-gcc-linux".

   ```
   PATH="/usr/local/bin:/home/hugo/pin-gcc-linux"
   ```

   Para que a mudança tenha efeito na sessão atual, basta executar o comando:

   ```
   source /etc/environment
   ```

Após os passos anteriores, o usuário já pode compilar e executar *Pintools* em plataformas Linux.

1.2.3.3. Compilação e Execução

Com o *framework* devidamente instalado em sua máquina, o programador pode compilar e executar sua primeira *Pintool*. É importante notar, nesse ponto, que as as *Pintools* são bibliotecas de carga dinâmica e, como tal, possuem implementações diferentes de acordo com o sistema operacional utilizado, o que se reflete nos formatos das ferramentas em cada sistema. Para Windows, elas têm formato **dll**, enquanto para Linux, o formato é **so**. Como primeiro exemplo[2], uma *Pintool* simples será compilada e executada. A *Pintool* de escolha para ilustrar o processo de compilação e execução é a **inscount0.cpp**, que simplesmente conta o número de instruções executadas pela aplicação alvo. Para Windows, a aplicação instrumentada será a calculadora (**calc.exe**) e para Linux, a aplicação instrumentação será o comando **ls**, que lista os arquivos do diretório atual. As Tabelas 1.2 e 1.3 mostram os comandos de compilação para Windows e Linux em uma máquina Intel. Em negrito, os comandos para execução do *framework*.

[2]No momento do desenvolvimento deste trabalho, a versão do Pin utilizada é a 3.6, lançada em Fevereiro de 2018.

	Comandos
32bit	cd source\tools\ManualExamples make dir TARGET=ia32 obj-ia32/inscount0.dll **pin -t obj-ia32\inscount0.dll -- calc.exe**
64bit	cd source\tools\ManualExamples make dir obj-intel64/inscount0.dll **pin -t obj-intel64\inscount0.dll -- calc.exe**

Tabela 1.2. Comandos para compilação da ferramenta "inscount0.cpp"e execução do Pin em máquinas Intel 32 e 64 bit com sistema operacional Windows.

	Comandos
32bit	cd source/tools/ManualExamples make TARGET=ia32 obj-ia32/inscount0.so **./pin -t obj-ia32/inscount0.so -- /bin/ls**
64bit	cd source/tools/ManualExamples make obj-intel64/inscount0.so **./pin -t obj-intel64/inscount0.so -- /bin/ls**

Tabela 1.3. Comandos para compilação da ferramenta "inscount0.cpp"e execução do Pin em máquinas Intel 32 e 64 bit com sistema operacional Linux.

Para que consiga utilizar a estrutura dos arquivos *Makefile* do Pin, ao escrever suas *Pintools*, o programador deve incluir seu código fonte em C++ em um dos seguintes diretórios:

source / tools / ManualExamples
source / tools / SimpleExamples

Ao executar a ferramenta "inscount0", será gerado pelo Pin um arquivo chamado "inscount.out", que contém sua saída. A execução do *framework* através da linha de comandos segue a seguinte estrutura:

pin [Pin Args] [-t <Pintool> [Pintool Args]] – <App> [App Args]

Nessa chamada, **[Pin Args]** refere-se aos argumentos passados ao próprio *framework*. Existe uma grande variedade de argumentos que podem ser passados ao Pin e eles definem diversas funcionalidades. Por exemplo, o argumento **-pid** seguido do número de um processo sendo executado diz ao Pin para instrumentar esse processo. Todos os argumentos do Pin estão descritos em sua API. A opção **-t** especifica qual será a *Pintool* (<Pintool>) utilizada durante a instrumentação. Além disso, as próprias *Pintools* também podem receber argumentos. Nesse caso, eles ([Pintool Args]) são especificados logo após o nome da ferramenta. Finalmente, os caracteres **--** separam o Pin e a *Pintool* utilizada da aplicação instrumentada. O arquivo executável da aplicação é especificado em **<App>**. Ele também pode receber argumentos que, por sua vez, são especificados em **[App Args]**.

1.2.4. Estrutura das Pintools

Esta seção tem o objetivo de fornecer ao leitor o conhecimento necessário para a criação de sua primeira Pintool. Para isso, teremos como referência a mesma *Pintool* da seção anterior, "inscount0.cpp"[3], disponível no pacote do Pin, na pasta "source/tools/ManualExamples". Mesmo que essa seja uma ferramenta simples, ela contém todas as partes fundamentais de uma *Pintool*. Logo, a estrutura dessa ferramenta será analisada com detalhes a seguir.

```
31 #include <iostream>
32 #include <fstream>
33 #include "pin.H"
```

A primeira coisa a ser feita em uma *Pintool* é a inclusão do arquivo de cabeçalho **pin.H**. É através dele que a ferramenta terá acesso a toda a API do Pin. Começando pela função principal da ferramenta:

```
80 int main(int argc, char * argv[])
81 {
82     // Inicia o pin
83     if (PIN_Init(argc, argv)) return Usage();
```

A função **Pin_Init** é responsável por inicializar o *framework*. Ela devolve um valor booleano que indica se houve algum erro na chamada do Pin. No código da ferramenta, caso isso aconteça, ela imprime uma mensagem informativa sobre sua chamada e termina a execução. Em seguida, usa-se o código:

```
88     INS_AddInstrumentFunction(Instruction, 0);
```

O uso dessa função no código revela não só o nível de granularidade usado pelo programador, mas também o nome da rotina de instrumentação da ferramenta (Instruction). Aqui, o programador instrumenta todas as instruções (INS) da aplicação alvo. Como o nome da própria função da API sugere (*AddInstrumentFunction*), o usuário define sua rotina de instrumentação ao passar a função *Instruction* para esse procedimento da API do Pin. Como já discutido, a rotina de instrumentação define onde o código de análise será inserido. Estudando o código da função *Instruction*, é possível entender como isso acontece no Pin.

```
45 VOID Instruction(INS ins, VOID *v)
46 {
47     // Insere uma chamada para "docount" antes de todas as
          instruções
48     INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount,
          IARG_END);
49 }
```

Vemos que essa rotina de instrumentação recebe um objeto do tipo INS. Trata-se da instrução a ser instrumentada. O corpo da rotina faz uso de outra função da API do Pin.

[3]Os códigos completos das *Pintools* analisadas estão disponíveis no repositório: https://github.com/ha2398/jai-2018-pin

A função **INS_InsertCall** define onde, em relação à instrução, o código de análise será inserido. Aqui, a opção do programador foi inserir código de análise antes da execução da instrução, o que é especificado pelo parâmetro **IPOINT_BEFORE**. Em seguida, a função recebe um ponteiro de função (**AFUNPTR)** que indica a rotina de análise propriamente dita. Além disso, a lista de argumentos - vazia, neste caso - vem após a especificação da rotina de análise e sempre termina com o identificador **IARG_END**. A rotina de análise é então definida na *Pintool*:

```
41 // Essa função é executada antes que toda instrução execute
42 VOID docount() { icount++; }
```

Como definido pela chamada de **INS_InsertCall**, o código da função **docount** será inserido antes de toda instrução executada. Essa é uma função simples, que apenas incrementa um contador das instruções encontradas até o momento. Voltando à função **main** da *Pintool*, o código é finalizado com:

```
90     // Registra a função "Fini" para ser chamada quando a aplica
          ção termina
91     PIN_AddFiniFunction(Fini, 0);
92
93     // Começa o programa, nunca retorna.
94     PIN_StartProgram();
95
96     return 0;
97 }
```

A função **PIN_AddFiniFunction** especifica uma função definida pelo programador (**Fini**) para ser executada ao fim da execução da aplicação alvo. No caso da *Pintool* "inscount0.cpp", a função **Fini** simplesmente escreve o valor do contador de instruções em um arquivo. A última chamada da função principal da ferramenta é **PIN_StartProgram**, que nunca devolve nenhum valor. Essa função é responsável por finalmente executar a aplicação instrumentada após todas as definições de rotinas de inicialização, finalização, análise e instrumentação.

1.2.5. Tópicos e API

O objetivo desta seção é fornecer uma visão detalhada sobre alguns tópicos de maior relevância dentro da API do *framework*. Sendo assim, a seção não tem como objetivo ser um guia completo de todas as funções fornecidas pelo Pin, mas um panorama que introduzirá algumas de suas funcionalidades mais interessantes ao leitor.

O primeiro tópico a ser discutido é a **passagem de parâmetros** para as rotinas de análise. Como discutido na Seção 1.2.4, o Pin define algumas rotinas cujo propósito é especificar quais serão as funções de análise, para qual tipo de granularidade elas serão aplicadas e onde nas sequências de código da aplicação o código de instrumentação será inserido. Dentro da rotina de instrumentação (especificada por alguma função **AddInstrumentFunction**), a rotina de análise é especificada com alguma função **InsertCall**. Como exemplo, temos a invocação de função a seguir.

```
1 INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) count, IARG_END);
```

As funções **InsertCall** têm uma estrutura bem definida. Nelas, o primeiro argumento é um objeto do tipo que define a granularidade de instrumentação (nesse exemplo, o objeto **ins**, de tipo **INS**, que define a granularidade de instrumentação de instruções). Em seguida, o ponto de inserção do código de análise é especificado, em relação à unidade de granularidade escolhida. A Tabela 1.4 mostra as opções disponíveis.

Ponto de Inserção	Descrição
IPOINT_BEFORE	Insere o código de análise antes da sequência de instruções definida pela unidade de granularidade. Disponível para INS e RTN.
IPOINT_AFTER	Insere o código de análise depois da sequência de instruções definida pela unidade de granularidade. Disponível para INS e RTN.
IPOINT_ANYWHERE	Insere o código de análise dentro da sequência de instruções definida pela unidade de granularidade. Disponível para BBL e TRACE.
IPOINT_TAKEN_BRANCH	Insere o código de análise quando o fluxo de execução encontra uma instrução de desvio que é tomado. Disponível para INS.

Tabela 1.4. Pontos de inserção de código de análise em relação à unidade de granularidade utilizada.

Após especificar o ponto de inserção do código de análise, o programador diz ao *framework* qual deve ser a função que contém o código de análise a ser executado e quais são os argumentos que essa função recebe. Como o número de argumentos não tem um tamanho fixo, o Pin define uma sintaxe específica para definir a lista de argumentos da função de análise, como mostrado a seguir.

```
1 ... (AFUNPTR) analysis, [IARGLIST], IARG_END);
```

No exemplo acima, a função de análise recebe o nome **analysis**. Além disso, **IARGLIST** define toda a lista de parâmetros que ela recebe. Essa lista pode receber tanto atributos do Pin (**IARG**) quanto variáveis definidas pelo usuário. No segundo caso, o programador deve passar como parâmetro, antes da variável em si, o seu tipo. É importante notar que alguns dos **IARGs** requerem argumentos adicionais. A chamada de função seguinte exemplifica esse processo:

```
1 BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)docount,
2   IARG_UINT32, BBL_NumIns(bbl), IARG_END);
```

Nesse caso, o código da função de análise **docount** será inserido antes dos blocos básicos do programa. Também está especificado que a função **docount** recebe um valor de tipo **UINT32** (resultado da chamada **BBL_NumIns(bbl)**).

Um dos tópicos citados anteriormente como exemplos de aplicação da técnica de IDB é a detecção de erros e análise de memória. Sendo assim, é importante estudar como

fazer **leituras do espaço de endereçamento** e obter informações sobre a memória através do Pin. Para isso, usaremos a *Pintool* "pinatrace.cpp", disponível no diretório “source/-tools/ManualExamples”. Essa ferramenta instrumenta cada instrução da aplicação alvo, porém faz uso da API do Pin para instrumentar somente as instruções que fazem leituras e escritas na memória. Sua função de instrumentação é mostrada a seguir.

```
53 /**
54  * Chamada para toda instrução e somente adiciona código de
55  * análise para instruções de leitura e escrita em memória.
56  */
57 VOID Instruction(INS ins, VOID *v)
58 {
59     /**
60      * O uso da função INS_InsertPredicatedCALL faz com
61      * que a instrumentação seja chamada se, e somente se,
62      * a instrução da aplicação alvo for de fato executada.
63      */
64     UINT32 memOperands = INS_MemoryOperandCount(ins);
65 
66     // Itera sobre cada operando de memória da instrução.
67     for (UINT32 memOp = 0; memOp < memOperands; memOp++)
68     {
69         if (INS_MemoryOperandIsRead(ins, memOp))
70         {
71             INS_InsertPredicatedCall(
72                 ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
73                 IARG_INST_PTR,
74                 IARG_MEMORYOP_EA, memOp,
75                 IARG_END);
76         }
77 
78         /**
79          * Importante notar que, em algumas arquiteturas,
80          * um único operando de memória pode ser tanto lido
81          * quanto escrito em uma mesma instrução. Nesse caso,
82          * a ferramenta realiza a instrumentação duas vezes,
83          * uma para a leitura e outra para a escrita.
84          */
85         if (INS_MemoryOperandIsWritten(ins, memOp))
86         {
87             INS_InsertPredicatedCall(
88                 ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
89                 IARG_INST_PTR,
90                 IARG_MEMORYOP_EA, memOp,
91                 IARG_END);
92         }
93     }
94 }
```

Na ferramenta, a identificação das instruções que referenciam a memória é feita através da chamada de **INS_MemoryOperandCount**, que devolve o número de operandos de memória que a instrução contém. Caso esse valor seja 0, o laço **for**, onde a instrumentação é aplicada, não é executado. Dentro desse laço, a ferramenta checa se o operando de memória da instrução é lido ou escrito, e adiciona a função de análise adequada a cada caso. Para essa ferramenta, em particular, as funções de análise simplesmente escrevem em um arquivo os endereços de memória referenciados.

A ferramenta "pinatrace.cpp"monitora o espaço de endereçamento através do monitoramento de cada instrução, verificando aquelas que alteram a memória diretamente. Entretanto, esse não é o único modo de inspecionar a memória usando Pin. O *framework* também fornece funções para leitura direta dos *bytes* em um endereço de memória, como mostrado a seguir.

```
1 PIN_SafeCopy(VOID* dst, const VOID* src, size_t size);
2 PIN_SafeCopyEx(VOID* dst, const VOID* src, size_t size,
3   EXCEPTION_INFO * pExceptInfo);
```

A função **PIN_SafeCopy** copia para o destino **dst** o número de *bytes* especificados em **size** da região de memória endereçada por **src**. O *framework* garante que a região de memória acessada tenha, de fato, o conteúdo original da memória da aplicação instrumentada. A função **PIN_SafeCopyEx** tem a mesma funcionalidade, além de fornecer informação detalhada caso algum erro tenha ocorrido. Nesse caso, **pExceptInfo** armazena as informações de erro.

Em conjunto com o tópico de inspeção de memória de aplicação, para obter o estado completo da máquina é necessário saber o valor que seus registradores assumem em algum momento. Pin possui estruturas de dados que fornecem uma abstração para esse processo de **recuperação de contexto**. Os tipos **CONTEXT** e **PHYSICAL_CONTEXT** armazenam todos os valores dos registradores presentes na máquina. Com eles é possível tanto fazer a leitura desses dados quanto modificá-los. A diferença entre os dois tipos se refere ao fato de que o primeiro fornece o contexto da máquina como a aplicação veria caso estivesse executando sem ser instrumentada. Já o segundo fornece o verdadeiro contexto da máquina, que é afetado pela interferência do *framework*. Para propósitos gerais, o programador deve fazer uso do tipo **CONTEXT**, que pode ser obtido nas rotinas de instrumentação. O tipo **PHYSICAL_CONTEXT** é utilizado para obter informações de depuração em caso de erros e exceções.

Para recuperar o contexto da máquina durante as rotinas de instrumentação, o programador pode optar pelos argumentos **IARG_CONTEXT** ou **IARG_CONST_CONTEXT**. A primeira opção permite leitura e escrita em registradores, enquanto a segunda permite somente leitura. Um exemplo de instrumentação com o uso dessas estruturas é mostrado a seguir.

```
1 VOID analysis(CONTEXT *ctxt) {
2   ...
3 }
4
5 ...
6
```

```
7  VOID instrumentation(INS ins, VOID *v) {
8      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) analysis,
9              IARG_CONST_CONTEXT, IARG_END);
10 }
```

Uma vez obtido o objeto do tipo **CONTEXT**, o programador pode obter o valor de qualquer registrador da máquina (para esse processo, o Pin só fornece suporte para arquiteturas Intel). A relação completa dos registradores das arquiteturas Intel, juntamente com seus nomes dentro do *framework* podem ser encontrados no Guia do Usuário do Pin [Intel 2018b]. A fim de ilustração, alguns exemplos são mostrados abaixo.

```
1  VOID analysis(CONTEXT *ctxt) {
2      ADDRINT registerEAX;
3      ADDRINT registerESI;
4
5      registerEAX = PIN_GetContextReg(ctxt, REG_EAX);
6      registerESI = PIN_GetContextReg(ctxt, REG_ESI);
7  }
```

A fim de controlar a interferência de fatores externos ao processo da aplicação, o programador pode estar interessado em monitorar **eventos assíncronos** como os sinais das plataformas Linux ou as chamadas de procedimento assíncronas (APC - *Asynchronous Procedure Calls*) das plataformas Windows. Atualmente, existem duas formas de tratar esses eventos utilizando Pin, como descrito a seguir.

- **PIN_InterceptSignal** Essa função permite à *Pintool* interceptar os sinais recebidos pela aplicação e tratá-los como desejar. Dessa forma, uma função é definida para ser executada sempre que um sinal é recebido pela aplicação. Esse sinal pode ou não ser encaminhado para o programa instrumentado. Para chamar essa função, o programador deve especificar qual sinal deverá ser interceptado e qual função deverá ser chamada caso o sinal seja enviado à aplicação. O exemplo abaixo ilustra a função. É importante notar que essa função só pode ser utilizada em sistemas Linux.

```
1      int main(int argc, char * argv[])
2      {
3        ...
4          PIN_InterceptSignal(SIGSEGV, Intercept, 0);
5          ...
6      }
7
8      static BOOL Intercept(THREADID, INT32, CONTEXT *,
9        BOOL, const EXCEPTION_INFO *, void *)
10     {
11         /**
12          * Tratamento do sinal.
13          */
14     }
```

Na *Pintool* acima, a função **Intercept** será executada sempre que a aplicação instrumentada receber um sinal **SIGSEGV**. As funções de tratamento de sinal passadas como argumento para a função **Pin_InterceptSignal** devem possuir o mesmo cabeçalho e receber, portanto, 6 valores. Esses valores representam, na ordem da lista de argumentos: o ID da *thread* que recebeu o sinal; o número do sinal; o contexto da máquina quando a aplicação recebeu o sinal; um booleano que indica se há uma função já registrada para tratar o sinal em questão; um objeto com uma descrição de exceção, caso o sinal represente uma; o valor passado para a função na chamada de **PIN_InterceptSignal** (no exemplo acima, 0). A função devolve *true* caso tenha sido bem sucedida ao interceptar o sinal.

- **PIN_AddContextChangeFunction** Essa função registra uma função que é executada sempre que a aplicação troca de contexto devido ao recebimento de algum sinal das plataformas Linux ou um APC do Windows. Geralmente, essa função é mais utilizada quando há somente a necessidade de notificar a *Pintool* sobre o recebimento de sinais e trocas de contexto. O código abaixo ilustra seu uso.

```
1   int main(int argc, char * argv[])
2   {
3     ...
4       PIN_AddContextChangeFunction(OnAPC, 0);
5       ...
6   }
7
8   static void OnAPC(THREADID, CONTEXT_CHANGE_REASON,
9     const CONTEXT *, CONTEXT *, INT32, VOID *)
10  {
11      /**
12       * Tratamento da troca de contexto.
13       */
14  }
```

Assim como na função anterior, as funções registradas através da chamada de **PIN_AddContextChangeFunction** devem ter seu cabeçalho definido com uma estrutura definida pelo Pin. Aqui, a lista de argumentos deve receber os tipos mostrados no código, que representam, na ordem da lista: o ID da *thread* que trocou de contexto; a causa da troca de contexto (APC, exceção, sinal, etc); o contexto da máquina antes da troca de contexto; o contexto da máquina após a mudança de contexto; uma informação adicional que depende da causa da troca de contexto; o valor passado para a função na chamada de **PIN_AddContextChangeFunction**.

Como visto na Seção 1.2.3.3, o Pin admite argumentos passados pela linha de comando no momento de sua invocação. Além disso, as próprias *Pintools* podem receber argumentos. Para isso, Pin fornece uma interface para adição de parâmetros obrigatórios e opcionais chamados de **KNOBs**. Cada KNOB define um argumento que a *Pintool* receberá pela linha de comando. O código abaixo ilustra o uso de KNOBs. Nele, o KNOB declarado especifica que o nome (*string*) do arquivo de saída da *Pintool* será definido pela

linha de comando através da *flag* **-o**. Além disso, também especifica um valor padrão caso a *flag* não esteja presente na invocação do *framework*.

```
1  KNOB<string> knobArquivoSaida(KNOB_MODE_WRITEONCE, "pintool",
2    "o", "output.log", "Nome do arquivo de saida");
3
4  static ofstream arquivoSaida;
5
6  int main(int argc, char *argv[])
7  {
8    ...
9    arquivoSaida.open(knobArquivoSaida.Value().c_str());
10   ...
11 }
```

1.2.6. Documentação e Suporte

Mesmo que existam guias e tutoriais do Pin no *website* do *framework*, muitas das funções de sua API não são muito intuitivas de serem usadas ou apresentam documentação insuficiente. Dessa forma, o programador deve procurar suporte em outros sítios. Abaixo encontra-se uma relação com as principais mídias de suporte ao Pin.

- **Guia do usuário**: O guia do usuário [Intel 2018b] do Pin é a documentação mais completa do *framework*. No guia, o programador encontra uma descrição de todas as funções da API do Pin, além de exemplos de *Pintools* e algumas discussões sobre o funcionamento do *framework*.

- **Tutorial do Pin**: Apresentado no simpósio CGO[4] de 2013, o tutorial do Pin [Devor 2013] fornece um compilado das principais funcionalidades do *framework*, com exemplos de código e discussões sobre sobrecarga de tempo de execução, desafios do *framework* tanto em plataformas Windows quanto Linux, etc.

- **Grupo *Pinheads***: Fórum de discussão [Intel 2004] sobre o *framework*. Nesse grupo, o programador tem acesso a um acervo de questões e respostas feitas por outros programadores que utilizam o Pin. Além disso, também pode se inscrever e postar suas próprias perguntas (fórum em inglês).

- ***Pintools* de exemplo**: Um bom método para estudar o funcionamento da interface do Pin e a criação de *Pintools* é através do próprio código fonte de outras *Pintools*. Por isso, as ferramentas de exemplo disponíveis no próprio *kit* do *framework* são referência de documentação. Essas ferramentas estão disponíveis a partir do diretório raiz onde o Pin é extraído, nas pastas "source/tools/SimpleExamples"e "source/tools/ManualExamples".

[4]Simpósio Internacional de Geração e Otimização de Código (*International Symposium on Code Generation and Optimization*).

1.3. Estudos de casos

Os casos estudados nesta seção compreendem a análise de situações em que os autores deste curso utilizaram a IDB para realizar trabalhos de pesquisa com foco na arquitetura x86. Trata-se, portanto, de exemplos de códigos de instrumentação criados pelos autores com finalidades diversas, todas na área de segurança de software. Esses códigos são apresentados e seus detalhes são minuciosamente explicados. O objetivo é ilustrar os conceitos apresentados na Seção 1.2 do texto, consolidando o aprendizado para que os leitores consigam desenvolver seus próprios códigos de instrumentação. Além disso, esta seção serve também como uma referência rápida para consulta de como se deve utilizar várias APIs importantes do Pin, já que muitas delas possuem uma documentação limitada.

1.3.1. Conceitos de segurança

Conforme mencionado, os estudos de casos envolvem a análise de programas e o desenvolvimento de protótipos de proteções elaborados no contexto de segurança de software. Para facilitar o pleno entendimento dos códigos de instrumentação a serem discutidos, antes apresentamos alguns conceitos de segurança de software que serão exercitados nesses códigos. O objetivo desta subseção é somente garantir que o leitor seja capaz de entender o contexto em que se encaixa cada um dos exemplos estudados. Não se pretende efetuar uma análise profunda desses assuntos, já que isso fugiria do escopo deste curso.

1.3.1.1. Estouro de Memória na Pilha

O conceito de estouro de memória na pilha, comumente referenciado pela expressão em inglês *Stack Buffer Overflow*, é importante para o entendimento dos três exemplos de código de IDB a serem apresentados nas Seções 1.3.2 a 1.3.4. Para compreendê-lo, é necessário assimilar como funciona a divisão do espaço de endereçamento virtual de um processo em segmentos. A Figura 1.3 ilustra a disposição geral dos segmentos de memória alocados para um processo executado por um sistema Linux em uma arquitetura x86 de 32 bits. Apesar de alguns detalhes serem ligeiramente diferentes em outros ambientes, a disposição e a função dos segmentos é bastante similar. Por questões didáticas, omitimos algumas informações, como aquelas referentes a espaços de deslocamento utilizados para a randomização do endereço inicial dos segmentos. Fazemos isso pois o intuito é focar no entendimento da função exercida pelo segmento de pilha.

Cada processo possui um espaço de endereços virtuais exclusivo, conforme representado na Figura 1.3. No caso de arquiteturas de 32 bits, esse espaço equivale a um bloco de 4GB. Os endereços virtuais, usados pelas instruções do programa, são mapeados para endereços reais da memória através de tabelas mantidas pelo *kernel* do Sistema Operacional (SO). Como o próprio kernel do SO é um processo, ele tem uma porção do espaço de endereços virtuais reservada para si dentro da faixa de endereços de todo processo (normalmente, do endereço 0xC0000000 ao endereço 0xFFFFFFFF). O restante (do endereço 0x00000000 ao endereço 0xBFFFFFFF), que equivale a 3GB, é destinado ao uso de cada processo de usuário. Os 3GB de memória dedicados ao processo são divididos em segmentos, de acordo com o tipo de dados que armazenam.

A *Pilha* guarda variáveis locais (pertencentes ao escopo de uma única função) e

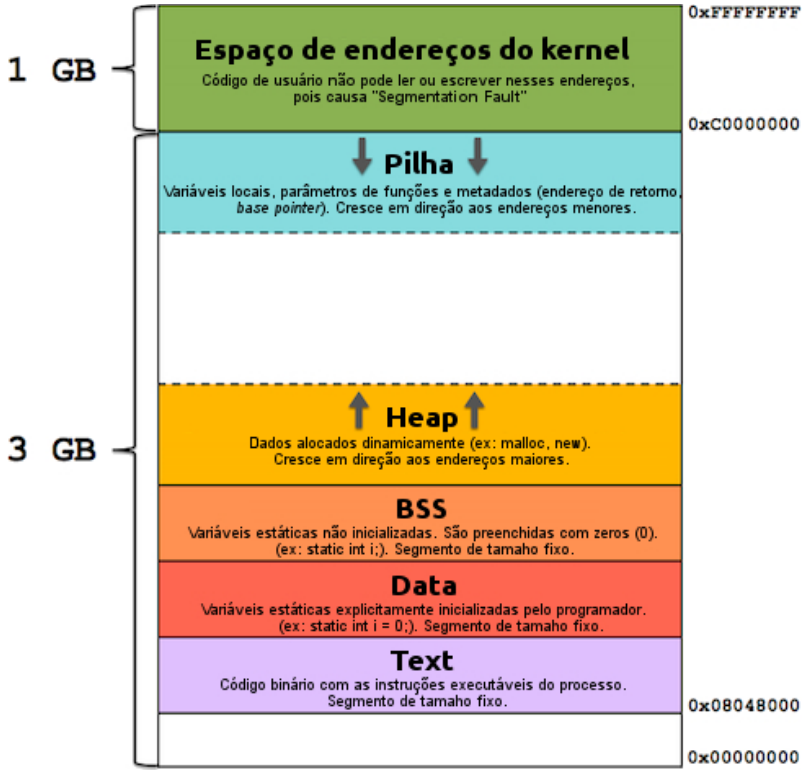

Figura 1.3. Distribuição dos segmentos de memória de um processo [Tolomei 2015].

metadados usados pelo processador para controlar chamadas de funções. O conjunto de valores armazenados para cada função executada por um programa é chamado de *Frame* da função. Cada *Frame* acomoda os valores atribuídos às variavés locais, os parâmetros de chamada da função e seu endereço de retorno. Dependendo do nível de otimização utilizado ao compilar o programa, a pilha pode armazenar também o endereço base do *Frame* da função para onde o fluxo de execução deve retornar. Nesse caso, o endereço base de cada função é mantido em um registrador denominado *Base Pointer*. O segmento de Pilha obedece às características de funcionamento da estrutura de dados de mesmo nome. Ou seja, dados novos são empilhados no topo da estrutura e são os primeiros a serem removidos. Para isso, o processador possui um registrador que indica a posição de topo da pilha, denominado *Stack Pointer*. A Pilha normalmente é posicionada nos endereços de memória mais altos, logo abaixo do espaço reservado para o kernel do SO, e cresce no sentido dos endereços de memória menores.

Durante a chamada de uma função qualquer, a pilha é alimentada com valores seguindo uma sequência de passos. Após a execução desses passos, a pilha ilustrada na parte esquerda da Figura 1.4 se transforma na pilha apresentada na parte direita da mesma figura. Posteriormente, quando a função chamada retorna, a pilha volta ao estado indicado no lado esquerdo da Figura 1.4. O processo de retorno de uma função consiste em realizar as seguintes operações, inversas ao procedimento de chamada: 1) depois de executar as instruções previstas em seu código e antes de retornar, a função chamada faz o registrador *Stack Pointer* apontar novamente para o endereço onde foi armazenado o endereço base

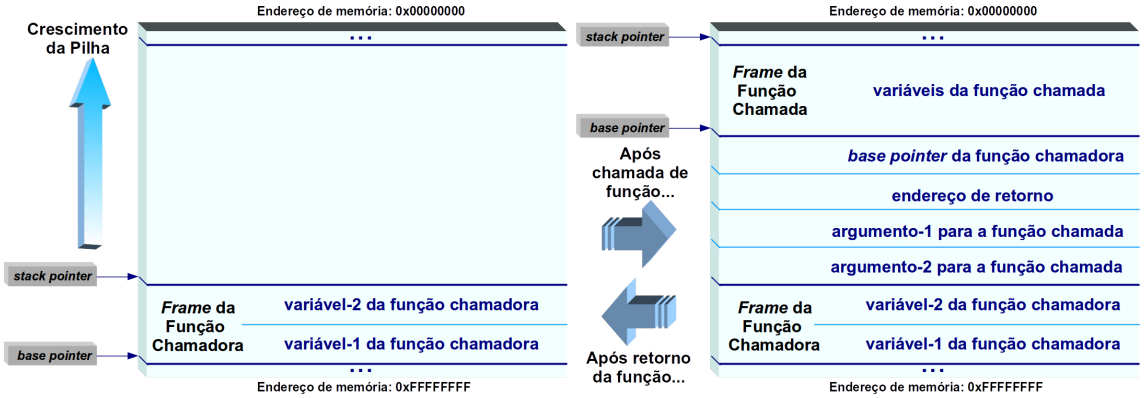

Figura 1.4. Estados da pilha em chamada e retorno de função [Tymburibá et al. 2012].

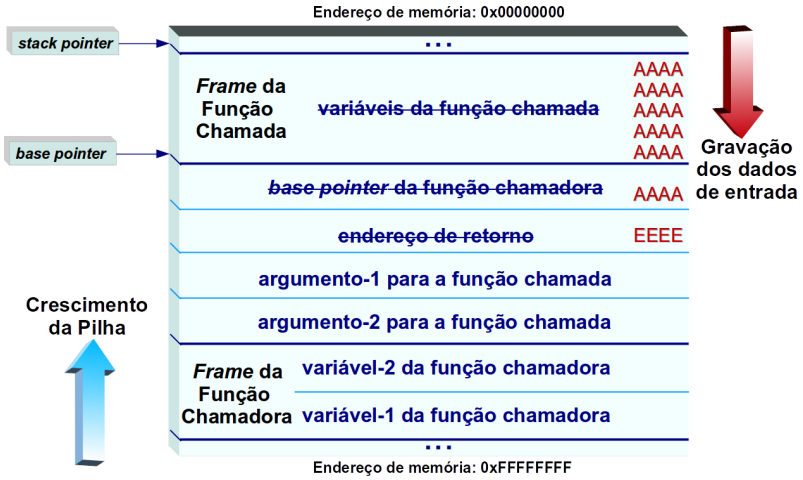

Figura 1.5. Sobrescrita de endereço de retorno ocasionada por um estouro de memória na pilha [Tymburibá et al. 2012].

do *Frame* pertencente à função chamadora; 2) o endereço base do *Frame* pertencente à função chamadora é desempilhado e restabelecido no registrador *Base Pointer*; 3) o endereço de retorno é desempilhado e o fluxo de execução é desviado para esse endereço.

O estouro de memória na pilha consiste em enviar como entrada para a aplicação uma quantidade de dados maior do que o espaço alocado na pilha para as variáveis locais da função. Para isso, basta que um usuário envie como entrada para uma aplicação que não checa o tamanho de suas entradas, uma sequência de dados maior do que a área reservada para essa entrada de usuário. Nesse cenário, uma entrada de tamanho devidamente calculado pode, portanto, extravasar os limites do *Frame* da função e sobrescrever seu endereço de retorno. Assim, quando a instrução de desvio para o endereço de retorno é executada, o fluxo de execução é transferido para um endereço escrito pelo atacante na pilha, conforme representado na Figura 1.5.

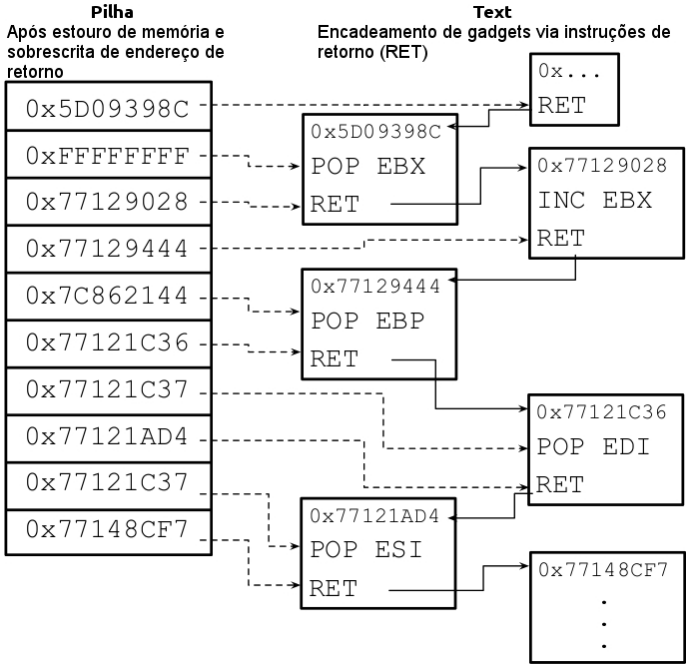

Figura 1.6. Exemplo de encadeamento de *gadgets* em um ataque do tipo ROP.

1.3.1.2. Return-Oriented Programming

Return-Oriented Programming, ou simplesmente ROP, é a principal técnica empregada atualmente por atacantes para executar códigos maliciosos em aplicações vulneráveis. Os exemplos de código de IDB apresentados nas Seções 1.3.2 a 1.3.4 fazem parte de trabalhos de pesquisa, encabeçados pelos autores deste curso, que visam propor proteções contra ataques ROP. Por conta disso, esta subseção é dedicada a introduzir os conceitos relacionados a ROP. Com isso, espera-se que os leitores compreendam a motivação e o contexto em que se enquadram os exemplos de IDB a serem apresentados.

Quando os primeiros ataques de corrupção da memória se tornaram públicos, era possível executar códigos maliciosos simplesmente inserindo-os na pilha junto com os dados que causavam o estouro da memória, exatamente como ilustrado na Figura 1.5 [One 1996]. Assim, ao induzir a sobrescrita de um endereço de retorno, bastava forçar a transferência do fluxo de execução para o endereço onde o código malicioso era armazenado na pilha. Com o intuito de impedir esse tipo de ataque, foi criado um marcador (bit de execução) que impede a execução de instruções localizadas fora do segmento executável do processo (*Text*) [Kanellos 2004]. Dessa forma, atualmente atacantes estão impedidos de transferir o fluxo de execução diretamente para códigos maliciosos inseridos na Pilha, já que esse segmento não possui permissão de execução, por ser unicamente destinado a dados. Diante desse obstáculo, o artifício encontrado por atacantes foi reutilizar instruções do próprio programa, já que os bytes correspondentes a essas instruções obrigatoriamente precisam ter permissão de execução, para que o processo funcione.

ROP baseia-se justamente no reuso de código para superar a proteção oferecida pelo bit de execução. Para isso, encadeia pequenos trechos de código da própria aplicação alvo (ou de bibliotecas utilizadas por ela), denominados *gadgets*. Para conseguir

esse encadeamento, a última instrução de cada trecho de código escolhido deve executar um desvio, conforme ilustrado na Figura 1.6. A ideia original do ROP utiliza *gadgets* finalizados com instruções de retorno (RET) para interligar as frações de código escolhidas [Shacham 2007]. Daí surgiu o nome da técnica. Posteriormente, foi demonstrado que também é possível encadear *gadgets* através de instruções de desvio indireto do tipo jump incondicional (JMP) [Checkoway et al. 2009, Chen et al. 2011] ou do tipo chamada de função (CALL) [Carlini and Wagner 2014, Göktas et al. 2014]. Através desse encadeamento de sequências de código, atacantes são capazes de superar a proteção do bit de execução para executar códigos maliciosos arbitrários.

A Figura 1.6 representa parte da estrutura de um *exploit* ROP disponível publicamente [Blake 2011]. Os dados inseridos pelo usuário na pilha, que ocasionam o estouro de memória e a consequente sobrescrita de um endereço de retorno, estão indicados à esquerda na figura, em formato hexadecimal. As linhas tracejadas que partem desses dados apontam qual instrução os utiliza. No lado direito da Figura 1.6, as caixinhas representam os *gadgets* que, encadeados, compõem o ataque. O encadeamento dessas pequenas sequências de instruções é representado na Figura 1.6 por linhas contínuas. Note que, nesse exemplo, a transição de um *gadget* para outro sempre ocorre ao executar uma instrução de retorno (RET). Essas instruções desviam o fluxo de execução (retornam) para o endereço posicionado no topo da pilha (indicados pelas linhas tracejadas). Como o conteúdo da pilha é inicialmente sobrescrito por dados introduzidos pelo usuário, dessa forma um atacante consegue controlar o encadeamento de *gadgets*, garantindo que computações arbitrárias possam ser executadas a seu critério.

1.3.2. Simulando o LBR

Last Branch Record (LBR) é um conjunto de registradores, presentes nos processadores modernos da Intel, que registram os últimos desvios de fluxo de execução tomados por um programa [Kleen b]. Esses dados de desvios recentes são importantes para diversas aplicações, como análise de desempenho e implementação de operações de memória transacional [Kleen a]. No contexto de segurança de software, pode ser usado para identificar situações onde uma determinada aplicação pode estar sendo alvo de um ataque ROP. Essa identificação se baseia no fato de que durante um ataque ROP, a correspondência ideal entre chamadas de procedimentos (um tipo particular de desvios) e instruções de retorno é geralmente quebrada. Em função dessas características de comportamento dinâmico inerentes a um ataque ROP, a utilização de uma ferramenta de IDB pode ser efetiva na identificação dessas tentativas de ataque. Esta seção detalha a implementação de uma *Pintool* que simula a estrutura do LBR de forma a utilizar a informação registrada para identificar possíveis ataques ROP, além de conseguir determinar informações úteis sobre a correspondência entre instruções de chamada de procedimento (CALLs) e instruções de retorno (RETs).

A ferramenta analisada será a **lbrmatch.cpp**. Primeiramente, o seu cabeçalho é definido como mostrado a seguir.

```
1 /**
2  * Autor: Hugo Sousa (hugosousa@dcc.ufmg.br)
3  *
4  * lbrmatch.cpp: Pintool usada para simular a estrutura de
```

```
5   * um LBR para instruções de chamada de função (CALL) e
6   * checar por correspondências entre elas e instruções de
7   * retorno.
8   */
9
10  #include "pin.H"
11
12  #include <iostream>
13  #include <fstream>
14
15  using namespace std;
16
17  KNOB<string> outFileKnob(KNOB_MODE_WRITEONCE, "pintool",
18    "o", "lbr_out.log", "Nome do arquivo de saída.");
19
20  KNOB<unsigned int> lbrSizeKnob(KNOB_MODE_WRITEONCE,
21    "pintool", "s", "16", "Número de entradas do LBR.");
```

Nas linhas 17 e 20 temos a definição de 2 parâmetros de linha de comando aceitos pela *Pintool*. O primeiro (*flag* **-o**) define o nome do arquivo de saída e o segundo (*flag* **-s**) define o número de entradas do LBR a ser simulado. De acordo com o manual da Intel, esse número varia nos valores de 4, 8 e 16 entradas [Guide 2011]. Quanto maior for esse número, maior será o número de chamadas de função aninhadas que o LBR será capaz de armazenar, como mostrado na parte seguinte do código.

```
23  /**
24   * Estrutura de dados LBR (Last Branch Record).
25   */
26
27  /**
28   * Uma entrada do LBR nessa Pintool será composta pelo endereço
29   * da instrução CALL e um booleano que indica se esse é um CALL
30   * direto (true) ou indireto (false).
31   */
32  typedef pair<ADDRINT, bool> LBREntry;
33
34  class LBR {
35  private:
36    LBREntry *buffer;
37    unsigned int head, tail, size;
38  public:
39    LBR(unsigned int size) {
40      this->size = size;
41      head = tail = 0;
42      buffer = (LBREntry*) malloc(sizeof(LBREntry) * (size + 1));
43    }
44
45    bool empty() {
46      return (head == tail);
```

```
47    }
48
49    void put(LBREntry item) {
50      buffer[head] = item;
51      head = (unsigned int) (head + 1) % size;
52
53      if (head == tail)
54        tail = (unsigned int) (tail + 1) % size;
55    }
56
57    void pop() {
58      if (empty())
59        return;
60
61      head = (unsigned int) (head - 1) % size;
62    }
63
64    LBREntry getLastEntry() {
65      if (empty())
66        return make_pair(0, false);
67
68      unsigned int index = (unsigned int) (head - 1) % size;
69
70      return buffer[index];
71    }
72  };
```

Cada entrada do LBR simulado nessa *Pintool* guarda dois valores: o endereço da instrução de chamada (CALL) de função e um valor que indica se essa instrução é um CALL direto ou indireto. Além disso, como o LBR tem um número limitado de entradas, caso ele esteja cheio no momento de uma nova adição (método **put**), a entrada mais antiga será sobrescrita. Prosseguindo, a *Pintool* define sua que a instrumentação será aplicada a cada traço da aplicação alvo e itera sobre os blocos básicos de cada um.

```
168     TRACE_AddInstrumentFunction(InstrumentCode, 0);
```

```
122 VOID InstrumentCode(TRACE trace, VOID *v) {
123   /**
124    * Função de instrumentação da Pintool.
125    *
126    * Cada bloco básico tem um único ponto de entrada
127    * e um único ponto de saída. Assim, CALLs e RETs
128    * somente podem ser encontradas ao fim de blocos
129    * básicos.
130    */
131
132   for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl =
        BBL_Next(bbl)) {
133
134     INS tail = BBL_InsTail(bbl);
```

```
135
136     if (INS_IsRet(tail)) {
137       INS_InsertCall(tail, IPOINT_BEFORE, (AFUNPTR) doRET,
138         IARG_BRANCH_TARGET_ADDR, IARG_END);
139     } else if (INS_IsCall(tail)) {
140       INS_InsertCall(tail, IPOINT_BEFORE, (AFUNPTR) doCALL,
141         IARG_INST_PTR, IARG_END);
142     }
143   }
144 }
```

Aqui, o código de análise é diferente para CALLs e RETs. Para os primeiros, a função de análise é **doCALL**, que recebe o endereço da instrução como parâmetro. Para os segundos, a função de análise é **doRET**, que recebe o endereço de retorno da instrução como parâmetro. Como mostrado no código seguir, a função **doCALL** simplesmente adiciona entradas ao LBR, enquanto **doRET** verifica se o endereço de retorno da instrução corresponde ao endereço imediatamente posterior ao endereço da instrução CALL no topo do LBR. Caso essa correspondência exista, a aplicação alvo está sob funcionamento regular e, caso contrário, pode estar sob um ataque ROP. É importante notar que nesse código de exemplo, a distinção entre CALLs diretos e indiretos não é explorada, entretanto, isso pode ser feito caso seja de interesse do programador, através da estrutura das entradas do LBR.

```
84  VOID doRET(ADDRINT returnAddr) {
85    /**
86     * Função de análise para instruções de retorno.
87     *
88     * @returnAddr: Endereço de retorno.
89     */
90
91    LBREntry lastEntry;
92
93    /**
94     * Instrução CALL anterior ao endereço de retorno
95     * pode estar de 2 a 7 bytes antes dele.
96     *
97     * callLBR é um objeto da classe LBR.
98     */
99    lastEntry = callLBR.getLastEntry();
100   for (int i = 2; i <= 7; i++) {
101     ADDRINT candidate = returnAddr - i;
102
103     if (candidate == lastEntry.first) {
104       calllLBRMatches++;
105       break;
106     }
107   }
108
109   callLBR.pop();
```

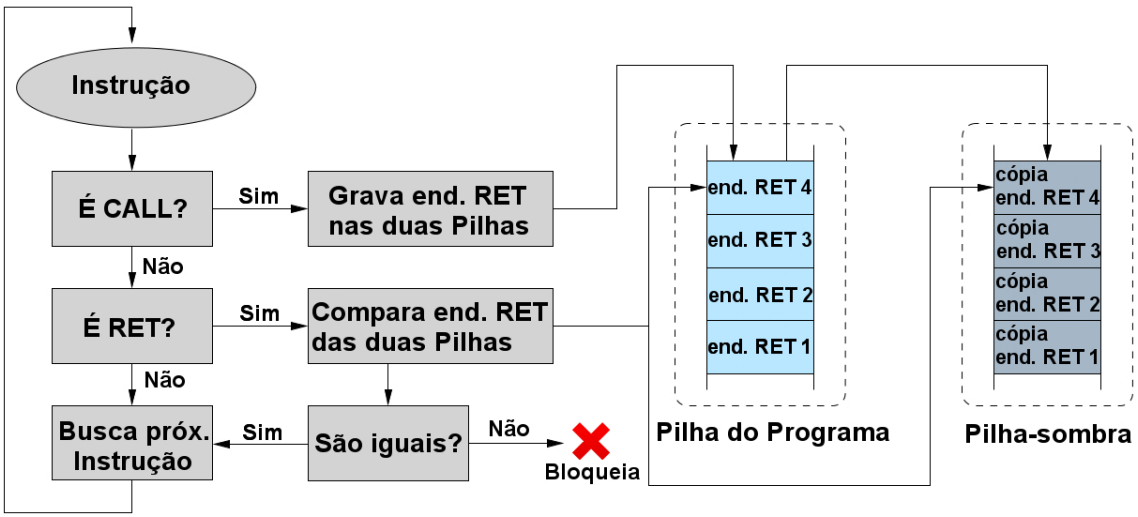

Figura 1.7. Proteção baseada em pilha-sombra [Davi et al. 2011].

```
110 }
111
112 VOID doCALL(ADDRINT addr) {
113   /**
114    * Função de análise para instruções de chamada de função.
115    *
116    * @addr: O endereço da instrução.
117    */
118
119   callLBR.put(make_pair(addr, true));
120 }
```

1.3.3. Pilha-Sombra

Pilha-Sombra é a tradução da expressão *Shadow-Stack*, do inglês, e refere-se a um conceito amplamente utilizado na área de segurança de software. Esse conceito estabelece que é possível impedir diversos ataques de corrupção da memória através da criação de uma segunda pilha de execução, protegida, onde são armazenadas cópias dos endereços de retorno de procedimentos. Dessa forma, pode-se comparar tais endereços com aqueles que a aplicação efetivamente utiliza para desviar o fluxo de execução, evitando que corrupções maliciosas do conteúdo da pilha acarretem no sequestro do fluxo de execução de um programa [Vendicator 2000]. Conforme ilustrado na Figura 1.7, sempre que uma instrução de chamada de uma função (CALL) é executada, além de ser anotado na tradicional pilha do processo, o endereço de retorno é também escrito em uma estrutura de pilha adicional (a pilha-sombra), protegida contra a escrita por instruções do processo. Essa proteção é necessária para evitar que um atacante altere os endereços de retorno nas duas pilhas, fazendo-os coincidir e, consequentemente, superando a proteção. No momento em que instruções de retorno são executadas (RET), checa-se a coincidência dos endereços entra as duas pilhas. Assim, se o endereço de retorno armazenado na pilha do processo for alterado, o ataque será identificado.

Diversas implementações desse conceito, inclusive utilizando IDB, já foram propostas na literatura, com o objetivo de evitar ataques como estouro de memória na pilha (Seção 1.3.1.1) e ROP (Seção 1.3.1.2). A seguir, apresentamos o código-fonte de uma implementação do conceito de Pilha-Sombra elaborada pelos autores deste curso no *framework* de instrumentação dinâmica Pin.

```
1  #include "pin.H"           // para usar APIs do Pin
2  #include <stack>           // para usar estrutura de dados Pilha
3  #include <stdio.h>         // para usar "fprintf" e "snprintf"
4  #include <stdlib.h>        // para usar "calloc"
5  #include <string.h>        // para usar "memset" e converter nú
       meros para string
6  #include <sstream>         // para converter números para string
7  #include <fstream>         // para imprimir no arquivo de saída
8  #include <sys/time.h>      // para registro do tempo de
       processador usado pelo algoritmo
9  #include <sys/resource.h> // para registro do tempo de
       processador usado pelo algoritmo
10
11 static TLS_KEY chave_tls;           // chave para acesso ao
       armazenamento local (TLS) das threads
12 static std::ofstream arquivo_saida; // arquivo onde a saída é
       escrita
13
14 void Uso(){
15    fprintf(stderr, "\nUso: pin -t <Pintool> [-o <
          NomeArquivoSaida>] [-logfile <NomeLogDepuracao>] -- <
          Programa alvo>\n\n"
16                    "Opções:\n"
17                    "  -o       <NomeArquivoSaida>\t"
18                    "Indica o nome do arquivo de saida (padrão:
                         $PASTA_CORRENTE/pintool.out)\n"
19                    "  -logfile <NomeLogDepuracao>\t"
20                    "Indica o nome do arquivo de log de depuracao
                          (padrão: $PASTA_CORRENTE/pintool.log)\n\n
                         ");
21 }
22
23 static string converte_double_string(double valor){
24    ostringstream oss;
25    oss << valor;
26    return(oss.str());
27 }
28
29 void IniciaThread(THREADID tid, CONTEXT * contexto, int flags,
       void * v){
30    stack<ADDRINT> *pilhaSombra = new stack<ADDRINT>();
31    PIN_SetThreadData(chave_tls, pilhaSombra, tid);
32 }
```

As linhas 1 a 9 importam as bibliotecas do Pin e de C++ necessárias. Em seguida, as linhas 11 e 12 criam duas variáveis globais. A primeira (chave_tls), é utilizada para diferenciar pilhas-sombras pertencentes a *threads* distintas em aplicações *multi-thread*. A segunda (arquivo_saida), referencia o arquivo onde os resultados são escritos. A função **Uso** simplesmente imprime uma mensagem no *prompt* de comandos indicando as opções de uso da Pintool. Por sua vez, a função **converte_double_string** faz o que seu nome sugere: recebe um valor do tipo *double* como parâmetro, converte-o para o tipo *string*, e devolve o *string* convertido.

Uma vez que cada *thread* deve possuir sua própria pilha-sombra, a função **IniciaThread** é usada para instanciar um objeto do tipo **stack** no TLS (*Thread Local Storage*). O TLS é um espaço de armazenamento criado pela API do Pin que oferece a opção de reservar e gerenciar, de forma eficiente, áreas de memória exclusivas para *threads*. Qualquer *thread* de um processo pode guardar e recuperar dados no seu próprio espaço, referenciando-o através de uma chave única (vide linha 11). Como o Pin cria um identificador (ID) único para cada thread, normalmente utiliza-se o próprio ID de cada thread como sua chave de acesso ao TLS. Como veremos adiante no código, a função **IniciaThread** é posteriormente registrada para ser executada sempre que uma nova *thread* for criada. A função **Fim** (linhas 34 a 47), é chamada quando a aplicação instrumentada termina de executar. Neste exemplo, ela calcula o tempo de CPU (usuário + sistema) consumido pelo processo e imprime os resultados no arquivo de saída.

Conforme ilustrado na Figura 1.7, o mecanismo de pilha-sombra exige duas funções de análise: uma para tratar as instruções de chamada de função (CALL) e outra para tratar as instruções de retorno de chamadas de funções (RET). Em nosso código, a função **AnaliseCALL** é registrada junto ao Pin para executar sempre que uma instrução CALL for executada. Essa função grava o endereço de retorno recebido como parâmetro na pilha-sombra da *thread* cujo ID também é recebido como parâmetro.

```
34 void Fim(INT32 codigo, void *v){
35    // salva instante atual para registrar o momento de término
36    time_t data_hora = time(0);
37
38    // calcula consumo total de tempo de CPU pelo processo (usuá
         rio + sistema)
39    struct rusage ru;
40    getrusage(RUSAGE_SELF, &ru);
41    double tempo_fim = static_cast<double>(ru.ru_utime.tv_sec) +
         static_cast<double>(ru.ru_utime.tv_usec * 0.000001) +
42                       static_cast<double>(ru.ru_stime.tv_sec) +
                            static_cast<double>(ru.ru_stime.tv_usec
                             * 0.000001);
43
44    // imprime no arquivo de saída os resultados
45    arquivo_saida << " #### Instrumentação finalizada em " <<
         converte_double_string(tempo_fim) << " segundos" << endl;
46    arquivo_saida << " #### Fim: " << string(ctime(&data_hora))
         << endl;
47 }
```

```
48
49 void PIN_FAST_ANALYSIS_CALL AnaliseCALL(THREADID tid, ADDRINT
       endereco){
50    // obtém ponteiro para a pilha sombra
51    stack<ADDRINT> *pilhaSombra = static_cast<stack<ADDRINT> *>(
          PIN_GetThreadData(chave_tls, tid));
52    // empilha o endereço de retorno na pilha sombra da thread
53    pilhaSombra->push(endereco);
54 }
```

Nas linhas 56 a 102, aparece a função **AnaliseRET**, registrada junto ao Pin para executar sempre que uma instrução RET for executada. É ela que checa se o endereço de retorno corresponde ao endereço anotado no topo da pilha-sombra. Note que nas linhas 80, 85, 95 e 100 são utilizadas travas de sincronização de *threads* do próprio Pin, para evitar que *threads* concorrentes escrevam simultaneamente no arquivo de saída.

```
56 void PIN_FAST_ANALYSIS_CALL AnaliseRET(THREADID tid, CONTEXT *
       contexto){
57    // inicializa endereço de retorno anotado na pilha original
          da thread
58    ADDRINT end_ret_original = 0;
59
60    // inicializa endereço de retorno anotado na pilha sombra
61    ADDRINT end_ret_sombra = 0;
62
63    // recupera endereço do topo da pilha original
64    ADDRINT * ptr_topo_pilha = (ADDRINT *) PIN_GetContextReg(
          contexto, REG_STACK_PTR);
65
66    // copia conteúdo do topo da pilha (endereço de retorno) para
          a variável inicializada
67    PIN_SafeCopy(&end_ret_original, ptr_topo_pilha, sizeof(
          ADDRINT));
68
69    // obtém ponteiro para a pilha sombra
70    stack<ADDRINT> *pilhaSombra = static_cast<stack<ADDRINT> *>(
          PIN_GetThreadData(chave_tls, tid));
71
72    // checa se há algum endereço anotado na pilha sombra
73    if(pilhaSombra->size() != 0){
74       // obtém endereço de retorno anotado no topo da pilha
             sombra
75       end_ret_sombra = pilhaSombra->top();
76       // se os endereços de retorno não coincidirem, sinaliza a
             suspeita de ataque ROP
77       if(end_ret_sombra != end_ret_original){
78
79          // Ativa uma trava interna do Pin para evitar que
              threads concorrentes escrevam simultaneamente no LOG
```

```
80            PIN_LockClient();
81
82            arquivo_saida << " #### Suspeita de ataque ROP! O
                 endereço de retorno " << hexstr(end_ret_original,
                 sizeof(ADDRINT)) << " não coincide com o endereço
                 anotado na pilha sombra (" << hexstr(end_ret_sombra,
                  sizeof(ADDRINT)) << ")" << endl;
83
84            // Libera trava
85            PIN_UnlockClient();
86         }
87         // desempilha o endereço anotado no topo da pilha sombra
88         pilhaSombra->pop();
89      }
90      else{
91         /* se uma instrução RET está sendo executada e não há
               endereço de retorno na pilha sombra,
92            significa que a paridade CALL-RET foi violada */
93
94         // Ativa uma trava interna do Pin para evitar que threads
               concorrentes escrevam simultaneamente no LOG
95         PIN_LockClient();
96
97         arquivo_saida << " #### Suspeita de ataque ROP! Não há
               nenhum endereço de retorno anotado na pilha sombra e o
               programa pretende retornar para o endereço de retorno "
                << hexstr(end_ret_original, sizeof(ADDRINT)) << endl;
98
99         // Libera trava
100        PIN_UnlockClient();
101     }
102  }
```

Em seguida, apresentamos a função **InstrumentaCodigo** (linhas 104 a 127), que na função **main** (linhas 129 a 165) é registrada junto ao Pin para executar a instrumentação do código. Ela registra junto ao Pin as funções **AnaliseCALL** e **AnaliseRET**, já apresentadas. Para isso, usa o conceito de BBLs (*Basic Blocks*), evitando a instrumentação de todas as instruções executadas. Ao invés disso, por questões de eficiência, checa apenas a última instrução de cada BBL, já que todo BBL possui um único ponto de saída. Note que utilizamos a opção **IPOINT_ANYWHERE** ao registrar a função **AnaliseCALL**. Essa opção permite que o Pin agende a chamada da função de análise em qualquer lugar do BBL, para obter um melhor desempenho. No caso do registro da função **AnaliseRET**, a opção **IPOINT_ANYWHERE** não pôde ser usada pois, entre as diversas instruções de um BBL, o topo da pilha pode variar em relação àquele válido no momento em que a instrução de retorno (RET) for executar. Também por questões de desempenho (passagem de argumentos otimizada pelo Pin), a opção **IARG_FAST_ANALYSIS_CALL** é utilizada em ambos os casos.

```
104 void InstrumentaCodigo(TRACE trace, void *v){
```

```
105     // percorre todos os BBLs
106     for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl =
            BBL_Next(bbl)){
107
108         // obtém a última instrução do BBL
109         INS ins = BBL_InsTail(bbl);
110
111         // se a última instrução do BBL for uma instrução CALL
112         if( INS_IsCall(ins) ){
113             // Registra a função "AnaliseCALL" para ser chamada
                    quando o BBL executar,
114             // passando o ID da thread e o endereço de retorno a
                    ser empilhado.
115             BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)
                    AnaliseCALL, IARG_FAST_ANALYSIS_CALL, IARG_THREAD_ID
                    , IARG_ADDRINT, INS_Address(ins) + INS_Size(ins),
                    IARG_END);
116         }
117         else{
118             // se a última instrução do BBL for uma instrução RET
119             if(INS_IsRet(ins)){
120                 // registra a função "AnaliseRET" para ser chamada
                        imediatamente antes de uma instrução RET executar
                        ,
121                 // passando o ID da thread e o ponteiro para o
                        contexto de execução (Pilha, regs, etc).
122                 INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)
                        AnaliseRET, IARG_FAST_ANALYSIS_CALL,
                        IARG_THREAD_ID, IARG_CONTEXT, IARG_END);
123             }
124         }
125     }
126 }
```

Finalmente, apresentamos a função **main** que, neste exemplo, tem a função de tratar opções da linha de comandos e arquivos de saída, inicializar o Pin e a aplicação a ser monitorada, instanciar o TLS e registrar as funções apresentadas anteriormente (**Fim**, **IniciaThread** e **InstrumentaCodigo**) junto ao Pin.

```
129 int main(int argc, char *argv[]){
130
131     // Usado para receber da linha de comandos (opção -o) o nome
            do arquivo de saída. Se não for especificado, usa-se o
            nome "Pintool.out"
132     KNOB<string> KnobArquivoSaida(KNOB_MODE_WRITEONCE, "pintool",
            "o", "pintool.out", "Nome do arquivo de saida");
133
134     // Inicializa o Pin e checa os parâmetros
135     if(PIN_Init(argc, argv)){
```

```
136         // imprime mensagem indicando o formato correto dos parâ
                metros e encerra
137         Uso();
138         return(1);
139     }
140 
141     // Abre o arquivo de saída no modo apêndice. Se não for
            passado um nome para o arquivo na linha de comandos, usa "
            Pintool.out"
142     arquivo_saida.open(KnobArquivoSaida.Value().c_str(), std::
            ofstream::out | std::ofstream::app);
143 
144     // obtem e imprime no arquivo de saída o momento em que a
            execução está iniciando
145     time_t data_hora = time(0);
146     arquivo_saida << endl << " #### Inicio: " << string(ctime(&
            data_hora));
147 
148     // obtém a chave para acesso à área de armazenamento local
            das threads (TLS)
149     chave_tls = PIN_CreateThreadDataKey(0);
150 
151     // registra a função "Fim" para ser executada quando a aplica
            ção for terminar
152     PIN_AddFiniFunction(Fim, NULL);
153 
154     // registra a função "IniciaThread" para ser executada quando
             uma nova thread for iniciar
155     PIN_AddThreadStartFunction(IniciaThread, NULL);
156 
157     // registra a função "InstrumentaCodigo" para instrumentar os
             "traces"
158     TRACE_AddInstrumentFunction(InstrumentaCodigo, NULL);
159 
160     // inicia a execução do programa a ser instrumentado e só
            retorna quando ele terminar
161     PIN_StartProgram();
162 
163     // encerra a execução do Pin
164     return(0);
165 }
```

1.3.4. Janela Deslizante

Janela deslizante é uma estratégia de proteção contra ataques ROP criada por um dos autores deste curso. Essa estratégia foi implementada em um protótipo denominado RipRop utilizando-se o instrumentador Pin [Tymburibá et al. 2015]. Nesta seção, são detalhados os procedimentos de implementação adotados, incluindo as abordagens de otimização de código empregadas com o intuito de minimizar o *overhead* do protótipo. Antes disso,

porém, explicamos os conceitos relacionados à solução de janela deslizante.

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 estado da memória ou de um registrador de forma a comprometer o ataque. Essa alteração de estado é comumente chamada por atacantes de "efeito colateral" de um *gadget*. A fim de evitar esses efeitos colaterais, quase sempre os *gadgets* escolhidos pelos atacantes são extremamente curtos.

Por evitar os mencionados "efeitos colaterais", ataques ROP acabam apresentando uma elevada concentração de instruções de desvio indireto em um curto espaço de tempo. Diante dessa constatação, diversos autores investiram esforços em uma estratégia de controle da frequência de instruções de desvio indireto como forma de detectar a execução de cadeias de *gadgets* [Chen et al. 2009, Davi et al. 2009, Min et al. 2013, Tymburibá et al. 2014]. Uma dessas estratégias é a janela deslizante, ilustrada na Figura 1.8, que 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 [Tymburibá et al. 2014]. Para definir o valor ideal desse limiar, é possível tanto estabelecer um valor padrão, com base na análise de um conjunto de aplicações, quanto efetuar uma etapa de análise estática do código com cada software que se pretende proteger, a fim de estabelecer o limiar máximo atingido por aquela aplicação [Emílio et al. 2015].

A lógica de funcionamento da janela deslizante segue o esquema ilustrado na Figura 1.9. Assumindo-se que a janela possui um tamanho N, pode-se dizer que a função da janela é permitir a contagem do número de desvios indiretos executados nas últimas N instruções. Nessa janela, as posições correspondentes às instruções de desvio indireto são anotadas com um bit 1 e as demais instruções são representadas pelo bit 0. Ao executar qualquer instrução, a janela precisa ser atualizada. Para evitar um *overhead* excessivo decorrente da análise de todas as instruções de um programa, novamente exploramos o conceito de bloco básico (*Basic Block*, ou BBL). Assim, insere-se um código de análise para um BBL, ao invés de avaliar cada instrução do programa, tornando a instrumentação mais eficiente. 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.

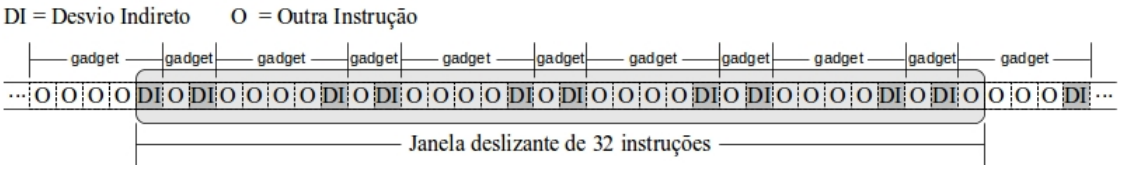

Figura 1.8. Funcionamento de uma janela deslizante durante a execução de *gadgets* ROP [Tymburibá et al. 2016].

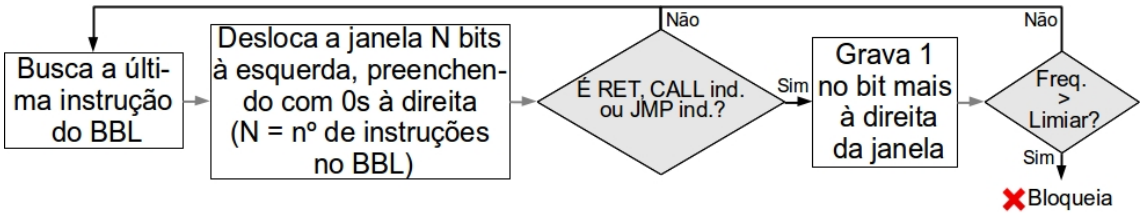

Figura 1.9. Lógica de funcionamento de uma janela deslizante para contagem de desvios indiretos executados [Tymburibá et al. 2014].

Depois de capturar a última instrução do BBL, a Pintool 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. 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, a Pintool 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, a Pintool sinaliza em um arquivo de saída a ocorrência de um ataque ROP e encerra a execução da aplicação.

A seguir, antes de apresentar o código que implementa a janela deslizante no Pin, discutimos brevemente algumas decisões de implementação tomadas com o objetivo de otimizar a execução da Pintool, para garantir o melhor tempo de execução possível.

A instrução POPCNT. Considerando-se que na arquitetura x86 existem instruções de hardware que permitem deslocar os bits de um registrador e contar a quantidades de bits ativos [Intel 2016], a execução das operações de atualização das janelas não acarretou em um *overhead* tão elevado quanto se fossem utilizados laços de repetição. A instrução SHL (deslocamento lógico para a esquerda, em direção aos bits mais significativos), por exemplo, usa um operando para indicar a quantidades de bits a serem deslocados de um registrador. Os bits menos significativos deslocados pela instrução são preenchidos com o valor zero (0). Isso evita a necessidade de executar um laço iterativo para deslocar a estrutura que representa a janela de instruções, o que exigiria um esforço computacional maior. Para contar a quantidade de bits ativos em um operando, foi utilizada a instrução POPCNT (Contagem de população). Trata-se de uma instrução introduzida em 2007 pela AMD em sua microarquitetura denominada Barcelona [AMD 2017]. Os processadores da família Intel Core introduziram a instrução POPCNT junto com a extensão do conjunto de instruções SSE4.2 [Intel 2018a]. Desde então, os processadores produzidos por esses e outros fabricantes contam com uma instrução equivalente. Assim como no deslocamento da janela de instruções, o uso dessa instrução de hardware evita o elevado custo computacional de percorrer a janela para contar os bits ativos, garantindo uma considerável melhoria de desempenho.

Linhas de Cache da CPU. Uma questão estrutural inerente à arquitetura dos processadores que costuma ocasionar a degradação do desempenho de aplicações que executam várias *threads* é o problema do falso compartilhamento (*false sharing*). Essa situação ocorre quando múltiplas *threads* acessam diferentes partes de uma mesma linha de cache da CPU e ao menos um desses acessos é uma escrita [Intel 2018b]. Para manter a coerência dos dados em memória, o computador copia os dados da cache de uma CPU para a outra, mesmo que as *threads* concorrentes não estejam efetivamente compartilhando nenhum dado. Normalmente, problemas de falso compartilhamento entre *threads* podem ser evitados ajustando-se o tamanho das estruturas de dados para que cada estrutura ocupe uma linha de cache diferente. Nossa implementação da janela deslizante trata aplicações que lançam múltiplas *threads* através da criação de uma janela de instruções independente para cada *thread*, assim como realizado com a pilha-sombra, na Seção 1.3.3. No entanto, isso não impede que mais de uma janela ocupe a mesma linha de cache da CPU, o que pode eventualmente acarretar no problema de falso compartilhamento. Para evitar essa situação indesejada, forçamos a alocação de uma estrutura de dados inútil junto com a estrutura que representa a janela de instruções. O tamanho dessa estrutura de dados inerte é calculado para que a soma da área de armazenamento ocupada pela janela com o espaço ocupado por essa estrutura inútil corresponda ao tamanho exato de uma linha de cache da CPU. Assim, a Pintool força o armazenamento dos dados de cada *thread* em uma linha de cache diferente, evitando o problema de falso compartilhamento. Para garantir o correto funcionamento desse mecanismo de prevenção do problema de falso compartilhamento, deve-se registrar no código-fonte da Pintool o tamanho do complemento da linha de cache (linha 11 no código abaixo). Em ambientes Linux, uma forma de identificar o tamanho da linha de cache (em bytes) usada pelo processador do equipamento onde se pretende usar a Pintool é executar o comando *getconf LEVEL1_DCACHE_LINESIZE*.

O código da Pintool que implementa a janela deslizante está listado abaixo. Assim como no exemplo da pilha-sombra, o código começa com a importação de bibliotecas e a declaração de variáveis globais (linhas 1 a 16). Em seguida (linhas 18 a 21), a estrutura usada para representar a janela de instruções das *threads* é declarada, com capacidade de 32 bits (1 bit por instrução). Note que o arranjo **lixo** é criado com o tamanho exato para forçar a janela de cada *thread* a ocupar sua própria linha no cache da CPU, a fim de evitar perdas de desempenho decorrentes do problema de *false sharing*.

```
1  #include "pin.H"           // para usar APIs do Pin
2  #include <stdio.h>         // para usar I/O
3  #include <stdlib.h>        // para usar exit()
4  #include <string.h>        // para converter números para string
5  #include <sstream>         // para converter números para string
6  #include <fstream>         // para imprimir no arquivo de saída
7  #include <sys/time.h>      // para registro do tempo de
       processador usado pelo algoritmo
8  #include <sys/resource.h> // para registro do tempo de
       processador usado pelo algoritmo
9
10 static const UINT32 tam_janela = 32;              // constante
       que indica o tamanho da janela em bits
```

```
11 static const UINT32 COMPLEMENTO_LINHA_CACHE = 60; // tamanho da
      linha da cache (64 bytes) - tamanho da janela
12 static const UINT32 limiar_padrao = 10;           // valor de
      limiar padrao pré-estabelecido para a janela de 32
13 static const UINT32 MASCARA_UM = 1;               // máscara
      usada para setar o bit menos significativo da janela
14 static std::ofstream arquivo_saida;               // arquivo
      onde a saída é escrita
15 static UINT32 limiar;                             // valor de
      limiar checado durante a execução
16 static TLS_KEY chave_tls;                         // chave para
      acesso ao armazenamento local (TLS) das threads
17
18 struct JanelaThread{
19    UINT32 janela_bits; // buffer que guarda os bits (janela)
20    UINT8 lixo[COMPLEMENTO_LINHA_CACHE]; // área inútil usada
         para ocupar uma linha inteira da cache
21 };
```

Assim como no exemplo da pilha-sombra, algumas funções existem para exibir uma mensagem sobre o uso correto dos parâmetros de linha de comando e para converter tipos de valores (linhas 23 a 43). A função **IniciaThread** (linhas 45 a 54), chamada ao iniciar uma nova thread, aloca espaço para a janela da nova *thread* no TLS.

```
23 void Uso(){
24    fprintf(stderr, "\nUso: pin -t <Pintool> [-l <Limiar>] [-o <
         NomeArquivoSaida>] [-logfile <NomeLogDepuracao>] -- <
         Programa alvo>\n\n"
25                    "Opções:\n"
26                    "  -l       <Limiar>\t"
27                    "  -o       <NomeArquivoSaida>\t"
28                    "Indica o nome do arquivo de saida (padrão:
                         $PASTA_CORRENTE/pintool.out)\n"
29                    "  -logfile <NomeLogDepuracao>\t"
30                    "Indica o nome do arquivo de log de depuracao
                          (padrão: $PASTA_CORRENTE/pintool.log)\n\n
                         ");
31 }
32
33 static string converte_double_string(double valor){
34    ostringstream oss;
35    oss << valor;
36    return(oss.str());
37 }
38
39 static string converte_ulong_string(unsigned long int valor){
40    ostringstream oss;
41    oss << valor;
42    return(oss.str());
```

```
43 }
44
45 void IniciaThread(THREADID thread_id, CONTEXT *
      contexto_registradores, int flags_SO, void *v){
46    // Aloca espaço para a janela e guarda endereço em apontador
47    JanelaThread* janela_ptr = new JanelaThread;
48
49    // Inicializa a janela
50    janela_ptr->janela_bits = 0x00000000;
51
52    // Armazena a janela na área de armazenamento (TLS) da thread
53    PIN_SetThreadData(chave_tls, janela_ptr, thread_id);
54 }
```

A função **Fim** (linhas 56 a 69) tem o mesmo propósito da função de mesmo nome no exemplo anterior (Seção 1.3.3). Já a função **DeslocaJanela** (linhas 71 a 108) corresponde ao código de análise. Nessa rotina, a janela deslizante é deslocada, de acordo com os parâmetros recebidos, e um alerta é emitido em caso de superação do limiar de desvios indiretos. O primeiro parâmetro recebido por essa função, **thread_id**, é usado para recuperar a janela da *thread*. O segundo parâmetro, **num_bits_shift**, indica o número de instruções executadas no BBL. Esse valor corresponde ao número de bits que devem ser deslocados na janela de instruções. O terceiro parâmetro, **desvio_indireto**, indica se a última instrução do BBL é um desvio indireto (TRUE) ou não (FALSE). Se a última instrução do BBL for um desvio indireto, o bit menos significativo da janela é ligado.

```
56 void Fim(INT32 codigo, void *v){
57    // salva instante atual para registrar o momento de término
58    time_t data_hora = time(0);
59
60    // calcula consumo total de tempo de CPU pelo processo (usuá
         rio + sistema)
61    struct rusage ru;
62    getrusage(RUSAGE_SELF, &ru);
63    double tempo_fim = static_cast<double>(ru.ru_utime.tv_sec) +
         static_cast<double>(ru.ru_utime.tv_usec * 0.000001) +
64                       static_cast<double>(ru.ru_stime.tv_sec) +
                            static_cast<double>(ru.ru_stime.tv_usec
                             * 0.000001);
65
66    // imprime no arquivo de saída os resultados
67    arquivo_saida << " #### Fim: " << string(ctime(&data_hora));
68    arquivo_saida << " #### Instrumentação finalizada em " <<
         converte_double_string(tempo_fim) << " segundos" << endl
         << endl;
69 }
70
71 void PIN_FAST_ANALYSIS_CALL DeslocaJanela(THREADID thread_id,
      UINT32 num_bits_shift, BOOL desvio_indireto){
72    // variável auxiliar: número de bits setados na janela
```

```
73      UINT32 num_bits_setados;
74
75      // obtém ponteiro para a janela da thread
76      JanelaThread *janela_ptr = static_cast<JanelaThread *>(
            PIN_GetThreadData(chave_tls, thread_id));
77
78      // se o número de bits a deslocar é menor que 32 e maior que
            0
79      if(num_bits_shift < tam_janela){
80          // executa shift de num_bits_shift
81          janela_ptr->janela_bits <<= num_bits_shift;
82      }
83      else{ // se o número de bits a deslocar é maior ou igual ao
            tamanho da janela
84          // zera o buffer
85          janela_ptr->janela_bits = 0x00000000;
86      }
87
88      // seta o bit menos signif. da janela se a última instrução
            do BBL for um desvio indireto
89      if(desvio_indireto){
90          janela_ptr->janela_bits |= MASCARA_UM;
91      }
92
93      // usa a instrução de HW POPCNT para contar o número de bits
            setados na janela
94      num_bits_setados = __builtin_popcount(janela_ptr->janela_bits
            );
95
96      // se o número de bits setados for maior do que o limiar
97      if(num_bits_setados > limiar){
98          // Ativa uma trava interna do Pin para evitar que threads
                concorrentes escrevam simultaneamente no arquivo de saí
                da
99          PIN_LockClient();
100
101         // imprime mensagem no arquivo de saída
102         arquivo_saida << " ####  Suspeita de ataque ROP! O limiar
                de " << converte_ulong_string(static_cast<unsigned long
                 int>(limiar)) <<
103                          " foi superado pelo seguinte valor: " <<
                             converte_ulong_string(static_cast<
                             unsigned long int>(num_bits_setados))
                             << endl;
104
105         // Libera trava
106         PIN_UnlockClient();
107     }
108 }
```

A função **InstrumentaCodigo** (linhas 110 a 121) é registrada junto ao Pin para executar a instrumentação do código. Ela registra a função **DeslocaJanela** junto ao Pin para ser disparada sempre que a última instrução de um BBL estiver para ser executada.

```
110 void InstrumentaCodigo(TRACE trace, void *v){
111    // percorre todos os BBLs
112    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl =
          BBL_Next(bbl)){
113       // se a última instrução do BBL for um desvio indireto
114       if( INS_IsIndirectBranchOrCall(BBL_InsTail(bbl)) ){
115          BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)
                DeslocaJanela, IARG_FAST_ANALYSIS_CALL,
                IARG_THREAD_ID, IARG_UINT32, BBL_NumIns(bbl),
                IARG_BOOL, TRUE, IARG_END);
116       }
117       else{// se a última instrução do BBL NÃO for um desvio
             indireto
118          BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)
                DeslocaJanela,  IARG_FAST_ANALYSIS_CALL,
                IARG_THREAD_ID, IARG_UINT32, BBL_NumIns(bbl),
                IARG_BOOL, FALSE, IARG_END);
119       }
120    }
121 }
```

Finalmente, apresentamos a função **main** (linhas 123 a 162), que executa um papel muito semelhante à função de mesmo nome do exemplo anterior (Seção 1.3.3).

```
123 int main(int argc, char *argv[]){
124    // Usado para receber da linha de comandos (opção -o) o nome
          do arquivo de saída. Se não for especificado, usa-se o
          nome "Pintool.out"
125    KNOB<string> KnobArquivoSaida(KNOB_MODE_WRITEONCE, "pintool",
           "o", "pintool.out", "Nome do arquivo de saida");
126
127    // Usado para receber da linha de comandos (opção -l) o valor
           de limiar a ser usado. Se não for especificado, usa-se o
          limiar padrão
128    KNOB<UINT32> KnobEntradaLimiar(KNOB_MODE_WRITEONCE, "pintool"
          , "l", converte_ulong_string(static_cast<unsigned long int
          >(limiar_padrao)), "Valor de limiar a ser usado pela
          protecao");
129
130    // Inicializa o Pin e checa os parâmetros
131    if(PIN_Init(argc, argv)){
132       // imprime mensagem indicando o formato correto dos parâ
             metros e encerra
133       Uso();
134       return(1);
135    }
```

```
136
137     // Abre o arquivo de saída no modo apêndice. Se não for
            passado um nome para o arquivo na linha de comandos, usa "
            Pintool.out"
138     arquivo_saida.open(KnobArquivoSaida.Value().c_str(), std::
            ofstream::out | std::ofstream::app);
139
140     // obtem e imprime no arquivo de saída o momento em que a
            execução está iniciando
141     time_t data_hora = time(0);
142     arquivo_saida << endl << " #### Inicio: " << string(ctime(&
            data_hora));
143
144     // Obtém valor do limiar. Se não for passado na linha de
            comandos, usa limiar padrão
145     limiar = KnobEntradaLimiar.Value();
146     arquivo_saida << " #### Valor do limiar: " <<
            converte_ulong_string(static_cast<unsigned long int>(
            limiar)) << endl;
147
148     // registra a função "Fim" para ser executada quando a aplica
            ção for terminar
149     PIN_AddFiniFunction(Fim, NULL);
150
151     // registra a função "IniciaThread" para ser executada quando
             uma nova thread for iniciar
152     PIN_AddThreadStartFunction(IniciaThread, NULL);
153
154     // registra a função "InstrumentaCodigo" para instrumentar os
             "traces"
155     TRACE_AddInstrumentFunction(InstrumentaCodigo, NULL);
156
157     // inicia a execução do programa a ser instrumentado e só
            retorna quando ele terminar
158     PIN_StartProgram();
159
160     // encerra a execução do Pin
161     return(0);
162 }
```

1.4. Considerações finais

Neste trabalho, a técnica de Instrumentação Dinâmica de Binários e o *framework* Pin foram apresentados. Com o material aqui reunido, o leitor tem o arcabouço necessário para escrever suas próprias ferramentas de IDB. O *framework* Pin mostra-se uma boa opção para o programador de ferramentas de IDB pois, além dos motivos apresentados na Seção 1.1.2.3, ele possui uma grande comunidade online que mantém-se ativa. Diante das capacidades do *framework*, espera-se que ele continue a ser muito utilizado na academia

ou até mesmo que seja ainda mais usado. Isso porque, se as tecnologias emergentes tendem a ser mais complexas, o uso de um *framework* como Pin se torna cada vez mais necessário para estudá-las (Seção 1.1.1). De fato, a variedade de usos de ferramentas de análise dinâmica existentes atualmente indica que há muito progresso tecnológico sendo feito nessa área. A Figura 1.10 ilustra essa variedade.

De maneira geral, através da Figura 1.10, é possível identificar pontos de interesse de pesquisa que podem crescer em relevância nos próximos anos. O primeiro deles se refere ao uso de técnicas de análise dinâmica em aplicações *Web*. Se por um lado a análise estática dessas aplicações era suficiente no passado, atualmente existe a necessidade do uso de análise dinâmica. Isso se deve ao fato de que hoje essas aplicações tendem a utilizar cada vez mais interfaces de interação com o usuário e códigos *Javascript*, que tornam ainda mais complexa a estrutura de navegação [Cornelissen et al. 2009]. Além disso, vemos que os artigos que tratam do tema de análise dinâmica costumam focar nos métodos de avaliação por estudos de caso. À medida que as técnicas de análise dinâmica se tornem mais populares, possivelmente surgirá a necessidade de aumentar o uso de *benchmarks* para avaliação do desempenho das ferramentas, já que a sobrecarga em tempo de execução é fator importante quando se fala de análises dinâmicas e, em particular, instrumentação dinâmica de binários.

Em relação às atividades em que a análise dinâmica são empregadas, as visualizações são as mais populares. Conforme discutido na Seção 1.1.1 e considerando a crescente complexidade dos sistemas computacionais, esse fato provavelmente tenderá a

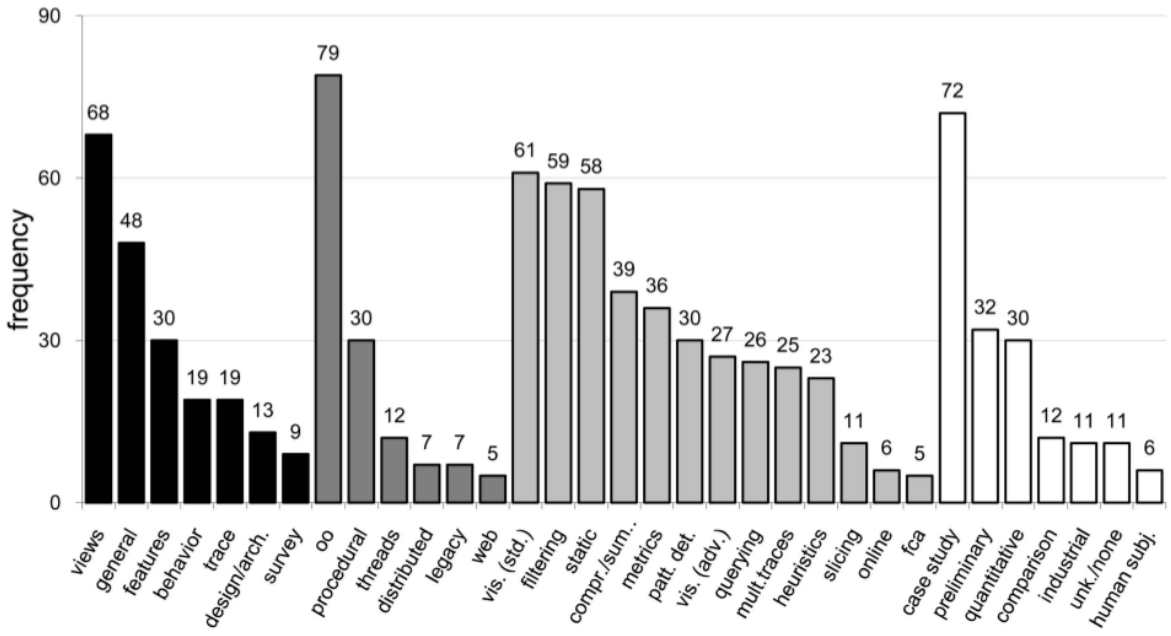

Figura 1.10. Frequência de atributos encontrados nos artigos sobre análise dinâmica até o ano de 2009 [Cornelissen et al. 2009]. As cores indicam a categoria do atributo. Da esquerda para a direita: atividade (preto), aplicações alvo (cinza escuro), método de análise dinâmica utilizado na condução da atividade (cinza claro) e método de avaliação através do qual a abordagem é validada (branco). Essa figura tem como referência 176 **artigos analisados.**

ser mantido no futuro. Entretanto, outras aplicações, como aquelas discutidas na Seção 1.1.2.1, deverão se popularizar com a difusão das técnicas de análise dinâmica.

Por fim, a Figura 1.10 revela que o uso de informações estáticas é o terceiro método mais utilizado. Levando em consideração que as análises estáticas também possuem vantagens (Seção 1.1.2.2), os dados da figura mostram que a abordagem mista (ou híbrida) tem ganhado popularidade. Isso acontece porque, utilizando essa abordagem, é possível combinar vantagens tanto das análises dinâmicas quanto estáticas.

Referências

[AMD 2017] AMD (2017). AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. `https://support.amd.com/TechDocs/24594.pdf`.

[Blake 2011] Blake (2011). MY MP3 Player 3.0 m3u Exploit DEP Bypass. `http://www.exploit-db.com/exploits/17854/`.

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

[Checkoway et al. 2009] Checkoway, S., Feldman, A. J., Kantor, B., Halderman, J. A., Felten, E. W., and Shacham, H. (2009). Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. *EVT/WOTE*, 2009.

[Chen 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 *International Conference on Information Systems Security*, pages 163–177. Springer.

[Chen et al. 2011] 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 6th ACM Symposium on Information, Computer and Communications Security*, pages 20–29. ACM.

[Cornelissen et al. 2009] Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702.

[Davi et al. 2009] Davi, L., Sadeghi, A.-R., and Winandy, M. (2009). Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54. ACM.

[Davi et al. 2011] Davi, L., Sadeghi, A.-R., and Winandy, M. (2011). Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM.

[Devor 2013] Devor, T. (2013). Pin cgo tutorial. *CGO'13*.

[Emílio et al. 2015] Emílio, R., Tymburibá, M., and Pereira, F. (2015). Inferência estática da frequência máxima de instruções de retorno para detecção de ataques rop. In *Anais do XV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 2–15. SBC.

[Ferreira et al. 2014] Ferreira, M. F. T. et al. (2014). Rip-rop: uma proteção contra ataques de execução de código arbitrário baseados em return-oriented programming.

[Garnett 2003] Garnett, T. (2003). *Dynamic optimization of IA-32 applications under DynamoRIO*. PhD thesis, Massachusetts Institute of Technology.

[GNU 2018] GNU (2018). The GNU Operating System and The Free Software Movement. `https://www.gnu.org/home.en.html`.

[Göktas et al. 2014] 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.

[Guide 2011] Guide, P. (2011). Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2.

[Intel 2004] Intel (2004). Pin Dynamic Binary Instrumentation Tool - Yahoo Groups. `https://groups.yahoo.com/neo/groups/pinheads/info`.

[Intel 2012] Intel (2012). Pin - A Dynamic Binary Instrumentation Tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[Intel 2016] Intel (2016). Intel 64 and IA-32 Architectures Software Developer's Manual. `https://software.intel.com/en-us/articles/intel-sdm`.

[Intel 2018a] Intel (2018a). Intel Architecture Instruction Set Extensions and Future Features Programming Reference. `https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf`.

[Intel 2018b] Intel (2018b). Pin 3.6 user guide. `https://software.intel.com/sites/landingpage/pintool/docs/97554/Pin/html/`.

[Kanellos 2004] Kanellos, M. (2004). AMD, Intel put antivirus tech into chips. `https://www.zdnet.com/article/amd-intel-put-antivirus-tech-into-chips/`.

[Kleen a] Kleen, A. LWN.net advanced usage of last branch records. `https://lwn.net/Articles/680996/`. Acessado em: 19-04-2018.

[Kleen b] Kleen, A. LWN.net an introduction to last branch records. `https://lwn.net/Articles/680985/`. Acessado em: 19-04-2018.

[Laurenzano et al. 2010] Laurenzano, M. A., Tikir, M. M., Carrington, L., and Snavely, A. (2010). Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. 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 *Acm sigplan notices*, volume 40, pages 190–200. ACM.

[Min et al. 2013] 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.

[Nethercote 2004] Nethercote, N. (2004). Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory.

[Nethercote and Seward 2007a] Nethercote, N. and Seward, J. (2007a). How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM.

[Nethercote and Seward 2007b] Nethercote, N. and Seward, J. (2007b). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

[One 1996] One, A. (1996). Smashing the stack for fun and profit. *http://phrack.org/issues/49/14.html*.

[Reddi et al. 2004] Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. (2004). Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, page 22. ACM.

[Rodríguez et al. 2014] Rodríguez, R. J., Artal, J. A., and Merseguer, J. (2014). Performance evaluation of dynamic binary instrumentation frameworks. *IEEE Latin America Transactions*, 12(8):1572–1580.

[Romer et al. 1997] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., and Chen, B. (1997). Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop*, volume 1997, pages 1–8.

[Shacham 2007] 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 conference on Computer and communications security*, pages 552–561. ACM.

[Tolomei 2015] Tolomei, G. (2015). In-Memory Layout of a Program (Process). `https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/`.

[Tymburibá et al. 2015] Tymburibá, M., Emilio, R., and Pereira, F. (2015). Riprop: A dynamic detector of rop attacks. In *Proceedings of the 2015 Brazilian Congress on Software: Theory and Practice*, pages 1–8.

[Tymburibá et al. 2016] Tymburibá, M., Moreira, R. E., and Quintão Pereira, F. M. (2016). Inference of peak density of indirect branches to detect rop attacks. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 150–159. ACM.

[Tymburibá et al. 2012] Tymburibá, M., Rocha, T., Martins, G., Feitosa, E., and Souto, E. (2012). Análise de vulnerabilidades em sistemas computacionais modernos: Conceitos, exploits e proteções. In dos Santos, A., Santin, A., Maziero, C., and da S. Gonçalves, P. A., editors, *Minicursos do XII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, chapter 1, pages 2–51. Sociedade Brasileira de Computação, Porto Alegre.

[Tymburibá et al. 2014] Tymburibá, M., Santos, A., and Feitosa, E. (2014). Controlando a frequência de desvios indiretos para bloquear ataques rop. In *Anais do XIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 223–236. SBC.

[Uh et al. 2006] Uh, G.-R., Cohn, R., Yadavalli, B., Peri, R., and Ayyagari, R. (2006). Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*. Citeseer.

[Uhlig and Mudge 1997] Uhlig, R. A. and Mudge, T. N. (1997). Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170.

[Vendicator 2000] Vendicator (2000). Stack Shield: A stack smashing technique protection tool for Linux. `http://www.angelfire.com/sk/stackshield/`.