Mensurando a Eficiência do Controle de Integridade de Fluxo Através do Contexto Dinâmico

Pedro T. Delboni[1] **, João Moreira**[1] **, Sandro Rigo**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
13.083-852 – Campinas – SP – Brazil

[pedro.delboni, joao.moreira]@lsc.ic.unicamp.br, sandro@ic.unicamp.br

Abstract. *Control-flow Hijack is a type of attack that execute arbitrary computations through the corruption of code pointers in the target system. These vulnerabilities are mitigated through control-flow integrity policies that restrict the possible targets of these pointers. Since it is inviable to compute a perfect restriction, metrics have been proposed to compare the efficiency of different policies. In this paper we propose two metrics that take into consideration the dynamic context of a program's execution and show how they can be used to direct the development of more restrictive control-flow policies.*

Resumo. *Sequestro de controle de fluxo é um ataque que explora corromper a memória responsável por um ponteiro de código para executar computação arbitrária no sistema alvo. Essas vulnerabilidades são mitigadas por políticas de integridade de fluxo de controle (CFI) que restringem os possíveis destinos de saltos que usam esses ponteiros. Como calcular a restrição perfeita é inviável computacionalmente, usam-se métricas para comparar a segurança de políticas diferentes. Neste artigo propomos duas métricas que levam em consideração o contexto dinâmico de execução e mostramos como elas podem ser usadas para direcionar o desenvolvimento de políticas de controle de fluxo mais restritivas.*

1. Introdução

Programas escritos com linguagens de nível intermediário como C e C++ são passíveis a ataques que visam corromper a memória. Se a memória for responsável por um ponteiro de código, o atacante pode sequestrar o fluxo de execução do programa e realizar computação arbitrária no sistema alvo [Shacham 2007, Checkoway et al. 2010, Schuster et al. 2015]. Como forma de mitigação para tais problemas de segurança foram propostas políticas que visam garantir a integridade de fluxo de controle (CFI)[Abadi et al. 2005], que são alterações no programa para reduzir os possíveis destinos quando se usa um ponteiro de código. Idealmente, essas políticas deveriam restringir ao máximo possível o fluxo sem bloquear destinos corretos, mas, para essas linguagens, é computacionalmente impossível saber os destinos válidos de um ponteiro. Criaram-se então métricas que visam comparar a segurança de diferentes políticas, porém medir o quanto um programa é vulnerável a um ataque não é fácil, pois, a não ser que já se

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001

saiba a priori da existência da vulnerabilidade, não é possível determinar se um programa está protegido ou não. As métricas propostas utilizam-se de parâmetros diferentes para identificar zonas de potencial vulnerabilidade, portanto cada uma identifica um tipo de problema diferente, devendo então ser usadas em conjunto para analisar o programa. Um parâmetro importante para caracterizar a segurança que é ignorado pelas métricas existentes é o contexto dinâmico da execução das aplicações-alvo [Burow et al. 2017b], pois ele ajuda a definir a superfície de ataque, que são as partes do programa que são acessíveis ao atacante.

Nesse artigo nós apresentamos as seguintes contribuições:

- Duas métricas para a avaliação de CFI que levam em consideração o contexto dinâmico;
- Uma avaliação comparativa entre métricas dinâmicas e estáticas para fluxos de execução do *Kernel* Linux;
- Uma ferramenta de perfilamento de fluxo de execução para o *Kernel* Linux.

Nós avaliamos as métricas propostas através de diferentes fluxos de execução e obtivemos resultados que indicam que métricas dinâmicas avaliam positivamente melhorias de segurança que não seriam identificadas ou seriam vistas como prejudiciais pelas métricas estáticas. Além disso, a avaliação do fluxo de execução do *kernel* indica que as políticas atuais podem ser ineficientes quando considerado o contexto dinâmico de execução. Isso demonstra que a inclusão do contexto torna as métricas propostas mais robustas que as métricas estáticas atuais.

Este artigo está organizado da seguinte forma: a Seção 2 apresenta em mais detalhes ataques de sequestro de controle de fluxo e como a CFI visa mitigar esses ataques. A Seção 3 apresenta métricas aceitas para comparar implementações de CFI e explica como elas diferem entre si. A Seção 4 propõe novas métricas que levam em consideração o fluxo de execução de um programa para avaliar a efetividade da proteção. A Seção 5 descreve um experimento onde aplicamos uma implementação de CFI no *Kernel* Linux e mostramos como as métricas propostas se comportam em diferentes fluxos de execução. Em seguida, propomos um reforço para as políticas de CFI e apresentamos como ele é interpretado como um aumento de vulnerabilidade para métricas que ignoram o contexto dinâmico. A Seção 6 termina o artigo com as conclusões da pesquisa desenvolvida.

A maioria das referências contidas nesse artigo são para publicações em inglês. Nós vamos traduzir os nomes e termos criados por eles, mas manteremos os acrônimos originais, pois eles já estão bem difundidos na literatura.

2. Contexto

Nesta Seção apresentamos os ataques de sequestro de controle de fluxo, explicando três exemplos. Em seguida, apresentamos a integridade de fluxo de controle (CFI), que é uma classe de técnicas que visam mitigar esses ataques [Abadi et al. 2005].

2.1. Sequestro de controle de fluxo (CFH)

Um dos primeiros ataques do qual se tem registro que realiza um CFH é o *Stack Smashing* [One 1996]. Esse ataque visa utilizar uma falha de memória para causar um *buffer overflow* inserindo na memória um código malicioso e sobrescrevendo o endereço

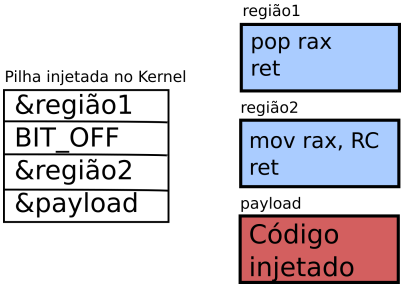

Figura 1. Exemplo de ROP em um *kernel*.

de retorno da função em execução pelo endereço do código injetado. Dessa forma, quando a função terminava, ao invés de retornar para o ponto onde foi chamada, ela faria um salto para o código malicioso. Atualmente é possível prevenir ataques de injeção de código usando processadores com a funcionalidade *Write XOR Execute*, que não permite o processador escrever em região de memória que é executada, nem executar instruções em região de dados.

Três novos ataques foram desenvolvidos que visam explorar as mesmas vulnerabilidades de corrupção de memória e ainda conseguir realizar computação arbitrária, mas sem precisar injetar código. Esses ataques são Programação Orientada a Retorno (ROP) [Shacham 2007], Programação Orientada a Saltos (JOP) [Checkoway et al. 2010] e Programação Orientada a Objetos Falsos (COOP)[Schuster et al. 2015].

2.1.1. Programação orientada a retorno (ROP)

No ataque ROP [Shacham 2007] um atacante usa uma vulnerabilidade de memória para sobrescrever a pilha de execução de um programa por uma pilha falsa. O objetivo da pilha falsa é concatenar pequenas regiões de código já existente em memória que estão logo antes de uma instrução de retorno. Ao substituir a pilha por uma sequência contendo os endereços dessas regiões e outros dados necessários, o sistema alvo pularia para a primeira região ao encontrar a primeira instrução de retorno. Como cada região é seguida por outra instrução de retorno, ao chegar nela o sistema saltaria para a próxima região indicada na pilha, executando as instruções na ordem desejada pelo atacante.

Para ilustrar o ataque, vamos mostrar um exemplo onde um atacante aplica ROP no *kernel* de um sistema operacional para forçar o processador a desabilitar o isolamento entre espaços de endereçamento para em seguida saltar para um código injetado.

Nos sistemas computacionais atuais, o espaço de endereçamento da memória principal é dividido entre espaço do *kernel* e espaço de aplicações do usuário. As páginas de memória de cada um dos espaços são marcadas com *bits* específicos que sinalizam a qual espaço elas pertencem. Como os usuários possuem controle sobre os programas que executam no seu espaço de endereçamento, os processadores modernos impedem que tais páginas de memória sejam acessadas dentro de um contexto de execução previlegiado. Sem tal proteção, um usuário malicioso poderia, por exemplo, corromper um ponteiro de código dentro do *kernel*, fazendo-o apontar para uma página de memória do espaço de usuário, que esteja sob seu controle, sequestrando então o fluxo de execução com privilégios. Tal isolamento entre o *kernel* e espaço de usuário é configurado através de *bits*

específicos dentro de registradores de controle do processador.

Na Figura 1 podemos ver duas regiões que serão usadas no ataque, um espaço onde o atacante já injetou o código e a pilha falsa. Assumimos que as vulnerabilidades de memória já foram exploradas e vamos explicar o resto do ataque.

- O primeiro valor da pilha é o endereço da *região1*. O processador vai tirar esse endereço da pilha e saltar para ele. A instrução seguinte diz para tirar um valor de pilha e colocar no registrador *rax*. O valor encontrado na pilha é *BIT_OFF*, que será responsável por desabilitar o isolamento.
- A instrução seguinte é um retorno. Como *BIT_OFF* foi removido da pilha, o valor seguinte é o endereço da *região2*. A primeira instrução dessa região move o valor do registrador *rax* para o registrador de controle, desabilitando a proteção que bloqueia o processador de executar código em memória mapeada como espaço de usuário.
- O último retorno contém o endereço do código injetado, que agora pode ser executado já que o isolamento foi desativado pelo ataque.

Esse é um ataque bem simples, porém em um código complexo o suficiente (basta incluir a *libc*) o atacante consegue montar ataques que são Turing-completos apenas pela concatenação de regiões.

2.1.2. Programação orientada a saltos (JOP)

Esse ataque é bem semelhante ao ROP [Checkoway et al. 2010], mas ao invés de concatenar regiões antes de uma instrução de retorno, o JOP busca três coisas:

- Regiões seguidas por um salto para o endereço em um dos registradores de propósito genérico que chamaremos de *RX*.
- Uma região que será usada como trampolim. Essa região precisa ter uma pilha de dados que contenha estruturas com ponteiros de função e um ciclo que desempilhe uma estrutura e chame a função apontada.
- Uma forma de corromper RX para apontar para o trampolim.

Se essas três condições forem satisfeitas, JOP é tão eficaz quanto ROP.

2.1.3. Programação orientada a objetos falsos (COOP)

No COOP [Schuster et al. 2015] o atacante tenta explorar propriedades de linguagens orientadas a objeto (como C++), onde a implementação de interfaces e heranças levam a muitas chamadas de função indiretas. O ataque busca uma região do programa onde exista um ciclo que chame métodos de objetos organizados em um vetor. As chamadas de métodos são traduzidas em ponteiros de código no binário final. Para concatenar código existente e realizar computação arbitrária, é necessário apenas corromper o vetor substituindo o endereço do método chamado pelo endereço do desejado.

2.1.4. Capacidades atuais de CFH

Técnicas que visam mitigar o ataque aleatorizando os endereços de partes do código não são suficientes pois é possível montar ataques que conseguem encontrar as regiões necessárias [Shacham et al. 2004, Snow et al. 2013]. Técnicas que visam proteger a pilha [Cowan et al. 1998, Chen et al. , Chiueh and Hsu 2001, Prasad et al. 2003] também não são suficientes pois o JOP não precisa corromper endereços de retorno. As técnicas de defesa mais aceitas para CFH são CFI e integridade de ponteiro de código (CPI) [Kuznetsov et al. 2014], e já existem ataques que mostram que uma implementação que não restrinja o fluxo o suficiente ainda pode ser contornada [Schuster et al. 2015, Evans et al. 2015b, Evans et al. 2015a].

2.2. Integridade de fluxo de controle (CFI)

CFI são políticas que visam modificar um programa de forma que ele verifique antes de usar um ponteiro de código se esse ponteiro é válido ou não. Ponteiros de código são variáveis que contem um endereço dentro da região a executada de um programa e são usados para realizar saltos indiretos. Os casos de uso desses ponteiros são retornos de função (que retornam para o endereço salvo ao entrar nela) e chamadas a ponteiros de função, onde o endereço da função a ser chamada não é fixo, e sim um parâmetro definido na execução do programa. Uma primeira implementação foi proposta junto ao conceito de CFI [Abadi et al. 2005] e ela visa instrumentar binários da seguinte forma:

- Insere um marcador (valor fixo) antes de cada função.
- Insere um marcador diferente depois da chamada de cada função.
- Adiciona instruções antes de cada retorno para validar se o endereço a ser retornado possui o marcador de retorno e retorna para depois do marcador.
- Adiciona instruções antes de cada chamada de função indireta para validar se o endereço antes dela possui um marcador.

Isso faz com que ponteiros de função só sejam chamados se eles apontarem para o início de uma função, e retornos só sejam feitos para endereços que seguem uma chamada. Essa técnica se mostrou insuficiente para se proteger contra ataques mais complexos [Schuster et al. 2015, Evans et al. 2015b].

Novas implementações foram propostas que ao invés de usar um instrumentador de binários, usam o compilador. Dessa forma é possível ter mais informações sobre as indireções (saltos indiretos), possibilitando a criação de heurísticas que focam em reduzir os possíveis destinos dos saltos indiretos [Tice et al. 2014].

Atualmente chamadas para ponteiros de códigos podem ser protegidas por marcadores, ou por tabelas de saltos [Tice et al. 2014] e retornos também podem ser protegidos por marcadores ou outros mecanismos de proteção de pilha, porém suas implementações levam a um aumento de latência que em diversos casos pode ser inaceitável [Dang et al. 2015].

3. Métricas de avaliação de segurança estáticas

Como diferentes implementações de CFI usam políticas diferentes, criou-se a necessidade de métricas que tentam quantificar a segurança de um programa para que seja possível

compará-las. Nesta Seção, apresentaremos três métricas encontradas na literatura. Essas métricas são extraídas de uma inspeção do binário ou do código, sem a necessidade de detalhes da execução do programa, ou seja, são puramente estáticas. A próxima Seção discutirá as métricas propostas neste trabalho.

Um parâmetro que pode ser extraído de análise estática é a quantidade de destinos possíveis de uma indireção. Como o CFH é baseado em levar o programa para fazer um salto que deveria ser ilegal, identifica-se como uma zona vulnerável uma que possui uma quantidade de destinos elevada. Uma das dificuldades de extrair esse parâmetro é que ele é dependente da implementação, por exemplo: no caso da instrumentação explicada na seção 2.2, os destinos possíveis de um salto são todos os endereços do programa que possuem o marcador que é verificado pelo salto.

3.1. Redução média de destinos por indireção (AIR)

A AIR [Zhang and Sekar 2013] mede a média da restrição de cada indireção do programa. Saltos restritos contribuem de forma positiva a métrica, aumentando ela.

$$AIR = \frac{1}{n}\sum_{i=1}^{n} 1 - \frac{|T_i|}{S} \tag{1}$$

Na equação 1, temos como n o número de saltos indiretos, S como o total de destinos possíveis em um programa desprotegido e T_i como o número de destinos possíveis depois de aplicadas as políticas de CFI para uma indireção i. Essa métrica varia entre zero e um, onde zero quer dizer que não existe restrição nenhuma e um que ele restringe todos os destinos possíveis. O problema com essa métrica é que S é proporcional a quantidade de *bytes* de um programa, pois todos os endereços são possíveis em um programa desprotegido. Dessa forma até as medidas menos restritivas já vão ser muito próximas a 1, o que dificulta a comparação entre elas, pois um aumento na restrição terá um impacto insignificante na métrica.

3.2. Média de destinos permitidos por indireção (AIA)

A AIA [Ge et al. 2016] mede a média de destinos possíveis por indireção. Ela funciona como o inverso da AIR, que mede a restrição dos destinos, mas ao invés de normalizar a quantidade pelo máximo possível, ela trabalha com valores absolutos. Isso tem como intuito facilitar a comparação de implementações de CFI diferentes.

$$AIA = \frac{1}{n}\sum_{i=1}^{n} |T_i| \tag{2}$$

Na equação 2 os identificadores possuem o mesmo significado que na AIR. Essa métrica varia entre a menor quantidade de destinos possíveis de um salto e a maior, o que pode ser de zero até o tamanho do programa. Zero indica uma política que bloqueia qualquer salto indireto, e o tamanho do programa que todos os saltos podem ir para qualquer ponto do programa. A vantagem dessa métrica é que, em um mesmo programa, se uma política *A* restringe os destinos possíveis pela metade que uma *B*, então a AIA de *A* seria metade de *B*. A desvantagem dessa métrica é que reduzir a quantidade de destinos de um salto com muitos destinos é muito mais importante do que reduzir o de um com poucos, pois um ponteiro com muitos destinos é mais fácil de ser explorado por um atacante, mas para a métrica isso é indiferente.

3.3. Segurança quantitativa (QS)

A QS [Burow et al. 2017a] define o conceito de classe de equivalência, que é o conjunto de destinos possíveis de um salto indireto. Dois saltos diferentes podem ter como destino a mesma classe de equivalência (caso eles possam ir para os mesmos destinos), ou classes diferentes. As classes de equivalência de um programa são a união das classes de todos os saltos presentes nele. A métrica melhora (aumenta) com o número de classes de um programa, pois assumindo que o número de saltos se mantém, aumentar o número de classes tende a levar a uma redução do número de destinos dela. Ela também usa a maior classe de equivalência para caracterizar a vulnerabilidade do programa de forma que diminuir essa classe também impacta positivamente na métrica. Portanto QS foca na quantidade de classes de equivalência, enquanto AIA e AIR focam nos tamanhos das classes.

$$QS = EC\frac{1}{LC} \tag{3}$$

A equação 3 tem EC como a quantidade de classes de equivalência e LC como o tamanho da maior classe (quantidades máximas de destino de um salto). O problema com essa métrica é que qualquer redução de tamanho em classes de equivalência que não seja a maior delas não possui impacto no resultado, apesar de ser uma alteração benéfica do ponto de vista de segurança.

4. Propostas de métricas que consideram o contexto dinâmico

O contexto dinâmico possui informações que só podem ser obtidas através da execução de um programa, como quais as funções mais chamadas e quantidade de memória necessária para a execução de um programa a partir de uma determinada entrada. Em programas grandes é comum a existência de partes que raramente ou nunca são executadas. Além disso, regiões que fazem parte do fluxo de execução principal do programa são as de mais fácil acesso para atacantes [Burow et al. 2017b]. Usar o fluxo de execução como parâmetro de uma métrica pode nos permitir identificar qual política melhor protege o fluxo principal do programa, ou até, dado uma mesma política, comparar a sua eficiência em diferentes fluxos de execução do mesmo programa. Essa Seção apresenta duas novas métricas que estamos propondo neste trabalho e que usam como parâmetro o contexto dinâmico. Na Seção 5 discutiremos o comportamento dessas métricas em experimentos.

4.1. Segurança quantitativa dinâmica (DQS)

A DQS leva em consideração os mesmos elementos que a QS, mas também analisa a distribuição de acessos a classes de equivalência. Acessos distribuídos em classes diferentes impactam de forma positiva na métrica. O objetivo da mesma é identificar fluxos que mostram um abuso de uma classe. A partir desses resultados é possível que estratégias sejam utilizadas para tentar melhorar a restrição da classe em questão.

$$DQS = EC\frac{1}{LC}DD \tag{4}$$

$$DD = \frac{1}{MAC}\sum_{i=1}^{n}\frac{A_i}{TAC} \tag{5}$$

A Equação 5 apresenta a métrica distribuição dinâmica (DD), onde temos n como o número de classes, TAC como o total de acessos a todas as classes de equivalência, A_i

como o total de acessos para a classe i e MAC como o maior valor entre todos os Ai. Dessa forma, DD é a quantidade média de acessos a classes dividido pela quantidade máxima de acessos a classe. Ela varia entre zero e um, onde um quer dizer que os saltos indiretos de um fluxo de execução estão perfeitamente distribuídos entre as classes e vai se aproximando a zero à medida que a distribuição foca mais em uma classe do que em outras. DQS (Equação 4) é a multiplicação de QS (Fórmula 3) por DD, portanto é igual ou inferior a QS. Dessa forma ela possui as qualidades de QS e ainda permite extrair informações sobre a distribuição do fluxo de execução.

4.2. Média de destinos permitidos por acesso AIAA

A AIAA é a média de destinos possíveis para as indireções executadas pelo programa. Como uma indireção com muitos destinos é identificada como uma vulnerabilidade em potencial, a métrica expande esse conceito para identificar fluxos que possibilitam muitas variações de destinos como vulneráveis.

$$AIAA = \frac{1}{n}\sum_{i=1}^{n}|T_i| \tag{6}$$

Na equação 6, n é a quantidade total de saltos indiretos em um fluxo de execução e T_i é a quantidade de destinos permitidos pelo salto executado i. Essa métrica é muito semelhante a AIA (Equação 2), mas ao invés de fazer a média de destinos permitidos para cada salto do programa, ela faz para cada salto executado. Enquanto a AIA dá o mesmo peso para todos os saltos indiretos que existem no código, na AIAA o peso está diretamente ligado a quantidade de vezes que o salto foi utilizado. Em ambas as equações, a medida varia entre o tamanho da menor e maior classe de equivalência. Em um caso onde um programa tem poucos saltos para uma classe grande, mas esses saltos são executados frequentemente AIA resultaria em uma métrica baixa, o que leva a interpretação de que o código está seguro. Por outro lado, AIAA, ao considerar o contexto dinâmico, deslocaria o valor da métrica de modo a expor o fluxo como vulnerável. Por usar o fluxo de execução como parâmetro ao invés das indireções presentes no binário, AIAA desconsidera saltos que não são executados, mas que ainda poderiam ser acessíveis a um atacante, porém ela é eficiente para dizer se um fluxo específico oferece uma superfície de ataque grande.

5. Experimentos

Nessa Seção vamos usar as métricas apresentadas para avaliar a segurança do *Kernel* Linux dada uma política de CFI de granularidade fina (que usa técnicas entre as mais eficientes para restringir o fluxo de execução). Nela iremos explicar como foi feito para extrair as métricas, propor fluxos específicos para serem avaliados, interpretar os resultados de cada métrica individualmente, propor uma forma de aumentar a segurança do programa que seria identificada por métricas dinâmicas mas não por estáticas e por fim ver conclusões possíveis ao combinar a interpretação das diferentes métricas e do código.

5.1. Metodologia

Para avaliar a métrica usamos um *Kernel* Linux modificado para funcionar com uma implementação de CFI de granularidade fina [**?**] que leva em consideração o cabeçalho das funções e o tipo dos ponteiros de função para reduzir o número de destinos possíveis tanto das chamadas indiretas quanto dos retornos. Desenvolvemos um perfilador para chamadas do *kernel* (KCperf) que consiste em:

- Um passo do LLVM (versão 8.0.0) que instrumenta todas as chamadas de função de forma a criar uma variável global para cada uma, guardando um inteiro, e coloca antes da chamada as operações necessárias para aumentar em um o valor da variável. As variáveis são colocadas em uma mesma região do binário.
- Um módulo do *kernel* que permite leitura de regiões do mesmo especificadas pelo usuário.
- Um conjunto de scripts em Python que analisa o binário do *kernel* para identificar o endereço da região com as variáveis (que será informado para o módulo no momento de leitura) e que processa o resultado da leitura retornando a quantidade de vezes que cada chamada foi executada.

O perfilador é executado antes e depois da execução de um fluxo desejado. A diferença entre as duas execuções representa apenas as chamadas feitas durante o fluxo.

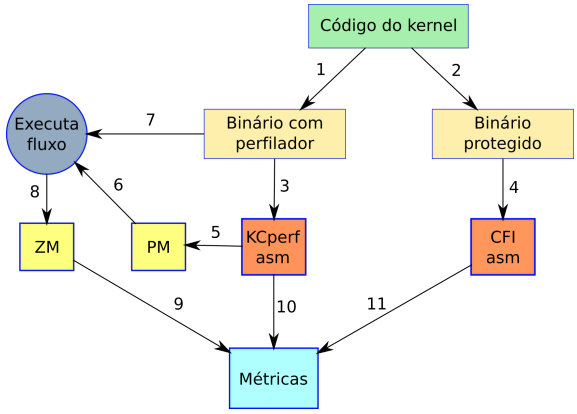

Figura 2. Processo de obtenção de métricas

A Figura 2 ilustra o processo de obtenção de métricas. Ela é dada da seguinte forma: O código do *kernel* é compilado duas vezes, uma com CFI e outra com a instrumentação do KCperf. Da compilação com o KCPerf extraímos informações sobre a localização das variáveis usadas para o perfilamento (identificadas como PM na Figura). Essas informações serão usadas para extrair os valores do módulo que criamos. Executamos um fluxo desejado para o *kernel* e extraímos os dados do perfilamento (identificados como ZM na Figura). Por fim combinamos os resultados do perfilamento com o resultado da linguagem de montagem de ambas as compilações do *kernel* para extrair as métricas desejadas.

Apesar do KCperf apenas perfilar as chamadas de função e não os retornos, as métricas ainda podem ser calculadas pois DQS conta saltos para classes de equivalência. Como nós sabemos a classe de equivalência do retorno da função chamada, se ela foi chamada n vezes, ela também vai retornar n vezes. A AIAA faz a média do uso de saltos, se uma função é chamada n vezes, qualquer distribuição de retorno escolhida da mesma função contribui da mesma forma para a métrica.

No nosso experimento fizemos o perfilamento de três fluxos de execução:

- O *benchmark Apache* do conjunto provido pela *Phoronix* [pho] (versão 1.7.x).
- O subconjunto referente a sistemas operacionais do *LMbench* [lmb] (versão 3.0.a9).

Tabela 1. Avaliação da proteção do *kernel* por métricas estáticas

Métrica	AIA	QS
Valor	231.92	2.64763

Tabela 2. Avaliação da proteção de fluxos do *kernel* por métricas dinâmicas

Fluxo	AIAA	DQS
pts/apache	563.65	0.01876
lmbench/os	527.46	0.02421
ocioso	1940.38	0.00635

- O *Kernel* Linux (versão 4.9) rodando em modo ocioso (*idle*).

O perfilamento dos fluxos do *LMbench* e *Phoronix* representam a execução encadeada de dez instâncias de cada *benchmark*. No caso do fluxo do *kernel* ocioso, ele foi registrado durante 10 minutos. As métricas extraídas representam o valor para um fluxo inteiro.

Algo importante a se entender é que um fluxo de execução ocioso para o *kernel* não quer dizer que ele está parado. Enquanto a máquina estiver ligada ele vai estar em um ciclo de execução pequeno, mas ainda ativo. Consideramos importante perfilar esse fluxo pois da mesma forma que ele está presente enquanto o *kernel* está ocioso, ele também está presente enquanto o sistema está trabalhando com outras coisas. O objetivo é identificar como esse fluxo contribui para as métricas propostas e a segurança do sistema. O *Apache* foi escolhido pois no trabalho que apresenta o CFI usado nos nossos experimentos, ele foi o que apresentou o maior aumento de latência após aplicada a proteção. O *LMbench* foi escolhido por ser um *benchmark* que tem como objetivo estressar o sistema operacional, que é o contexto no qual estamos trabalhando.

5.2. Resultados e análise

Extraímos do *kernel* protegido as métricas estáticas AIA (Equação 2) e QS (Equação 3), que podem ser vistas na Tabela 1. Para os fluxos de execução do Apache, LMbench e ocioso extraímos as métricas propostas AIAA (Equação 6) e DQS (Equação 4) que podem ser vistas na Tabela 2. A extração das métricas estáticas foi feita pois elas têm parâmetros em comum com as dinâmicas, o que torna possível uma comparação entre elas para novas conclusões.

Para ajudar na análise vamos usar os gráficos que se encontram na Figura 3. Os gráficos possuem no eixo y a porcentagem de vezes que uma chamada foi feita em um fluxo de execução (em relação ao total de chamadas). No eixo x temos as 100 chamadas mais executadas de forma ordenada. Em todos os casos as 100 chamadas representam mais que 40% do total.

5.2.1. Analisando a Segurança Quantitativa Dinâmica (DQS)

Podemos ver que as medidas de DQS são muito menores do que a QS original (estática) para todos os fluxos de execução. Isso mostra que os saltos indiretos estão sendo, em

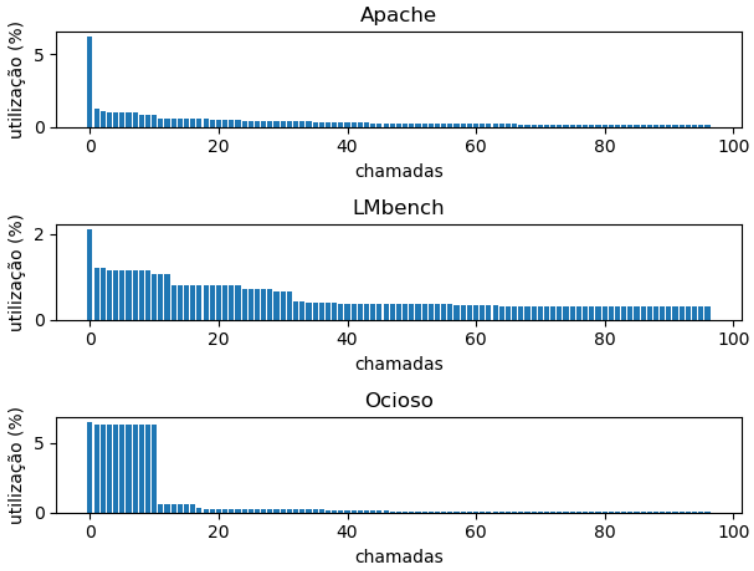

Figura 3. Concentração de saltos

sua maioria, para classes de equivalência semelhante. Relacionando esse resultado com a Figura 3 podemos perceber que muitos dos saltos são representados por apenas 20 chamadas de funções dentre todas executadas. Isso confirma a concentração de saltos para uma mesma classe. Vemos também que o fluxo do *Apache* tem os seus saltos concentrados em menos chamadas que o fluxo ocioso, porém a DQS obtida para ele foi melhor. Isso indica que o fluxo ocioso, apesar de usar mais vezes saltos diferentes, tem como destino de seus saltos classes de equivalência semelhantes, evidenciando que usar menos saltos não é a mesma coisa que acessar menos classes de equivalência. *Lmbench*, por ser um *benchmark* que tem como objetivo explorar o sistema operacional, foi o que apresentou melhor resultado. Baseado nessa análise das medidas obtidas e da Figura 3 mostramos que a métrica DQS representa com boa fidelidade a distribuição de uso de classes de equivalência, conforme proposto na Seção 4.

5.2.2. Analisando a Média de destinos permitidos por acesso (AIAA)

Comparando a AIAA para os diferentes fluxos com a AIA extraída estaticamente vemos que as medidas dinâmicas foram sempre maiores que o dobro da original. Para que isso aconteça a distribuição de indireções por classe de equivalência executadas deve ser diferente da distribuição por classe de equivalência existente no programa, e mais voltadas para classes maiores. Comparando as métricas dos fluxos entre si temos novamente a do fluxo do LMbench com mais indireções mais seguras (para classes menores) e a do fluxo ocioso com mais indireções vulneráveis (para classes maiores). É importante notar que isso não é a mesma coisa que dizer que o fluxo do LMbench é mais seguro. Como dito no início da Seção, o fluxo ocioso está sempre presente, logo as indireções vulneráveis dele ainda estão presentes no LMbench. O caso contrário também seria possível, o resultado da métrica ser maior para o fluxo alternativo que para o principal indica que ele é mais vulnerável por conter mais indireções para classes de equivalência maiores, mas isso não foi visto nesse experimento.

Tabela 3. Métricas após expansão de funções (inlining)

AIA	AIAA (lmbench/os)
231.98	527.44

5.2.3. Interpretações dos resultados

Juntando as interpretações dos resultados da DQS e da AIAA percebemos que o fluxo ocioso do *kernel* é o mais vulnerável dos apresentados. Porém, esse fluxo está contido dentro dos fluxos do Apache e do LMbench. Logo podemos afirmar que qualquer fluxo de execução dentro do *kernel* é tão vulnerável quanto, ou mais, que o fluxo ocioso. A conclusão é que precisamos aplicar políticas que sejam mais eficientes para proteger o fluxo ocioso, entretanto isso é complicado, pois os 60 saltos mais feitos são quase exclusivamente para ponteiros de função e a maioria do mesmo tipo. As políticas de CFI que utilizamos já estão entre as mais restritas para esse tipo de salto, e como podemos ver na Figura 3, esses saltos correspondem a maioria dos executados. Isso mostra que o *kernel* está sempre executando a sua parte mais vulnerável, e que nossas ferramentas são limitadas para proteger esse caso.

5.2.4. Propondo novas políticas de CFI

Para a AIAA, resultados positivos comparando fluxos diferentes não indicam que o fluxo é mais seguro (como explicado na Seção 5.2.2), porém resultados positivos comparando AIAA para o mesmo fluxo são mais importantes, pois se a AIAA diminui, a quantidade média de indireções possíveis também diminuiu. Vamos agora propor uma forma de melhorar a segurança do programa que terá resultados positivos para AIAA, mas negativos para AIA.

Como as métricas dinâmicas são baseadas em quantidades de acesso, propor políticas de CFI que visam remover saltos perigosos mais usados teria resultados positivos para as métricas. Uma forma de fazer isso é pela expansão (conhecida como *inlining* na literatura de compiladores) de chamadas diretas, pois elas removem o retorno da chamada.

Para exemplificar, nós escolhemos trabalhar com o fluxo de execução do LMbench e expandimos a chamada para *syscall_return_slowpath()* feita pela função *do_syscall_64()*. Ela foi a oitava função mais acessada no fluxo e possui cabeçalho do tipo *void (void)* (funções que nem recebem argumentos, nem retornam valores) que a indireção gerada pelo seu retorno é para uma das maiores classes de equivalência do *Kernel* Linux.

Comparando os valores da Tabela 3 com os das Tabelas 1 e 2, podemos ver que AIA indicou que a política foi menos segura, enquanto AIAA indicou o oposto. Uma consequência de expandir uma função é que todas as chamadas feitas pela função expandida agora serão feitas pela função que a chamava. Dessa forma o número de chamadas do sistema e o tamanho das classes de equivalência aumentam, o que pode ser visto como um problema pela AIA e QS. O código expandido, entretanto, é o mesmo que o da função que era chamada, então mesmo que existam mais regiões para o atacante usar, essas regiões são redundantes, não oferecendo capacidades novas ao ataque.

Isso mostra que AIAA é mais robusta para identificar melhorias de segurança, mas ainda é possível que se façam manipulações no programa para aumentar a segurança e piorar a métrica, como por exemplo remover um salto para uma classe de equivalência menor que a média.

6. Conclusão

Esse artigo propõe pela primeira vez métricas que utilizam o contexto dinâmico de execução para avaliar políticas de CFI. Mostramos que elas conseguem identificar melhorias em um programa que seriam ou ignoradas ou até mesmo vistas como pioras na segurança aplicada por métricas estáticas já existentes na literatura. Através dessas métricas dinâmicas, conseguimos também mostrar que proteger o *kernel* Linux com CFI é um desafio maior do que o esperado, pois o mesmo está acessando constantemente seções de código vistas como zonas vulneráveis até pelas soluções mais robustas de CFI. Como trabalhos futuros pretendemos explorar mais formas de aumentar a segurança de um programa levando em conta o seu fluxo de execução. A expansão de função usada como exemplo nesse trabalho não pode ser feita indiscriminadamente, pois pode aumentar o tamanho do binário para limites inaceitáveis e influenciar negativamente no desempenho do programa devido ao impacto a cache do processador. Desejamos explorar novas formas de aplicar a expansão de maneira a não gerar esse indesejável efeito colateral.

Referências

Lmbench. `https://sourceforge.net/projects/lmbench/`. Accessed: 2020-07-31.

Phoronix test suit. `https://www.phoronix-test-suite.com/`. Accessed: 2020-07-31.

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2005). Control-flow integrity: Principles, implementations, and applications. *ACM SIGSAC Conference on Computer and Communications Security (CSS)*.

Burow, N., Carr, S., Nash, J., Larsen, P., Franz, M., and Brunthaler, S. (2017a). Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*

Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. (2017b). Control-flow integrity: Precision, security, and performance). *ACM Computing Surveys*, 50(16).

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. *7th ACM conference on Computer and communications security - CCS Í0*.

Chen, X., Slowinska, A., Andriesse, D., Bos, H., and Giuffrida, C. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries.

Chiueh, T. and Hsu, F. (2001). Rad: A compile-time solution to buffer overflow attacks. *IEEE Distributed Computing Systems*.

Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., , and Zhang, Q. (1998). Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. *8st USENIX Security Symposium*.

Dang, T., Maniatis, P., and Wagner, D. (2015). The performance cost of shadow stacks and stack canaries. *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*.

Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H. (2015a). Missing the point(er): On the effectiveness of code pointer integrity. *IEEE Symposium on Security and Privacy*.

Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., and Sidiroglou-Douskos, S. (2015b). Control jujutsu. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pages 901–913.

Ge, X., Talele, N., Payer, M., and Jaeger, T. (2016). Fine-grained control-flow integrity for kernel software. *IEEE European Symposium on Security and Privacy, EURO S and P*.

Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., , and Song, D. (2014). Code-pointer integrity. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

One, A. (1996). Smashing the stack for fun and profit. *Phrack Magazine 49(14)*.

Prasad, M., Chiueh, T., and Brook, T. S. (2003). A binary rewriting defense against stack based buffer overflow attacks. *USENIX Annual Technical Conference*.

Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A., and Holz, T. (2015). Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. *IEEE Symposium on Security and Privacy*.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). *Proceeding CSS 07 Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561.

Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. *Proceeding CCS '04 Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307.

Snow, K., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A. (2013). Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. *Proceedings - IEEE Symposium on Security and Privacy*.

Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, U., and Lozano, L. (2014). Enforcing forward-edge control-flow integrity in gcc & llvm. *Proceedings of the 23rd USENIX Security Symposium*.

Zhang, M. and Sekar, R. (2013). Control flow integrity for cots binaries. *USENIX Security Symposium*.