Efficient and Precise Dynamic Construction of Control Flow Graphs

Andrei Rimsa
Departamento de Computação
CEFET-MG
Belo Horizonte, MG, Brazil
andrei@cefetmg.br

José Nelson Amaral
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
jamaral@ualberta.ca

Fernando Magno Quintão Pereira
DCC – UFMG
Belo Horizonte, MG, Brazil
fernando@dcc.ufmg.br

Resumo

The extraction of high-level information from binary code is an important problem in programming languages, whose solution supports the detection of malware in binary code and the construction of dynamic program slices. The Control Flow Graph is one of the instruments used to represent the structure of binary programs. Most solutions to reconstruct CFGs from binary programs rely on purely static techniques, based either on data-flow analyses, or in type inference. In contrast, in this work we use a purely dynamic approach to such a purpose. Our technique can be used alone, or in combination with static analysis tools. We demonstrate that it is possible to verify completeness in several real-world programs. We also show how to combine our technique with DynInst, the current state-of-the-art static CFG reconstructor. By providing DynInst with extra information, we improve its capacity to deal with indirect jumps. Our dynamic CFG reconstructor has been implemented on top of valgrind. When applied on cBench, this implementation is able to completely cover 36% of all the functions available in that suite. It adds an average overhead of 43x onto the execution of the original programs. Although expressive, this overhead is almost four times lower than the overhead of DCFG, a tool distributed by Intel, and built on top of PinPlay.

CCS Concepts • **Software and its engineering → Compilers**; *Language features*; • **Computer systems organization → *Complex instruction set computing***.

Keywords Grafo de fluxo de controle, análise dinâmica

ACM Reference Format:
Andrei Rimsa, José Nelson Amaral, and Fernando Magno Quintão Pereira. 2019. Efficient and Precise Dynamic Construction of Control Flow Graphs. In *XXIII Brazilian Symposium on Programming Languages (SBLP 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3355378.3355383

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SBLP 2019, September 23–27, 2019, Salvador, Brazil
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7638-9/19/09...$15.00
https://doi.org/10.1145/3355378.3355383

1 Introdução

O grafo de fluxo de controle (GFC) é uma estrutura de dado usada para representar programas [2, p.525]. Nesse grafo direcionado vértices representam *blocos básicos*, isto é, sequências maximais de instruções sem desvios, exceto, possivelmente, no final. Arestas representam fluxos possíveis dentro do texto do programa. Em outras palavras, uma aresta entre os blocos b_1 e b_2 indica que é possível que o bloco b_2 siga a execução do bloco b_1. Desde que foi introduzido, nos anos 70 [3], GFCs têm sido usados na implementação das mais variadas técnicas de análise e otimização de programas. Tipicamente, o GFC é produzido a partir do código fonte de um programa, contudo, há também muito interesse em extrair tal estrutura a partir de código binário [6, 16, 17, 20]. Entretanto, enquanto a reconstrução a partir de código fonte é um problema bem resolvido, a reconstrução exata de GFCs a partir da representação binária do programa é um problema indecidível. A indecidibilidade advém da inabilidade de um reconstrutor de prever o destino de desvios indiretos.

Existem duas formas de extrair um GFC a partir da representação binária de um programa: o método estático e o método dinâmico. O primeiro deles consiste na extração a partir do texto de um código binário. Essa técnica é usada em várias ferramentas, algumas, como DynInst[12], já bastante conhecidas. A segunda alternativa busca reconstruir o GFC a partir de traços observados durante a execução do programa alvo. A reconstrução dinâmica não é tão popular quanto sua contra-partida estática. A única ferramenta publicamente conhecida da qual estamos cientes é DCFG –uma extensão da ferramenta PinPlay da Intel.

Esses dois métodos, estático e dinâmico, embora vertentes do mesmo problema, levam a resultados diferentes. A técnica estática é conservadora: o GFC produzido pode ter arestas que nunca serão, de fato, exercitadas durante a execução do programa. A técnica dinâmica, por outro lado, pode deixar de adicionar arestas ao GFC –arestas são criadas somente se elas descrevem fluxos observados durante a execução do programa. Essas diferenças levam a aplicações distintas. A

reconstrução estática de GFCs é muito utilizada tanto em análises de segurança, que visam encontrar vulnerabilidades de software [18, 21], quanto em otimização de programas binários [14, 25]. A reconstrução dinâmica, por sua vez, é usada como uma forma de suporte a técnicas de fatiamento dinâmico (*dynamic slicing*) [1, 10]. Embora o problema de fatiamento dinâmico de programas seja antigo, tendo sido introduzido três décadas atrás, neste artigo defendemos a tese de que o método de reconstrução dinâmica de GFCs ainda carece de melhorias. A ferramenta proposta, disponível em https://github.com/rimsa/CFGgrind, lida com as seguintes limitações do estado da arte na área:

Completude. Conforme descrito por Xu *et al.* [22], um dos principais problemas da abordagem dinâmica é a sua inabilidade de obter um GFC completo, isto é, ter todos os blocos básicos que integram um programa. Contudo, os resultados de experimentos neste trabalho indicam que é possível obter GFCs completos para uma vasta quantidade de GFCs, mesmo que a análise dinâmica use somente uma entrada do programa alvo.

Acurácia. Uma das contribuições deste artigo é mostrar como o mecanismo de reconstrução dinâmico de GFCs pode ser usado para complementar as informações produzidas por uma ferramenta estática. Mostraremos, na seção 2.2, como nossa técnica é capaz de refinar a informação produzida por DynInst, por exemplo.

Eficiência. Possivelmente, o grande entrave à utilização de técnicas dinâmicas de reconstrução é sua eficiência. Contudo, neste trabalho descreveremos detalhes de implementação que nos deixam criar um reconstrutor dinâmico sobre a ferramenta valgrind [13] que aumenta o seu tempo de execução em torno de 45 vezes. A fim de dar ao leitor uma perspectiva sobre este número, nossa implementação é quatro vezes mais rápida do que DCFG, ferramenta da Intel.

2 Contextualização

A modelagem dos fluxos de execução de um programa é uma etapa essencial para a condução de várias análises feita por um compilador, como por exemplo análises de fluxo de dados [2]. Para isso é usualmente construída uma representação, em formato de um grafo direcionado, conhecida como grafo de fluxo de controle (GFC). A seção 2.1 provê uma definição de GFC centrada sobre funções, e a seção 2.2 discute as limitações da reconstrução estática desse tipo de grafo.

2.1 Definições Básicas

Vários trabalhos utilizam um único GFC para todo o programa, onde blocos básicos - sequências contíguas de instruções executadas na ordem dada sem desvios - são conectados por arestas para definir os possíveis caminhos de execução [4, 12, 20]. Nesse modelo, funções são subgrafos de todo o GFC, e arestas são marcadas como intraprocedurais e interprocedurais. As primeiras conectam blocos dentro da mesma função, ao passo que as segundas conectam blocos em funções diferentes. Note que duas ou mais funções podem compartilhar um mesmo bloco básico ou uma função pode ter mais de um nó de entrada. Esse trabalho diverge dessa noção de um único GFC para todo o programa em favor de uma definição mais específica: cada função tem seu próprio GFC, em par com implementações encontradas em compiladores industriais. Essa distinção é relevante, pois permite rastrear com mais precisão as entradas e saídas de cada função. Nesse caso, um GFC possui apenas uma entradas, embora possa compartilhar instruções com outros GFCs. Assim, segue sua definição formal:

Definição 2.1. *Grafo de Fluxo de Controle (GFC) é uma tupla* $G = (endereço, N)$*, tal que:*

- *endereço é a posição de memória da primeira instrução;*
- *N é um conjunto de nós (n), tal que* $n = (Preds, Succs, tipo)$*:*
  - *Preds é um conjunto de nós predecessores* ($Preds \subset N$) *conectados à entrada de n;*
  - *Succs é um conjunto de nós sucessores* ($Succs \subset N$) *conectados à saída de n;*
  - *tipo pode ser uma das seguintes classificações:*
    - ∗ *entrada indica o ponto de início desse GFC. Deve ter apenas um successor* ($|Succs| = 1$) *e nenhum predecessor* ($Preds = \emptyset$)*.*
    - ∗ *saída indica o ponto de saída desse GFC onde o fluxo de execução retorna para seu chamador. Deve ter pelo menos um predecessor* ($|Preds| > 0$) *e nenhum sucessor* ($Succs = \emptyset$)*.*
    - ∗ *término indica o ponto de término desse GFC onde o fluxo de execução é interrompido e programa termina sua execução. Deve ter pelo menos um predecessor* ($|Preds| > 0$) *e nenhum sucessor* ($Succs = \emptyset$)*.*
    - ∗ $bloco(Instrs, Chamadas, modo \in \{direto, indireto\})$ *indica um bloco básico que possui: um conjunto ordenado não vazio de instruções* ($Instrs = \{i_1, i_2, \ldots, i_n\}$*;* $i_j = (endereço, tamanho, asm)$) *que devem ser executadas sequencialmente, sem desvios; um conjunto de GFCs* ($Chamadas$) *que são chamados por esse bloco básico; e um modo de desvio (modo, se direto ou indireto).*

Exemplo 2.2. *A Figura 1 mostra uma função* (a) *que aplica uma função genérica em um inteiro e seu GFC equivalente* (b)*. O nó com oval preenchido representa o nó de entrada do GFC (entrada), com oval dupla o de saída (saída), com quadrado duplo o de término (término). Os nós blocos básicos possuem instruções e podem conter uma seção com as chamadas para outros GFCs e um modo de desvio* ($bloco(Instrs, Chamadas, modo)$)*. O modo de desvio indireto é representado por uma aresta pontilhada apontando para um ponto de interrogação.*

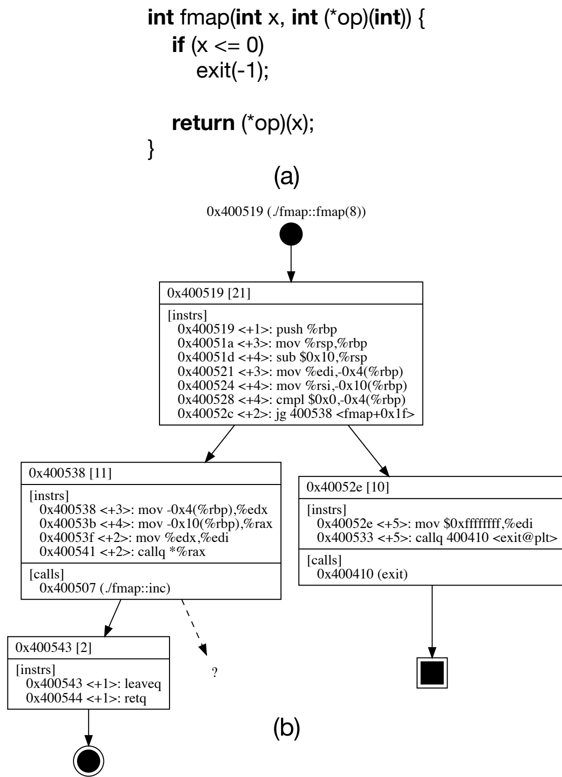

Figura 1. Exemplo de código (a) com seu respectivo diagrama GFC (b) de acordo com a Definição 2.1.

2.2 Limitações de Análises Estáticas

Muita da atenção voltada para a reconstrução de GFC a partir de códigos binários foi dada para abordagens utilizando análise estática [5, 9, 19, 20]. Umas das ferramantas mais completas para obtenção de GFCs estaticamente encontradas publicamente, DynInst [12], ilustra como é complexa a tarefa de analisar binários. Dificuldades advêm de construções como bytes que não são instruções, funções com entradas ambíguas, instruções sobrepostas, fluxos indiretos, funções que não retornam, e funções que compartilham código, por exemplo. Embora ferramentas desse tipo sejam extremamente rápidas elas sofrem com problemas de precisão levando a obtenção de GFCs parciais e até errôneos.

Um dos casos mais críticos, e potencialmente a maior fonte de imprecisão da abordagem estática, é quando há presença de fluxos indiretos, ou seja, instruções de salto (*jump*) e chamadas de funções *call* sobre registradores. Como esses alvos são dinamicamente calculados é difícil determiná-los estaticamente. Eles são comumente utilizados em desvios baseados em ponteiros, funções virtuais e para alguns casos de implementações do comando *switch*. DynInst [12] consegue reconstruir boa parte das construções de comandos switch, mas falha ao lidar com os outros dois tipos listados.

Exemplo 2.3. *A Figura 2 mostra que DynInst não foi capaz de identificar blocos básicos no caso de desvios baseados em ponteiros - código dentro do retângulo pontilhado em (a) e (b) não presente em (c). Além disso, DynInst também não foi capaz de reconhecer funções inteiras na presença de chamadas virtuais - função* twice *de (d) inexistente em (e). Os GFCs (c) e (e) foram gerados com a ferramenta auxiliar (*cfg_to_dot*) distribuída junto com DynInst. Desvios indiretos são representados com uma aresta para o nó especial* -1*. Esses GFCs tiveram nós removidos para fins de apresentação, sem prejuízo a argumentação. Os dois programas (a) e (d) foram compilados sem otimização e com a remoção dos símbolos de depuração no binário produzido (*stripped*).*

3 Reconstrução Dinâmica de GFCs

Este trabalho explora a construção de grafos de fluxo de controle rastreando dinamicamente os caminhos tomados pelas instruções durante a execução de um programa. A cobertura do código depende das entradas fornecidas - problema da sensibilidade da entrada - e os desvios não tomados previnem a visão completa de todos os caminhos possíveis. Contudo, a Seção 3.1 estende a definição de GFCs para incluir caminhos não tomados e permite criar o conceito de completude para eles. Uma outra limitação da abordagem dinâmica é a sobrecarga imposta pela instrumentação durante a execução. A Seção 3.2 discute estratégias para minimizar esse impacto.

3.1 Sobre Completude de GFCs

Instruções de desvios podem ser classificadas como diretas ou indiretas, incondicionais ou condicionais. Desvios diretos codificam o endereço alvo na própria instrução, ou a partir de um offset que pode ser calculado estaticamente, enquanto desvios indiretos utilizam o conteúdo de um registrador como parâmetro para o salto. Desvios incondicionais transferem o controle da execução para o endereço alvo imediatamente. Desvios condicionais podem saltar para o endereço alvo se determinada condição for satisfeita ou para a instrução imediatamente posterior caso contrário. Instruções de retorno (*ret*) são sempre indiretas e incondicionais. Já instruções de chamada de funções (*call*) são sempre incondicionais, mas podem ser diretas ou indiretas. Por fim, instruções de salto (*jump*) podem ter qualquer uma dessas combinações.

Para a reconstrução de GFC dinamicamente, a definição 2.1 cobre todas as instruções incondicionais, sejam elas diretas ou indiretas, já que o alvo de qualquer instrução será sempre seguido. No caso de desvios condicionais, um dos endereços - aquele que será seguido pela execução - também será sempre coberto. Para o outro endereço têm-se duas opções: se for indireto está coberto pela marcação de indireção na definição GFC, mas se for direto não é possível saber seu contéudo, já que esse caminho não será exercitado. A fim de lidar com esses caminhos não visitados, criamos a noção de *nó fantasma*: um bloco básico especial que carrega apenas

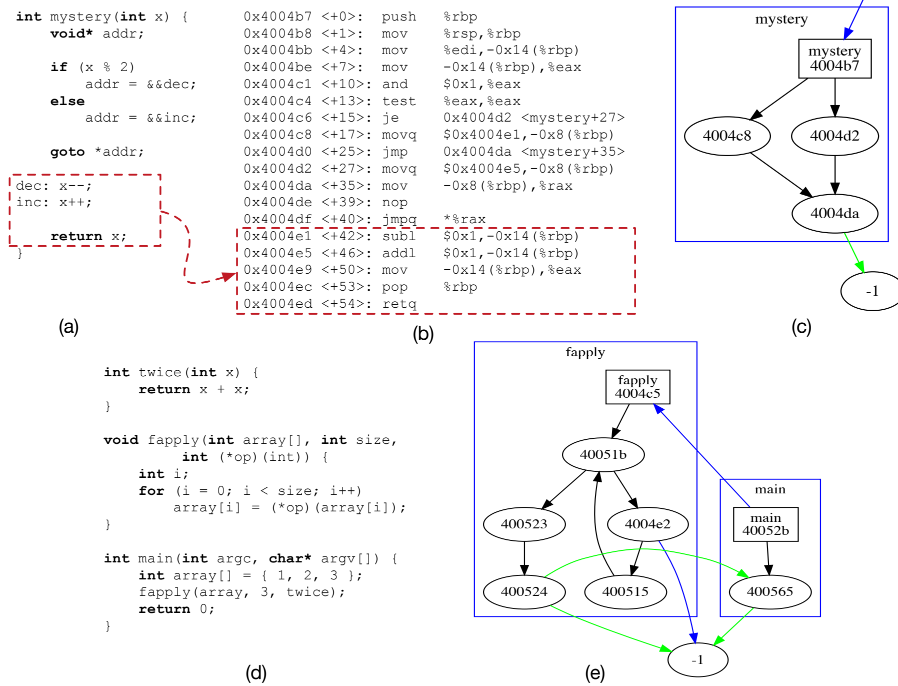

Figura 2. Um exemplo com desvios baseados em ponteiros (a) com sua respectiva seção em assembler de x86 (b) e seu GFC parcial (c). Outro exemplo com chamada de função virtual (d) e seu respectivo GFC parcial (e).

esse endereço. Assim, a definição de GFC pode ser estendida para GFCs dinâmicos da seguinte maneira:

Definição 3.1. *GFC dinâmico é uma tupla $G' = (endereço, N')$, tal que:*

- *endereço é um endereço de memória de como em um GFC G;*
- *N′ é um conjunto de nós, onde:*
  - *Inclui os nós N de um GFC G ($N \subset N'$);*
  - *$n = (Preds, Succs, fantasma(endereço))$ é um nó fantasma ($n \in N'$) deve ter pelo menos um predecessor ($|Preds| > 0$), nenhum sucessor ($Succs = \emptyset$) e um tipo fantasma que possui um endereço de memória (fantasma(endereço)).*

A definição 3.1 nos leva naturalmente ao conceito de *completude*:

Definição 3.2. *Um GFC dinâmico $G' = (a, N')$ é completo se, um nó $n = (Preds, Succs, tipo) \in N'$:*

- *Não é fantasma ($tipo \neq fantasma(\dots)$), e;*
- *Não é um bloco básico com modo indireto ($tipo \neq bloco(\dots, \dots, indireto)$).*

A presença de nós fantasmas indica a existência de caminhos não tomados, enquanto a de desvios indiretos não permite inferir todos os possíveis caminhos de execução. Para todos os outros tipos de construções têm-se os fluxos bem definidos. Portanto, um grafo é completo se todos os seus caminhos são conhecidos. Note que o problema de determinar, estaticamente, que um GFC dinâmico será completo para uma determinada entrada é indecidível. A indecibilidade desse problema pode ser facilmente inferida a partir do teorema de Rice [15], o qual impõe uma barreira teórica à obtenção de todos os fluxos possíveis na execução de um programa.

3.2 Detalhes de Implementação

O Rastreador de Funções. A construção do GFC dinâmico se dá via o *rastreamento* de instruções binárias durante a execução do programa alvo. No contexto deste trabalho, o rastreador é valgrind. Em termos abstratos, o rastreador de instruções é uma máquina M, que lê o próximo conjunto de instruções em vias de serem executadas, até uma instrução de desvio (*jump, call, ret*). Esse conjunto de instruções forma uma sequência ordenada $R = \{i_1, i_2, \dots, i_n\}$. Note que R não é necessariamente um bloco básico de um GFC. É possível que durante o rastreamento do programa R precise ser dividido em sequências menores de instruções. A divisão acontece quando algum desvio salta para uma instrução $i_j \in R$, $i_1 < i_j \leq i_n$. Nesse caso, R será particionado em duas sequências: $R_x = \{i_1, \dots, i_{j-1}\}$ e $R_y = \{i_j, \dots, i_n\}$.

Nosso algoritmo de reconstrução de GFCs mantém, para cada GFC, um apontador para o último ponto de sua execução processado até o momento. Esse apontador é referenciado como *pendente* no restante do texto. Além disso, o algoritmo

mantém uma pilha das chamadas de funções, onde em cada nível tem-se um GFC e seu respectivo apontador *pendente*. Nosso algoritmo recebe, do rastreador, sequências de instruções (R) e a analisa. Caso a sequência termine com chamada ou retorno de funções, o algoritmo atualiza a pilha de GFCs. Caso a última instrução seja um salto, ele atualiza o GFC corrente. Nosso algoritmo é aplicável em ambientes de execução concorrente, pois mantemos uma pilha distinta para cada linha de execução.

Construção do GFC via refinamentos sucessivos. Nosso reconstrutor de GFCs mantém uma lista de blocos básicos, com as arestas conhecidas. Durante a execução do programa alvo, o algoritmo recebe sequências R do rastreador e as processa. O processamento ocorre em duas fases: a primeira onde são processadas as instruções do grupo (Algoritmo 1) e a segunda onde são processados os desvios ao final da execução dessas instruções, com seus possíveis endereços alvos (Algoritmo 2). O algoritmo 1 varre as instruções em R, e, para cada instrução i, atualiza ou não o GFC corrente. Atualizações consistem em: (i) incrementar o tamanho do bloco corrente se a instrução i puder ser adicionada a ele; (ii) transformar um nó fantasma em um bloco real, caso i faça parte de um bloco fantasma; ou (iii) criar novos blocos caso i salte para fora de R. A fim de reconhecer novas arestas no GFC, o algoritmo 1 sempre mantém uma cópia da última instrução processada, a qual é chamada *pendente*.

Algorithm 1: Processamento de grupo de instruções

```
   Input: R = {i_1, i_2, ..., i_n}
1  Function processarGrupo(R):
2      foreach i ∈ R do
3          if proxInstr(pendente) == i then
4              // nada a fazer
5          else if temProxSucc(nó(pendente), i) then
6              if éNóFantasma(nó(i)) then
7                  transformarParaBloco(nó(i), i)
8              end
9          else
10             if i ∈ GFC→I then
11                 if ~ éPrimeiraInstrDoBloco(i) then
12                     dividirBloco(nó(i), i)
13                 end
14                 conectarBlocos(nó(pendente), nó(i))
15             else
16                 if podeAnexarInstr(nó(pendente), i) then
17                     adicionarInstr(nó(pendente), i)
18                 else
19                     criarNovoBloco(i)
20                     conectarBlocos(nó(pendente), nó(i))
21                 end
22             end
23         end
24         pendente = i
25     end
26 end
```

O algoritmo 2 marca desvios como diretos ou indiretos conforme o caso. Esse algoritmo atualiza a pilha de GFCs, caso encontre instruções de chamada e retorno de funções. Para cada endereço alvo em instruções de desvios, inclusive aquele seguinte em desvios condicionais, o algoritmo 2 verifica se já existe uma instrução com esse endereço no GFC. Se existir, essa deve ser a primeira instrução do bloco, dividindo o bloco se não for o caso, para ser conectado ao bloco *pendente*. Caso contrário, conecta ele a um nó fantasma que deve ser criado. No término do programa são criadas arestas para o nó de termino para a instrução pendente e para todas que estiverem ativas na pilha de chamadas.

Otimizações. Embora os algoritmos descritos sejam capazes de rastrear o fluxo de execução corretamente, ainda têm um impacto relevante no tempo de execução de todo o programa. Uma das avenidas exploradas por este trabalho para melhorar o desempenho foi o uso de memoização. Uma vez processada a instrução *pendente* sobre um grupo de instruções R, obtém-se uma nova instrução *pendente*. Assim, pode-se criar o mapeamento $(i_{atual}, R) \longrightarrow i_{proximo}$ a ser usado nas próximas execuções. Caso um novo caminho seja tomado, esse mapeamento é atualizado. Além disso, a maior parte da execução do algoritmo 1 é concentrada nos dois primeiros testes

Algorithm 2: Finalizar o processamento do bloco de acordo com a última instrução

```
1  Function finalizarBloco():
2      definirModo(nó(pendente), tipoIndireção(pendente))
3      switch tipoInstr(pendente) do
4          case call do
5              novoGFC = obterGFC(obterAlvo(pendente))
6              adicionarChamada(nó(pendente), novoGFC)
7              empilhar(GFC, pendente)
8              GFC = novoGFC
9              pendente = nóEntrada(GFC)
10         case ret do
11             conectarBlocos(nó(pendente), nóSaída(GFC)
12             (GFC, pendente) = desempilhar()
13         otherwise do
14             foreach alvo ∈ obterAlvos(pendente) do
15                 if ∃{i ∈
                     obterInstruções(CFG) | obterEndereço(i) =
                     alvo} then
16                     if ~ éPrimeiraInstrDoBloco(i) then
17                         dividirBloco(nó(i), i)
18                     end
19                     conectarBlocos(nó(pendente), nó(i))
20                 else
21                     conectarBlocos(nó(pendente),
                         criarFantasma(alvo))
22                 end
23             end
24         end
25     end
26 end
```

condicionais, já que uma vez definido um caminho dentro do GFC, esse caminho pode ser executado múltiplas vezes. Assim, algumas otimizações podem ser implementadas sobre o algoritmo 1. Por exemplo, se o tamanho do bloco básico corrente bate com o tamanho da sequência R, então pode-se processar diretamente a última instrução daquela sequência. Além disso, as instruções são mantidas em uma tabela hash para fácil manipulação. Na seção 4 analisaremos o impacto dessas otimizações.

4 Avaliação Empírica

Nessa seção apresentaremos números para demonstrar que as técnicas descritas neste trabalho são eficientes e úteis. Com tal propósito, analisaremos os programas disponíveis em cBench *Collective Benchmark*, uma suite para otimizações de programas e arquiteturas, disponível em http://ctuning.org/wiki/index.php/CTools:CBench. Essa coleção de benchmarks contém 32 programas implementados em C, cada um em torno de 20 entradas diferentes. Essa abundância de entradas torna cBench ideal para estudos baseados em perfilamento de código. As simulações foram executadas em uma máquina Linux com processador Intel Xeon de 2.4GHz com 16 núcleos e 32GB de memória RAM. Apenas o programa office_ispell da suite cBench não poder ser concluído por não executar em arquiteturas 64 bits.

Eficiência de execução. A construção de um GFC dinâmico é um processo lento: código antes executado nativamente passa a ser interceptado pelo rastreador de instruções. Nosso processo de reconstrução do GFC pode tornar a execução de um programa até 45 vezes mais lenta. Contudo, ferramentas similares sofrem perdas ainda mais sérias. A figura 3 corrobora essa observação comparando nossa ferramenta, CFGgrind executada sobre valgrind e DCFG sobre PinPlay.

Cada barra dupla na figura 3 representa a reconstrução de GFCs para os oito (8) primeiros programa de cBench, exceto a última que mostra a média artimética deles. Para cada um deles, a barra esquerda mostra o tempo de execução de nossa ferramenta CFGgrind sobre valgrind. Para referência, o tempo de CFGgrind é dado pela composição da execução dos plugins de valgrind nulgrind - sem instrumentação - e callgrind - rastreador de funções. nulgrind adiciona um slowdown médio de ~4 vezes, callgrind de ~30 vezes e CFGgrind de ~45 vezes sobre o programa. Já o plugin DCFG sobre PinPlay adiciona um slowdown médio de ~170 vezes sobre o programa, dado pela barra da direita. Conforme pode ser visto na figura, CFGgrind é substancialmente mais rápido que o reconstrutor de GFCs da Intel, com speedup médio de ~4 vezes. O trabalho de Yount *et al.* [24] não provê detalhes suficientes sobre o reconstrutor de GFCs e PinPlay é código-fechado, o que impossibilita uma análise comparativa aprofundada com nossa ferramenta.

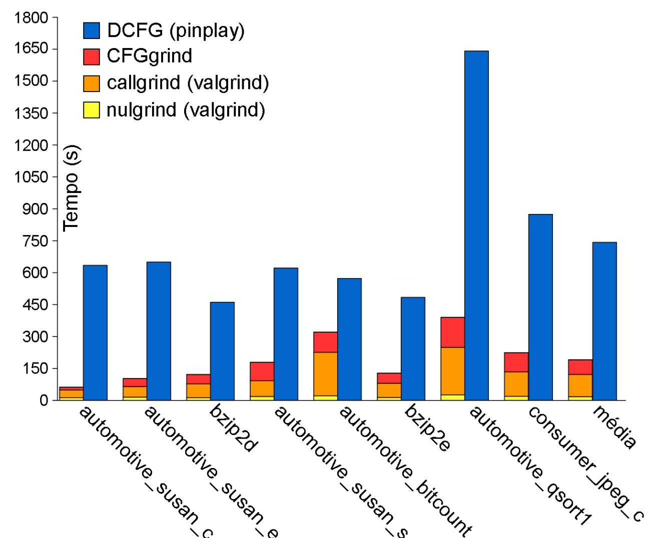

Figura 3. Comparação entre o tempo de execução de PinPlay e CFGgrind.

Acurácia. Para validar a precisão de nossa ferramenta comparamos com o reconstrutor estático estado-da-arte DynInst. Foram removidos todos os símbolos de depuração para obtenção dos dados da Tabela 1, já que é comum encontrar programas distribuídos sem essas informações. Do total de 7131 funções existentes nessa suite, CFGgrind encontrou 2680 (37.6%) enquanto DynInst 2938 (41.2%). Dessas, 1579 (22.1%) foram encontradas por ambas as análises, 1101 (15.4%) somente por CFGgrind e 1359 (19.1%) apenas por DynInst. A análise estática tem dificuldade de lidar com binários sem símbolos de depuração, enquanto a análise dinâmica tem problemas devido a cobertura das entradas. Vale ressaltar que é difícil quantificar esses diferenças com granularidade fina, já que imprecisões em ambas as análises podem gerar colisões nessas métricas. Por exemplo, saltos podem ser considerados chamadas numa análise e vice-versa em outra. Embora a análise estática tenha encontrado mais blocos básicos, e por conseguinte mais arestas, instruções e chamadas, as encontradas pela análise dinâmica são caminhos reais que foram tomados durante a execução.

Completude. Conforme discutido na seção 1, um GFC dinâmico é uma aproximação dos comportamentos possíveis de um programa. A qualidade desta aproximação depende da cobertura do programa obtida com as entradas utilizadas para obter o GFC. Caso um GFC não possua desvios indiretos e sua reconstrução dinâmica não apresente, ao final do processo, nós fantasmas, então, de acordo com a definição 3.2, esse GFC é dito *completo*. Usando uma entrada (a entrada rotulada como zero) para cada os programas disponíveis no benchmarks cBench, pudemos observar que 36% de todos os GFCs são completos.

Tabela 1. Comparação de acurácia das ferramentas CFGgrind (dinâmica) e DynInst (estática).

	CFGgrind (A)	DynInst (B)	$A \cap B$	$A \setminus B$	$B \setminus A$
GFCs	2680	2938	1579 (58.9%/53.7%)	1101 (41.1%)	1359 (46.3%)
Blocos básicos	32076	73002	22474 (70.1%/30.8%)	9602 (29.9%)	50528 (69.2%)
Arestas	51129	106936	35633 (69.7%/33.3%)	15496 (30.3%)	71303 (66.7%)
Instruções	157978	318700	118635 (75.1%/37.2%)	39343 (24.9%)	200065 (62.8%)
Chamadas	7309	17698	3942 (53.9%/22.3%)	3367 (46.1%)	13756 (77.7%)

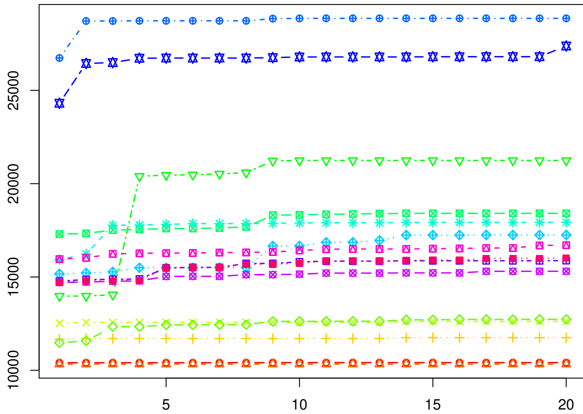

Figura 4. Crescimento do número de instruções observadas conforme mais execuções (com entradas diferentes) são avaliadas. O eixo-X contém um ponto para cada entrada diferente, e o eixo-Y descreve o número de instruções vistas. Cada linha representa uma distribuição cumulativa.

Com o uso de mais entradas, mais GFCs podem ser completados. Entretanto, nossos experimentos revelam que esse crescimento é modesto. A figura 4 provê alguns dados que suportam essa afirmação. Na figura, separamos 15 programas disponíveis em cBench, e os executamos com as 20 entradas disponíveis. A fim de simplificar a figura, omitimos os nomes dos benchmarks. Esses nomes são imateriais para essa análise, pois o comportamento de todos os benchmarks é muito similar. A conclusão desse experimento é que a primeira entrada já é suficiente para mostrar quase todas as instruções em um dado programa. Somente em três, dentre os quinze benchmarks mostrados na figura 4 pudemos observar um crescimento de mais de 5% no número de instruções vistas com as 20 entradas, quando comparado apenas com as instruções observadas com a primeira entrada.

5 Trabalhos Relacionados

A técnica recentemente proposta por Yount *et al.* [24] para reconstrução dinâmica de grafos de fluxo de controle é, possivelmente, o trabalho mais próximo deste artigo. Aquela técnica foi incorporada à ferramenta PinPlay, e constitui hoje, a única forma publicamente disponível de reconstruir GFCs dinamicamente. A avaliação experimental neste artigo revela que a implementação de PinPlay é substancialmente mais lenta que nosso reconstrutor: a nossa ferramenta é quatro vezes mais rápida. Parte dessa eficiência deve-se às várias melhorias incorporadas em nosso algoritmo, e descritas na seção 3. Contudo, PinPlay possui objetivos diferentes dos nossos. O trabalho original de Yount *et al.* buscava comparar o desempenho de binários diferentes produzidos para o mesmo programa. Esses binários seriam, por exemplo, resultado de aplicações de conjuntos distintos de otimizações.

Conforme discutido na seção 1 deste trabalho, a principal utilização de grafos de fluxo de controle dinâmicos é na implementação de técnicas de fatiamento dinâmico (*dynamic slicing*) de programas. O problema de fatiamento dinâmico foi introduzido por Korel e Laski em 1988 [10], porém a formulação de Agrawal e Horgan [1] é, hoje, a mais adotada. Há várias implementações de técnicas de fatiamento dinâmico –algumas bastante recentes, como os trabalhos de Hu *et al.* [7] e Lin *et al.* [11]. Uma diferença muito importante entre esses trabalhos e a técnica descrita neste artigo é que nós derivamos o grafo de fluxo de controle diretamente a partir da execução do programa binário, sem que qualquer instrumentação de código seja necessária. Por outro lado, os trabalhos acima citados produzem a fatia de programa via instrumentação do código fonte. Kargén e Shahmehri [8] provêm uma ilustração precisa dessa abordagem: o compilador insere instrumentação no código alvo, e tal instrumentação constrói *uma trilha de execução* do programa.

Também relacionadas ao nosso trabalho são as diversas ferramentas que buscam entender e modificar programas binários. Exemplos incluem BitBlaze [18], BOLT [14] e Janus [25]. Tais ferramentas são, essencialmente, analisadores de programas binários. Tais análises requerem que esses programas sejam colocados em um formato "processável", isto é, representados por uma estrutura de dados. Tal estrutura não é um grafo de fluxo de controle – esse nível de informação não é necessário, dado o objetivo dessas ferramentas. Assim, em vez de produzirem um grafo de fluxo de controle, essas ferramentas produzem uma sequência de blocos básicos que denota o caminho mais comum trilhado durante sua execução. Os grafos que produzimos poderiam ser usados por quaisquer dessas três ferramentas, porque eles contém

estritamente mais informação: não somente sequências de instruções, mas os possíveis caminhos elas formam.

Finalmente, existe uma vasta literatura relacionada à construção de grafos de fluxo de controle a partir da análise estática de programas. Acreditamos que DynInst [12] represente o ápice da área atualmente. A seção de trabalhos relacionados do artigo que descreve essa ferramenta, inclusive, contém um interessante levantamento desse campo de pesquisa. Embora muito útil – e muito mais popular que a reconstrução dinâmica de grafos de fluxo – as técnicas estáticas possuem várias limitações, conforme descrito por Ye *et al.* [23]. Várias dessas limitações foram descritas na seção 2 deste trabalho. Técnicas dinâmicas não vêm, contudo, suprir tais limitações, posto que elas também possuem seus limites. Ao contrário, a reconstrução dinâmica complementa as técnicas estáticas, seja eliminando caminhos impossíveis, seja descobrindo novos caminhos. Nesse sentido, nossas técnicas podem ser usadas para melhorar a precisão de DynInst, conforme discutido na seção 2.

6 Conclusão

Esse artigo apresentou uma contribuição para a a viabilidade e a utilidade de reconstrução dinâmica de GFCs. No esforço de desenvolver esse estudo, criamos a ferramenta CFGgrind. Essa é quatro vezes mais rápida que DCFG, software similar da Intel. Os resultados experimentais indicam que é possível obter completude para uma quantidade não trivial de GFCs. Especificamente, uma entrada somente é suficiente para atingir cobertura total de 36% de todos os GFCs produzidos para os programas de cBench. A infra-estrutura descrita neste artigo está disponível. Esperamos, no futuro, poder usá-la para descobrir código malicioso em programas.

Agradecimentos

Fernando Quintão teve o suporte financeiro do CNPq. José N. Amaral teve o apoio do programa UFMG-IEAT enquanto esteve no Brasil. Esta pesquisa somente foi possível devido à licença que Andrei Rimsa obteve do CEFET-MG.

Referências

[1] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *PLDI*. ACM, New York, NY, USA, 246–256.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Boston, Massachusetts, USA.

[3] Frances E. Allen. 1970. Control flow analysis. *SIGPLAN Not.* 5 (1970), 1–19. Issue 7.

[4] A. R. Bernat and B. P. Miller. 2012. Structured Binary Editing with a CFG Transformation Algebra. In *Working Conference on Reverse Engineering*. ACM, New York, NY, USA, 9–18.

[5] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. 2007. Static Analysis on x86 Executables for Preventing Automatic Mimicry Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Bernhard M. Hämmerli and Robin Sommer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 213–230.

[6] Cristina Cifuentes and K. John Gough. 1995. Decompilation of Binary Programs. *Softw. Pract. Exper.* 25, 7 (1995), 811–829.

[7] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. 2018. BinMatch: A Semantics-based Hybrid Approach on Binary Code Clone Analysis.

[8] Ulf Kargén and Nahid Shahmehri. 2015. Turning Programs Against Each Other: High Coverage Fuzz-testing Using Binary-code Mutation and Dynamic Slicing. In *ESEC/FSE*. ACM, New York, NY, USA, 782–792.

[9] Daniel Kästner and Stephan Wilhelm. 2002. Generic Control Flow Reconstruction from Assembly Code. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES/SCOPES '02)*. ACM, New York, NY, USA, 46–55. https://doi.org/10.1145/513829.513839

[10] B. Korel and J. Laski. 1988. Dynamic Program Slicing. *Inf. Process. Lett.* 29, 3 (1988), 155–163.

[11] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. 2018. Break the Dead End of Dynamic Slicing: Localizing Data and Control Omission Bug. In *ASE*. ACM, New York, NY, USA, 509–519.

[12] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *ISSTA*. ACM, New York, NY, USA, 24–35.

[13] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*. ACM, New York, NY, USA, 89–100.

[14] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *CGO*. IEEE Press, Piscataway, NJ, USA, 2–14.

[15] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 1 (1953), 358–366.

[16] B. Schwarz, S. Debray, and G. Andrews. 2002. Disassembly of Executable Code Revisited. In *WCRE*. IEEE Computer Society, Washington, DC, USA, 45–.

[17] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993. Binary Translation. *Commun. ACM* 36, 2 (1993), 69–81.

[18] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ICISS*. Springer-Verlag, Berlin, Heidelberg, 1–25.

[19] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. 2000. On the Static Analysis of Indirect Control Transfers in Binaries. In *In PDPTA*. 1013–1019.

[20] H. Theiling. 2000. Extracting safe and precise control flow from binaries. In *Conference on Real-Time Computing Systems and Applications*. ACM, New York, NY, USA, 23–30.

[21] Mateus Tymburibá, Rubens E. A. Moreira, and Fernando Magno Quintão Pereira. 2016. Inference of Peak Density of Indirect Branches to Detect ROP Attacks. In *CGO*. ACM, New York, NY, USA, 150–159. https://doi.org/10.1145/2854038.2854049

[22] Liang Xu, Fangqi Sun, and Zhendong Su. 2010. Constructing Precise Control Flow Graphs from Binaries.

[23] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. 2017. A Survey on Malware Detection Using Data Mining Techniques. *ACM Comput. Surv.* 50, 3 (2017), 41:1–41:40.

[24] Charles Yount, Harish Patil, Mohammad S. Islam, and Aditya Srikanth. 2015. Graph-matching-based simulation-region selection for multiple binaries. In *ISPASS*. IEEE Computer Society, Washington, DC, USA, 52–61.

[25] Ruoyu Zhou and Timothy M. Jones. 2019. Janus: Statically-driven and Profile-guided Automatic Dynamic Binary Parallelisation. In *CGO*. IEEE Press, Piscataway, NJ, USA, 15–25.