ANÁLISE ESPARSA DE FLUXO DE INFORMAÇÃO

BRUNO RODRIGUES SILVA

ANÁLISE ESPARSA DE FLUXO DE INFORMAÇÃO

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira

Belo Horizonte

Abril de 2016

Silva, Bruno Rodrigues

S586a Análise esparsa de fluxo de informação. / Bruno Rodrigues Silva. — Belo Horizonte, 2016
xxiv, 83 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais — Departamento de Ciência da Computação.
Orientador: Fernando Magno Quintão Pereira

1. Computação — Teses. 2. Compiladores — Teses.
3. Criptografia de dados(Computação) — Teses.
I. Orientador. II. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Análise esparsa de fluxo de informação

BRUNO RODRIGUES SILVA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ALAN MITCHELL DURHAM
Departamento de Ciência da Computação - USP

PROF. DIEGO DE FREITAS ARANHA
Instituto de Computação - UNICAMP

PROF. FELIPE MAIA GALVÃO FRANÇA
COPPE - UFRJ

PROF. LEONARDO BARBOSA E OLIVEIRA
Departamento de Ciência da Computação - UFMG

PROF. MARIO SÉRGIO FERREIRA ALVIM JÚNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 19 de abril de 2016.

Dedico este trabalho de tese aos meus pais Adão e Rosa que nunca mediram esforços em me oferecer a melhor educação que suas condições permitiam. Sei muito bem o orgulho que eles sentem nesse momento, o que me traz uma satisfação imensa por lhes proporcionar essa alegria. Pai, mãe, obrigado! Dedico também a Ana Carolina que pacientemente suportou minhas ausências durante o desenvolvimento desta pesquisa. Obrigado por compreender a importancia deste trabalho de tese para minha vida e carreira!

Agradecimentos

Agradeço ao PPGCC/UFMG por todo suporte e infraestrutura para a realização dessa pesquisa. Agradeço aos colegas do Laboratório de Linguagens de Programação que muito me ajudaram nas fases iniciais de aprendizado do compilador LLVM. Agradeço também ao apoio financeiro da CAPES e Intel por meio do projeto E-CoSoc. Ao professor Diego Aranha do IC/UNICAMP, o meu muito obrigado pelos importantes testes que ele realizou na nossa ferramenta FlowTracker. Finalmente, agradeço de forma especial ao professor Fernando M. Q. Pereira que me acolheu como seu pupilo nesse mundo louco dos compiladores e que de forma muito especial soube compreender minhas dificuldades técnicas e pessoais durante o curso. Sem você, seria impossível eu ter chegado até aqui. Muito obrigado!

"Nunca ande pelo caminho traçado,
pois ele conduz somente até onde os outros já foram."
(Alexander Graham Bell)

Resumo

Palavras-chave: Computação, Compiladores, Criptografia de dados..

Análises de fluxo de informação usam o conceito de Grafo de Dependência do Programa (GDP) como uma estrutura de dados de suporte. Esse grafo utiliza a noção de dependência de controle introduzida por Ferrante *et al.* para representar fluxos implícitos de informação. A limitação dessa abordagem é que ela pode criar $O(|I| \times |E|)$ arestas de fluxo implícito no GDP, onde I são as instruções no programa, e E são as arestas do seu respectivo grafo de fluxo de controle. Esta tese mostra que é possível construir análises de fluxo de informação usando uma noção diferente de dependência implícita, que gera um número de arestas linear no número de definições mais o número de usos de variáveis. Apresenta-se aqui então um algoritmo que obtém essas dependências em uma única travessia na árvore de dominância do programa. Essa eficiência é possível devido a uma propriedade chave de programas no formato de atribuição única estática (SSA): a definição de uma variável domina todos os seus usos. Esse algoritmo corretamente implementa o conhecido sistema de tipos seguros de Hunt e Sands. Assim, contrariamente à sua formulação original, que requer $O(I^2)$ espaço e tempo para programas estruturados, o presente trabalho requer somente $O(I)$. Esse arcabouço teórico aqui desenvolvido foi usado para construir FlowTracker, uma ferramenta que detecta diversas vulnerabilidades de software relacionadas ao fluxo de informação, tais como: vazamento de endereços, estouro de arranjo, estouro de inteiro e canais laterais baseados em tempo. Também pode ser usado como uma forma de obter um *slice* de programa de forma rápida. Um *slice* de programa diz respeito a um conjunto de instruções que podem afetar algum ponto de interesse do código fonte. FlowTracker é capaz de manipular programas com mais de 1 milhão de instruções Assembly em menos que 200 segundos, e cria 24% menos arestas de fluxo implícito que a técnica de Ferrante *et al.*

Abstract

Information flow analyses traditionally use the Program Dependence Graph (PDG) as a supporting data structure. This graph relies on Ferrante *et al.*'s notion of control dependences to represent implicit flows of information. A limitation of this approach is that it may create $O(|I| \times |E|)$ implicit flow edges in the PDG, where I are the instructions in a program, and E are the edges in its control flow graph. This tesis shows that it is possible to compute information flow analyses using a different notion of implicit dependence, which yields a number of edges linear on the number of definitions plus uses of variables. Our algorithm computes these dependences in a single traversal of the program's dominance tree. This efficiency is possible due to a key property of programs in Static Single Assignment form (SSA): the definition of a variable dominates all its uses. Our algorithm correctly implements Hunt and Sands system of security types. Contrary to their original formulation, which required $O(I^2)$ space and time for structured programs, we require only $O(I)$. We have used our ideas to build FlowTracker, a tool that detect several software vulnerabilities related to the information flow: address leak, buffer overflow and time based side-channel. Our theoretical framework can be used also to implement program slices. A program slice is a set of instructions that can affect some point of interest in the source code. FlowTracker handles programs with over one-million Assembly instructions in less than 200 seconds, and creates 24% less implicit flow edges than Ferrante *et al.*'s technique.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

1.1 Motivação

Análises de Fluxo de Informação são bastante úteis para provar certas propriedades em programas de computador, tais como, confidencialidade e integridade em *softwares* sensíveis a ataques. Desde a publicação do pioneiro trabalho de Denning & Denning [1977], essas análises têm se expandido fortemente em ambas direções teóricas e práticas. Hoje, existem formas puramente estáticas [Denning & Denning, 1977; Hunt & Sands, 2006], puramente dinâmicas [Zhang et al., 2011] e híbridas [Huang et al., 2004] de rastrear a propagação de dados no código de um programa, e devido à importância de segurança e privacidade, tais técnicas estão sendo aprimoradas.

Uma implementação de análise de fluxo deve rastrear não somente a propagação de informação explícita, mas também a implícita. Diz-se que existe um fluxo explícito de informação de uma variável u para uma variável v se v é definida por uma instrução que usa u. Por outro lado, informação flui implicitamente de um predicado p para uma variável v se p é usado em um desvio que controla uma atribuição de v, por exemplo: "$v = p\ ?\ 0 : 1$". Detectar o fluxo explícito de informação em um programa é uma prática comum na literatura de compiladores [Denning & Denning, 1977]. Porém, rastrear o fluxo implícito é um desafio maior [Russo & Sabelfeld, 2010]. Hoje, esses fluxos implícitos são aproximados via noção de *Dependência de Controle* definida no trabalho de Ferrante *et al.*[Ferrante et al., 1987, p.323] - o qual especifica uma metodologia de rastreamento de fluxo implícito largamente utilizada em algumas análises e transformações realizadas por compiladores [Hammer et al., 2006; Hammer & Snelting, 2009; Horwitz et al., 1988; Ranganath et al., 2007; Reps & ang, 1988; Snelting, 1996; Taghdiri et al., 2011; Wasserrab et al., 2009].

No trabalho de Ferrante *et al.* dependências explícitas (dados) e implícitas (con-

trole) são organizadas em uma estrutura de dados chamada *Program Dependence Graph* (PDG). Essa abstração foi originalmente projetada para facilitar tarefas de compilação tais como escalonamento de instruções e paralelização de código [Pingali & Bilardi, 1997]. O uso desse trabalho em implementação de análises de fluxo de informação, embora provavelmente correto [Hammer & Snelting, 2009; Horwitz et al., 1988; Ranganath et al., 2007; Reps & ang, 1988; Wasserrab et al., 2009], possui um ponto negativo: ele pode criar $O(|I| \times |E|)$ arestas de fluxo implícito no PDG, onde I são as instruções no programa, e E são as arestas no seu Grafo de Fluxo de Controle - GFC. Essa alta complexidade não é um limite puramente teórico: é possível construir cenários que evidenciam o pior caso, devido a laços *do-while* aninhados, ou devido aos chamados *ladder graphs* [Cytron et al., 1990, Fig.3]. Embora existam formas de compactar arestas de dependência de controle [Pingali & Bilardi, 1997], a complexidade de tempo quadrática não pode ser evitada caso necessário listar todas elas.

1.1.1 Ataques de Fluxo Contaminado

Análise de fluxo pode ser utilizada como uma ferramenta de detecção de duas categorias bastante conhecidas de ataques baseados no fluxo de informação. Na primeira categoria, conhecida como *Ataque de Fluxo Contaminado*, estamos interessados em saber se existe um caminho através do qual a informação pode fluir de uma entrada pública, que um adversário pode controlar, para uma operação sensível no programa. Se este caminho não passa por uma função capaz de filtrar possíveis entradas maliciosas, o adversário pode utilizá-lo para atacar o sistema.

Este tipo de ataque é muito comum em sistemas *Web*, que geralmente inicia com entrada de dados inserida por um adversário remoto que cuidadosamente a escolhe como forma de corromper o sistema de execução do aplicativo *Web*. Como exemplos de vulnerabilidades *Web* definidas como Ataques de Fluxo Contaminado, podemos citar: *SQL Injection*, *cross-site scripting*, *upload* de arquivo malicioso, execução de comandos não desejáveis e ataques ao sistema de arquivos [Scott & Sharp, 2003], [Wassermann & Su, 2007], [Xie & Aiken, 2006]. Esses ataques são baseados no mesmo padrão, que consiste na exploração remota da interface pública do aplicativo. Neste contexto, a interface pública é a *Web* e a vulnerabilidade é a inexistência de funções sanitizadoras nos dados de entrada antes destes serem usados pelo programa. Funções sanitizadoras buscam filtrar as entradas maliciosas daquelas inofensivas.

Fluxo contaminado é um problema que foi formalizado por Orbæk & Palsberg [1997] como uma instância de *type-checking*. Eles escreveram um sistema de tipos para linguagem λ-calculus, e provaram que se um programa passa na verificação de tipos,

então ele não é vulnerável a ataques por fluxo contaminado. Dez anos mais tarde, Jovanovic et al. [2006] implementaram um algoritmo que soluciona tal problema para a linguagem PHP 4.0. Esse algoritmo é uma análise de fluxo de dados baseada no sistema de tipos de *Orbæk e Palsberg* e possui complexidade $O(V^2)$ para o caso médio e $O(V^4)$ para o pior caso. Entretanto, a solução de Orbæk & Palsberg [1997] quando vista como um problema de fluxo de dados, admite complexidade $O(V^3)$ no pior caso.

Rimsa et al. [2011] melhoram a complexidade das soluções anteriores para o problema de fluxo contaminado. Para isso, eles propõem uma solução quadrática sobre o número de variáveis de um programa fonte. A ideia chave por trás dessa abordagem é a conversão do programa para uma representação intermediária chamada *e-SSA - (Extended Static Single Assignment)* introduzida por Bodik et al. [2000], a qual pode ser gerada em tempo linear sobre o tamanho do programa. Esta representação intermediária torna possível resolver o problema de fluxo contaminado através de uma análise esparsa, que associa restrições diretamente à variáveis do programa, ao invés de associá-las à variáveis em cada ponto do programa.

Ataques por fluxo contaminado não são exclusividade de sistemas *Web*. Programas *standalone* escritos em linguagem C e C++ também são alvos desses ataques. Essas são linguagens inseguras, fracamente tipadas e que permitem o uso de técnicas maliciosas, tal como o estouro de arranjo. *Estouro de arranjo* acontece quando uma quantidade de dados de tamanho suficientemente grande é escrita em um arranjo eventualmente extrapolando o limite dele e causando uma sobreposição de valores nas porções de memória adjacentes a esse limite. Programas vulneráveis a ataques por estouro de arranjo podem ter suas entradas públicas de dados cuidadosamente escolhidas pelo adversário. Normalmente, ele possui a intenção de sobrescrever dados sensíveis ou redirecionar o fluxo de controle para outra localização na memória de instruções. Esse comportamento foi observado em *worms* como o *Code Red* [Moore et al., 2002] e o *SQL Lammer*[1], que juntos comprometeram centenas de milhares de sistemas de computação no mundo.

Estouros de arranjo podem ser detectados através de técnicas de instrumentação de código ou modificações no código fonte. *Stack Guard* [Cowan et al., 1998] e *ProPolice* são soluções que detectam estouro de arranjo pela inserção de canários[2] na pilha para possibilitar a detecção de reescritas inválidas no endereço de retorno da função. Entretanto, essas ferramentas falham por não detectar todo tipo de estouro de arranjo, visto ser possível modificar dados sensíveis na pilha sem alterar o endereço de retorno. Em Santos [2013] é apresentada uma ferramenta de auxilio ao programador, que gera

[1] http://paintsquirrel.ucs.indiana.edu/pdf/SLAMMER.pdf
[2] http://en.wikipedia.org/wiki/Buffer_overflow_protection

anotações no arquivo fonte de programas C e C++, evidenciando possíveis problemas de estouro de arranjo. Tais problemas são detectados através de três análises do código fonte: análise de intervalo das variáveis que são índices de acesso ao arranjo; análise simbólica para assegurar que índices serão menores que o tamanho do arranjo e uma análise de inferência do tamanho do arranjo. Essas três análises são implementadas como módulos para o compilador LLVM [Lattner & Adve, 2004].

Uma possível solução para o problema de estouro de arranjo é a inserção de *ABC - Array Bound Checks* nos acessos ao arranjo. ABC pode ser considerado como uma forma de instrumentação de código e obviamente levam a um aumento significativo do tempo de execução.

1.1.2 Vazamento de informação sigilosa

Ataques que buscam obter conhecimento sigiloso referente aos dados manipulados por um programa em execução são a segunda categoria de ataques baseados no fluxo de informação. Deseja-se verificar se, durante a execução de um programa, alguma informação sensível alcança qualquer um dos canais públicos disponíveis, permitindo que um agente externo, adversário ou usuário tenha conhecimento desses dados confidenciais. Certas informações de um programa não devem alcançar funções de saída públicas, tais como as funções *printf*, *fwrite* etc. Um exemplo típico de informação que não deve chegar ao conhecimento dos usuários do sistema são os endereços de memória internos do programa. Isto é, os endereços de variáveis locais, dinâmicas e globais. Quando um programa permite que qualquer endereço possa atingir alguma função pública, temos uma vulnerabilidade de vazamento de endereços.

Os sistemas operacionais modernos utilizam um mecanismo de proteção para o vazamento de endereços chamado *ASLR - Address Space Layout Randomization* [Bhatkar et al., 2003; Shacham et al., 2004]. Esse mecanismo consiste em carregar os módulos binários que correspondem ao programa executável em diferentes endereços cada vez que o programa é executado. Essa técnica visa proteger o *software* de ataques como *return-to-libc*[Shacham et al., 2004] e *ROP - return-oriented -programming*[Buchanan et al., 2008; Shacham, 2007]. Entretanto, mesmo um sistema protegido por ASLR pode estar vulnerável caso algum de seus programas permita que endereços alcancem funções públicas de saída. No trabalho de Quadros et al. [2012] é possível encontrar um exemplo de ataque via estouro de arranjo que só foi possível graças ao vazamento de endereços. Neste exemplo, o adversário assume o controle de uma máquina 32 bits sob o sistema operacional Ubuntu 11.10, um sistema protegido por ASLR.

O vazamento de informação pode ocorrer via fluxo explícito diretamente ou indi-

retamente. Quando uma variável apontadora p, que contém um endereço de memória é passada como parâmetro para uma função de saída pública, temos um vazamento de endereços de forma explicitamente direta. Se o conteúdo dessa variável é transferido para outra variável p' e esta é passada como parâmetro para a função de saída pública, temos um vazamento de endereços explicitamente indireto. Porém, os vazamentos de endereços podem também ocorrer por meio dos fluxos implícitos. Detectar tais vazamentos é mais desafiador do que quando eles ocorrem por fluxo explícitos.

Denning & Denning [1977] formalizaram o problema de vazamento de informação e propuseram uma solução via análise estática de código. Entretanto, esta solução não considera vazamentos de informação por fluxo implícitos. Em 2006, Hunt e Sands apresentaram um sistema de tipos que é expressivo o suficiente para detectar vazamentos por fluxos implícitos de informação [Hunt & Sands, 2006]. Em termos simples, esse sistema associa a cada valor usado no programa um tipo, normalmente H para dados de alta segurança, e L para dados de baixa. Se informação de tipo H pode ser usada em alguma função que um adversário consegue observar, então o programa não passa na verificação de tipos. O sistema de tipos de Hunt e Sands motivou diversos trabalhos na área de segurança computacional, como é possível observar pela sua alta taxa de citações.

Quadros & Pereira [2011, 2012] apresentam uma análise estática que implementa o sistema de tipos de Denning & Denning [1977] lidando com o problema de vazamento de endereços via fluxo explícito. Nessa implementação o sistema de tipos foi convertido em um problema de busca em grafos de dependências construído a partir do fluxo de dados explícito detectado no código do programa, de forma que existe uma vulnerabilidade de vazamento de endereços por fluxo explícito se existir um caminho a partir do qual uma fonte de informação sigilosa alcança uma função pública de saída no grafo de dependência de dados. Em Quadros et al. [2012] uma análise dinâmica por meio de instrumentação de código foi concebida a fim de reduzir o número de falsos positivos e com isso aumentar a precisão dos resultados da análise estática anteriormente concebida. Entretanto, as análises estáticas e dinâmicas de Quadros et al não lidam com fluxo implícito, portanto elas são inconsistentes e geram falsos negativos. Isto é, tais soluções permitem que vulnerabilidades de vazamento de endereços via fluxo implícito ainda estejam presentes no programa analisado.

Análises de fluxo podem ser bastante úteis na detecção das vulnerabilidades de fluxo contaminado e vazamento de informação. Entretanto, uma análise consistente, isto é, com ausência de falsos negativos, deve considerar não somente o fluxo explícito de dados, mas também o fluxo implícito devido ao controle. Portanto, análises de fluxo de informação precisam ser capazes de detectar as dependências de dados e de controle.

A detecção das dependências de dados é uma tarefa sem grandes desafios, pois a mesma está evidenciada na sintaxe do programa e pode ser facilmente obtida por um *parser* de compilador. Já as dependências de controle exigem um esforço maior. Como já informado, Ferrante *et al.* propuseram um algoritmo capaz de detectar as dependências de controle, porém, esta tese advoga que sua alta complexidade quadrática poderia ser reduzida usando uma noção diferente de dependência implícita, que evitaria o seu pior caso. Ao contrário da ideia de Ferrante *et al.* em que dependência de controle determina quais instruções podem, ou não podem *executar*, Análises de fluxo no estilo de Denning & Denning [1977] estão mais focadas em determinar quais valores podem *influenciar* outros valores. Isso pode ser usado na concepção de um novo algoritmo de detecção dessas dependências, mas com um custo linear.

Importantes vulnerabilidades que aterrorizam os desenvolvedores poderiam ser detectadas de forma estática e automática através de uma análise de fluxo incorporada no compilador. Obviamente, a fim de evitar falsos negativos, essa análise deverá ser conservadora, o que por outro lado pode incorrer em alguns falsos positivos. Entretanto, é notável o benefício de se garantir que um sistema esteja livre de vulnerabilidades graves como: vazamento de endereços, estouro de arranjo e de inteiro, canais laterais, *SQL Injection* etc. Uma análise de fluxo é uma poderosa ferramenta de auxílio na detecção destas e outras vulnerabilidades, pois uma vez conhecido todo o fluxo e dados e de controle, pode-se customizar a análise a fim de rastrear qualquer tipo de vulnerabilidade ligada ao fluxo. As análises podem ser categorizadas entre análise esparsa e análise densa. Uma análise densa associa informação a cada ponto do programa. Um ponto do programa pode ser considerado aquele entre duas sentenças consecutivas. Já uma análise esparsa associa informação a cada variáveis do programa. Análises esparsas tendem a ser muito mais rápidas e com baixa demanda por recursos de armazenamento.

1.2 Tese

O principal objetivo desta tese é propor e avaliar uma análise esparsa de fluxo de informação para a detecção estática de vulnerabilidades relacionadas ao fluxo de dados e/ou de controle em programas de computador. Assim, esta tese defende que análises estáticas esparsas *são consistentes na detecção de importantes vulnerabilidades e de rápida execução* inclusive em programas reais com mais de 600 mil linhas de código. Entende-se por consistente, uma análise que não gera falsos negativos.

1.3 Contribuições Teóricas

Duas grandes contribuições para o estado da arte de análise de fluxo de informação podem ser citadas como fruto desta tese:

1. Um novo algoritmo para detecção de dependências implícitas e consequente construção do grafo de dependências, aqui chamado de grafo SSA, o qual é bastante utilizado como uma estrutura de dados que facilite a análise de fluxo. Esse novo algoritmo evita os piores casos da técnica criada por Ferrante *et. al* [Ferrante et al., 1987] largamente utilizada por outros trabalhos relacionados.

2. Redução da complexidade de implementação do sistema de tipos seguros de Hunt e Sands [Hunt & Sands, 2006]. Isso é alcançado basicamente pela utilização da representação de programas na forma *Static Single Assignment - SSA*, o que torna a análise esparsa.

1.4 Contribuições Práticas

O arcabouço teórico aqui construído pode e será utilizado nesta tese para a detecção consistente de vulnerabilidades de *software* relacionadas ao fluxo de informação. Essa detecção foi realizada por *FlowTracker* uma ferramenta desenvolvida no contexto deste trabalho de tese, que é capaz de analisar programas estaticamente e descobrir as vulnerabilidades listadas abaixo.

1. **Detecção de vazamento de informações sigilosas:** Um dos mais sérios problemas de segurança computacional é o vazamento de informações sigilosas. Para prevenir esse tipo de vulnerabilidade, o programador deve evitar que dados sigilosos alcancem um canal público durante a execução do programa. Uma das formas de evitar tal vulnerabilidade é observar se no fluxo explícito e implícito de informação existe alguma possibilidade desses dados alcançarem funções de saída como *printf* e *fprintf*, por exemplo. A fim de ajudar desenvolvedores a encontrar tais fluxos problemáticos, projetou-se e implementou-se uma ferramenta que analisa o código de programas detectando estaticamente, a possibilidade de uma informação sigilosa alcançar um canal público. Essa ferramenta é descrita em detalhes no Capítulo 4.

2. **Detecção de potenciais estouros de arranjo:** Estouros de arranjo, uma vulnerabilidade de *software* bastante conhecida na literatura especializada em

segurança, são frequentemente utilizados por adversários que têm o objetivo de corromper o fluxo de controle do programa. Isto é possível em linguagens fracamente tipadas como C e C++ onde os acessos aos arranjos não são verificados. Uma possível solução é a inserção de código para verificação de todos os acessos aos arranjos de forma a evitar aqueles fora dos limites. Entretanto o custo em tempo de execução seria proibitivo. Este trabalho propõe a detecção de potenciais estouros de arranjo via análise estática de código, o que permitiria ao desenvolvedor, a inserção de código de verificação apenas nos traços estáticos possivelmente vulneráveis. Verificou-se que cerca de 41% do código fonte dos benchmarks SPEC CPUINT 2006 estão vulneráveis a este tipo de ataque. Este resultado pode ser visualizado no Capítulo 4, Subseção 4.7.3.

3. **Detecção de potenciais estouros de inteiro:** Em 1996, o foguete Ariane 5 foi perdido devido a um estouro de inteiro – uma falha que custou mais de US$370 milhões [Dowson, 1997]. Esse tipo de problema ocorre quando após diversas transformações sobre um dado inteiro, este acaba por ultrapassar o seu valor máximo permitido devido ao espaço em bits limitado por cada arquitetura. A análise estática aqui proposta é capaz de detectar caminhos no programa que possam levar inteiros operados por operações aritméticas que podem ser "arredondadas", isto é, podem gerar estouro aritmético até instruções de carga e armazenamento na memória, pois adversários podem usar estouro de inteiro para habilitar estouro de arranjo. A ferramenta descrita no Capítulo 4 também é capaz de detectar esse problema.

4. **Detecção de canais laterais baseados em tempo:** Canais laterais baseados em tempo são vulnerabilidades ligadas à implementação de sistemas criptográficos e que permitem ao adversário conhecer acerca de uma informação sigilosa através de minuciosas observações do tempo de execução do programa. Mascaramento e sistemas de tipos já foram propostos objetivando mitigar esse problema. Esta tese propõe uma alternativa a essas soluções, focada em análise estática de fluxo de informação aplicando-se essa análise na biblioteca NaCl e em porções da OpenSSL, foi possível validar a boa qualidade da primeira e reportar vários traços vulneráveis na segunda. Este resultado pode ser visualizado no Capítulo 4.7.3, Subseção 4.7.1.

A técnica de análise esparsa e estática aqui explorada pode ser utilizada também como ferramenta de construção de *Program slices* em tempo linear. *Program Slicing* é um mecanismo que pode ser incorporado no compilador e utilizado para obter a

parte - *slice* - do código fonte que potencialmente afeta os valores armazenados em algum ponto de interesse. A construção de *slices* requer a obtenção da relação de dependência explícita e implícita entre as variáveis do programa. Essas duas relações podem ser explicitadas no PDG de Ferrante *et al.*, que possui alta complexidade, gerando no seu limite superior $O(|I| \times |E|)$ arestas de fluxo implícito, conforme já mencionado. Esta tese mostra que é possível construir análises de fluxo de informação usando uma noção diferente de dependências implícitas, que gera um número de arestas linear sob o número de definições somado ao número de usos das variáveis. O algoritmo aqui apresentado, o qual pode ser visto como uma grande inovação ao estado da arte, obtém essas dependências em uma única travessia sobre a árvore de dominância[3] do programa.

A eficiência do novo algoritmo é possível devido à exploração de uma propriedade chave de programa na forma *SSA*: a definição de uma variável domina todos os seus usos. Com o novo algoritmo é possível corretamente implementar o sistema de tipos seguros de Hunt & Sands [2006] o qual existia apenas na esfera teórica e não possuía, até onde se constatou nesta tese, uma implementação e uso prático. Além disso, conseguiu-se uma complexidade de espaço e tempo da ordem de $O(I)$, contrariamente à sua formulação original, que requeria $O(I^2)$ espaço e tempo para programas estruturados. Com esse sistema de tipos implementado e utilizando o novo algoritmo, pôde-se implementar *FlowTracker* eficientemente.

1.5 Publicações

Durante o desenvolvimento desta tese, o estudante publicou seis trabalhos nas principais conferências de compiladores e conferências de segurança da informação no Brasil, sendo quatro deles premiados. Também foi gerada uma publicação na importante conferência *Compiler Construction 2016 (CC'16)*. Além de uma submissão para o periódico *TOPLAS* que está sob revisão. Vale ressaltar que todas as submissões para conferências nacionais foram aceitas para publicação, evidenciando a boa aceitação desta pesquisa pela comunidade de compiladores e segurança em nosso país.

O primeiro trabalho foi publicado na seção de ferramentas do *Congresso Brasileiro de Software: Teoria e Prática (CBSoft 2013)* [Rodrigues et al., 2013b] e tratava da apresentação de uma ferramenta para a detecção de vazamentos de endereços em programas escritos na linguagem C/C++. Tal ferramenta foi descontinuada por utilizar um algoritmo de detecção de fluxos implícitos bastante ineficiente. Porém ela foi a

[3] A definição de árvore de dominância pode ser encontrada no capítulo 3 - Seção 3.2.

primeira ferramenta, até onde se tem conhecimento, capaz de detectar estaticamente vazamento de informação sigilosa de forma consistente, isto é, sem a possibilidade de falsos negativos visto que a mesma considerava tanto os fluxos implícitos quanto explícitos de informação.

O segundo trabalho relacionado à esta tese foi publicado em *XIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2013)* [Rodrigues et al., 2013a] que foi agraciado com o prêmio de menção honrosa. Nesse trabalho, todo arcabouço teórico utilizado na concepção da ferramenta apresentada no CBSoft 2013, é apresentado e detalhado. Pode-se dizer que esse é o trabalho precursor de todas as ideias que foram desenvolvidas nesta tese, pois com ele pôde-se observar a necessidade de um algoritmo mais eficiente para a detecção dos fluxos implícitos e consequente geração das arestas de controle no grafo SSA que será detalhado no Capítulo 3.

No *XVIII Simpósio Brasileiro de Linguagens de Programação (SBLP 2014)* [Rodrigues, 2014b] foi publicado o terceiro trabalho oriundo desta tese que foi contemplado com o prêmio de terceiro melhor artigo da conferência. Nesse trabalho, apresentou-se pela primeira vez o novo algoritmo para detecção de fluxo implícitos que diferentemente da abordagem de Ferrante *et al.*, é capaz de criar no máximo $O(I)$ arestas de controle, onde I é o número de instruções do programa.

Ainda em 2014, no *XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2014)* foi publicado o trabalho *Detecção Estática e Consistente de Potenciais Estouros de Arranjos* [Rodrigues, 2014a] que apresentava uma abordagem de detecção de possíveis estouros de arranjos por meio de análise estática de fluxo de informação. Nesse trabalho foi possível mostrar que a solução de segurança que esta tese havia proposto no ano anterior era mais genérica do que se imaginava e podia ser facilmente adaptada para a detecção de outras vulnerabilidades além do vazamento de informação sigilosa.

Em 2015 o estudante publicou no *Congresso Brasileiro de Software: Teoria e Prática (CBSoft 2015)* um trabalho premiado como segundo melhor artigo da seção de ferramentas e titulado *FlowTracker - Detecção de Código Não Isócrono via Análise Estática de Fluxo* [Rodrigues et al., 2015b]. Este trabalho apresenta a versão final da ferramenta de análise estática de fluxo de informação utilizando o algoritmo desenvolvido no ano anterior, bem como várias melhorias como disponibilidade de acesso *online*[4] via uma interface *Web* além de possibilitar ao usuário informar por meio de uma sintaxe XML simples quais dados do código fonte serão considerados como sigilo-

[4]http://cuda.dcc.ufmg.br/flowtracker/

sos. *FlowTracker* é então capaz de detectar canais laterais baseados no tempo através da construção do grafo de dependências e busca por caminhos conectando tais dados à instruções de desvio condicionais e indexação de memória.

Ainda em 2015, foi publicado um trabalho também premiado como segundo melhor artigo na conferência *XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2015)* [Rodrigues et al., 2015a]. Nele, foi apresentado o problema de canais laterais baseados em tempo com vistas à solução proposta nesta tese. Foi também apresentado, de maneira formal a corretude do algoritmo desenvolvido no ano anterior além de uma análise do mesmo quanto à escalabilidade, adaptabilidade e efetividade.

Finalmente em 2016 publicou-se na *25th International Conference on Compiler Construction (CC 2016)* [Rodrigues et al., 2016] um artigo apresentando o algoritmo que foi desenvolvido nesta tese, bem como uma análise formal e empírica de suas vantagens sobre a abordagem de Ferrante *at. al* que até então é o estado da arte do que diz respeito ao rastreamento de fluxo implícito de informação. Acredita-se que com a divulgação desse resultado em CC 2016, a comunidade de compiladores passe a utilizar esse novo algoritmo em aplicações que demandem o rastreamento consistente e eficiente de fluxo de informação implícito.

1.6 Organização do trabalho

Esta Tese está organizada da seguinte forma. O Capítulo 2 apresenta o *background* e os trabalhos relacionados, sendo dividido da seguinte forma. Primeiramente apresentam-se alguns problemas práticos relacionados à segurança e que podem se detectados em tempo de compilação utilizando a abordagem defendida neste trabalho de tese. O primeiro problema versa sobre o vazamento de informação sigilosa, mais especificamente, o vazamento de endereços. O segundo problema abordado é o estouro de arranjo e o terceiro é um importante problema que pode existir em implementações vulneráveis de conhecidos e robustos algoritmos criptográficos, chamado de vazamento de informação por canais laterais baseados em tempo. Em seguida, o mesmo capítulo aborda a técnica de *Program Slicing* que para ser realizada necessita do completo rastreamento dos fluxos implícitos de informação e portanto pode ser beneficiada com o algoritmo de detecção de tais fluxos desenvolvido nesta Tese. Por fim, este capítulo cita o grafo de dependências de Ferrante *et al.*[Ferrante et al., 1987, p.323] que é a solução de linha de base usada como comparação à abordagem aqui defendida. O Capítulo 3, como núcleo desta Tese, apresenta a abordagem de rastreamento de fluxo implícito criada

neste trabalho, bem como uma implementação eficiente do famoso sistema de tipos de Hunt e Sands, que usa tal abordagem de forma esparsa e com isso é utilizado para detectar os problemas relacionados acima. Finalmente no Capítulo 4 apresenta-se a ferramenta FlowTracker criada a partir da implementação esparsa do sistema de tipos de Hunt e Sands e focada na detecção de canais laterais baseados em tempo. Entretanto, tal ferramenta pôde ser facilmente adaptada para detectar os demais problemas de segurança citados.

Capítulo 2

Background e Trabalhos Relacionados

Este trabalho utiliza os conceitos de análise estática e análise esparsa de fluxo de informação para a detecção consistente de algumas das principais vulnerabilidades relacionados ao fluxo de informação em programas de computador. Baseado nisto a Seção 2.2 apresenta algumas aplicações da análise esparsa defendida nesta tese, sendo elas: a detecção de vazamento de endereços cuja descrição se encontra na Subseção 2.2.1; a detecção de possíveis estouros de arranjo que é descrito na Subseção 2.2.2 e a detecção de canais laterais baseados em tempo descritos na Subseção 2.2.3. Por fim é descrita a técnica de *Program Slicing* na Subseção 2.2.4 que também pode ser beneficiada pelo arcabouço teórico defendido nesta tese. Finalmente a Seção 2.3 descreve o estado da arte no que diz respeito ao rastreamento de fluxos implícitos, mais especificamente o trabalho de Ferrante *at. al* e a literatura que o vem utilizando desde a sua introdução na comunidade de compiladores. A Seção 5.1 apresenta as considerações finais deste capítulo.

2.1 Conceitos básicos

2.1.1 Grafo de Fluxo de Controle

Um *Grafo de Fluxo de Controle - GFC* é um grafo dirigido que possui dois vértices especiais Start e End, de forma que todo vértice é alcançável a partir de Start, nenhum vértice alcança Start, todo vértice alcança End, e End não alcança qualquer vértice. Vértices em um grafo de fluxo de controle são chamados *Blocos Básicos*. Um bloco básico é a maior sequência de instruções, com as seguintes propriedades:

1. O fluxo de controle pode somente iniciar um bloco básico pela sua primeira instrução. Isto é, não é possível realizar saltos de algum ponto do programa fora do bloco básico para alguma instrução se não a primeira do bloco básico em questão.

2. O fluxo de controle pode somente deixar um bloco básico, por sua última instrução.

Em outras palavras, uma vez que se inicia a execução da primeira instrução de um bloco básico, é garantido que todas as demais instrução do mesmo bloco serão executadas. A primeira instrução de um bloco básico é chamada de instrução líder. Ela pode ser identificada via três propriedades:

1. A primeira instrução do código é uma instrução líder.

2. Qualquer instrução alvo de um desvio condicional ou incondicional será uma instrução líder.

3. Qualquer instrução que imediatamente segue um teste condicional ou um desvio incondicional, será uma instrução líder.

Um bloco básico consiste de uma instrução líder e todas as demais instruções até a próxima líder. A Figura 2.1 exemplifica esse conceito exibindo em (a) um simples trecho de programa e em (b) seu respectivo grafo de fluxo de controle. Existem dois caminhos a serem tomados a partir da execução do primeiro bloco básico ℓ_1-ℓ_2, isto é, pode-se seguir para dois possíveis blocos dependendo da avaliação do predicado p. Caso p seja verdadeiro, segue-se para o bloco ℓ_3, do contrário, o bloco ℓ_5 é executado. Além disso, um bloco básico pode ser executado mais de uma vez em caso de laços de repetição. Nesse caso haverá arestas direcionais saindo e entrando no mesmo bloco básico.

2.1.2 Dominância e Pós-Dominância

Um bloco B' domina outro bloco B se todo caminho a partir de Start para B deve passar por B'. Dualmente, B *pós-domina* B' se todo caminho a partir de B' para End deve passar por B. Seja dom_B o conjunto de blocos que *dominam* o bloco B, e seja pdom_B o conjunto de blocos que pós-dominam B. Se $B' \in \mathsf{dom}_B$, e para qualquer outro $B'' \in \mathsf{dom}_B, B'' \neq B'$ tem-se que $B' \in \mathsf{dom}_{B''}$, diz-se que B' é o *dominador imediato* de B, denotado como idom_B. Define-se o *pós-dominador imediato* de B como o dual do dominador imediato, e denotado por pdom_B. O dominador imediato e o pós-dominador

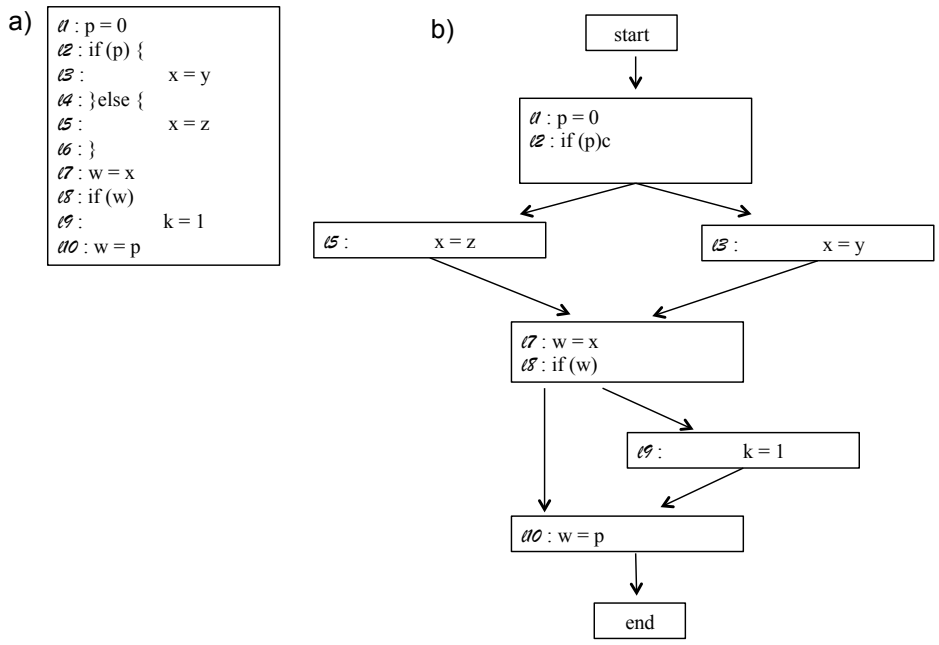


Figura 2.1. Exemplo de grafo de fluxo de controle.

imediato de qualquer bloco básico em um grafo de fluxo de controle é único [Appel & Palsberg, 2002]; assim, essas duas noções definem árvores: a *árvore de dominância* e a *árvore de pós-dominância*.

A Figura 2.2 exibe alguns exemplos de dominância e pós-dominância. No GFC mais a esquerda é possível ver que o bloco básico a domina b, pois todos os caminhos que levam o bloco Start (representado pelo círculo preenchido) ao bloco b, passam obrigatoriamente por a. Além disso, no mesmo GFC nota-se que o bloco b pós-domina o bloco a, pois todos os caminhos que levam do bloco a ao bloco End (representado pelo círculo semi-preenchido), passam obrigatoriamente por b.

Na mesma Figura 2.2 é possível ver no GFC central que o bloco n pó-domina m, mas o bloco m não domina n pois vê-se que há um caminho de Start diretamente para o bloco n. Por fim no GFC mais à direita vê-se que o bloco x domina o bloco y, porém y não pós-domina x, pois há um caminho conectando o bloco x ao bloco End e que não passa por y.

2.1.3 Região *Hammock*

De acordo com a definição encontrada em Ferrante et al. [1987], uma região *hammock* em um grafo de fluxo de controle é também conhecida como uma região de entrada única e saída única.

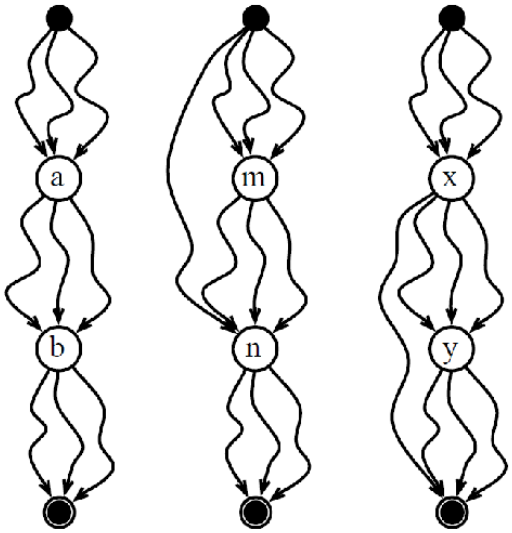


Figura 2.2. Exemplo de dominância e pós-dominância.

Definição 2.1.1 *Se G um grafo de fluxo e controle para o programa P. Uma região hammock H é um subgrafo induzido de G com um vértice V em H chamado vértice de entrada e um vértice W não pertencente ao subgrafo H chamado vértice de saída tal que:*

1. *Todas arestas de $(G - H)$ para H passam por V.*

2. *Todas arestas de H para $(G - H)$ passam por W.*

Ainda segundo Ferrante *et al.*, o seguinte teorema, cuja prova pode ser encontrada em Ferrante et al. [1987], destaca uma das propriedades cruciais de regiões *hammock* com respeito a dependência de controle.

Theorema 2.1.2 *Seja G um GFC para o programa P. Seja H uma região hammock de G. Se o bloco básico X está em H e Y está em $(G - H)$, então Y não é dependente de controle de X.*

Na Figura 2.1 (b) pode-se identificar 2 regiões *hammock*, a primeira delas corresponde as blocos básicos ℓ_1, ℓ_3, ℓ_5, ℓ_7, onde o vértice V é o bloco ℓ_1 e o vértice W é o bloco ℓ_7 A segunda região corresponde aos blocos ℓ_7, ℓ_9 e ℓ_10, onde V é o bloco ℓ_7 e W representa o bloco ℓ_10.

2.1.4 *Static Single Assingment Form* - Forma SSA

Muitas análises de fluxo de dados necessitam conhecer o local de uso de cada variável definida ou o local de definição de cada variável usada em uma expressão. A cadeia definição-uso é uma estrutura de dados que torna isso possível: Para cada sentença no grafo de fluxo de controle, o compilador pode manter uma lista de ponteiros para todos os locais de uso das variáveis definidas no código, e uma lista de ponteiros para todos os locais de definição de variáveis usadas. Desta maneira o compilador pode encontrar rapidamente o local de uso ou de definição de cada variável do programa.

Um aprimoramento dessa idéia de cadeia definição-uso é a forma de atribuição única estática, ou mais conhecida na literatura como *static single-assignment form*, ou mais ainda forma SSA. Uma representação intermediária na qual variáveis têm apenas um única definição no texto do programa. O único local de definição estático pode ser um laço de repetição que é executado muitas vezes, devido a isso, o termo estático é utilizado na sua nomenclatura, ao invés de apenas forma de atribuição única, nas quais variáveis nunca são definidas mais de uma vez. [Appel & Palsberg, 2002]

A forma SSA é útil por algumas razões:

1. Análises de fluxo de informação e algoritmos de otimização de código são mais simples quando operam sobre programas com apenas um local de definição para cada variável.

2. Se uma variável possui N usos e M definições (o qual ocupa $N + M$ instruções no programa), cadeias definição-uso ocupam um espaço proporcional a $N * M$, levando à um crescimento quadrático do espaço necessário para armazená-las. Em contrapartida a forma SSA possui crescimento linear sobre o tamanho do programa original.

3. Forma SSA simplifica a construção de importantes estruturas de dados muito utilizadas em otimização de código por parte dos compiladores modernos, tais como: árvore de dominância, árvore de pós-dominância e grafos de interferência.

A Figura 2.3 mostra que em um código sem instruções de desvio, é possível convertê-lo para a forma SSA fazendo com que cada instrução defina sempre uma nova variável invés de redefinir uma variável já existente. Cada nova definição de uma mesma variável (no exemplo, variável a) é modificada para definir uma nova variável $(a1, a2, ...)$, e cada uso da variável é modificado para usar a definição mais recente dela.

Porém, quando dois caminhos no grafo de fluxo de controle convergem para o mesmo bloco básico, não é obvio como ter apenas uma definição para cada variável.

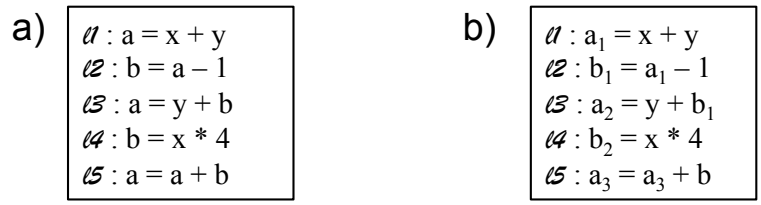


Figura 2.3. Código sem instrução de desvio, convertido para forma SSA.(Figura reproduzida do trabalho de Appel & Palsberg [2002])

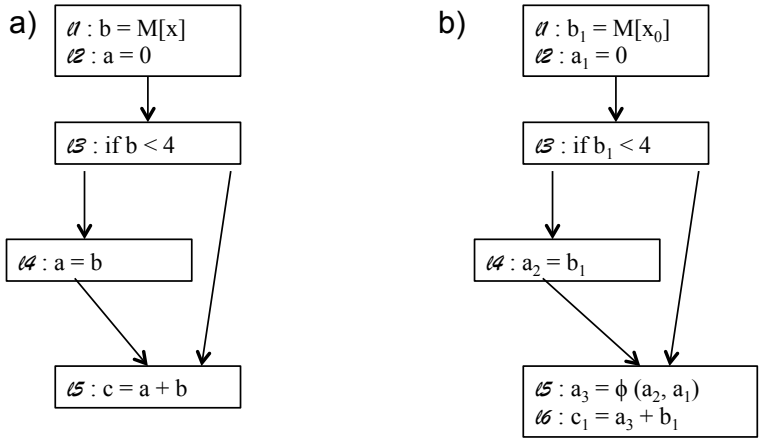


Figura 2.4. (a)Código com uma junção de fluxo de controle; (b) Representação na forma SSA. (Figura reproduzida do trabalho de Appel & Palsberg [2002])

Por exemplo, na Figura 2.4 (a), se há uma nova definição da variável a no bloco ℓ_1-ℓ_2 e no bloco ℓ_4, qual versão deve ser usada no bloco ℓ_5? Isto é, quando há sentença que possui mais que um antecessor, não existe a noção de "mais recente".

A fim de solucionar esse problema, foi introduzida uma função notacional conhecida como função ϕ. A Figura 2.4 (b) mostra que é possível combinar $a1$ definida no bloco ℓ_1-ℓ_2 e $a2$ definida no bloco ℓ_4 usando a função $a3 = \phi(a1, a2)$. Porém, não como uma função matemática ordinária, $\phi(a1, a2)$ retorna $a1$ se o controle alcança o bloco ℓ_5-ℓ_6 através da aresta $\ell_3 \rightarrow \ell_5$, e retorna $a2$ se o controle origina da aresta $\ell_4 \rightarrow \ell_5$.

2.2 Aplicações

2.2.1 Vazamento de endereços

Certas informações de um programa não devem alcançar funções de saída públicas, tais como as funções *printf*, *fwrite* etc. Estas informações sigilosas podem ser usadas como auxílio em um ataque ao sistema computacional. Um exemplo típico de informação

que não deve chegar ao conhecimento dos usuários do sistema são os endereços de memória internos do programa. Isto é, os endereços de variáveis locais, dinâmicas e globais. Quando um programa permite que qualquer endereço possa atingir alguma função pública, temos uma vulnerabilidade de vazamento de endereços.

Os sistemas operacionais modernos utilizam um mecanismo de proteção chamado *ASLR - Address Space Layout Randomization* [Bhatkar et al., 2003; Shacham et al., 2004]. Esse mecanismo consiste em carregar os módulos binários que correspondem ao programa executável em diferentes endereços cada vez que o programa é executado. Essa técnica visa proteger o software de ataques como *return-to-libc* [Shacham et al., 2004] e *ROP - return-oriented-programming* [Buchanan et al., 2008; Shacham, 2007].

Entretanto, mesmo um sistema protegido por ASLR pode estar vulnerável caso algum de seus programas permita que endereços alcancem funções públicas de saída. No trabalho de Quadros et al. [2012] é possível encontrar um exemplo de ataque via estouro de arranjo que só foi possível graças ao vazamento de endereços. Nesse exemplo, o adversário assume o controle de uma máquina 32 bits sob o sistema operacional Ubuntu 11.10, um sistema protegido por ASLR.

O vazamento de informação pode ocorrer via fluxo explícito diretamente ou indiretamente. Quando uma variável apontadora p que contém um endereço de memória é passada como parâmetro para uma função de saída pública, temos um vazamento de endereços de forma explicitamente direta. Se o conteúdo desta variável é transferido para outra variável p' e esta é passada como parâmetro para a função de saída pública, temos um vazamento de endereços explicitamente indireto. Porém, os vazamentos de endereços podem também ocorrer por meio dos fluxos implícitos. Detectar tais vazamentos é mais desafiador do que quando eles ocorrem por fluxo explícitos.

A Figura 2.5 exibe dois vazamentos de informação: No lado esquerdo podemos verificar um vazamento explícito de informação ocorrendo através do fluxo de dados, pois o conteúdo da variável *secret* que contém um endereço de memória retornado pela função *malloc*() é transferido para a variável x que por sua vez é impressa pela função de saída pública $printf()$. No lado direito vemos um vazamento de informação por fluxo implícito. Neste caso, o conteúdo da variável *secret* não é transferido e posteriormente impresso, mas é utilizado na condição do predicado que influenciará o fluxo de controle do programa. Caso seja impresso o valor 1, o usuário, que tem conhecimento do código fonte, poderá inferir que o conteúdo de *secret* é 0. Esse conhecimento pode ser crucial para um ataque ao sistema e deve ser fortemente evitado.

Vazamento de endereços é um problema bastante conhecido e discutido em *blogs* e *forums*: Como um exemplo Dion Blazakis explica como utilizar inferência de ponteiros

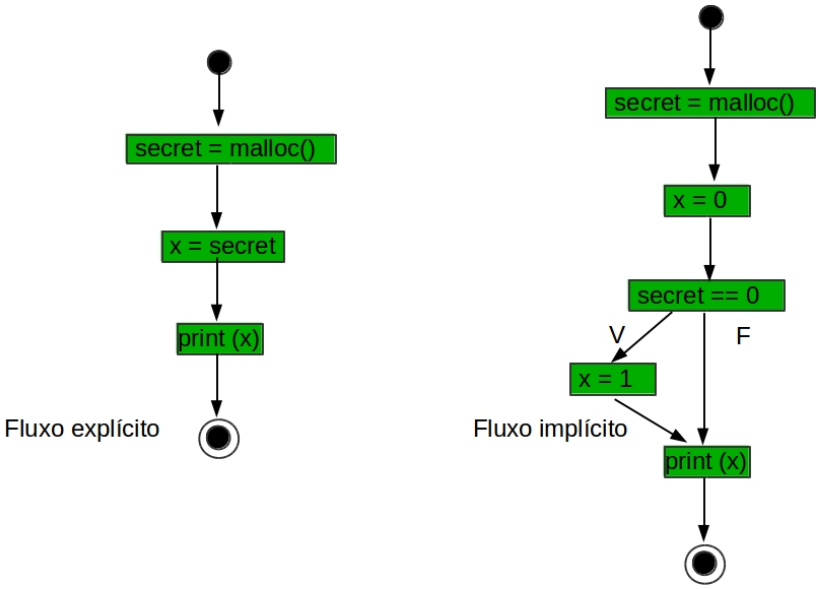


Figura 2.5. (a) Vazamento de informação (endereço) por fluxo explícito. (b) Vazamento de informação (endereço) por fluxo implícito.

para comprometer um compilador *JIT ActionScript* [1]. Esse tipo de vulnerabilidade também é mencionado por Fermin Serna como um sério problema[2]. Contudo, a academia ainda não concentrou seus esforços nesse problema, visto a inexistência de um número adequado de publicações que tratem desse assunto. Porém, as técnicas utilizadas nesta tese foram foco de muita pesquisa. Mais especificamente, este trabalho gera como produto final um *framework* do fluxo de informação, tal como proposto por Denning & Denning [1977].

2.2.2 Estouro de Arranjo

Um estouro de arranjo acontece quando este é preenchido com dados que ultrapassam os seus limites. Isso pode acontecer de forma proposital ou não, principalmente em linguagens fracamente tipadas e bastante utilizadas como C e C++ que não fazem verificação dos acessos ao arranjo. Isto possibilita a existência de uma grande quantidade de *worms* e vírus que contaminaram e contaminam milhões de dispositivos computacionais em todo o mundo. O estouro de arranjo pode sobrescrever os valores de variáveis locais e endereço de retorno de função, o que compromete todo o fluxo de dados e controle.

Usuários maliciosos de posse do código fonte podem manipular os dados de en-

[1] `http://www.semantiscope.com/research/BHDC2010`
[2] *The info leak era on software exploitation*, *slides* disponíveis on-line.

trada que são públicos a fim de estourar um arranjo e sobrescrever o endereço de retorno de uma função com informações que direcionem o fluxo de controle para funções de sistema tais como *telnet* e *shell* com os mesmos privilégios de sistema do programa atacado. Este redirecionamento pode causar desde a interrupção do serviço até a obtenção do controle total do sistema.

Uma solução empregada por muitos compiladores é a inserção de canários, que são valores aleatórios inseridos antes do valor de retorno de uma função. Além disso, é inserido a uma pequena seção de código de verificação antes do retorno, capaz de gerar uma exceção e encerrar o programa caso o canário tenha sido modificado. Portanto, caso um adversário tente sobrescrever o valor de retorno, inevitavelmente ele também sobrescreverá o canário que por sua vez será detectado no momento do retorno da função.

Entretanto, mesmo funções protegidas por canário estão vulneráveis ao ataque de estouro de arranjo [Maffra et al., 2013]. Isto acontece porque algumas variáveis locais podem ser alojadas na pilha da função após o espaço alocado para o arranjo e portanto, elas podem ser sobrescritas em um possível estouro. Um adversário pode, através de um estudo cauteloso do fluxo de dados/controle da função, escolher quais valores tais variáveis receberão e assim comprometer a execução de todo o programa.

O trabalho de Quadros *et. al.* [Quadros et al., 2012] demonstra como um adversário pode assumir o controle total de um sistema executando Ubuntu Linux, por meio de um ataque de estouro de arranjo. Em Maffra et al. [2013] pode se encontrar uma análise estática de código para a detecção de vulnerabilidade de ataque por estouro de arranjo em código compilado com canários. Entretanto, por não considerar as dependências de controle entre as variáveis e os predicados de instruções de desvio, essa análise não é consistente.

2.2.3 Canais Laterais Baseados em Tempo

Nas últimas décadas, pesquisadores descobriram que a resiliência de um algoritmo criptográfico depende não somente de seu projeto abstrato, mas também de sua implementação concreta [Kocher, 1996]. Em termos de projeto, um algoritmo criptográfico deve estar livre de vulnerabilidades teóricas que poderiam permitir a um adversário a chance de decifrar suas mensagens codificadas sem o devido direito a isso. Em termos de implementação, tal algoritmo deve estar livre de canais laterais. Um ataque por canal lateral busca coletar informações sigilosas relacionadas à chave ou estados internos secretos da implementação. Existem alguns tipos de ataque nesse sentido. Em alguns deles, adversários devem ter acesso ao sistema criptográfico alvo, de forma que

eles possam inserir falhas ou recuperar dados residuais da memória. Ataques menos invasivos também existem. Eles tentam detectar diferenças sensíveis no tempo de execução [Kocher, 1996], consumo de potência [Kocher et al., 1999], variações no campo eletromagnético [Quisquater & Samyde, 2001] ou ainda emanações acústicas [Genkin et al., 2014].

Canais laterais baseados em tempo permitem a um adversário monitorar pequenas flutuações no tempo de execução do algoritmo criptográfico. Essas variações são devidas aos desvios condicionais, otimizações no nível de instruções, desempenho da hierarquia de memória ou latência de comunicação. Ataques por análise de variação de tempo podem ser devastadores contra essas implementações inseguras. Por exemplo, eles são muito efetivos contra implementações ruins de *square-and-multiply* do algoritmo RSA e Diffie-Hellman ou software baseado em tabela da implementação do algoritmo AES. Contrário à crença popular, mesmo o ruído das conexões de rede não é suficiente para dificultar o vazamento de informação por análise de tempo [Brumley & Boneh, 2005].

Assegurar o comportamento de tempo constante de execução é uma proteção natural contra ataques por análise de variação de tempo. Em software, isso é alcançado pela programação sem instrução de desvio ou redução da dependência sobre dados pré-computados; ou pela seleção rigorosa de parâmetros com regularidade intrínseca [Bernstein, 2006]. Apesar desses bem conhecidos mecanismos de proteção, implementações criptográficas resistentes a ataques por análise de variação de tempo devem ainda ser cuidadosamente validadas quanto às suas propriedades isócronas, visto que canais laterais podem ser descuidadosamente inseridos por um programador não treinado ou mesmo por ferramentas de auxílio ao desenvolvimento. Auditoria nesses casos usualmente requer a examinação de código complexo por um profissional experiente, completamente consciente das características específicas da tecnologia envolvida. Embora existam ferramentas que auxiliem essa inspeção manual de código [Chen et al., 2014], acredita-se que muito ainda deve ser feito nessa direção. Em particular, não existe técnica automatizada que auxilie na validação do comportamento invariante de tempo de programas compilados. Esse é um problema sério, visto que compiladores podem inserir canais laterais durante a compilação e/ou otimização de programas que foram validados quanto à não possibilidade de ataques por análise de variação de tempo.

Canais laterais baseados no tempo são um problema bem conhecido. Muito do atual conhecimento sobre esse problema é devido a Kocher [1996]. A conjectura de Kocher - que o tempo de execução pode ser usado para obter informação sobre dados sensíveis - foi demonstrada por vários pesquisadores. Dhem et al. [2000] mostraram

como implementar um ataque por variação de tempo em uma implementação RSA executando em um *smart card*. Posteriormente, Brumley & Boneh [2005] mostraram que é possível recuperar informação sigilosa de uma implementação mesmo em face de ruídos introduzidos pela rede de comunicação. Eles foram capazes de recuperar a chave privada RSA de um sistema *Web* com políticas de segurança baseadas na biblioteca OpenSSL. Neste trabalho a máquina adversária e o servidor *Web* estavam localizados em prédios diferentes com três roteadores e vários *switches* entre eles. Poucos anos mas tarde, Brumley & Tuveri [2011] montaram uma recuperação completa da chave contra um servidor TLS que usava assinaturas ECDSA para autenticação. Para uma visão mais geral sobre o campo de ataques por variação de tempo, é recomendado um tutorial apresentado por Emmanuel Prouff na conferência CHES'13 [Prouff, 2013].

Esta tese reconhece dois tipos de vazamentos de informação baseados na análise do tempo de execução da implementação vulnerável. Na primeira categoria, agrupam-se vazamentos que ocorrem quando dados secretos determinam quais partes do código do programa serão executadas. Na segunda categoria encontram-se os programas nos quais a memória é indexada por informação sensível. Nesta seção, apresenta-se um exemplo de cada um desses tipos de vulnerabilidade. Para tanto, uma função simples será usada como exemplo. Ela recebe uma senha codificada como um arranjo de caracteres `pw`, e tenta fazer o casamento dessa cadeia contra outro arranjo `in`, que representa uma entrada fornecida por um usuário externo. Neste exemplo, considera-se que a entrada do usuário pode ser contaminada com dados de seu interesse.

Vazamento devido ao fluxo de controle. O programa na Figura 2.6 (a) contém um vazamento de informação baseado em tempo. Nesse exemplo, um adversário pode perceber quanto tempo leva para a função `isDiffVul1` retornar. Um retorno antecipado indica que o casamento na linha 4 falhou em um dos primeiros caracteres. Através da variação, em ordem lexicográfica, do conteúdo do arranjo `in`, o adversário pode reduzir de exponencial para linear a complexidade da busca pela senha.

Vazamento devido ao comportamento da memória *cache*. O programa na Figura 2.6 (b) é uma tentativa de remover o canal lateral baseado em tempo do programa apresentado na Figura 2.6 (a). A Função `isDiffVul2` usa uma tabela para verificar se os caracteres usados na senha `pw`, combinam com aqueles apresentados no arranjo de entrada *in*. Se todos os caracteres em ambas cadeias aparecem na mesma ordem, a função retorna verdade, por outro lado retorna falso. A senha `pw` não controla qualquer instruções de desvio na função `isDiffVul2`; porém, este código ainda apresenta um vazamento baseado no tempo de execução. Dados pertencentes à senha são usados para indexar memória na linha 6 do exemplo. Dependendo da distância relativa entre os caracteres de `pw`, algumas falhas de cache podem acontecer. Nesse caso, um adversário

pode obter informação sobre quão espaçados estão os elementos alfanuméricos de `pw`. A praticidade desse tipo de ataque foi demonstrada em trabalhos anteriores [Bernstein, 2005].

Removendo canais laterais baseados em tempo O programa na Figura 2.7 é uma versão sanitizada do exemplo de casamento de caracteres. Neste caso, independentemente do valor secreto armazenado em `pw`, a função `isDiffOk` segue exatamente o mesmo fluxo de controle. Em outras palavras, cada caminho dentro do programa será percorrido o mesmo número de vezes e na mesma ordem, não importando os dados de entrada. Mais ainda, a função `isDiffOk` irá indexar os mesmos blocos de memória, sempre na mesma ordem, e sempre com os mesmo intervalos entre sucessivos acessos, independentemente da entrada. Nesse caso, diz-se que informação secreta não influência no fluxo de controle nem na indexação de memória. A técnica descrita nesta tese é capaz de detectar os dois tipos de vulnerabilidades das Figuras 2.6 (a) and (b), e não reporta qualquer falso positivo para o programa apresentado na Figura 2.7.

Técnicas para detectar e evitar vazamentos por análise de variação de tempo.

Existem várias metodologias e linhas gerais para evitar canais laterais baseados em tempo. Agat [2000] propôs um sistema de tipos para transformar um programa vulnerável em outro seguro. Ele realizou essa transformação pela inserção de instruções inócuas nos blocos de desvio, para mitigar a diferença no tempo de execução de diferentes caminhos que podem ser tomados à partir de um teste condicional. De forma similar, Molnar et al. [2006] projetaram um tradutor C fonte-fonte que detecta e conserta vazamentos baseados no fluxo de controle. Contrariamente ao trabalho aqui apresentado, as abordagens de Agat e de Molnar *et al.* não podem lidar com

```
1 int isDiffVul1(char *pw, char *in) {
2   int i;
3   for (i=0; i<7; i++) {
4     if (pw[i]!=in[i]) {
5       return 0;
6     }
7   }
8   return 1;
9 }
```

(a)

```
1 int isDiffVul2(char *pw, char *in) {
2   int i;
3   int isDiff = 0;
4   char array[128] = { 0 };
5   for (i=0; i<7; i++) {
6     array[pw[i]] += i;
7   }
8   for (i=0; i<7; i++) {
9     array[in[i]] -= i;
10  }
11  for (i=0; i<128; i++) {
12    isDiff |= array[i];
13  }
14  return isDiff;
15 }
```

(b)

Figura 2.6. (a) Programa no qual o fluxo de controle é controlado por informação sigilosa. (b) Programa que permite vazamento de informação devido ao comportamento da memória *cache*.

```
1 #define F(i) diff |= pw[i] ^ in[i]
2
3 int isDiffOk(char *pw, char *in) {
4   int diff = 0;
5   F(0);
6   F(1);
7   F(2);
8   F(3);
9   F(4);
10  F(5);
11  F(6);
12  F(7);
13
14  return (1 & ((diff - 1) >> 8)) - 1;
15 }
```

Figura 2.7. Implementação isócrona da função `isDiffVul1` e `isDiffVul2` na Figura 2.6.

vazamentos baseados no comportamento da *cache*.

Mais próximo à proposta desta tese, Lux & Starostin [2011] implementaram uma ferramenta que detecta vulnerabilidades de ataque por análise de tempo em programas Java. Entretanto o trabalho de Lux *et al*'s opera em uma linguagem de programação de alto nível, usando um conjunto de regras de inferência similares àquelas propostas por Hunt & Sands [2006]. Clama-se que a abordagem aqui defendida possui vantagens, porque atua diretamente na representação intermediária do compilador. Portanto, ela pode lidar com diferentes linguagens de programação e não precisa confiar no compilador quanto à não inserção de canais laterais de forma acidental no código executável. Além disso, o algoritmo aqui apresentado é substancialmente diferente de Lux *et al.*, porque ele pode lidar com programas não estruturados.

2.2.4 *Program Slicing*

Um *Program Slice* é a parte de um código fonte que potencialmente afeta os valores armazenados em algum ponto de interesse. O ponto de interesse é usualmente chamado de *slice criterion*. Em nosso caso, um *slice criterion* é uma variável de programa. A tarefa de encontrar a parte do programa que corresponde a uma variável v é chamado de *Program Slicing*. De acordo com Weiser, um *slice* S que corresponde à uma variável v é um programa reduzido, mas executável, obtido a partir do programa original P removendo sentenças que não influenciam na computação de v.

O conceito de *Program Slicing* foi introduzido por [Weiser, 1979, 1981, 1982]

e pode ser caracterizado como estático ou dinâmico. Técnicas de *slicing* estáticas realizam a análise em uma representação intermediária estática do código fonte para obter o *slice*. O código fonte é analisado e os *slices* são computados considerando todos os possíveis valores de entrada. Um *slice* estático contém todas as sentenças que podem afetar o valor de uma variável em um ponto do programa para qualquer entrada possível. Portanto, é necessário ser conservador na análise, o que leva a *slices* relativamente grandes. Isto é, *slices* estáticos podem conter algumas sentenças que não serão realmente executadas.

Korel e Lask [Korel, 1988] introduziram o conceito de *slicing* dinâmico que faz uso de uma execução particular de um programa. Um *slicing* dinâmico que corresponde a um *slicing criterion* para uma execução particular, contém sentenças que afetam o *slicing criterion* naquela execução particular. Portanto *slices* dinâmicos são usualmente menores que *slices* estáticos e são mais úteis em aplicações interativas como programas de *debugging* e testes. Um resumo dos principais algoritmos de *slicing* dinâmicos pode ser encontrado em Korel & Rilling [1998].

Program Slicing podem ainda ser caracterizados como *Backward* e *Forward Slicing*. Um *backward slicing* contém toda a parte do programa que pode direta ou indiretamente afetar o *slicing criterion*. Assim, um *backward slice* estático fornece a resposta para a seguinte questão: Quais sentenças **afetam** o *slicing criterion*? Já um *forward slice* correspondente a um *slicing criterion* $< p, V >$ contém toda a parte do programa que pode ser afetada pelas variáveis no conjunto V usadas ou definidas no ponto de programa p. Portanto, ele fornece uma reposta para a seguinte questão: Quais sentenças **são afetadas** pelo *slicing criterion*?

O principal uso de *program slicing* é como ferramenta de auxílio ao *debugging* de programas. *Slicing* permite ao desenvolvedor abstrair detalhes de um programa com erro que não são relevantes para corrigi-lo. Por exemplo, se desejamos encontrar um problema na sentença $*m = a$ na Figura 2.8, não precisamos verificar as sentenças em cinza no lado direito da figura. Portanto, a tarefa se torna mais fácil e rápida. *Program slicing* também é muito útil por estar relacionado aos fundamentos da maioria das implementações de análises de fluxo de informação. Por exemplo, para o problema de vazamento de endereços, um *backward slice* pode ser computado a partir de um canal público e em seguida pode ser verificado se ele alcança alguma operação que produz informação sigilosa. Para o problema de fluxo contaminado, um *forward slice* a partir de uma entrada pública pode ser computado e verificado se ele alcança alguma operação sensível no programa.

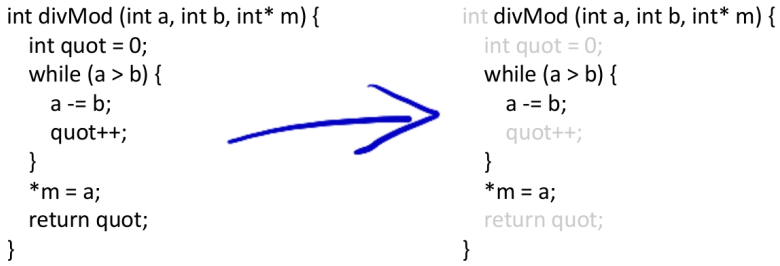


Figura 2.8. Exemplo de *Program Slicing*

2.3 O Grafo de Dependências de Ferrante

Em 1977 Jeanne Ferrante, Karl J. Ottenstein e Joe Warren publicaram o trabalho *The Program Dependence Graph and Its Use in Optimization* que se tornou uma importante referência no que se refere à detecção de dependências de controle e sua representação na forma de um grafo de dependências. Eles apresentaram nesse trabalho uma estrutura de dados chamada *program dependence graph - PDG* que torna explícita tanto as dependências de dados quanto de controle para cada operação em um programa. A motivação deles em conceberem o PDG foi o desenvolvimento de uma representação de programas que fosse útil para máquinas paralelas e vetoriais. O PDG representa um programa como um grafo no qual os vértices são sentenças e expressões com predicados (ou operadores e operandos). Uma aresta incidente a um vértice representa tanto os valores de dados os quais a operação do vértice depende como as condições de controle que a execução da operação depende. O conjunto de todas as dependências para um programa pode ser visto como uma ordem parcial induzida nas sentenças e predicados do programa que devem ser seguidas para preservar a semântica do programa original.

Ferrante *et al.* definem dependência como o resultado de dois efeitos distintos. Primeiramente, existe uma dependência entre duas sentenças se alguma variável em uma sentença pode ser definida com um valor incorreto caso se inverta a ordem de execução das sentenças. Por exemplo $(S1)$ $A = B * C$; seguida por $(S2)$ $D = A * E + 1$. Neste caso, $S2$ depende de $S1$, visto que executando a $S2$ antes de $S1$, resultaria no uso incorreto do valor de A por $S2$. Dependências deste tipo são dependências de dados.

O segundo tipo de dependência existe entre uma sentença e o predicado que imediatamente controla a execução da sentença. Por exemplo na sequencia $if\ (A)\ then\ B = C * D\ endif$ a sentença $B = C * D$ somente será executada caso o predicado A seja avaliado como verdadeiro. Dependências deste tipo são a noção de dependência de controle de Ferrante. Como será visto mais a frente, esta tese foca numa noção diferente de dependência de controle, na qual se está buscando saber quais valores

podem influenciar outros valores.

O primeiro passo da técnica de ferrante para obtenção das dependências de controle é a construção da árvore de pós-dominância[3] para um GFC incrementado. Para incrementar um GFC, basta adicionar um vértice com predicado especial *Entry* que tem uma aresta rotulada T direcionada ao vértice $START$ e outra aresta rotulada F direcionada ao vértice $STOP$. *Entry* corresponde a uma condição externa que causa a execução do programa. A Figura 2.10 mostra a árvore de pós-dominância para o GFC incrementado da Figura 2.9 (a). Na Figura 2.9 (b) é exibido o respectivo grafo de dependência de controle. Os vértices de região $R1$ até $R6$ e o vértice *Entry* foram inseridos para representar o conjunto de dependências de controle para cada vértice. As arestas de dependência de controle são representadas como linhas direcionais tracejadas. Por exemplo, os vértices 1 e 7 são dependentes de controle do vértice *Entry*.

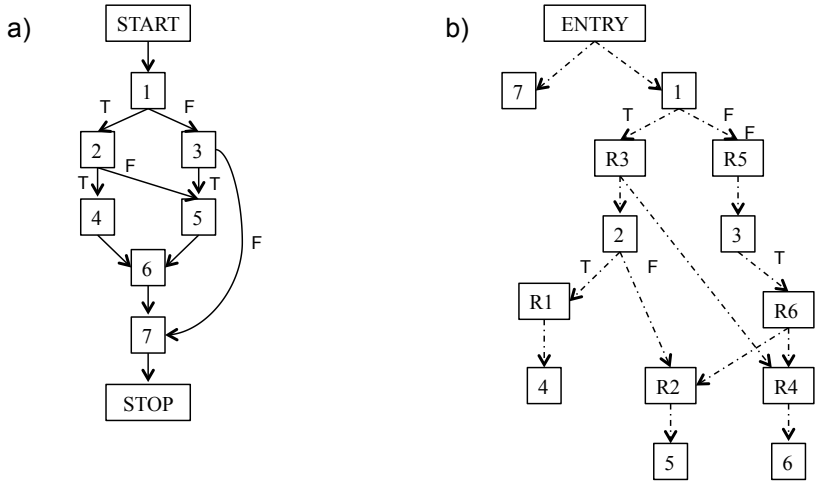


Figura 2.9. Grafo de fluxo de controle e seu subgrafo de dependência de controle. Figura reproduzida do trabalho de Ferrante et al. [1987].

Dada a árvore de pós-dominância, pode-se determinar a noção de Ferrante para dependência de controle através da análise de certas arestas do GFC anotando os vértices nos caminhos correspondentes na árvore. Seja S todas as arestas (A, B) no GFC tal que B não é um ancestral de A na árvore de pós-dominância (isto é, B não pós-domina A). Cada uma dessas arestas tem um rótulo associado T ou F. No exemplo da Figura 2.9 (a), $S = (Entry, Start), (1, 2), (1, 3), (2, 4), (2, 5), (3, 5)$. O algoritmo de determinação das dependências de controle procede examinando cada aresta (A, B) em S. Seja L o mais próximo ancestral comum de A e B na árvore de pós-dominância.

[3]Como será mostrado no Capítulo 3, a técnica desta tese utiliza ambas as árvore de dominância e de pós-dominância para obtenção de uma noção diferente de Ferrante *et al.* para dependência de controle, sem custo adicional de complexidade de tempo e espaço, evitando o pior caso da técnica de Ferrante *et al.*

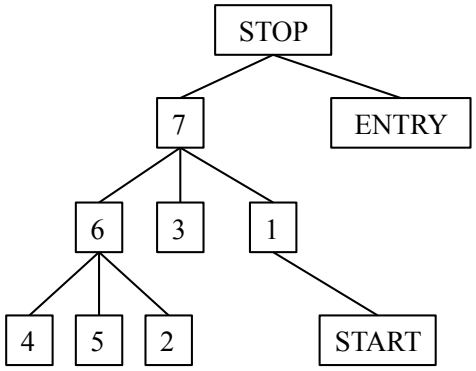


Figura 2.10. Árvore de pós-dominância do grafo de fluxo de controle incrementado. Figura reproduzida do trabalho de Ferrante et al. [1987].

Aresta de S examinada	Vértices marcados	Dependente de controle de	Rótulo
(ENTRY, START)	START, 1, 7	ENTRY	T
(1,2)	2,6	1	T
(1,3)	3	1	F
(2,4)	4	2	T
(2,5)	5	2	F
(3,5)	5,6	3	T

Figura 2.11. Dependências de controle determinadas para cada aresta em S. Figura reproduzida do trabalho de Ferrante et al. [1987].

Por construção, L não pode ser B. Advoga-se que L é A ou L é um ancestral de A na árvore de pós-dominância.[4]

Considera-se agora esses dois casos para L.

1. Caso 1. L = ancestral de A. Todos os vértices na árvore de pós-dominância no caminho de L para B, incluindo B mas não L, devem ser considerados como dependentes de controle de A.

2. Caso 2. $L = A$. Todos os vértices na árvore de pós-dominância no caminho de A para B, incluindo A e B, devem ser considerados como dependentes de controle de A. (Este caso captura dependências de laços de repetição).

Deve ficar claro que, dado (A, B), o efeito desejado será alcançado pela travessia inversa a partir de B na árvore de pós-dominância até alcançar o ancestral de A (se ele existir), tornando todos os vértices visitados antes do ancestral de A como dependentes de controle de A. Essa travessia única manipula ambos os casos acima visto que, para o Caso 1, A não estará no caminho para L e assim não será marcado. Após todas

[4] A prova dessa afirmação pode ser encontrada em Ferrante et al. [1987].

arestas em S serem examinadas, todas as dependências de controle segundo a noção de Ferrante terão sido encontradas. A tabela da Figura 2.11 mostra as dependências de controle que foram determinadas pelo exame de cada uma das arestas no conjunto S para o grafo da Figura 2.9. Visto que não há laços de repetição no GFC, todas as Dependências foram determinadas de acordo com o Caso 1 acima.

A próximo passo para a construção do PDG é a inserção dos vértices de região para representar o conjunto de condições de controle de um vértice e agrupar todos vértices com o mesmo conjunto de condições de controle juntos. Isso é realizado de forma que se um conjunto de dependências de controle contém outro, o conjunto contido será fatorado para a representação de dependência de controle do conjunto que o contém. Vértices de região também são inseridos de forma que vértices de predicado terão somente dois sucessores, como no grafo de fluxo de controle, hierarquicamente organizando as dependências de controle. Vértices de região podem ser vistos como uma forma limitada de eliminação de subexpressão comum para dependências de controle e uma forma extendida de construção de blocos básicos. Dois vértices tem os mesmos predecessores de dependência de controle se cada um tem arestas de dependência de controle a partir dos mesmos vértices e as respectivas arestas tem os mesmos rótulos: "T", "F", ou nenhum.

A noção de grafo de dependência de Ferrante *et al.* [Ferrante et al., 1987], foi aperfeiçoada em diferentes direções. Esses aperfeiçoamentos aceleraram a construção do grafo ou o fizeram mais eficiente do ponto de vista de espaço de armazenamento. Para reduzir o tempo de construção, Johnson e Pingali [Johnson et al., 1994] introduziram o *Program Structure Tree* (PST), que representa regiões *hammock* do programa, e, consequentemente, fornece uma forma rápida de determinar dependências de controle entre instruções. Em termos de espaço, Cytron e Ferrante [Cytron et al., 1990] introduziram o *Control Dependence Graph*, que rastreia dependências de controle de forma mais compacta. Em direção similar, Pingali e Bilardi [Pingali & Bilardi, 1997] projetaram o *Augmented Post-Dominator Tree*, uma estrutura que permite representar dependências implícitas eficientemente no que tange o espaço. Porém, apesar de todas essas vantagens, a noção de dependência de controle de Ferrante *et al.* ainda pode produzir um número quadrático de arestas denotando fluxos implícitos de informação. Assim, se necessário listar todas elas, como na construção do grafo de dependência, é possível construir piores casos onde o comportamento quadrático acontece.

Nenhuma dessas abordagem utiliza as propriedades do formato SSA para obter mais simplicidade ou eficiência. Essa é uma das grandes contribuições desta tese. Embora simplicidade seja algo relativo, advoga-se que esta tese introduz uma abordagem mais simples que os demais trabalhos anteriores apoiando em duas observações. Pri-

meiro, o protótipo executável exibido no Capítulo 3 ocupa apenas 27 linhas de código SML. Embora sua versão para o compilador LLVM seja maior – aproximadamente 600 linhas de código C++– esse tamanho extra é necessário apenas para manipular os diferentes tipos de instruções na representação intermediária de LLVM. Segundo, muitos dos trabalhos anteriores clamam como uma contribuição o fato de não requererem a árvore de dominância do programa analisado. Porém, essa estrutura está disponível sem custo adicional nos compiladores modernos, tais como *GCC*, *LLVM*, *Jikes* e *ICC*, porque ela já é usada na construção do formato SSA. Evitando usá-la, os algoritmos anteriores renunciam essa informação extremamente útil.

A forma GSA de Ottenstein *et al.* [Ottenstein et al., 1990] é o trabalho que mais se aproxima ao apresentado nesta tese. A forma GSA conecta funções ϕ a predicados que as controlam. Esta tese não conecta predicados a funções ϕ; porém, pode conectar predicados com parâmetros de funções ϕ sempre que estes parâmetros são definidos dentro da região de influência do respectivo predicado. Além disso, esta tese está solucionando um problema diferente com um algoritmo bastante distinto. Ottenstein *et al.* espera construir uma versão interpretável da forma SSA. Aqui, procura-se rastrear dependências de controle. GSA não é a forma que está sendo usada neste trabalho, porque ele não foi concebido como uma forma de implementar o sistema de tipos seguros.

2.4 Considerações Finais

Este capítulo apresentou na Seção 2.2 três tipos de problemas de segurança relacionados ao fluxo de informação que podem ser detectados com o auxílio do algoritmos e das ideias que serão discutidas no próximo capítulo. A Subseção 2.2.1 versou sobre o vazamento de endereços como um exemplo do vazamento de informação sigilosa. A Subseção 2.2.2 apresentou o problema de estouro de arranjo. Já a Subseção 2.2.3 descreveu o problema de vazamento de informação por canais laterais. Foi apresentado na Subseção 2.2.4 a técnica de *Program Slicing* que permite identificar partes de um código fonte que podem influenciar em algum ponto de interesse do mesmo código. Essa técnica também pode ser implementada com o auxílio do arcabouço teórico proposto e defendido nesta Tese. Finalmente este capítulo termina na Seção 2.3 apresentando o grafo de Ferrante como a linha de base de comparação com o algoritmo de rastreamento de fluxo implícito proposto neste trabalho e que será apresentado no Capítulo 3

Capítulo 3

Dependências Implícitas e o Sistema de Hunt e Sands

Este capítulo descreve em detalhes a dificuldade em se rastrearem os fluxos implícitos. Ele descreve também a solução que esta tese propõe, e também realiza uma comparação formal e empírica com a abordagem de Ferrante *et al.* mostrando que a nova solução avança o estado da arte nessa área. Este capítulo também descreve uma implementação do sistema de tipos seguros de Hunt e Sands de forma esparsa e portanto de execução mais rápida e com menor consumo de memória que a abordagem descrita em seu trabalho original [Hunt & Sands, 2006]. O sistema de tipos seguros de Hunt e Sands é um importante trabalho na área de análise estática de programas que visa detectar fluxos contaminados ou vazamentos de informação via fluxo de dados e controle.

3.1 Visão geral do problema

Análises de fluxo de informação tentam inferir propriedades de dados que um programa manipula. Típicas implementações de sistemas de fluxo associam entidades de programa (variáveis, pontos de programa, desvios, instruções) com pontos em um semi-reticulado que representa níveis de segurança. Sem perda de generalidade, esta tese assume que esse semi- reticulado tem três elementos: $\{H, L, \bot\}$, tal que $H > L > \bot$. O operador (comutativo) *meet* é definido como $H \wedge t = H, L \wedge \bot = L$, onde $t \in \{H, L, \bot\}$. Esse arcabouço teórico é baseado no trabalho de Denning & Denning [1977]. Sua análise de fluxo de informação original é insensível ao fluxo. Em outras palavras, o resultado de analisar a sequencia de comandos $C_1; C_2$ é o mesmo resultado produzido pela análise de $C_2; C_1$. Em termos de implementação, insensitividade ao fluxo implica que o tipo de uma variável é determinado do tipo mais restritivo que ele receber em qualquer lugar

do texto do programa.

Uma análise de informação insensível ao fluxo é imprecisa, porque ela não considera a ordem na qual as variáveis são definidas e usadas dentro do programa. Para resolver essa questão, em 2006 Hunt e Sands introduziram um sistema *sensível ao fluxo* de tipos seguros [Hunt & Sands, 2006]. Nesse sistema, o tipo de uma variável depende do ponto do programa onde a informação é consultada. A Figura 3.1 ilustra essa ideia. Deste ponto em diante, este texto usará $\bullet$ para denotar uma fonte de informação de baixo nível de segurança, e $\circ$ para denotar uma fonte de informação de alto nível de segurança. A variável z é a única inicialmente definida com o tipo H neste programa de exemplo. Na abordagem de Hunt e Sands, existe um ambiente de tipagem Γ, exibido na Figura 3.1 (b), combinando variáveis a tipos em cada *ponto de programa*. Um ponto de programa, neste caso, é qualquer região entre duas instruções – uma definição que inclui as arestas do grafo de fluxo de controle do programa.

Inicialmente no exemplo da Figura 3.1 (a) no ponto entre os rótulos: ℓ_1 e ℓ_2 a variável w é definida com o tipo L de baixo nível de segurança. O mesmo acontece nos pontos ℓ_2 - ℓ_3 e ℓ_3 - ℓ_4 atribuindo o tipo L para as variáveis x e y respectivamente. No ponto entre os rótulos ℓ_4 - ℓ_5 a variável z recebe o tipo H e no ponto entre os rótulos ℓ_5 - ℓ_6 o predicado p é definido com o mesmo tipo da variável x, isto é, tipo L.

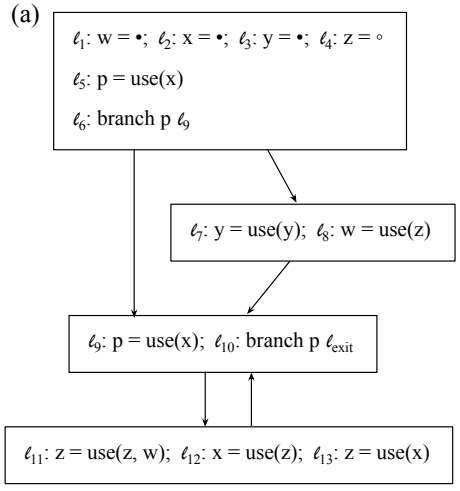
Nos pontos entre os rótulos ℓ_6 - ℓ_7 e ℓ_7 - ℓ_8 nada é alterado no ambiente de tipagem Γ, pois a variável y é redefinida com uma informação de tipo L. Porém no ponto entre os rótulos ℓ_8 - ℓ_9 a variável w recebe o tipo H pois ela é definida com o valor da variável z, uma informação de alto nível de segurança H. No outro caminho do desvio em ℓ_6, mais precisamente no ponto entre os rótulos ℓ_6 - ℓ_9 a variável p permanece com o tipo L pois é redefinida usando a variável x que é do tipo L. Caso o caminho tomado a partir do desvio seja esse, as variáveis w, x e y permaneceriam com o tipo L.

No ponto entre os rótulos ℓ_9 - ℓ_{10}, algumas constatações são importantes: primeiramente após o desvio no rótulo ℓ_6 existem dois caminhos possíveis a serem tomados, e como a análise é estática, uma abordagem conservadora deverá ser tomada considerando que ambos caminhos são possíveis de serem seguidos. Portanto, se em um caminho, uma variável tem tipo L e no outro caminho tipo H, deve-se assumir o tipo H. Logo, visto que a variável w assume tipo H no caminho ℓ_6 - ℓ_7 e tipo L no caminho ℓ_6 - ℓ_9, assume-se o tipo H para w em ℓ_9 - ℓ_{10}. A segunda constatação é que esse bloco básico ℓ_9 - ℓ_{10} é novamente acessado a partir do bloco ℓ_{11} - ℓ_{13} e como será visto no próximo parágrafo, do segundo acesso em diante a variável x estará "contaminada" com o tipo H e portanto devera ser considerado esse tipo para x.

Dando continuidade ao exemplo, no ponto entre os rótulos ℓ_{10} - ℓ_{11} nada ocorre no ambiente de tipagem. Passando então para o ponto entre os rótulos ℓ_{11} - ℓ_{12} observa-se

que a variável z é redefinida com a informação de z e w. Neste ponto w e z são do tipo H o que se mantém para z. No ponto entre os rótulos ℓ_{12} - ℓ_{13} é onde a variável x muda seu tipo original de L para H pois receber z que é do tipo H. Isso como dito no parágrafo anterior, fará com que o predicado p mude também seu tipo de L para H quando esse for redefinido com o valor x em ℓ_9 acessado pelo caminho ℓ_{13} - ℓ_9. Finalmente no ponto entre os rótulos ℓ_{13} - ℓ_9 nada de especial ocorre no ambiente de tipagem pois z já tem o tipo H e recebe uma informação também do tipo H.

Se visto como uma instancia de um arcabouço de fluxo de dados, a abordagem de Hunt e Sands seria classificada como uma análise *densa* [Choi et al., 1991; Oh et al., 2012; Tavares et al., 2014]. Uma análise densa associa pares formados por nomes de variáveis e pontos de programa com informação. Na Figura 3.1 é mostrado o resultado da inferência de tipos Hunt-Sands para o exemplo: esse resultado usa $O(V * N)$ espaço onde V é o número de variáveis do programa e N é o número de instruções. Visto que cada instrução em geral define uma nova variável tem-se que a complexidade tende a $O(V^2)$. Em contrapartida, uma análise *esparsa* combina informação diretamente a nomes de variáveis [Rimsa et al., 2014]. Uma análise de fluxo de dados deve ser aplicada em uma representação de programa que apresente a *Propriedade de Informação Única*, para ser implementada esparsamente [Tavares et al., 2014]. Essa propriedade existe se a informação que a análise associa com a variável v é invariante em cada ponto de programa onde v está viva. Hunt e Sands mostraram a existência de uma representação



(b)

	p	w	x	y	z
1-2:	[⊥,	L,	⊥,	⊥,	⊥]
2-3:	[⊥,	L,	L,	⊥,	⊥]
3-4:	[⊥,	L,	L,	L,	⊥]
4-5:	[⊥,	L,	L,	L,	H]
5-6:	[L,	L,	L,	L,	H]
6-7:	[L,	L,	L,	L,	H]
7-8:	[L,	L,	L,	L,	H]
8-9:	[L,	H,	L,	L,	H]
6-9:	[L,	L,	L,	L,	H]
9-10:	[H,	H,	H,	L,	H]
10-11:	[H,	H,	H,	L,	H]
11-12:	[H,	H,	H,	L,	H]
12-13:	[H,	H,	H,	L,	H]
13-9:	[H,	H,	H,	L,	H]

Figura 3.1. (a) Programa retirado de Hunt e Sands [Hunt & Sands, 2006, Fig.3] reescrito usando uma notação de baixo nível. (b) Tipo de cada variável em cada ponto de programa, como inferido usando o sistema de tipos insensível ao fluxo de Hunte Sands.

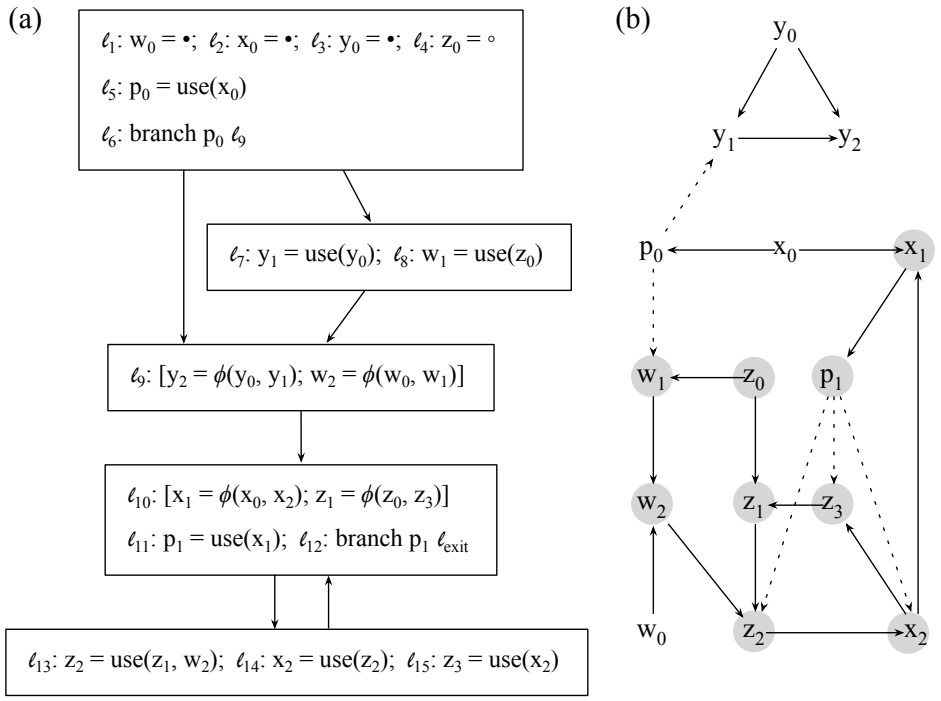


Figura 3.2. (a) Grafo de fluxo de controle da Figura 3.1 no formato SSA. (b) Grafo SSA construído com a técnica introduzida nesta tese. Vértices marcados em cinza deveriam ter o tipo H no sistema de Hunt e Sands [Hunt & Sands, 2006].

que os deixa aplicar seu sistema de tipos esparsamente [Hunt & Sands, 2006, Sec.7]. Porém, eles não apresentam uma forma de implementar tal análise; ao contrário, eles mostram que, dado o resultado de seu sistema de tipos *denso*, é possível derivar uma representação de programa que tem a propriedade de informação única. Na Seção 3.3.2, é mostrado que o sistema de tipos deles pode ser implementado esparsamente em programas no formato *Static Single Assigment (SSA)*. A Figura 3.2 (a) contém uma versão na forma SSA do exemplo em questão, e a parte (b) da mesma figura mostra um resumo do resultado obtido: cada variável tem um tipo único durante toda sua existência. Portanto, apesar do programa no formato em SSA ser maior que a sua versão original, a quantidade de informação criada durante a análise é muito inferior quando comparada a uma análise densa.

Tavares et al. [2014] proveram uma abordagem geral para produzir representações de programa que possuem a propriedade de informação única para análises de fluxo de dados. Sua técnica enquadra em análises nas quais toda informação que contribui para o tipo de uma variável está disponível em certo ponto do programa. Porém a abordagem de Tavares *et al* não pode ser diretamente aplicada para implementar o sistema de tipos seguros de Hunt e Sands. O problema neste caso é devido aos *fluxos*

implícitos. Diz-se que informação flui implicitamente de um predicado p para uma variável v se p é usado em um desvio que *controla* a definição de v [Russo & Sabelfeld, 2010]. Aqui, toma-se a definição de dependência de controle de Ferrante et al. [1987]:

Definição 3.1.1 (Ferrante'87) *Um vértice y é dependente de controle no grafo de fluxo de controle de um vértice x se: (i) existe um caminho, no grafo de fluxo de controle, de x para y, de forma que qualquer vértice neste caminho é pós-dominado*[1] *por y. (ii) x não é pós-dominado por y. Neste caso, diz-se que x* controla *a execução de y.*

Como um exemplo de dependência de controle, o desvio em ℓ_6 controla as atribuições nos rótulos ℓ_7 e ℓ_8 na Figura 3.1. Na abordagem Hunt-Sands, o tipo da variável w em ℓ_8 é determinado como um *meet* do tipo que z e p têm naquele ponto. Esse fato é não desejável: porque a variável p não está sintaticamente presente em ℓ_8, portanto não é possível aplicar a técnica de Tavares *et al* para gerar a representação do programa que tornaria esparsa a análise Hunt-Sands. Além desse problema, usualmente, quando aplicado em código de baixo nível, implementações de análises de fluxo de informação apoiam-se no algoritmo de Ferrante [Ferrante et al., 1987], e seus diversos aperfeiçoamentos [Cytron et al., 1990; Johnson & Pingali, 1993; Johnson et al., 1994; Pingali & Bilardi, 1997] para encontrar dependências de controle. Advoga-se aqui que essa noção de dependência de controle, embora correta (veja, por exemplo, Hammer & Snelting [2009]), fornece mais informação que o necessário para implementar o sistema de tipos seguros sensível ao fluxo de Hunt e Sands.

O programa na Figura 3.3 ilustra esse último ponto. Esse programa contém três laços *do-while* aninhados. A Figura 3.4 mostra dois grafos diferentes que representam as possíveis formas da informação fluir entre as variáveis no código. Esses grafos contêm um vértice para cada variável do programa. Uma aresta sólida de u para v representa uma *dependência de dados*. Uma variável v é dependente de dados de outra variável u se v é definida por uma instrução que usa u. Dependências de dados criam fluxos explícitos de informação. A linhas pontilhas representam fluxos implícitos. Na Figura 3.4 (a), usa-se a noção de dependência de controle de Ferrante para identificar esses fluxos implícitos, o que acontece em vários trabalhos anteriores [Hammer et al., 2006; Hammer & Snelting, 2009; Horwitz et al., 1988; Ranganath et al., 2007; Reps & ang, 1988; Snelting, 1996; Taghdiri et al., 2011; Wasserrab et al., 2009]. A Figura 3.4 (b) mostra os fluxos implícitos que foram encontrados usando o algoritmo que será introduzido nesta tese. Para diferenciar essas linhas das arestas de dependência de controle de Ferrante, a partir deste ponto elas serão chamadas de *dependências implícitas*.

[1] Para um definição formal de dominância e pós-dominância, veja a Seção 3.2.

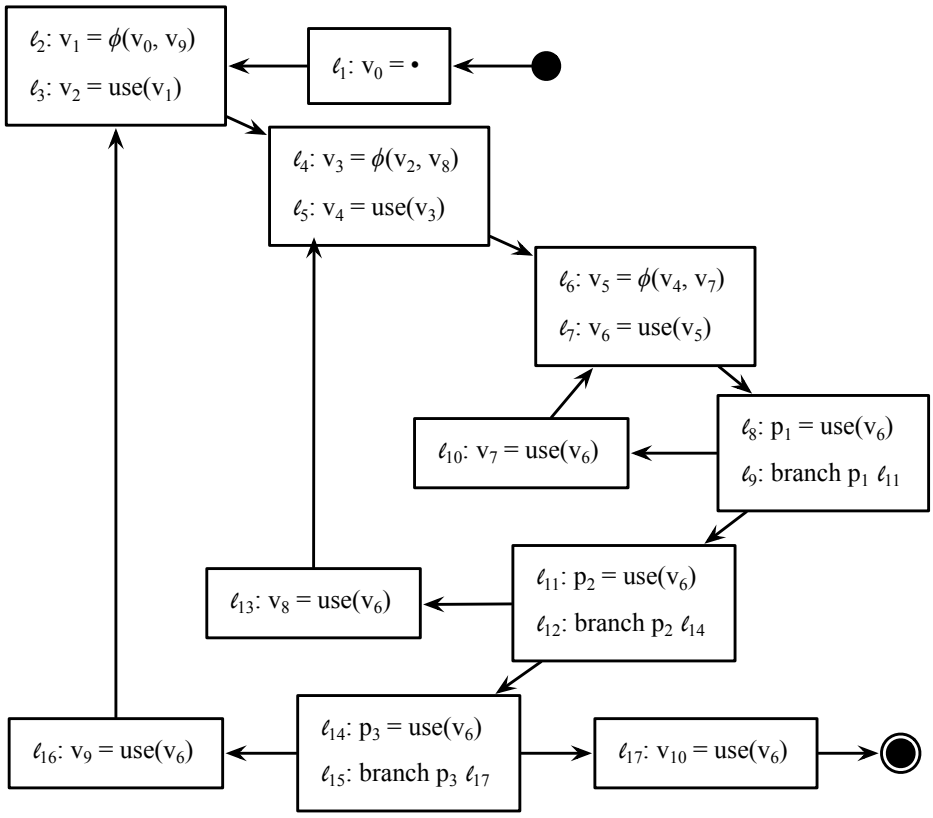


Figura 3.3. Grafo de fluxo de controle de três blocos *do-while* aninhados.

O conceito de dependência implícita aqui apresentado é fundamentalmente diferente daquele conceito de dependência de controle de Ferrante *et al.*, porque aqui se está interessado em rastrear quais *valores* – não quais instruções – um predicado controla. Como será mostrado na Seção 3.2, um predicado controla o valor das funções ϕ usadas em um programa na forma SSA. Essas instruções especiais permitem unificar fluxos implícitos. Por exemplo, o rótulo ℓ_7 na Figura 3.3 é dependente de controle – no estilo Ferrante – do rótulo ℓ_9. Porém, o predicado p_1, que determina a saída do desvio em ℓ_9, determina o valor de v_5, uma variável definida pela função ϕ em ℓ_6. O algoritmo desta tese será capaz de explorar essa corrente de dependências transitivas: $p_1 \rightarrow v_7 \rightarrow v_5 \rightarrow v_6$; assim, evita-se a necessidade de manter arestas tais como $p_1 \rightarrow v_6$.

O exemplo da Figura 3.3 é um caso extremo: sequências aninhadas de laços *do-while* são bem conhecidas por produzirem um número quadrático de arestas de dependência de controle. O mesmo é verdade para os chamados "grafos escada" [Cytron et al., 1990]. Este novo algoritmo que será introduzido nesta tese não sofre desse pior caso; além disso, para programas estruturados, suas dependências implícitas e as dependências de controle de Ferrante tendem a possuir número similar de arestas,

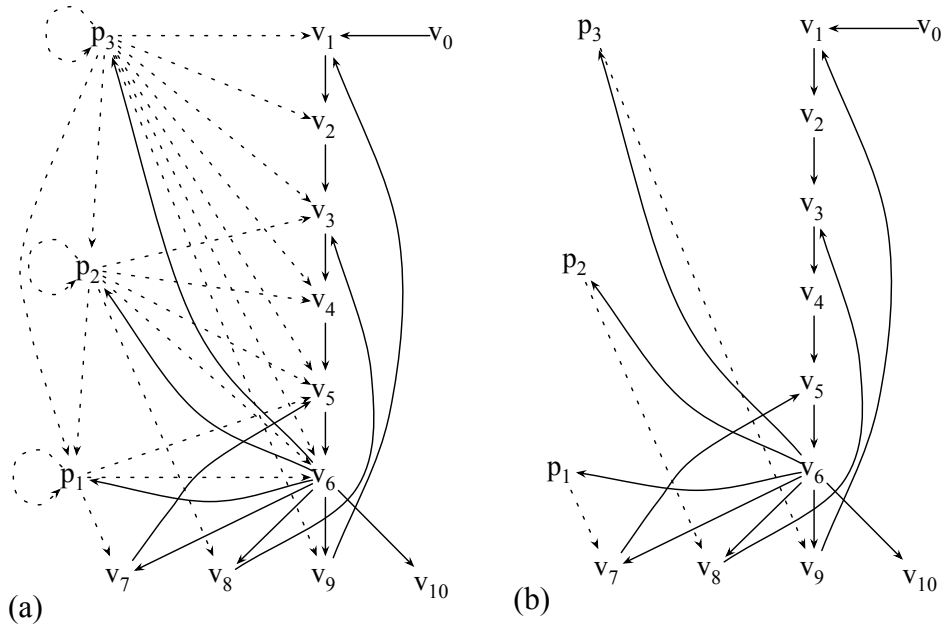


Figura 3.4. (a) Dependências de fluxo de controle como definidas por Ferrante *et al.*. (b) Dependências de fluxo de controle como definidas nesta tese.

embora o novo algoritmo as encontre de maneira mais rápida, como será demonstrado no próximo Capítulo.

3.2 Fluxo de Informação no Grafo SSA

Uma Linguagem Central.

O algoritmo desta tese será definido sobre uma linguagem central cuja sintaxe pode ser visualizada na Figura 3.5. A semântica dessa linguagem será definida na Seção 3.3. Um programa é uma sequência de blocos básicos; blocos básico são sequências de instruções e que finalizam com um desvio condicional ou com um desvio incondicional. Cada instrução é associada à um rótulo único. Serão reconhecidas sete categorias de instruções. Duas delas (`br` e `jmp`) mudam o fluxo de controle do programa. As outras (funções ϕ e as atribuições) transferem informação entre variáveis. Por simplicidade, se a variável definida por `sink` não é usada, ela não será mostrada nos exemplos.

3.2.1 Construção de Arestas de Dependência Implícita

Aqui representam-se dependências de fluxo de controle em um programa via o chamado *Grafo SSA* [Rastello, 2015]. A definição original dessa estrutura de dados é dirigida

Programas (*Prog*)	::=	$B_1, B_2, \ldots, B_n$
Blocos básicos (B)	::=	$P*, I*, T$
Rótulos (L)	::=	$\{\ell_1, \ell_2, \ldots\}$
Variáveis (V)	::=	$\{v_1, v_2, \ldots\}$
Terminadores (T)	::=	
– Desvio condicional	|	$\ell : \texttt{br}(v, \ell_x)$
– Desvio incondicional	|	$\ell : \texttt{jmp}(\ell_x)$
Funções ϕ (P)	|	$\ell : \overline{v} = \phi(\overline{v_1}, \ldots, \overline{v_n})$
Atribuições (I)	::=	
– Dado de baixa segurança	|	$\ell : v = \bullet$
– Dado de alta segurança	|	$\ell : v = \circ$
– Operação sensível	|	$\ell : v = \texttt{sink}(v)$
– Operação de leitura	|	$\ell : v = \texttt{use}(v_1, \ldots, v_n)$

Figura 3.5. A sintaxe da linguagem central.

à sintaxe: ele contém um vértice para cada variável do texto do programa, e uma aresta de u para v se v aparece no lado esquerdo de uma instrução que contém u no seu lado direito, evidenciando uma dependência explícita. Os grafos na Figura 3.4 são grafos SSA no sentido original, se considerarmos apenas arestas sólidas. O objetivo dessa Seção é incrementar esse grafo com arestas de dependência implícita. Seguindo o exemplo da Figura 3.4, deseja-se produzir o grafo na parte (b). A Figura 3.6 mostra o algoritmo que foi desenvolvido nesta tese para adicionar arestas de dependência implícita no grafo SSA. Esse algoritmo, somado à noção de dependência explícita, permite apresentar a definição desta tese para o grafo SSA:

Definição 3.2.1 (Grafo SSA) *Dado um programa* Prog *escrito na linguagem descrita na Figura 3.5, define-se seu Grafo SSA* $G = (V, E)$ *como:* V *contém 1 vértice para cada variável* v *no texto de* Prog*.* E *contém 1 aresta* $v \to u$ *se, e somente se,* u *depende, explicitamente ou implicitamente, de* v*. Diz-se ainda que* a *depende de* b *explicitamente se* b *aparece no lado direito da única instrução que define* a*. Dize-se também que* a *depende de* b *implicitamente se o Algoritmo da Figura 3.6 cria uma aresta de* a *para* b*.*

O algoritmo na Figura 3.6 está escrito em *Standard ML (SML)*. Colchetes vazios, isto é, [], denotam uma lista vazia. Dois pontos duplo, ::, são construtores de lista. Por exemplo, $(h :: t)$ é uma lista com cabeça em h, e cauda em t. O @ é uma concatenação de lista. O símbolo _ representa qualquer padrão. Uma construção tal como "map (fn a => (lb, a) defList)" converte cada elemento "a $\in$ defList" em um par "(lb, a)". Usa-se um opcode único ASG para representar os quatro tipos de atribuições diferentes da linguagem. A Figura 3.6 contém o algoritmo por completo,

```
1  datatype Instruction =
2    ASG of string * string list
3    PHI of string list * string list list
4
5  datatype DomTree =
6    BR of Instruction list * string * DomTree list * DomTree
7  | JMP of Instruction list * DomTree list
8
9  fun link [] _ = []
10   | link _ "" = []
11   | link ((ASG (a, _)) :: insts) lb = (lb, a) :: link insts lb
12   | link ((PHI (defList, _)) :: insts) lb =
13                   (map (fn a => (lb, a)) defList) @ link insts lb
14
15 fun visit (JMP (instructions, [])) pred =
16           link instructions pred
17   | visit (JMP (instructions, [child])) pred =
18           link instructions pred @ visit child pred
19   | visit (BR (instructions, new_pred, children, ipdom)) pred =
20       let
21         fun visit_every [] = []
22           | visit_every (child::rest) =
23             if child = ipdom
24             then visit child pred @ visit_every rest
24             else visit child new_pred @ visit_every rest
25       in
26         link instructions pred @ visit_every children
27       end
```

Figura 3.6. A versão SML/NJ do algoritmo básico que insere arestas de dependência implícita no grafo SSA.

isto é, esse programa pode ser testado em um interpretador SML. Nele, blocos básicos são representados como tuplas. Se um bloco B termina com um desvio condicional, então ele tem o formato "BR (*Instructions*, *Predicate*, *Children*, pdom_B)". Se B termina com um desvio incondicional, então ele tem o formato "JMP (*Instructions, Children*)". Em ambos os casos, sejam *Instructions* as operações no corpo de B e *Children* os blocos que B domina. Se B é do tipo BR, então *Predicate* é a variável usada como predicado nesse desvio. A Figura 3.7 mostra a estrutura de dados que representa o programa exibido na Figura 3.2.

A função `visit`, exibida na Figura 3.6 avança sobre a árvore de dominância do programa, rastreando o último predicado visualizado durante essa travessia. Por predicado, entende-se como a variável `pred` usada em um desvio condicional. Em um bloco básico B, `visit` invoca a função `link` para criar uma aresta a partir de `pred` para toda variável definida em B. Essa invocação acontece nas linhas 16, 18 e 26 da Figura 3.6. Após inspecionar o bloco B que termina em uma instrução $\texttt{br}(\texttt{new\_pred}, \ell)$, `visit` é invocado recursivamente sobre cada filho de B na árvore de dominância com `new_pred` sendo o predicado atual, a menos que esse filho é o pós-dominador imediato de B. Se

```
1 val t_exit = JMP ([], [])
2 val t13_15 = JMP (
3    [ASG ("z2", ["z1", "w2"]), ASG ("x2", ["z2"]), ASG ("z3", ["x2"])], [])
4 val t10_12 = BR ([PHI (["x1", "z1"], [["x0", "x2"], ["z0", "z3"]]),
5       ASG ("p1", ["x1"])], "p1", [t13_15, t_exit], t_exit);
6 val t9_9 = JMP ([PHI (["y2", "w2"], [["y0", "y1"], ["w0", "w1"]])], [t10_12])
7 val t7_8 = JMP ([ASG ("y1", ["y0"]), ASG ("w1", ["z0"])], [])
8 val t1_6 = BR ([ASG ("w0", []), ASG ("x0", []), ASG ("y0", []),
9       ASG ("z0", []), ASG("p0", ["x0"])], "p0", [t7_8, t9_9], t9_9);
```

Figura 3.7. A estrutura de dados que representa o programa exibido na Figura 3.2. A expressão (`visit t1_6 ''`) invoca o algoritmo da Figura 3.6 sobre essa estrutura de dados.

esse último caso acontece, então `new_pred` não é usado para substituir `pred` como predicado atual sendo rastreado. A aplicação de `visit` na estrutura exibida na Figura 3.7 gera as cinco arestas pontilhadas que estão evidenciadas na Figura 3.2 (b).

3.2.2 Exemplo

Como forma de melhor compreender o algoritmo de inserção de arestas de controle no grafo SSA, apresenta-se nesta Subseção um exemplo de funcionamento do algoritmo. A Figura 3.8 (b) exibe o grafo de fluxo de controle que será utilizado como exemplo de aplicação do algoritmo. O primeiro passo é a construção da respectiva árvore de dominância que pode ser visualizada na Figura 3.8 (a). Normalmente a árvore de dominância já é construída nas fases preliminares de compilação pela maioria dos compiladores modernos tais como *GCC*, *Jikes*, *ICC* e *LLVM* visto ser uma estrutura de dados usada na construção do formato SSA. Vale lembrar que a construção da árvore de dominância tem custo linear sobre o tamanho do código estático do programa.

O algoritmo atravessa a árvore de dominância, "memorizando" os predicados que são encontrados no caminho. Por exemplo, após caminhar no caminho destacado na árvore da Figura 3.8 (a), serão memorizados os predicados b_0, b_1 e b_2. Inicialmente o bloco básico $\ell_0 - \ell_2$ é visitado pois ele é a raiz da árvore. Nesse instante a pilha de predicados visitados está vazia. Visto que esse bloco básico termina com um desvio incondicional, não há necessidade de empilhar qualquer predicado.

Em seguida o bloco básico $\ell_3 - \ell_7$ é visitado com a pilha vazia. Visto que a pilha está vazia, não há necessidade de criar qualquer aresta de controle conectando as variáveis definidas nesse bloco à qualquer predicado. Porém, esse bloco termina com um desvio condicional `br` b_0 ℓ_8, ℓ_{23}. Nesse caso o algoritmo empilha o predicado b_0. Assim será possível conectar esse predicado as variáveis definidas no bloco básico filho de $\ell_3 - \ell_7$, criando as respectivas arestas de controle no grafo SSA desse exemplo.

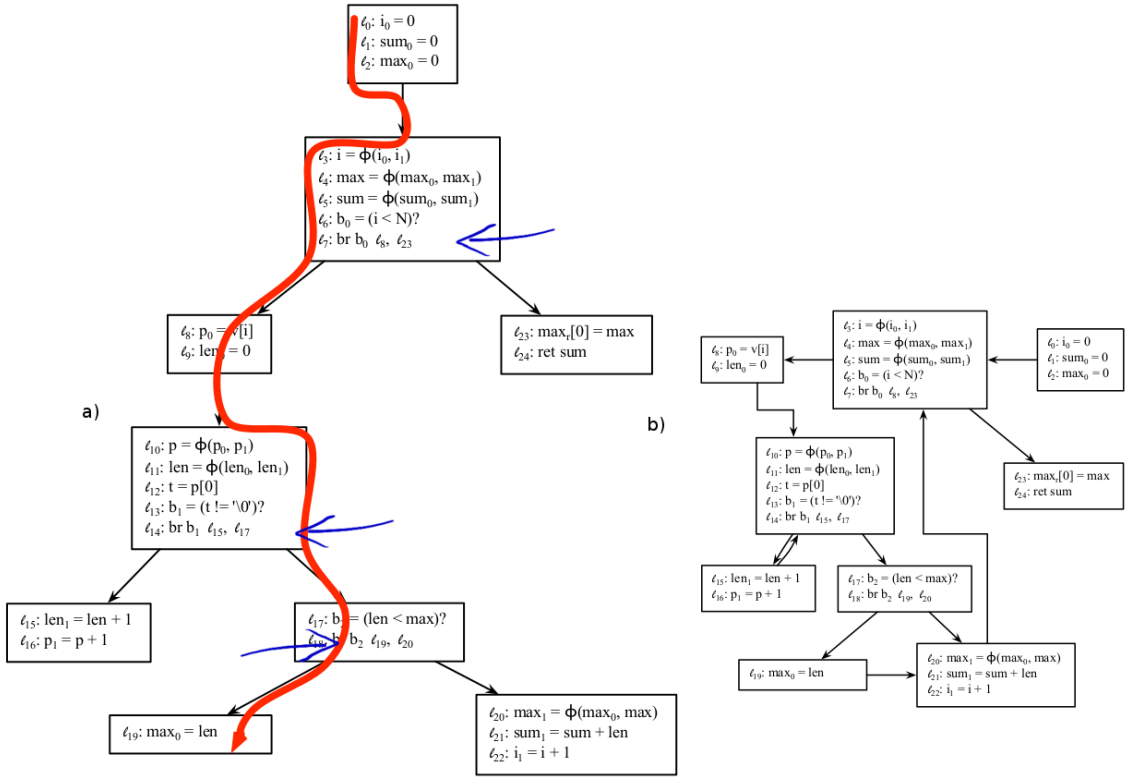


Figura 3.8. (a) Árvore de dominância e (b) respectivo grafo de fluxo de controle utilizados como exemplo de aplicação do algoritmo de inserção de arestas de controle no grafo SSA.

Agora, o algoritmo visita os dois filhos do bloco $\ell_3 - \ell_7$. A ordem de visita não é importante e ambos são visitados com a pilha contendo o predicado b_0 no topo. Portanto, b_0 é conectado às variáveis p_0 e len_0 do bloco $\ell_8 - \ell_9$. Porém, b_0 não será conectado a qualquer variável do bloco básico $\ell_{23} - \ell_{24}$ visto que esse bloco é um pós-dominador do bloco básico $\ell_3 - \ell_7$ onde b_0 foi definido. Além disso, o algoritmo desempilha o topo da pilha recebida pelo bloco $\ell_{23} - \ell_{24}$.

Em seguida o algoritmo visita o bloco básico $\ell_{10} - \ell_{14}$ com a pilha contendo o predicado b_0 no topo. Duas ações serão realizadas. Primeiro, deve-se conectar as variáveis definidas nesse bloco ao predicado do topo da pilha, isto é, b_0. Segundo, deve-se atualizar o predicado atual, empilhando o predicado b_1, antes de visitar os filhos do bloco básico $\ell_{10} - \ell_{14}$.

O algoritmo visita agora o bloco básico $\ell_{15} - \ell_{16}$ com a pilha contendo b_1 no topo e b_0 abaixo dele. Assim, o algoritmo conecta as variáveis definidas nesse bloco com o predicado atual, isto é o predicado b_1, o qual é o topo da pilha nesse momento. Visto que o bloco $\ell_{15} - \ell_{16}$ não tem filhos, não há o que se fazer. Agora o algoritmo visita

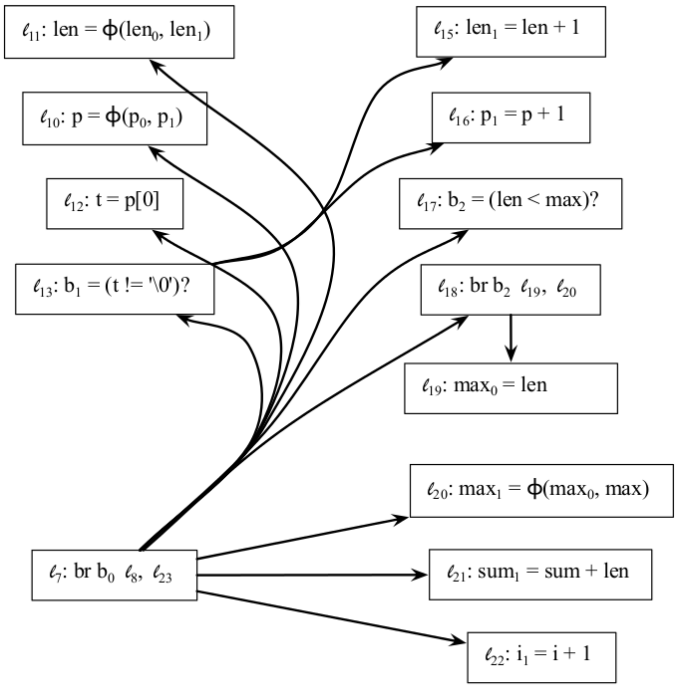


Figura 3.9. Arestas de controle detectadas a partir da execução do algoritmo proposto nesta tese sobre o exemplo da Figura 3.8.

o outro filho de $\ell_{10} - \ell_{14}$, isto é, o bloco básico $\ell_{17} - \ell_{18}$, o qual é um pós-dominador de $\ell_{10} - 1_4$. Sendo assim, o predicado b_1 que é o topo da pilha é desempilhado e o predicado considerado pelo algoritmo como controlador das variáveis definidas em $\ell_{17} - \ell_{18}$ será o predicado b_0. Após isso, o predicado b_2 do bloco básico atualmente visitado é empilhado.

Finalmente o algoritmo visita os filhos do bloco $\ell_{17} - \ell_{18}$. No caso do filho ℓ_{19} o predicado b_2, topo atual da pilha é considerado como controlador da variável max_0 definida nesse bloco. Porém no caso do filho $\ell_{20} - \ell_{22}$, será desempilhado o predicado b_2 pois esse bloco básico é pós-dominador do bloco $\ell_{17} - \ell_{18}$. Portanto, o predicado que resta na pilha, isto é, b_0 será considerado o controlador das variáveis max_1, sum_1 e i_1

A Figura 3.9 mostra todas as arestas de controle detectadas a partir da execução do algoritmo proposto nesta tese sobre o exemplo da Figura 3.8. É possível notar que o algoritmo reconhece a transitividade na relação de dependência de controle. Por exemplo: pela Definição 3.1.1, o predicado b_0 também controla as variáveis definidas nos rótulos ℓ_{15}, ℓ_{16} e ℓ_{19}. Porém não há aresta conectando diretamente b_0 à essas variáveis. Na maioria das análises de fluxos é suficiente existir um caminho entre o predicado e todas as variáveis que ele controla, não necessitando haver um caminho

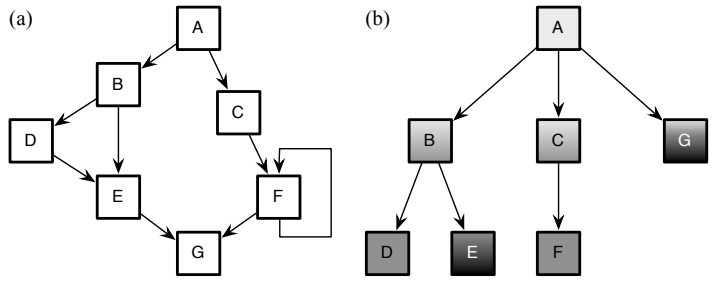


Figura 3.10. (a) Grafo de fluxo de controle. Cada caixa representa um bloco básico. (b) Árvore de dominância.

direto para cada uma.

3.2.3 Propriedades Estruturais do Algoritmo

Define-se a noção de *Região de influência* como:

Definição 3.2.2 (Região de Influência) *A Região de Influência IR_B de um bloco B que termina em um desvio condicional é um conjunto de blocos básicos. Ele contém o bloco B' se, e somente se: (i) $B \in \mathsf{dom}_{B'}$; (ii) $B' \notin \mathsf{pdom}_B$; (iii) não existe B'' tal que $B'' \in \mathsf{pdom}_B$ e $B'' \in \mathsf{dom}_{B'}$.*

A *região de influência imediata* de B é um subconjunto de sua região de influência, a qual é denotada por IIR_B, e definida como:

Definição 3.2.3 (Região de Influência Imediata) *A região de influência imediata IIR_B de um bloco B contém blocos B' se, e somente se: (i) $B' \in \mathsf{IR}_B$, e (ii) não existe bloco B'', $B'' \neq B$, tal que $B' \in \mathsf{IIR}_{B''}$. Diz-se que B é a cabeça de IIR_B.*

A Figura 3.10 ilustra essas definições. $\mathsf{IR}_A = \{B, C, D, E, F\}$ e $\mathsf{IR}_B = \{D\}$. O Bloco C não possui região de influência, porque ele não termina em um desvio condicional. Tem-se que $\mathsf{IIR}_A = \{B, C, F\}$ e $\mathsf{IIR}_B = \{D\}$. A noção de região de influência e região de influência imediata possuem interpretações geométricas. A região de influência de um bloco B é cada outro bloco que é alcançável a partir de B em uma travessia na árvore de dominância do programa, interrompendo a travessia em qualquer bloco em pdom_B. A região de influência imediata de B é obtida por uma travessia na árvore de dominância, novamente iniciando em B, e parando em qualquer bloco que pós-domina B, e também em qualquer bloco B' que termina com um desvio condicional (inclusive, isto é, $B' \in \mathsf{IIR}_B$ e $B' \notin \mathsf{pdom}_B$). O algoritmo da Figura 3.6 conecta um

predicado p as variáveis definidas dentro da região de influência imediata do bloco onde p é usado. Essa afirmação é um corolário do Lema 3.2.4, que segue abaixo:

Lema 3.2.4 *Se (i) B é o dominador imediato de B' ($B = \mathsf{idom}_{B'}$) e (ii) B' pós-domina B ($B' \in \mathsf{pdom}_B$), então B' é o pós-dominador imediato de B ($B' = \mathsf{pdom}_B$).*

> **Prova:** A prova é por contradição. Assuma a existência de um bloco B'', tal que $B'' \neq B'$, e $B'' = \mathsf{pdom}_B$. Mostra-se que (i.b) B'' deve dominar B' e (ii.b) B domina B''. (iii.b) Existe um caminho a partir de B'' para B', porque ambos blocos pós-dominam B, e B'' é o pós-dominador imediato. (iv.b) Se B não domina B'', então existe um caminho a partir de Start para B'' tal que não passa por B. Mas iii.b e iv.b implica que B' não pós-domina B, contradizendo ii.a; assim, (v.b) B domina B''. (vi.b) Se B'' não domina B', então existe um caminho a partir de Start para B' que não passa por B''. Se esse caminho não passa por B, então B não domina B', contradizendo i.a; assim, (vii.b) esse caminho passa por B. Se esse caminho não passa por B'', então B'' não pós-domina B, contradizendo a suposição inicial. Assim, (viii.b) B'' domina B'. v.b e viii.v contradizem i.a. □

Lema 3.2.5 *A função `visit` cria uma aresta de dependência implícita entre um predicado `pred` e uma variável v se, e somente se, `pred` é usado em um desvio condicional que finaliza um bloco B, e v é definida dentro de um bloco $B', B' \in \mathsf{IIR}_B$.*

> **Prova:**
> $\Rightarrow$: a prova é por indução na altura da árvore de dominância. Caso base: se $B = \mathsf{idom}_{B'}, B' \neq \mathsf{pdom}_B$, então B' não pode ser um pós-dominador de B, devido ao Lema 3.2.5. Neste caso, $B' \in \mathsf{IIR}_B$. Passo de indução: assumindo que o ancestral B'' de B' estava em IIR_B. Se B'' não é a cabeça de uma nova região de influência, então ele finaliza em uma instrução `jmp`, e `pred` é propagado para B' nas linhas 15 e 17 de `visit`.
> $\Leftarrow$: se $B' \in \mathsf{IIR}_B$, então B domina B'. Se $B = \mathsf{idom}_{B'}$, então B' será visitado com `pred`, na linha 24 da Figura 3.6. Por outro lado, B domina um bloco B'', que domina B'. Novamente, B' será visitado na linha 24. B'' não pode terminar em uma instrução de desvio condicional, porque $B' \in \mathsf{IIR}_B$. Assim, `visit` será chamada novamente na linha 18. Conclui-se a prova por indução na distância entre B'' para B. □

Se S é um conjunto, então uma relação de equivalência total em S consiste de uma coleção de conjuntos $\{S_1, S_2, \ldots, S_n\}$ tal que (i) todo elemento de S pertence a um, e somente um conjunto $S_i, 1 \leq i \leq n$; e (ii) todo $S_i, 1 \leq i \leq n$ é um subconjunto

de S. O Lema 3.2.7 determina a relação de equivalência total dentro da região de influência de um bloco básico. Este lema será necessário na prova do Teorema 3.2.8. Porém, antes de exibir a prova do Lema 3.2.7, segue o Lema 3.2.6.

Lema 3.2.6 *Se B domina B' e $\mathsf{IR}_B \cap \mathsf{IR}_{B'} \neq \emptyset$, então $\mathsf{IR}_B \supseteq \mathsf{IR}_{B'}$.*

> **Prova:** Se $B = B'$, então a prova do lema é trivial. Por outro lado, a partir de uma interseção não vazia, sabe-se que existe um bloco B'', $B'' \neq B' \neq B$, tal que: (i) $B \in \mathsf{dom}_{B''}$; (ii) $B' \in \mathsf{dom}_{B''}$; (iii) $B'' \notin \mathsf{dom}_B$; (iv) $B'' \notin \mathsf{dom}_{B'}$; e (v) não existe B_1 tal que: (v.a) $B_1 \in \mathsf{dom}_{B''}$; (v.b) $B_1 \in \mathsf{pdom}_B$; e (v.c) $B_1 \in \mathsf{pdom}_{B'}$. Fatos (i-ii) somado a $B \in \mathsf{dom}_{B'}$ assegura que existe um caminho de B para B' para B'' na árvore de dominância. Fatos (v.a-c) asseguram que não existe pós-dominador de B no caminho B para B'. Portanto, $B' \in \mathsf{IR}_B$. Agora, se B' domina um bloco B_p que pós-domina B, então B_p também pós-domina B'; de outra forma, poderia-se alcançar End indo de B para B'. Assim, $\mathsf{IR}_{B'} \subset \mathsf{IR}_B$.

Lema 3.2.7 *O conjunto de regiões de influência imediata dentro da região de influência de um bloco B determina uma relação de equivalência total.*

> **Prova:** Deve ser mostrada duas condições: (Condição 1) Para todo bloco $B' \in \mathsf{IR}_B$, existe um bloco único $B'' \in \mathsf{IR}_B$, tal que $B' \in \mathsf{IIR}_B$. Primeiro, se B' está dentro de uma certa região de influência imediata, então essa região é única, devido à Definição 3.2.3. Segundo, se não existe bloco B'' tal que $B' \in \mathsf{IIR}_{B''}$, então $B' \in \mathsf{IIR}_B$, novamente, por Definição 3.2.3. (Condição 2) Toda região de influência imediata que compartilha um bloco com IR_B é um subconjunto de IR_B. Esse fato é verdade devido ao Lema 3.2.6.

Os Lemas 3.2.5 e 3.2.7 fornecem o arcabouço necessário para mostrar que o algoritmo da Figura 3.6 cria uma corrente de dependências transitivas que conecta um predicado com todas as atribuições de variáveis que este predicado controla. Porém, essa afirmação somente é verdade se houver garantia de que todo predicado é definido imediatamente antes de seu uso em um desvio. Para assegurar essa propriedade é suficiente substituir todo teste condicional como "$\mathtt{br}(v, \ell)$" por duas instruções em sequência: "$v' = v; \mathtt{br}(v', \ell)$", onde v' é uma variável nova que será usada somente nesse desvio condicional. Essa transformação ocorre em tempo constante por desvio; assim, ela é linear sobre o número de desvios condicionais que existem em um programa. Note que os programas que foram mostrados nas Figuras 3.1, 3.2 e 3.3 estão atualmente nesse formato, isto é, todo predicado é definido imediatamente antes do desvio onde ele é usado.

Theorema 3.2.8 *Se um bloco básico B termina com um desvio condicional "$\texttt{br}(p, \ell)$", então a função* `visit` *cria uma corrente de dependências implícitas conectando p a todas variáveis definidas dentro de IR_B.*

> **Prova:** A prova se dá por indução sobre a altura da árvore de dominância. Caso base: O Lema 3.2.5 assegura que será conectado p diretamente a toda variável definida dentro de IIR_B. Se um bloco $B' \in \mathsf{IIR}_B$ termina com "$\texttt{br}(p', \ell')$", então sabe-se que p' é definida dentro de B'. Assim, `visit` conectará p à p'. A prova continua por invocação indutiva de "`visit` B' p'", como o Lema 3.2.7 assegura que as regiões de influência imediata dentro de uma região de influência determinam uma relação de equivalência total. □

3.2.4 Lidando com código Não-Estruturado e Código na forma SSA Não-Convencional

Devido às razões discutidas na Seção 3.3.1, o novo algoritmo cria uma corrente de arestas de dependência entre um predicado p, usado em um desvio tal como $\texttt{br}(p, \ell)$, e cada variável definida por uma função ϕ que p controla. Essa propriedade não se manterá se os argumentos de uma função ϕ não são definidos dentro da região de influência de p, como o Teorema 3.2.8 permite concluir. Tal situação pode acontecer, por exemplo, em programas *não-estruturados*, e em programas na forma de *atribuição única estática não-convencional (CSSA)*.

De acordo com Ferrante et al. [1987], uma *Região Hammock* dentro de um grafo de fluxo de controle é um conjunto de blocos entre um desvio e seu pós-dominador. Um programa é não-estruturado se ele contém pelo menos 1 região *hammock* R que pode ser acessada por diferentes blocos fora de R. Por exemplo, o programa na Figura 3.11 (a) é não-estruturado. Um programa P está na forma CSSA se ele não contém duas variáveis *ϕ-relacionadas* simultaneamente vivas no mesmo ponto de programa. Citando Pereira & Palsberg [2009], duas variáveis, v_1 e v_2, são ϕ-relacionadas em um programa na forma SSA se (i) elas são usadas na mesma função ϕ, como parâmetros ou definição, ou (ii) existe uma terceira variável v_3, de forma que v_1 e v_2 são ϕ-relacionadas à v_3. O programa na Figura 3.11 (b) está na forma CSSA, porque as variáveis ϕ-relacionadas x_0 e x_1 se sobrepõem.

O algoritmo na Figura 3.6 não irá criar qualquer aresta de dependência implícita em nenhum dos programas da Figura 3.11. Esse fato é indesejável, pois para ambos os casos o predicado p_0, controla as funções ϕ. Existem duas maneiras de lidar com esse fato: pode-se alterar o algoritmo para adicionar arestas de controle entre um predicado

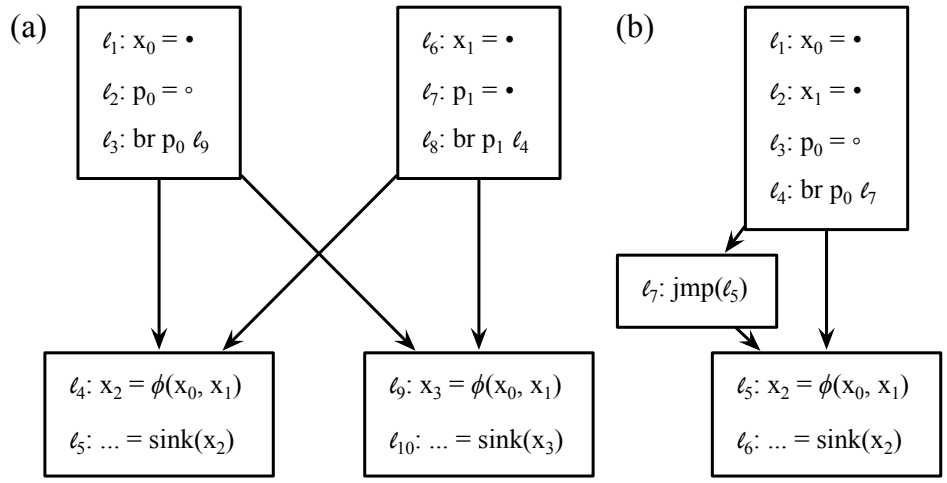


Figura 3.11. (a) Grafo de fluxo de controle não-estruturado. (b) Programa na forma não-convencional SSA.

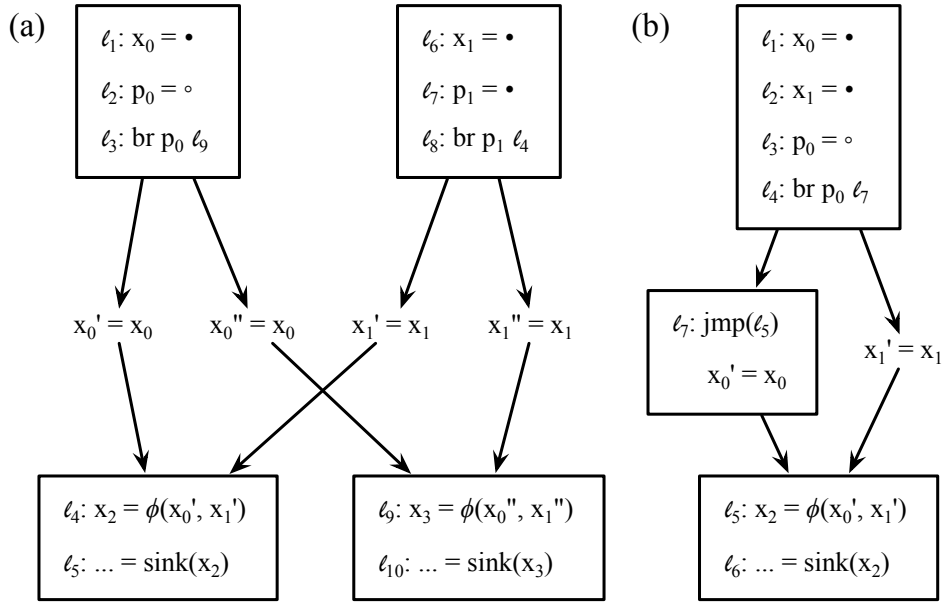


Figura 3.12. Programas da Figura 3.11, após realizar a divisão da linha de vida à la Sreedhar Sreedhar et al. [1999].

p e as funções ϕ que existem na *Fronteira de Dominância*[2] do bloco onde p é usado; ou pode-se fazer a divisão da linha de vida, seguindo a abordagem proposta por Sreedhar et al. [1999] para converter um programa na forma CSSA para a representação SSA. Essa última estratégia foi usada nesta tese, para simplificar as provas de corretude.

A divisão da linha de vida de uma variável v é o conjunto de pontos de programa

[2]para definição de Fronteira de Dominância, veja a Definição 3.2.9

onde v está viva. Em programas na forma SSA, v está viva em todo ponto que é alcançável a partir de uma travessia inversa no GFC de seu programa, iniciando em um local onde v é usada, e parando no – único – local onde v é definida [Appel & Palsberg, 2002, ch.15]. Para implementar a divisão da linha de vida, (i) quebram-se arestas críticas[3] e (ii) divide-se a linha de vida dos argumentos das funções ϕ nos antecessores do bloco onde essas funções ϕ estão localizadas. A divisão da linha de vida de uma variável v em uma aresta do GFC $\ell_i \to \ell_j$ se dá por (i) inserção de uma cópia $v' = v$ na aresta, v' sendo uma variável nova; (ii) renomeação de todo uso de v dominado por ℓ_j para v' e (iii) inserção de funções ϕ no programa para recriar a propriedade SSA. Visto que se usa o método de Sreedhar et al. [1999] nesta tese, passo (iii) não é necessário. A Figura 3.12 ilustra esse processo. É dividida a linha de vida dessas variáveis, o que chama-se *escaping*. Define-se *escaping* de variáveis da seguinte forma:

Definição 3.2.9 (*Escaping* de variáveis) *Diz-se que v "escape" a região de influência de um desvio "ℓ_b : $\texttt{branch}(p, \ell)$", localizado no rótulo ℓ_b se, e somente se v está viva em uma aresta $\ell_o \to \ell_d$, de forma que: (i) ℓ_o é dominada por ℓ_b; e (ii) ℓ_d não é dominada por ℓ_b. No jargão de compiladores, diz-se que a aresta $\ell_o \to \ell_d$ está na* fronteira de dominância *de ℓ_b.*

Lema 3.2.10 *Se é dividida a linha de vida de cada variável que "escape" a região de influência de um condicional "ℓ : $\texttt{br}(p, \ell')$" localizado no rótulo ℓ, então assegura-se que ℓ domina o local de definição de toda variável que passou pelo processo de "escape".*

> **Prova:** Pela Definição 3.2.9, divide-se a linha de vida nas arestas cuja origem é dominada por ℓ. Assim, o local de definição de cada nova variável é dominado por ℓ. Após a renomeação que sucede a divisão da linha de vida, somente as novas definições das variáveis originais que passaram pelo processo de "escape" podem ser usadas fora da região de influencia de ℓ. Essas são as novas variáveis que passaram por tal processo, novamente de acordo com a Definição 3.2.9. □

A estratégia de divisão de linha de vida possibilita o algoritmo proposto nesta tese estabelecer dependências de controle entre o predicado p que controla um desvio, e cada função ϕ que pode ser controlada por tal desvio. Essa propriedade é descrita formalmente no Teorema 3.2.11.

[3] Uma aresta do GFC é crítica se ela conecta um bloco básico com múltiplos sucessores a um bloco básico com múltiplos antecessores.

Theorema 3.2.11 *Após a divisão da linha de vida, assegura-se que o algoritmo proposto nesta tese cria um caminho de arestas de dependência entre um predicado p usado em "*`branch`*(p, ℓ')", e cada variável definida por uma função ϕ que é controlada por esse desvio.*

> **Prova:** Um desvio pode somente controlar uma função ϕ se essa função usa uma variável v viva na fronteira de dominância desse desvio [Appel & Palsberg, 2002, Ch.15]. Pelo Lema 3.2.10, sabe-se que o desvio domina a aresta de fluxo de controle onde v é definida. O algoritmo na Figura 3.6 irá criar uma aresta entre p e v, de acordo com o Teorema 3.2.8. □

Complexidade. A função `visit` cria no máximo uma aresta de dependência implícita chegando em qualquer vértice no grafo SSA de um programa. Esse resultado, justificado pelo Lema 3.2.12, permite definir um limite superior para a complexidade de espaço requerido pelo algoritmo introduzido nesta tese.

Lema 3.2.12 *O algoritmo na Figura 3.6 assegura que um vértice tem no máximo 1 aresta de controle direcionada a ele.*

> **Prova:** Cada vértice é visitado somente uma vez; assim, `visit` é invocada nesse vértice não mais que uma vez. □

O Lema 3.2.12 refere-se à variáveis na forma SSA. Programas na forma SSA contém mais variáveis que suas versões originais; porém, esse crescimento ainda é linear [Sreedhar & Gao, 1995]. A estratégia de divisão de linha de vida de Sreedhar, que é usada para lidar com programas não estruturados, cria uma nova variável para cada argumento de uma função ϕ. Assim, `visit` pode criar uma aresta de dependência implícita por variável no programa em SSA, e uma aresta por argumento de uma função ϕ. Isso ainda é linear sobre o tamanho do programa original, medido como o número de definições + número de usos de variáveis. Além disso, Benoit *et al.* mostraram que uma variável raramente será usada mais que 5 vezes em *benchmarks* do mundo real [Boissinot et al., 2008]. Assim, o novo algoritmo aqui proposto será linear sobre o número de variáveis de programa na prática, mesmo para códigos não estruturados. Note que usa-se o método ingênuo introduzido por Sreedhar *et al.* (Method I) [Sreedhar et al., 1999, Sec 4.1]. Nesse mesmo artigo, eles discutiram duas outras formas de divisão de linha de vida (Método II e III), que introduzem um pouco menos novas variáveis.

$$\textbf{lookup}([(v,x) :: S], v) \Rightarrow x \quad \frac{v' \neq v \qquad \textbf{lookup}(S, v) \Rightarrow x}{\textbf{lookup}([(v', x') :: L], v) \Rightarrow x}$$

$$\frac{v \in S_v}{\textbf{lookupFirst}([(v,x) :: S], S_v) \Rightarrow x}$$

$$\frac{v' \notin S_v \qquad \textbf{lookupFirst}(S, S_v) \Rightarrow x}{\textbf{lookupFirst}([(v', x') :: S], S_v) \Rightarrow x}$$

$$\frac{\forall v_i \in P, \textbf{lookup}(S, v_i) = x_i \qquad x_1 \sqcup \ldots \sqcup x_n = x}{\textbf{join}(S, P) \Rightarrow x}$$

Figura 3.13. Biblioteca de gerenciamento da pilha.

3.3 Propriedades Semânticas

Nesta seção, discutem-se as garantias que o novo algoritmo provê para seus usuários. Para tanto, as Figuras 3.13, 3.14 e 3.15 definem a semântica da linguagem exibida na Figura 3.5. A relação $\xrightarrow{i}$ (Figura 3.14) descreve a semântica das operações de dados e aritméticas, e a relação $\xrightarrow{e}$ (Figura 3.15) fornece a semântica das operações que alteram o fluxo de controle do programa. *State* é dado como uma tripla:

- `pc`: o contador de programa, que indexa um vetor I de instruções. Esse elemento é somente usado na relação $\xrightarrow{e}$.
- S: a pilha de variáveis, que combina nomes a valores. Os possíveis valores são $\bullet$ ou $\circ$. Define-se o operador *meet* $\sqcup$ de forma que $\bullet \sqcup \bullet = \bullet$ e $\circ \sqcup x = x \sqcup \circ = \circ$ para qualquer x.
- P: o conjunto de predicados *ativos*. Dize-se que um predicado p está ativo se o fluxo do programa está atravessando a região de influência de um bloco básico que termina com $\texttt{br}(p, \ell)$.

Novamente, seja (::) uma construção de lista. Atribuições adicionam uma nova informação no topo da pilha S. Sempre que necessário recuperar o valor x associado com uma variável v, faz-se uma varredura na pilha, do topo para a base, procurando pelo par (v, x). Esta pesquisa é realizada pela função **lookup**, exibida na Figura 3.13. Quando interpretando uma função ϕ tal como $v = \phi(v_1, v_2)$, procura-se na pilha pela primeira ocorrência de qualquer informação contendo v_1 ou v_2. Essa ação é implementada pela função **lookupFirst**, também exibida na Figura 3.13. Procurar pelo primeiro parâmetro de uma função ϕ é correto para programas estritos, isto é, programas nos quais cada variável é definida antes de serem usadas. Esta é uma das principais propriedades de programas na forma SSA, como afirmado por Zhao et al. [2013]. Por

$$P \vdash \langle v = \circ, S\rangle \xrightarrow{i} (v, \circ) : S$$

$$P \vdash \langle v = \bullet, S\rangle \xrightarrow{i} (v, \mathbf{join}(S, P)) : S$$

$$\frac{\mathbf{lookup}(S, v') \Rightarrow \bullet \qquad \mathbf{join}(S, P) = \bullet}{P \vdash \langle v = \mathtt{sink}(v'), S\rangle \xrightarrow{i} (v, \bullet) : S}$$

$$\frac{\mathbf{lookupFirst}(S, \{v_1, v_2\}) \Rightarrow x}{P \vdash \langle v = \phi(v_1, v_2), S\rangle \xrightarrow{i} (v, x) : S}$$

$$\frac{\mathbf{lookup}(S, v_i) \Rightarrow x_i \qquad x = x_1 \sqcup \ldots \sqcup x_n \sqcup \mathbf{join}(S, P)}{P \vdash \langle v = \mathtt{use}(v_1, \ldots, v_n), S\rangle \xrightarrow{i} (v, x) : S}$$

Figura 3.14. Semântica das operações de dados e aritméticas.

$$\frac{I[\mathtt{pc}] = \mathtt{jmp}(\ell) \qquad I \vdash \langle \ell, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}{I \vdash \langle \mathtt{pc}, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}$$

$$\frac{I[\mathtt{pc}] = \mathtt{br}(v, \ell) \qquad I \vdash \langle \ell, S, P \cup \{v\}\rangle \xrightarrow{e} \langle S', P'\rangle}{I \vdash \langle \mathtt{pc}, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}$$

$$\frac{I[\mathtt{pc}] = \mathtt{br}(v, \ell) \qquad I \vdash \langle \mathtt{pc} + 1, S, P \cup \{v\}\rangle \xrightarrow{e} \langle S', P'\rangle}{I \vdash \langle \mathtt{pc}, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}$$

$$\frac{I[\mathtt{pc}] = \mathtt{pdom}(P") \qquad I \vdash \langle \mathtt{pc} + 1, S, P \setminus P"\rangle \xrightarrow{e} \langle S', P'\rangle}{I \vdash \langle \mathtt{pc}, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}$$

$$\frac{P \vdash \langle I[\mathtt{pc}], S\rangle \xrightarrow{i} S' \qquad I \vdash \langle \mathtt{pc} + 1, S', P\rangle \xrightarrow{e} \langle S", P"\rangle}{I \vdash \langle \mathtt{pc}, S, P\rangle \xrightarrow{e} \langle S", P"\rangle}$$

Figura 3.15. A semântica operacional das instruções que modificam o fluxo de controle do programa. Desvios condicionais são não-determinísticos: qualquer resultado de sua avaliação é válido para os propósitos desta tese.

simplicidade, este texto assume que cada bloco básico contém no máximo uma função ϕ em seu início. Para manipular múltiplas funções ϕ, Zhao *et al.* procuram por uma lista de argumentos, ao invés de um.

Para modelar os efeitos de $\ell : \mathtt{br}(p, \ell')$ sobre atribuições que acontecem dentro da região de influência de ℓ, usa-se o ambiente de predicados ativos P. Sempre que

atravessar um desvio, adiciona-s p para P. O efeito de p termina em pdom_ℓ, então retorna-se dentro da região do programa que necessariamente será atravessada se o rótulo ℓ for visitado, independente do valor de p. Contrariamente à linguagem de programa While usada no sistema de tipos seguros padrão [Hunt & Sands, 2006; Russo & Sabelfeld, 2010], a linguagem didática definida nesta tese não oferece sintaxe para indicar pós-dominância. Assim, assume-se uma instrução especial ℓ_p : $\mathtt{pdom}(P')$ no rótulo $\ell_p = \mathsf{pdom}_\ell$. Seja P' o conjunto de variáveis usadas em desvios que são pós-dominados por ℓ_p. Como exibido na Figura 3.15, essa instrução remove do conjunto de predicados ativos toda variável em P.

3.3.1 De Dependências para Tipos

Se um programa I, no estado $\langle S, P\rangle$ não pode realizar um passo, então diz-se que esse programa está *travado*. Existem várias maneiras nas quais programas, escritos usando a linguagem deste trabalho de tese, podem travar. A máquina abstrata deste trabalho pode travar se: (i) o contador de programas apontar para uma instrução não existente (Figura 3.15); (ii) tentar utilizar uma variável não definida (veja a Figura 3.14); ou (iii) `sink` receber um argumento do tipo $\circ$. Visto que aqui não se está interessado em grafos de fluxo de controle mal formados, não há preocupações com (i). Zhao et al. [2013] mostraram que (ii) não pode acontecer em programas na forma SSA, isto é, programas nos quais qualquer uso de uma variável é dominado por sua definição. Programas que não apresentam as condições (i) e (ii) são ditos *bem formados*. Daqui em diante, somente serão considerados programas bem formados. Assim, assume-se que somente a condição (iii) pode parar a máquina abstrata. Note que não é modelado terminação nesta semântica: a existência de uma instrução `halt` é imaterial para essa formalização. As dependências explícitas, somadas às dependências implícitas criadas pelo procedimento exibido na Figura 3.6 permitem definir uma forma de atribuir tipos à variáveis:

Definição 3.3.1 (Tipos como Alcançabilidade no Grafo SSA) *Seja* Prog *um programa, $G = (V, E)$ seu grafo SSA, e $V_\circ$ o conjunto formado pelas variáveis v definidas como $v = \circ$ em* Prog. *Se $v \in V_\circ$, então v tem tipo H. O tipo de uma variável u alcançável a partir de qualquer variável $v \in V_\circ$ através de uma travessia de G é H; por outro lado, será L.*

Note que a Definição 3.3.1 está lidando com um reticulado de segurança de peso dois. Poder-se-ia usar reticulados mais expressivos adaptando a Definição 3.3.1 apropriadamente, por exemplo, o tipo de uma variável v é a união sobre todos os caminhos

de tipos de todas as variáveis que alcançam v. Restringe-se então a discussão no resto dessa seção a um reticulado mais simples; O Teorema 3.3.3 descreve a noção de não-interferência aqui utilizada. Para prová-lo, será necessário o Corolário 3.3.2 obtido a partir do Teorema 3.2.8.

Corolário 3.3.2 *Se uma variável v é definida dentro da região de influência de um rótulo ℓ : $\texttt{br}(v', \ell')$, e v' tem tipo H, então v tem tipo H de acordo com a Definição 3.3.1.*

> **Prova:** Pelo Teorema 3.2.8, se v é criado dentro da região de influência de ℓ, então o algoritmo da Figura 3.6 cria uma corrente de dependências implícitas entre v' e v O resultado é obtido a partir da Definição 3.3.1. □

Theorema 3.3.3 *Se uma variável tem tipo L, ela não pode ser atribuída a um valor $\circ$.*

> **Prova:** A prova consiste de uma indução sobre o comprimento do traço de instruções executado até a atribuição de v. Será realizado uma análise de caso sobre as instruções usadas para definir v. O caso $v = \circ$ é impossível, pela trivial inversão da relação de tipagem exibida na Definição 3.3.1. Se $\ell : v = \bullet$, então seu valor depende de P, o predicado que influencia ℓ. A partir do Corolário 3.3.2, se qualquer desses predicados tem tipo H, então v terá tipo H, contradizendo a hipótese. Se $\ell : v = \texttt{use}(v_1, \ldots, v_n)$, então o tipo de v depende do tipo de cada v_i e P. Todas as variáveis v_i devem ter tipo L, e por indução, elas terão o valor $\bullet$. A análise de P é similar ao caso anterior. A análise de $v = \phi(v_1, v_2)$ é similar, assumindo que cada v_1 é definida seguindo a divisão de linha de vida de Sreedhar et al. [1999], como discutido na Seção 3.2.4. Neste caso, de acordo com o Teorema 3.2.11, os argumentos da função ϕ são definidos dentro da região de influência dos predicados ativos em P. □

A noção de progresso é obtida a partir do Teorema 3.3.3. Ela é declarada como:

Corolário 3.3.4 (Progresso) *Se um programa bem formado não contém uma instrução $v' = \texttt{sink}(v)$ tal que v tem tipo H, então esse programa não pode travar.*

Progresso significa que uma instrução sorvedouro não receberá um valor do tipo H. Por outro lado, nada é mencionado sobre a execução de instruções sorvedouro dentro de regiões de programa de alto nível de segurança. Como apontado por Russo & Sabelfeld [2010], eventos como esse podem também permitir o vazamento de informação sensível. Por sorte, é fácil adaptar a semântica aqui proposta para modelar esse tipo

$CS\,[\![\textbf{skip}]\!] \xrightarrow{t} ;$	$CS\,[\![\mathrm{v_s}^{t} := \mathrm{E}^{\Gamma}]\!] \rightarrow \mathrm{I};\ \mathrm{v_s} = \texttt{use}(x)$
$CS\,[\![\mathrm{D}_1; \mathrm{D}_2]\!] \xrightarrow{t}$ $CS\,[\![\mathrm{D}_1]\!]; CS\,[\![\mathrm{D}_2]\!]$	$CS\,[\![$ **if** E^{Γ} $(\mathrm{D}_1; \boldsymbol{\Gamma}' = \boldsymbol{\Gamma}_1)$ $(\mathrm{D}_2; \boldsymbol{\Gamma}' = \boldsymbol{\Gamma}_2)$ $]\!] \xrightarrow{t}$ $\ell:$ I; $\texttt{br}(x, \ell)$; $CS\,[\![\mathrm{D}_1]\!]$; $CS\,[\![\mathrm{D}_2]\!]$; $\boldsymbol{\Gamma}' = \phi(\boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2)$

$CS\,[\![\boldsymbol{\Gamma}' = \boldsymbol{\Gamma}_0;$ **while** E^{Γ} $(\mathrm{D}_n; \boldsymbol{\Gamma}' = \boldsymbol{\Gamma}_n)\,]\!] \xrightarrow{t}$

$\ell_0:\ \boldsymbol{\Gamma}' = \phi(\boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2);\ \mathrm{I};\ \texttt{br}(x, \ell_n);\ CS\,[\![\mathrm{D}_n]\!];\ \texttt{jmp}(\ell_0);\ \ell_n:$

Figura 3.16. Tradução da linguagem de alto nível While com tipos fixados [Hunt & Sands, 2006, Fig.4] para a linguagem de baixo nível definida nesta tese. Cada expressão E^{Γ} é traduzida como uma sequência de instruções I que produzem um resultado final para uma variável x.

de vazamento indireto. Cada operação sorvedouro define uma variável, por exemplo, $\ell : v = \texttt{sink}(v')$. O nível de segurança em ℓ é dado pelo tipo de v.

3.3.2 Equivalência com o Sistema de Tipos de Hunt e Sands

A noção de dependências explícita e implícita definida nesta tese fornece uma forma eficiente de implementar o sistema de tipos de Hunt e Sands [Hunt & Sands, 2006]. Para suportar essa afirmação, a Figura 3.16 provê uma tradução da linguagem While para a linguagem definida na Figura 3.5. O ponto de início é a versão tipos fixados de While que Hunt e Sands definiram na Seção 7 de seu trabalho. Como Hunt e Sands especularam [Hunt & Sands, 2006, Sec7.6], essa versão é muito próxima à forma SSA. Assim, a tradução foi realizada juntando, via funções ϕ, as variáveis que eles definiram no fim das construções de fluxo de controle. Hunt e Sands não definiram a semântica de While; porém, Russo e Sabelfeld fizeram isso posteriormente Russo & Sabelfeld [2010]. Assim, para mostrar a corretude da tradução, essa última formalização será usada. No Teorema 3.3.5, seja a relação $\xrightarrow{r}$ os passos definidos por Russo e Sabelfeld para sua versão da linguagem While.

Theorema 3.3.5 *Seja C uma sentença de While tal que $CS[\![C]\!] \xrightarrow{t} I$, e seja S uma pilha. Se $\langle C, S\rangle \xrightarrow{r} \langle stop, S'\rangle$ então existe um conjunto P' ativo tal que $I \vdash \langle \ell_0, S, \{\}\rangle \xrightarrow{e} \langle S', P'\rangle$.*

Prova: A prova é por indução na forma da árvore de derivação de $\xrightarrow{r}$, seguindo a técnica discutida por [Nielson & Nielson, 1992, p.80]. Para cada pilha inicial e cada árvore de derivação na semântica natural de While existe uma sequência finita de computação em I que produz a mesma pilha final, desde que cada variável é definida, antes de ser utilizado, ou possui um valor inicial armazenada na pilha de outra forma. □

Pode-se mostrar que o algoritmo aqui apresentado, quando aplicado na linguagem de baixo nível que obtém-se via o compilador exibido na Figura 3.16, corretamente implementa as regras de tipagem de Hunt e Sands. Isso é formalizado no Teorema 3.3.6. Sem perda de generalidade, assume-se um sistema de tipos com somente dois tipos, por exemplo, H e L. Para generalizar esse sistema de tipos para um semi-reticulado maior, seja o tipo de uma variável v a união sobre todos os caminhos de tipos de cada variável que pode influenciar v. Novamente, variável u influencia variável v se, e somente se, existe um caminho no grafo SSA a partir de u para v.

Theorema 3.3.6 *Seja C uma sentença de While tal que $CS[\![C]\!] \xrightarrow{t} I$. O sistema de tipos de Hunt e Sands atribui a variável $v_x \in C$ ao tipo H, se e somente se, v_x tem tipo H de acordo com a definição 3.3.1.*

Prova: A prova é por análise de caso em cada uma das sete regras que são parte da versão algorítmica do sistema de tipos de Hunt e Sands [Hunt & Sands, 2006, Tabs. 1 & 2]. Aqui, inicia-se sobre o caso se-então-senão, com as seguintes regras de tipagem:

$$\frac{\Gamma \vdash E : t \qquad p \sqcup t \vdash \Gamma\{C_i\}\Gamma', i = 1, 2}{p \vdash \Gamma\{\mathit{if}\ E\ C_1\ C_2\}\Gamma'}$$

e a tradução em tipos fixados [Hunt & Sands, 2006, Tab. 4] se dá pela expressão abaixo, onde $\Gamma' = \Gamma'_1 \sqcup \Gamma'_2$:

$$\frac{\Gamma \vdash E : t \qquad p \sqcup t \vdash \Gamma\{C_i \rightsquigarrow D_i\}\Gamma'_i, i = 1, 2}{p \vdash \Gamma\{\mathit{if}\ E\ C_1\ C_2 \rightsquigarrow \mathit{if}\ E^{\Gamma}\ (D_1; \Gamma' := \Gamma_1)(D_2; \Gamma' := \Gamma_2)\}\Gamma'}$$

Para esta última expressão, o tradutor $\xrightarrow{t}$ produz a sequência de instruções abaixo, onde $CS(E^{\Gamma}) = I$, e o valor da expressão produzida por I é armazenada em x:

$$I; \mathtt{br}(x, \ell); CS[\![D_1]\!]; \ell : CS[\![D_2]\!]; \Gamma' = \phi(\Gamma_1, \Gamma_2)$$

Por indução na árvore de derivação $\xrightarrow{t}$, tem-se que o tipo de x é o tipo de E^{Γ}, por exemplo, t. A partir do Teorema 3.2.8 somado ao Corolário 3.3.2, tem-se

que o tipo de qualquer variável definida dentro de $CS[\![D_1]\!]$ ou $CS[\![D_2]\!]$ será pelo menos t. E, novamente devido ao Teorema 3.2.8, tem-se que o tipo de x é a união do tipo de cada predicado p_i tal que I (e assim a definição de x) está dentro da região de influência de p_i. Este é o tipo "p", como definido por Hunt e Sands. Finalmente, Hunt e Sands definirão Γ' exatamente como as variáveis cujo tipo é dado pela união de tipos de variáveis em Γ_1 e Γ_2. Se existe um par $x_1 \in \Gamma_1$ e $x_2 \in \Gamma_2$, tal que essas variáveis correspondem a um dado $x \in \Gamma'$, então tem-se uma instrução $x = \phi(x_1, x_2)$, e o tipo de x é dado pela união dos tipos de x_1 e x_2, de forma que o grafo SSA irá conter as duas arestas de dependências de dados $x_1 \to x$ e $x_2 \to x$. Esta prova é similar para as outras seis regras. □

3.4 Considerações Finais

O presente capítulo evidenciou na Seção 3.1 que o rastreamento de dependências de controle, isto é, determinar fluxos implícitos é um problema importante para a criação do grafo de dependências que por sua vez pode ser usado na implementação do sistema de tipos de Hunt e Sands. Ferrante et al. [1987] propuseram uma solução para esse rastreamento focada em detectar quais instruções um predicado controla no texto do programa. Essa abordagem é diferente da solução proposta nesta tese, materializada na forma do algoritmo da Figura 3.6, visto que aqui o foco está em saber quais valores um predicado controla. A Seção 3.2 definiu uma linguagem central que foi usada na concepção do algoritmo, que possui complexidade linear de tempo e espaço. Ela também apresentou as propriedades estruturais do mesmo na forma de vários lemas, corolários e teoremas, bem como suas respectivas provas. A Seção 3.3 apresentou as propriedades semânticas da solução, mostrando que o grafo de dependências pode ser usado para implementar o sistema de tipos de Hunt e Sands. Finalmente, vale ressaltar que o novo algoritmo de rastreamento das dependências implícitas, somado à implementação esparsa do sistema de tipos de Hunt e Sands são as duas principais contribuições desta tese para o estado da arte em rastreamento de fluxos de informação em programas de computador.

Capítulo 4

FlowTracker: Uma Ferramenta de Análise Esparsa e Estática

Neste Capítulo apresenta-se a ferramenta FlowTracker, um módulo desenvolvido para o compilador LLVM 3.7 que implementa todas as ideias discutidas e apresentadas no Capítulo anterior. FlowTracker é capaz de analisar de forma muito rápida programas escritos em linguagem C ou C++ em busca de canais laterais baseados em tempo. Caso encontrado, o traço de instruções vulnerável é reportado ao desenvolvedor na forma de linhas respectivas ao programa testado, permitindo a fácil depuração do código e consequente remoção do problema por parte do programador. Neste Capítulo também são discutidos alguns detalhes de implementação tais como a propagação de dados através da memória e a inter-proceduralidade. Finalmente são apresentados resultados experimentais comprovando que FlowTracker é: (i) efetivo na detecção dos canais laterais, (ii) é adaptável, sendo capaz de detectar outras vulnerabilidades relacionadas ao fluxo de informação e (iii) é escalável, possibilitando analisar em tempo aceitável programas realmente grandes como o código fonte do compilador GCC.

4.1 Motivação

Para validar todas as ideias discutidas nesta tese, elas foram materializadas em FlowTracker, uma ferramenta de análise estática sensível ao fluxo e inter-procedural incorporada no compilador LLVM 3.7, que cria o grafo SSA a partir de um código fonte de entrada em linguagem C/C++[1]. Essa ferramenta pode ser configurada para rastrear diferentes tipos de informação: o usuário é livre para definir operações fonte e sorve-

[1]FlowTracker pode ser facilmente modificado para aceitar outras linguagens de programação.

douros. A inter-proceduralidade é implementada seguindo a descrição de Hammer & Snelting [2009]. Desde março de 2014, FlowTracker é usada para detectar vulnerabilidade de canais laterais baseado em tempo em implementações criptográficas.

Uma vez construído o grafo SSA, *FlowTracker*, no contexto da vulnerabilidade de canais laterais, inicia a busca por caminhos que conectam vértices que representam informações sigilosas a vértices sorvedouros, que representam predicados de instruções de desvio ou indexação de memória. Estas informações sigilosas devem ser informadas pelo usuário através de uma entrada para *FlowTracker* em formato *eXtensible Markup Language* (XML). Cada caminho encontrado por FlowTracker é reportado ao usuário que poderá identificar o traço de instruções correspondente em seu programa e corrigir o problema, basicamente modificando sua implementação a fim de quebrar a cadeia de dependências entre o sorvedouro e a informação sigilosa. *FlowTracker* executa em tempo linear sobre o tamanho do programa. A implementação de FlowTracker tem 4,771 linhas de código comentado C++. Esse tamanho é devido à necessidade de manipular cada tipo individual de instruções da representação intermediária do LLVM. Essa manipulação corresponde às linhas 9-13 da Figura 3.6.

4.2 Canais Laterais Induzidos pelo Compilador

Um compilador pode induzir canais laterais baseados em tempo enquanto realiza transformações em um código fonte que foi implementado de forma isócrona, isto é, sem existência de canais laterais em sua concepção. Um exemplo típico é devido aos operadores de curto-circuito de C e Java. Por exemplo, o código da Figura 4.1 (a) parece ser invariante de tempo. Porém, o compilador irá traduzir este código para uma versão vista na parte (b) da mesma Figura. Assim, acredita-se que técnicas que detectam comportamento invariante de tempo no nível de código fonte [Agat, 2000], embora muito úteis no processo de desenvolvimento de software, devem ser complementadas com uma validação no nível do compilador. Por exemplo, citando Agat [Agat, 2000, p.51]: *"A minor oversimplification in our semantics is that it neglects the unconditional jumps made in the machine code that a compiler produces for if- and while- commands"*. A abordagem descrita neste trabalho de tese não sofre dessa limitação.

```
int eq(char *p, char *q) {
  a0 = (p[0] == q[0]);
  a1 = (p[1] == q[1]);
  a2 = (p[2] == q[2]);
  return a0 && a1 && a2;
}
```

(a)

```
int eq(char *p, char *q) {
  if (p[0] != q[0])
    return false;
  else if (p[1] != q[1])
    return false;
  else
    return p[2] == q[2];
}
```

(b)

Figura 4.1. (a) Programa isócrono. (b) Código variante de tempo.

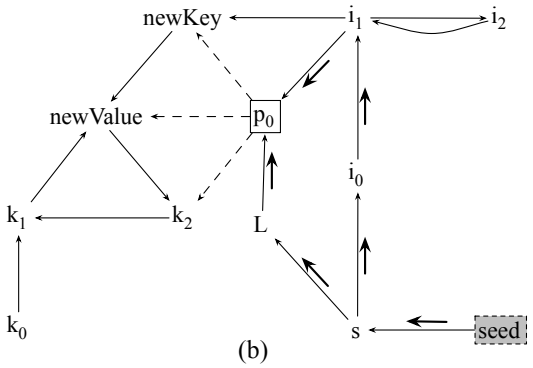
4.3 Derramamento de Registradores e Cópia Coalescente

Os desenvolvimentos apresentados na Seção 3.2 funcionam para qualquer programa na forma SSA com sentenças `goto`. FlowTracker executa imediatamente antes da fase de compilação conhecida como *Static Single Assignment Elimination*. Neste momento, o compilador substitui funções ϕ por cópias de instruções, efetivamente produzindo uma representação concreta do programa. Essa representação é então fornecida ao alocador de registradores. Assim, após a execução de FlowTracker, o programa pode ainda ser modificado pelo alocador de registradores. Essas modificações são causadas por *copia coalescente* ou por *derramamento de registradores*. Cópia coalescente é uma otimização que consiste em remover instruções de movimentação que leem e escrevem no mesmo registrador. Derramamento é o ato de mapear registradores para a memória. Tal mapeamento é realizado através da inserção de instruções de carga e armazenamento no programa.

Para execução correta de FlowTracker, é essencial compreender que nenhuma dessas transformações de programa - coalescência e derramamento - criam novas instruções de desvio no programa. Uma prova dessa última afirmação envolve a análise da implementação LLVM disponível em `lib/Target/X86`. Dados sigilosos podem ser enviados para a memória, devido ao derramamento. Porém, eles não podem ser usados para indexar a memória ou controlar desvios, se tal comportamento não está presente no programa original.

```
1 long foo(int seed) {
2   int s = random(seed);
3   int i = s;
4   int L = s + 8;
5   long k = 0;
6   for (; i < L; i++) {
7     char newKey = random(i) % 128;
8     int newValue = k | newKey;
9     k = newValue << 8;
10  }
11  return k;
12 }
```

(a)



(b)

Figura 4.2. (a) Programa invariante de tempo. (b) Grafo SSA do programa. O caminho falso positivo (que a presente análise manipula corretamente) está marcado com setas.

4.4 Lidando com laços constantes

Um programa pode conter um laço cujo predicado depende de uma informação sigilosa, mas que sempre itera o mesmo número de vezes. O programa na Figura 4.2 (a) ilustra essa situação. O laço na linha 6 sempre itera 8 vezes. Mesmo assim, o grafo de dependência deste programa, exibido na Figura 4.2 (b) contém um caminho a partir de *seed*, a informação sigilosa que deseja-se manter secreta, e p_0, o predicado que controla o laço. Esse caminho seria reportado como um falso positivo.

Para evitar falsos positivos como o da Figura 4.2, FlowTracker utiliza uma análise estática conhecida como *scalar evolution* [Grosser, 2011]. *Scalar evolution* fornece, para cada variável v, uma expressão afim, por exemplo, $scev(v) = a \times i + b$, que determina qual o valor de v durante a execução do programa. No exemplo da Figura 4.2 (a), tem-se que $scev(i) = s$, e que $scev(L) = s + 8$. A diferença entre essas duas expressões afins fornece o número de iterações do laço em questão. *Scalar evolution* admite uma implementação $O(N)$, onde N é o número de instruções no programa.

4.5 Manipulando Efeitos Colaterais

O grafo SSA representa relações entre variáveis de programa. Assim, é difícil rastrear, nessa estrutura de dados, dependências entre sentenças que podem produzir efeitos colaterais, se essas sentenças não definem novos nomes de variáveis. A Figura 4.3 ilustra esse problema. Esta Figura mostra o GFC de um laço no estilo *repeat*, após a divisão de linha de vida das variáveis, de acordo com o método discutido anteriormente. O predicado p_0 controla cada invocação, mas não a primeira, da função f, em ℓ_4. O

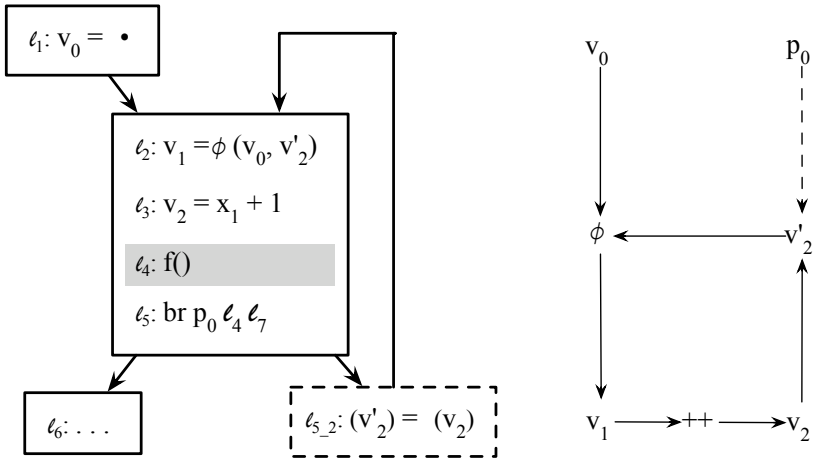


Figura 4.3. Não-SESE, isto é, Grafo *Não-Single Entry Single Exit* criado a partir do laço *repeat*, mais seu grafo SSA com arestas de dependência de controle.

grafo de dependência de Ferrante *et al.* permite representar essa relação trivialmente. Esta tese por outro lado, lida com esse tipo de dependência entre variáveis usando um "truque" para representar a dependência de controle entre p_0 e $f()$ em ℓ_4.

Para representar dependências entre predicados e sentenças que não definem variáveis, usa-se a seguinte abordagem: (i) define-se uma variável nova t no vértice *start* do GFC; (ii) adiciona-se um uso simulado de t no vértice *end* do GFC; (iii) adiciona-se um uso de t próximo à cada sentença que não cria uma variável; (iv) divide-se a linha de vida de t no início de cada bloco básico. Se o bloco tem múltiplos predecessores, faz-se essa divisão de linha de vida via uma função ϕ.

A Figura 4.4 mostra a versão transformada do programa da Figura 4.3. Neste exemplo, a primeira definição da variável t é t_0, e seu último uso acontece em ℓ_{5_6}. Uma nova versão de t é criada no início de cada bloco básico. Note que, uma vez dividida a linha de vida de uma variável, deve-se assegurar que o programa resultante está ainda na forma SSA. Assim, tem-se uma função ϕ em ℓ_2 juntando diferentes versões de t. Essa função ϕ substitui a redefinição de t em ℓ_2. Nessa nova versão do programa sabe-se que a invocação de t, a qual está agora sintaticamente ligada a t_1, depende de p_0, devido ao caminho $p_0 \rightarrow t_2 \rightarrow t_1$ no grafo SSA.

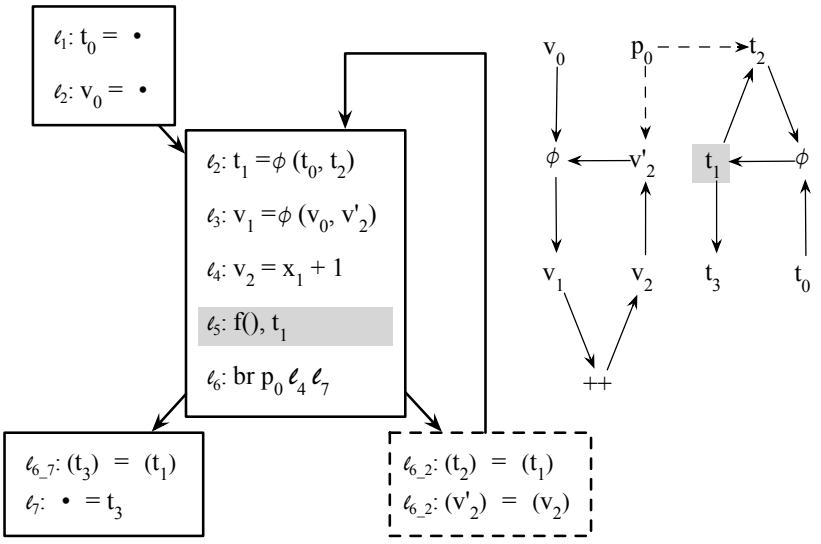


Figura 4.4. Programa da Figura 4.3, após a variável auxiliar t ser adicionada próximo as sentenças que não definem novas variáveis.

4.6 Inter-proceduralidade e Propagação de Memória

FlowTracker implementa uma análise inter-procedural, a qual não é sensível ao contexto. De forma a alcançar inter-proceduralidade, basta criar conexões entre os argumentos reais e os formais das funções. Similarmente, deve-se também criar uma conexão entre a variável retornada por uma função e a variável usada para ler este valor. Conexões entre argumentos formais e reais são criadas via funções ϕ. Assim, se um programa contém uma função f, declarada como $f(\ldots, v, \ldots)$, e n chamadas tais como $f(\ldots, a_i, \ldots), 1 \leq i \leq n$, então cria-se uma instrução $v = \phi(a_1, \ldots, a_n)$ para representar tais conexões.

Um último detalhe de implementação diz respeito a como manipular propagação de informação via memória. Até agora, tem-se assumido que toda variável está armazenada em um registrador. porém, informação flui pela memória sempre que se tem operações de carga e armazenamento. Em outras palavras, deseja-se rastrear fluxos tais como $v = \bullet, *m = v; w = *m; x = \texttt{sink}(w)$; Para rastrear esse tipo de fluxo, utiliza-se uma análise de ponteiros. Em termos concretos, o compilador LLVM provê cinco tipos diferentes de análises de ponteiros, as quais são utilizadas aqui para conhecer as diferentes localizações de memória que o programa manipula. A Figura 4.5 mostra o número de vértices de memória (*MemNodes*) no grafo SSA para os 50 maiores programas da coletânea de testes do LLVM. Esse número é contrastado com a quantidade de vértices

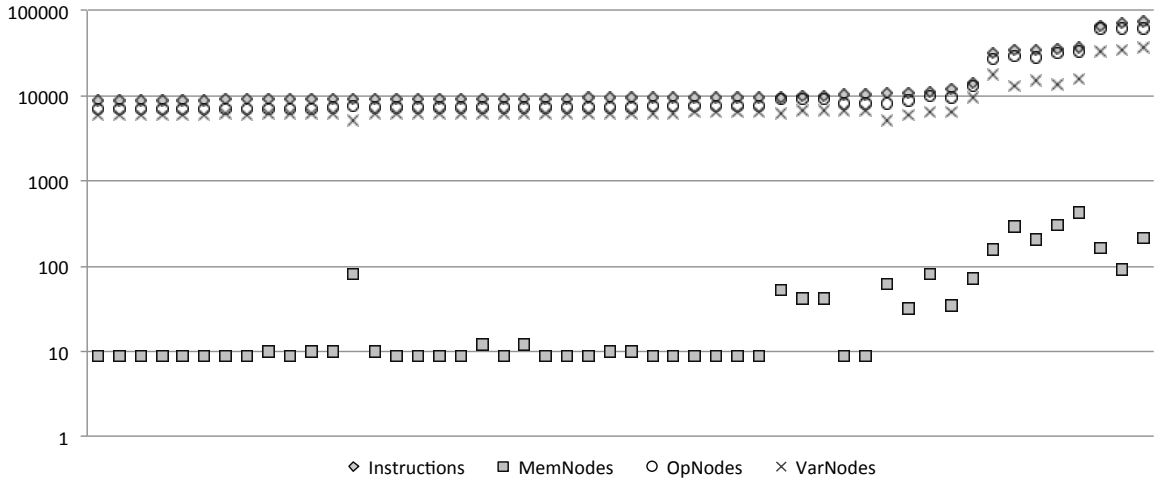


Figura 4.5. Número de vértices de memória no grafo SSA quando usando a análise de ponteiro *Basic* do LLVM.

que representam variáveis (*VarNodes*) e operações (*OpNodes*). O número de instruções de cada teste é denotado por *Instructions*. Observa-se significantemente menos vértices de memória que outros tipos de vértices. Essa discrepância é devida à imprecisão da análise de ponteiros usada pelo LLVM. Quanto menos precisa for a análise de ponteiros, menos vértices de memória haverá no grafo de dependências. O número de arestas de controle e de dados no grafo SSA não decresce significantemente se for desabilitada a análise de ponteiros. Arestas de dados reduzem de 1,774,594 para 1,772,298. Arestas de controle reduzem de 432,540 para 431,748. Como uma consequência do uso de uma análise de ponteiros simples, localizações de memória não são representadas na forma SSA. Assim, FlowTracker é sensível ao fluxo para *variáveis top-level*, e insensível ao fluxo para localizações de memória. No jargão LLVM, uma variável *top-level* é um nome que pode ser representado na forma SSA.

4.7 Avaliação

Esta Seção serve para responder a 3 perguntas de investigação (PI):

- PI1: Quão *efetivo* é FlowTracker para detectar vazamento de informação?
- PI2: Quão *escalável* é o algoritmo da Figura 3.6?
- PI3: Quão *adaptável* é FlowTracker para detectar outros tipos de vulnerabilidades de fluxo de informação?

4.7.1 PI1: Efetividade

A efetividade de FlowTracker foi avaliada em três diferentes formas: por terceiros através de um serviço *on-line*, pela aplicação sobre a biblioteca NaCl versão 20110221, e pela aplicação na biblioteca OpenSSL 1.0.2. FlowTracker foi disponibilizado como um serviço *on-line*[2] que permitiu a interação com usuários externos e avaliação de 12 *benchmarks*, variando de código trivial, até código complexo, tal como combinação de chaves baseada em curva, usada por LibSSH [Bernstein, 2006]. FlowTracker corretamente reportou vazamentos baseados em tempo para todos os exemplos onde era esperado um problema, e não disparou qualquer aviso para os casos onde não era esperado uma vulnerabilidade.

A efetividade de FlowTracker também foi avaliada com implementações populares de criptografia, iniciando com a biblioteca NaCl por seu comportamento de tempo constante. NaCl contém implementações de várias primitivas criptográficas, incluindo funções *hash*, códigos de autenticação de mensagem (MACs), encriptação autenticada, assinaturas digitais e encriptações de chave pública. Além das chaves secretas e públicas, entradas de função *hash* a mensagens em texto puro foram de forma conservadora marcadas como sensíveis. Como esperado, as propriedades isócronas da biblioteca NaCl foram formalmente verificadas e nenhum vulnerabilidade foi encontrada, conforme resultados anteriores [Almeida et al., 2013]. A análise verificou 12 implementações em linguagem C contidas nela, abrangendo 45 funções diferentes e acima de 6,000 linhas de código: HMAC baseado em SHA2, variantes da cifragem Salsa20, autenticador Poly1305, Curve25519 [Bernstein, 2006] e suas combinações.

FlowTracker não emitiu falsos positivos, de acordo com as definições de fluxo de informação e de não-interferência. Porém, falsos positivos ainda podem acontecer devido à semântica do programa que é analisado. Ao aplicar FlowTracker na implementação GLS254 [Oliveira et al., 2014], 4 traços de código foram apontados como vulneráveis, mas uma inspeção manual não revelou qualquer vetor de ataque. Mais precisamente, um laço crítico no código é escrito de seguinte forma: `for (i = 0; t[1]` $\neq$ `0; i++)`. A precisão de `t[1]` é fixada com alta probabilidade devido à um resultado matemático, mas este fato não pode ser determinado automaticamente pela ferramenta. Após os relatórios de FlowTracker, o código foi modificado e então a ferramenta não mais reportou qualquer aviso na nova versão. Isso demonstra a grande utilidade de verificação automática de propriedades criptográficas.

FLowTracker também foi aplicado à várias funções de OpenSSL. Contrariamente à NaCl, nesse caso foi possível identificar vários avisos de vulnerabilidade. Devido

[2] http://cuda.dcc.ufmg.br/flowtracker

às múltiplas interfaces para as mesmas primitivas, a análise foi restrita à operações críticas de segurança requeridas por RSA e Criptografia de Curva Elíptica, nomeadamente exponenciação modular e multiplicação escalar. FlowTracker foi capaz de encontrar centenas de traços vulneráveis na implementação dessas operações. Um exemplo particular de vulnerabilidade potencial foi a multiplicação escalar de Montgomery em uma curva binária (função `ec_GF2m_montgomery_point_multiply()` no arquivo `ec2_mult.c`). Apesar da resistência natural à ataques por canais laterais provida pela implementação segura dessa função, FlowTracker detectou 82 traços vulneráveis nesta instância específica. A suscetibilidade de canal lateral nesse pedaço de código foi recentemente demonstrada por um ataque baseado em tempo de cache *Flush+Reload* [Yarom & Benger, 2014], corroborando o que foi encontrado.

4.7.2 PI2: Escalabilidade

Para demonstrar a eficiência e a escalabilidade de FlowTracker, ela foi aplicada nos programas SPEC CPU 2006 e em outros programas C disponíveis na coleção de testes de LLVM. Note que esses programas não possuem código usado em aplicações criptográficas. Porém, eles são grandes o suficiente para fornecer uma ideia de (i) quanto tempo FlowTracker precisa para construir o grafo de dependências para programas grandes; (ii) quanta memória FlowTracker requer, e (iii) qual a relação entre o tamanho do programa e o tamanho de seu grafo SSA. Todos os número reportados aqui foram obtidos em um Intel Core I7 com 2.20 GHz de *clock*, oito núcleos, e 8GB de memória RAM. Cada núcleo tem uma *cache* de 6,144 KB. Essa máquina executava Linux Ubuntu 12.04.

A Tabela 4.1 mostra o tamanho do grafo SSA que foi produzido para a coleção de programas SPEC CPU 2006. É notável um número 1.5x maior de arestas de dependências de dados que variáveis no grafo SSA. Existem 2.5 variáveis por aresta de dependência de controle. Estes números indicam que o grafo é esparso, isto é, o número

Bench	perl	bzip2	mcf	go	hmmer	sjeng	libqu	h264	omnet	astar	xalanc	gcc
Vars (K)	434	27	3.9	234	108	43	9	234	161	13	1,038	1,249
Controle (K)	296	24	2.1	119	61	26	4.4	124	49	6.3	343	1,112
Dados (K)	607	41	6	332	149	62	15	340	239	20	1,471	1,780
Tempo (sec)	39.9	2.1	0.05	7.6	2.6	1.4	0.1	5.2	6.4	0.2	48.0	227.8

Tabela 4.1. Como FlowTracker escala: a coleção de testes SPEC. ***Vars***: número de variáveis em cada programa (número de vértices no grafo SSA). ***Controle***: número de arestas de dependência de controle inseridas pelo Algoritmo da Figura 3.6. ***Dados***: número de arestas de dependência de dados. ***Tempo***: tempo para construir o grafo SSA.

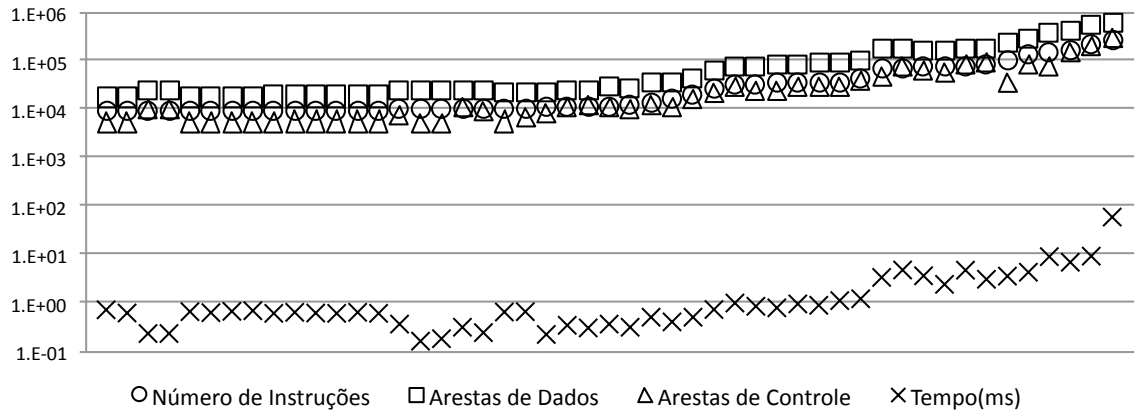


Figura 4.6. Tamanho dos programas (Número de Instruções) vs tamanho do grafo SSA (número de arestas) vs tempo (ms) para construir o grafo SSA.

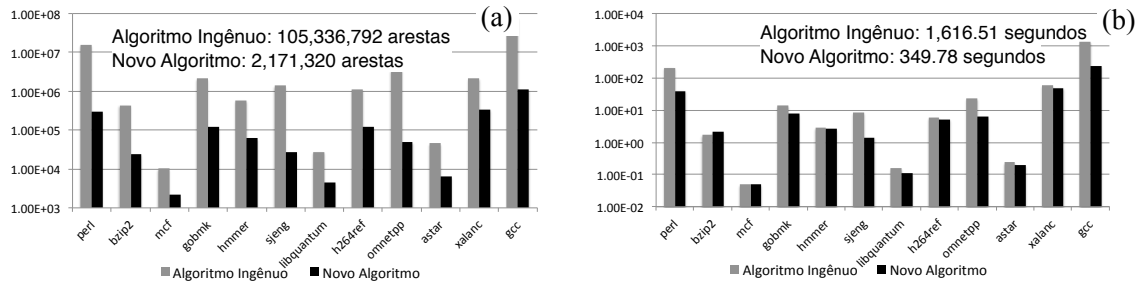


Figura 4.7. (a) Número de arestas inseridas com o algoritmo da Figura 3.6 vs sem a remoção de dependências transitivas (b) Tempo de execução do procedimento de construção do grafo, com e sem a remoção de dependências transitivas.

de arestas é linearmente proporcional ao número de vértices. A exploração da transitividade para remoção de dependências redundantes é essencial para assegurar essa propriedade. FlowTracker também foi aplicado nos 100 maiores programas da coleção de *benchmarks* de LLVM. A Figura 4.6 mostra esses números.

Para demonstrar a importância da remoção das dependências transitivas, foi considerado o comportamento de uma versão ingênua do algoritmo. Esta abordagem ingênua cria uma aresta entre o predicado p e uma variável v sempre que v é controlada por p, não considerando transitividade. A Figura 4.7 compara ambas versões do algoritmo que detecta dependências de controle. A Figura 4.7 (a) mostra que o algoritmo proposto é duas ordens de magnitude mais frugal que essa versão ingênua. Essa lacuna cresce com o tamanho do *benchmark*. Por exemplo, o grafo SSA do menor *benchmark*

SPEC, `mcf`, tem 3,940 vértices. Para esse *benchmark* foram criados 2,133 arestas de controle, enquanto a versão ingênua criou 10,241. O maior *benchmark*, `gcc`, fornece um grafo com 1,249,681 vértices. O algoritmo insere 1,112,889 arestas de controle no grafo, e a versão ingênua insere 78,522,510. Esta diferença no número de arestas de controle tem um impacto direto no tempo de execução do algoritmo, como mostra a Figura 4.7 (b). O algoritmo analisa o menor *benchmark* do conjunto, `mcf` em 0.05 segundos, e o maior, `gcc`, em 227 segundos. A versão ingênua toma 0.05 e 1,297 segundos, respectivamente. No total, o algoritmo leva 341.35 segundos para cobrir todo o SPEC CPU 2006, enquanto a versão ingênua necessita de 1,616.51 segundos.

4.7.3 PI3: Adaptabilidade

A maior motivação da implementação de FlowTracker foi descobrir canais laterais em implementações de algoritmos criptográficos. Porém, durante o projeto da ferramenta, foi percebido que ela é muito mais geral. Ela pode ser usada para implementar diferentes tipos de análises de fluxo de informação. Para fundamentar essa afirmativa, ela foi usada para descobrir três tipos diferentes de vulnerabilidades: vazamento de endereço, estouro de arranjo e estouro de inteiro. Cada uma dessas análises é parametrizada com um conjunto de operações *fonte*, e por um conjunto de funções *sorvedouro*. Deseja-se verificar se um programa contém um caminho de dependências a partir de um fonte para um sorvedouro. Este caminho é procurado via duas formas de atravessar o grafo SSA. Primeiro, uma travessia para frente, marcado cada vértice visitado a partir da fonte. Então uma travessia para trás, iniciando do sorvedouro, marcando todos os vértices alcançáveis neste caminho. A interseção dessas duas travessias indica uma fatia vulnerável do programa.

A Figura 4.8 mostra a porcentagem do grafo SSA dos *benchmarks* SPEC CPU 2006 que é considerada vulnerável. No caso de estouro de arranjo, considerou-se como fonte qualquer função de biblioteca cujo código não esteja disponível para o compilador. Sorvedouros são instruções de armazenamento na memória. Para estouro de inteiro foram consideradas como fonte as operações aritméticas que podem ser "arredondadas", isto é, podem gerar estouro aritmético. Como sorvedouro foram consideradas instruções de carga e armazenamento na memória, pois adversários podem usar estouro de inteiro para habilitar estouro de arranjo. Finalmente, para o problema de vazamento de endereço foi considerada como fonte qualquer operação que lê endereços de memória, e como sorvedouro qualquer função de biblioteca. Descoberta de endereços podem dar ao adversário, meios de contornar um mecanismo de proteção do sistema operacional chamado *Address Space Layout Randomization* (ASLR). Como um exemplo,

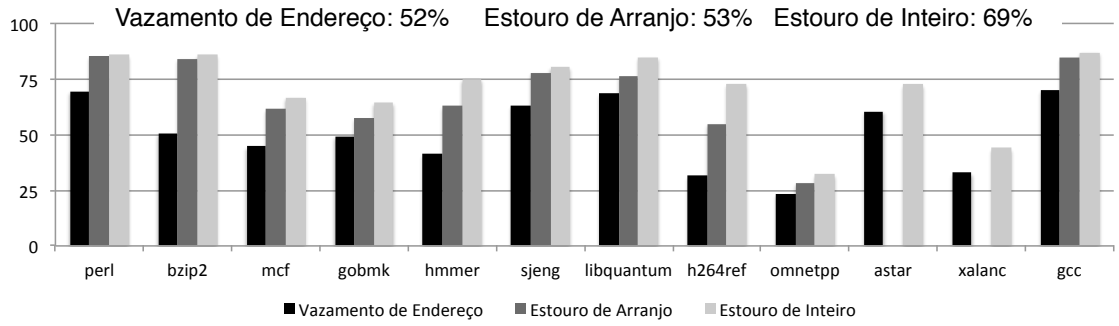


Figura 4.8. Porcentagem de grafo SSA "vulnerável" e que requer guardas para sanitizá-los.

está-se de volta

Dionysus Blazakis explicou como usar informação de endereço para comprometer um interpretador *ActionScript* [Blazakis, 2010].

Como a Figura 4.8 mostra, uma parte substancial de cada grafo SSA foi marcada como vulnerável. Isso é consequência de uma definição liberal de funções fonte. Note que a análise pode conter falsos positivos, isto é, um dado caminho dentro do programa nunca ser tomado dinamicamente. Cada vulnerabilidade demanda diferentes tipos de guarda. Estouros de inteiro podem ser evitados com instrumentação de Dietz *et al.* [Dietz et al., 2012]. Instruções de armazenamento que podem causar estouro de arranjo podem ser guardadas por ferramentas tais como *AddressSanitizer* [Serebryany et al., 2012]. E vazamentos de endereço pode ser prevenidos por uma adaptação da análise dinâmica de fluxo de dados de Chang *et al.* Ainda sobre essa última vulnerabilidade, foram inspecionados manualmente alguns grafos SSA, e encontrado vazamentos perigosos. Por exemplo, as funções `spec_fread` (em `bzip2/spec.c:187`), e `spec_fwrite` (em `bzip2/spec.c:262`) imprimem o endereço de arranjos que eles recebem como parâmetros; assim, eles dão ao adversário total conhecimento da informação secreta.

4.8 Considerações Finais

Este Capítulo apresentou FlowTracker, uma ferramenta de análise estática e esparsa de código C/C++ capaz de informar ao programador, traços de instruções estáticas vulneráveis a estouro de arranjo e de inteiro, vazamento de endereços e canais laterais baseados em tempo. Essa ferramenta foi implementada como um módulo para o compilador LLVM e está disponível para acesso on-line. Na Seção 4.2 foi descrito a possibilidade do compilador induzir canais laterais acidentalmente durante a otimização do

código intermediário. Na Seção 4.3 descreveu-se que o derramamento de registradores e cópia coalescente não interferem na garantia de ausência de falso negativo nos relatórios gerados por FlowTracker. Já na Seção 4.4 foi detalhada uma técnica para redução do número de falsos positivos. A Seções 4.5 e 4.6 descreveram respectivamente como resolver efeitos colaterais durante a análise e como lidar com inter-proceduralidade e propagação de informação pela memória. Finalmente a Seção 4.7 exibe os resultados experimentais de FlowTracker executado sobre a coletânea de testes do LLVM e sobre o conjunto de testes SPEC CPU 2006. FlowTracker foi uma das 70 propostas selecionadas entre 500 submissões de todas as áreas do conhecimento em uma chamada pública da FAPEMIG[3]. Essas 70 propostas foram apresentadas numa feira de tecnologia no ano de 2015 em Belo Horizonte, despertando o interesse da PRODEMGE[4], uma empresa de tecnologia da informação do Governo de Minas Gerais. Isso resultou em uma cooperação entre o Departamento de Ciência da Computação da UFMG e a PRODEMGE a fim de portar FlowTracker para a plataforma Java, fazendo-o capaz de analisar programas escrito em linguagem Java e assim possibilitar a certificação dos programas desenvolvidos pela PRODEMGE quanto à ausência de canais laterais de tempo e possibilidades de estouro de arranjo. Tal cooperação está atualmente em curso.

[3]http://www.fapemig.br/
[4]http://www.prodemge.mg.gov.br

Capítulo 5

Conclusão e Trabalhos Futuros

Este Capítulo apresenta na Seção 5.1 as conclusões deste trabalho de tese. Os trabalhos futuros vislumbrados pelo estudante podem ser encontrados na Seção 5.2.

5.1 Conclusão desta Tese

Este trabalho de Tese focou no rastreamento eficiente do fluxo de informação como forma de detectar vulnerabilidades relacionadas ao fluxo tais como vazamento de informação e fluxo contaminado. Para tanto foi necessário detectar as dependências explícitas relacionadas ao fluxo de dados entre as instruções do programa e as dependência implícitas relacionadas ao fluxo de controle. As dependências explícitas foram facilmente detectadas através de um único passo de compilador. Porém para a detecção do fluxo implícito houve um desafio maior. Inicialmente pensou-se em utilizar a abordagem de Ferrante *et al.* para detectar tais fluxos, entretanto essa abordagem poderia gerar no pior caso um número excessivo de arestas de controle no grafo de dependências, o qual é uma estrutura de dados auxiliar que permite representar ambos fluxos de informação.

Como forma de contornar esse inconveniente da abordagem de Ferrante *et al.*, foi projetado um algoritmo linear sobre o número de definições somado ao uso das variáveis do programa. Tal algoritmo é extremamente simples de ser implementado de forma a ser possível representá-lo com poucas dezenas de linhas de código SML. A grande ideia por trás de tal algoritmo é a utilização de uma estrutura de dados chamada árvore de dominância, já presente na maioria dos compiladores industriais. O texto desta Tese comprova que o algoritmo funciona inclusive para programas não estruturados e está correto, por meio de um conjunto de teoremas e suas respectivas provas.

Além do novo algoritmo, esta tese também contribui para o estado da arte de análise de fluxo, por meio de uma implementação esparsa do sistema de Tipos de Hunt e Sands. Essa implementação consegue ser esparsa por utilizar uma forma de representação de programas bastante conhecida e usada por compiladores modernos, incluindo GCC e LLVM, chamada forma SSA. Nessa forma de programas, cada variável pode ser definida apenas 1 vez no texto do programa. Isso permite associar facilmente informação à cada variável ao invés de associá-la à outros pontos que tornariam a análise densa, tal como o espaço entre uma instrução e outra.

O Sistema de tipos de Hunt e Sands implementado de forma esparsa foi utilizado nesta Tese para detectar 4 tipos de vulnerabilidades: vazamento de endereços, estouro de arranjo e de inteiro e canais laterais baseados em tempo. Isso foi possível com a implementação de uma ferramenta chamada FlowTracker que usa tal sistema de tipos e está disponível para *download* ou uso *on-line*, sendo capaz de detectar vulnerabilidades de canais laterais e reportá-las ao programador em tempo de compilação. FlowTracker se mostrou bastante escalável, sendo capaz de rastrear todo fluxo de informação e detectar vulnerabilidades em programas realmente grandes tais como compilador GCC que possui mais de 600K linhas de código fonte em poucos segundos. FlowTracker também se mostrou efetivo, pois foi capaz de certificar uma importante biblioteca criptográfica chamada NaCl, provando suas propriedades isócronas. Por outro lado FlowTracker reportou diversos problemas de segurança na conhecida biblioteca OpenSSL. Finalmente FlowTracker se mostrou adaptável, sendo facilmente customizado para rastrear as demais vulnerabilidades relacionadas ao fluxo.

Por meio destes resultados experimentais, conclui-se que analisadores estáticos são poderosas ferramentas que podem ser usadas para o rastreamento consistente (sem falsos negativos) de diversas vulnerabilidades. Também conclui-se que analisadores estáticos esparsos são mais eficientes do ponto de vista de consumo de memória e tempo de execução do que analisadores densos e podem desde agora ser implementados utilizando um novo algoritmo linear para o rastreamento dos fluxos implícitos de informação construído neste trabalho de tese, ao invés da abordagem clássica de Ferrante *et al.*, que embora correta, produz mais informação do que o necessário para tais analisadores e portando sofre de um alto custo de pior caso ($O(|I| \times |E|)$).

5.2 Trabalhos Futuros

Como trabalho futuro, deseja-se modificar os traços que contenham canais laterais de forma a interromper os caminhos que conectam informação sigilosa a predicados de

instruções de desvio, bem como à indexação de memória. Isso, obviamente deve ser feito mantendo a semântica do programa e com baixo *overhead*. Duas abordagens para tal feito poderiam ser experimentadas tais como:

1. Incluir instruções inócuas nas ramificações dos desvios de forma a fazê-los executar pelo mesmo tempo. Isso resolveria parte do problema, mas canais laterais devido ao comportamento de memória ainda continuariam a existir.

2. Substituir instruções de desvio dependentes de informação sigilosa e suas ramificações, por sequências de instruções que realizam as mesmas ações de ambas ramificações do desvio, prevalecendo apenas os resultados condizentes com a avaliação desvio na versão original do código fonte. Essa abordagem poderia imputar um alto *overhead*, mas é uma alternativa.

Um segundo trabalho futuro diz respeito a encontrar vulnerabilidade de canal lateral em programas importantes que estejam em uso atualmente e reportar tais problemas aos seus desenvolvedores, como forma de promover FlowTracker na comunidade internacional.

Um terceiro trabalho futuro, está sendo realizado no momento pela equipe da PRODEMGE e diz respeito à permitir que FlowTracker encontre canais laterais em aplicações Java. Para tanto, FlowTracker está sendo portado da plataforma LLVM para a plataforma SOOT[1] que é um analisador de código Java publicamente disponível para uso.

[1] http://sable.github.io/soot/

Referências Bibliográficas

Agat, J. (2000). Transforming out timing leaks. Em *POPL*, pp. 40--53. ACM.

Almeida, J. B.; Barbosa, M.; Pinto, J. S. & Vieira, B. (2013). Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 78(7):796--812.

Appel, A. W. & Palsberg, J. (2002). *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edição.

Bernstein, D. J. (2005). Cache-timing attacks on AES. Relatório técnico, University of Illinois at Chicago.

Bernstein, D. J. (2006). Curve25519: new Diffie-Hellman speed records. Em *PKC*, pp. 207--228. Springer.

Bhatkar, E.; Duvarney, D. C. & Sekar, R. (2003). Address obfuscation: an efficient approach to combat a broad range of memory error exploits. Em *USENIX Security*, pp. 105--120.

Blazakis, D. (2010). Interpreter exploitation. Em *WOOT*, pp. 1--9. USENIX.

Bodik, R.; Gupta, R. & Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. Em *PLDI*, pp. 321–333. ACM.

Boissinot, B.; Hack, S.; Grund, D.; de Dinechin, B. D. & Rastello, F. (2008). Fast liveness checking for SSA-form programs. Em *CGO*, pp. 35–44. IEEE.

Brumley, B. B. & Tuveri, N. (2011). Remote timing attacks are still practical. Em *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS'11, pp. 355--371, Berlin, Heidelberg. Springer-Verlag.

Brumley, D. & Boneh, D. (2005). Remote timing attacks are practical. *Computer Networks*, 48(5):701--716.

Buchanan, E.; Roemer, R.; Shacham, H. & Savage, S. (2008). When good instructions go bad: generalizing return-oriented programming to RISC. Em *CCS*, pp. 27--38. ACM.

Chen, Y.-F.; Hsu, C.-H.; Lin, H.-H.; Schwabe, P.; Tsai, M.-H.; Wang, B.-Y.; Yang, B.-Y. & Yang, S.-Y. (2014). Verifying Curve25519 software. Em *Proceedings of CCS*, pp. 299--309. ACM.

Choi, J.-D.; Cytron, R. & Ferrante, J. (1991). Automatic construction of sparse data flow evaluation graphs. Em *POPL*, pp. 55–66. ACM.

Cowan, C.; Pu, C.; Maier, D.; Hintony, H.; Walpole, J.; Bakke, P.; Beattie, S.; Grier, A.; Wagle, P. & Zhang, Q. (1998). Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. Em *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pp. 5--5, Berkeley, CA, USA. USENIX Association.

Cytron, R.; Ferrante, J. & Sarkar, V. (1990). Compact representations for control dependence. Em *PLDI*, pp. 337--351. ACM.

Denning, D. E. & Denning, P. J. (1977). Certification of programs for secure information flow. *Commun. ACM*, 20:504--513.

Dhem, J.-F.; Koeune, F.; Leroux, P.-A.; Mestre, P.; Quisquater, J.-J. & Willems, J.-L. (2000). A practical implementation of the timing attack. Em *Smart Card Research and Applications*, volume 1820 of *Lecture Notes in Computer Science*, pp. 167–182. Springer.

Dietz, W.; Li, P.; Regehr, J. & Adve, V. (2012). Understanding integer overflow in c/c++. Em *ICSE*, pp. 760--770. IEEE.

Dowson, M. (1997). The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84--.

Ferrante, J.; Ottenstein, K. J. & Warren, J. D. (1987). The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319--349.

Genkin, D.; Shamir, A. & Tromer, E. (2014). RSA key extraction via low-bandwidth acoustic cryptanalysis. Em *CRYPTO*, pp. 444--461. Springer.

Grosser, T. C. (2011). *Enabling Polyhedral Optimizations in LLVM*. Tese de doutorado, Universitat Passau.

Hammer, C.; Krinke, J. & Nodes, F. (2006). Intransitive noninterference in dependence graphs. Em *ISOLA*, pp. 119--128. IEEE.

Hammer, C. & Snelting, G. (2009). Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399--422.

Horwitz, S.; Prins, J. & Reps, T. (1988). On the adequacy of program dependence graphs for representing programs. Em *POPL*, pp. 146--157. ACM.

Huang, Y.-W.; Yu, F.; Hang, C.; Tsai, C.-H.; Lee, D. T. & Kuo, S.-Y. (2004). Securing web application code by static analysis and runtime protection. Em *WWW*, pp. 40--51.

Hunt, S. & Sands, D. (2006). On flow-sensitive security types. Em *POPL*, pp. 79--90. ACM.

Johnson, R.; Pearson, D. & Pingali, K. (1994). The program tree structure. Em *PLDI*, pp. 171--185. ACM.

Johnson, R. & Pingali, K. (1993). Dependence-based program analysis. Em *PLDI*, pp. 78–89. ACM.

Jovanovic, N.; Kruegel, C. & Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). Em *Symposium on Security and Privacy*, pp. 258--263. IEEE.

Kocher, P. C. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. Em *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pp. 104--113, London, UK, UK. Springer-Verlag.

Kocher, P. C.; Jaffe, J. & Jun, B. (1999). Differential power analysis. Em *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pp. 388--397, London, UK, UK. Springer-Verlag.

Korel, B. (1988). Dynamic program slicing. Em *Information Processing Letters*.

Korel, B. & Rilling, J. (1998). Dynamic program slicing methods. *Information and Software Technology*.

Lattner, C. & Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. Em *CGO*, pp. 75–88. IEEE.

Lux, A. & Starostin, A. (2011). A tool for static detection of timing channels in Java. *Journal of Cryptographic Engineering*, 1(4):303–313. ISSN 2190-8508.

Maffra, I. K. T.; Pereira, F. M. Q. & Oliveira, L. B. (2013). Detecção automática de vulnerabilidades em código protegido por canários. Em *SBSeg*, pp. 184 --197.

Molnar, D.; Piotrowski, M.; Schultz, D. & Wagner, D. (2006). The program counter security model: Automatic detection and removal of control-flow side channel attacks. Em *Proceedings of ICISC*, pp. 156--168, Berlin, Heidelberg. Springer.

Moore, D.; Shannon, C. & claffy, k. (2002). Code-Red: A case study on the spread and victims of an internet worm. Em *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, pp. 273--284, New York, NY, USA. ACM.

Nielson, H. R. & Nielson, F. (1992). *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc. ISBN 0-471-92980-8.

Oh, H.; Heo, K.; Lee, W.; Lee, W. & Yi, K. (2012). Design and implementation of sparse global analyses for c-like languages. Em *PLDI*, pp. 1–11. ACM.

Oliveira, T.; López, J.; Aranha, D. F. & Rodríguez-Henríquez, F. (2014). Two is the fastest prime: lambda coordinates for binary elliptic curves. *J. Cryptographic Engineering*, 4(1):3--17.

Orbæk, P. & Palsberg, J. (1997). Trust in the λ-calculus. *J. Funct. Program.*, 7(6):557--591.

Ottenstein, K. J.; Ballance, R. A. & MacCabe, A. B. (1990). The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. Em *PLDI*, pp. 257--271. ACM.

Pereira, F. M. Q. & Palsberg, J. (2009). SSA elimination after register allocation. Em *CC*, pp. 158 -- 173.

Pingali, K. & Bilardi, G. (1997). Optimal control dependence computation and the roman chariots problem. Em *TOPLAS*, pp. 462--491. ACM.

Prouff, E. (2013). Side channel attacks against block ciphers implementations and countermeasures. Tutorial presented in CHES.

Quadros, G. S. & Pereira, F. M. Q. (2011). Static detection of address leaks. Em *SBSeg*, pp. 23--37.

Quadros, G. S. & Pereira, F. M. Q. (2012). A static analysis tool to detect address leaks. Em *CBSoft – Tools*.

Quadros, G. S.; Souza, R. M. & Pereira, F. M. Q. (2012). Dynamic detection of address leaks. Em *SBSeg*, pp. 61--75.

Quisquater, J.-J. & Samyde, D. (2001). Electromagnetic analysis (ema): Measures and counter-measures for smart cards. Em *Smart Card Programming and Security*, pp. 200--210. Springer.

Ranganath, V. P.; Amtoft, T.; Banerjee, A.; Hatcliff, J. & Dwyer, M. B. (2007). A new foundation for control dependence and slicing for modern program structures. *TOPLAS*, 29(5).

Rastello, F. (2015). *SSA-based Compiler Design*. Springer, 1st edição.

Reps, T. & ang, W. (1988). The semantics of program slicing. Relatório técnico, University of Wisconsin – Madison.

Rimsa, A.; d'Amorim, M.; Pereira, F. M. Q. & da Silva Bigonha, R. (2014). Efficient static checker for tainted variable attacks. *Sci. Comput. Program.*, 80:91–105.

Rimsa, A.; d'Amorim, M. & Pereira, F. M. Q. a. (2011). Tainted flow analysis on e-SSA-form programs. Em *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pp. 124--143, Berlin, Heidelberg.

Rodrigues, B. (2014a). Detecção estática e consistente de potenciais estouros de arranjos. Em *Anais do XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*.

Rodrigues, B. (2014b). Um algoritmo linear para a construção de program slices. Em *Anais do XVIII Simpósio Brasileiro de Linguagens de Programação (SBLP 2014)*.

Rodrigues, B.; Aranha, D. & Pereira, F. M. Q. (2015a). Uma tecnica de análise estática para detecção de canais laterais baseados em tempo. Em *Anais do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*.

Rodrigues, B.; Aranha, D. & Pereira, F. M. Q. (2016). Sparse representation of implicit flows with applications to side-channel detection. Em *Procedings of 25th International Conference on Compiler Construction (CC 2016)*.

Rodrigues, B.; Pereira, F. M. Q. & Oliveira, L. B. (2013a). Uma representação intermendiária para a detecção de vazamentos implícitos de informação. Em *Anais do XIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pp. 212--225.

Rodrigues, B.; Pereira, F. M. Q.; Oliveira, L. B. & Loureiro, A. A. F. (2013b). Flow tracking: Uma ferramenta para detecção de vazamento de informações sigilosas. Em *Anais do CBSoft 2013*, Brasilia, DF, Brasil.

Rodrigues, B.; Ribeiro, L. R. & Pereira, F. M. Q. (2015b). FlowTracker - detecção de código não isócrono via análise estática de fluxo. Em *Anais do CBSoft 2015*.

Russo, A. & Sabelfeld, A. (2010). Dynamic vs. static flow-sensitive security analysis. Em *CSF*, pp. 186--199. IEEE Computer Society.

Santos, Henrique Nazaré, P. F. M. Q. O. L. B. (2013). Verificacao estática de acessos a arranjos em C. Em *Anais do XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pp. 198--211.

Scott, D. & Sharp, R. (2003). Specifying and enforcing application-level web security policies. *Trans. on Knowl. and Data Eng.*, 15:771--783.

Serebryany, K.; Bruening, D.; Potapenko, A. & Vyukov, D. (2012). Addresssanitizer: a fast address sanity checker. Em *USENIX*, pp. 28--28. USENIX Association.

Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). Em *CCS*, pp. 552--561. ACM.

Shacham, H.; Page, M.; Pfaff, B.; Goh, E.-J.; Modadugu, N. & Boneh, D. (2004). On the effectiveness of address-space randomization. Em *CSS*, pp. 298--307. ACM.

Snelting, G. (1996). Combining slicing and constraint solving for validation of measurement software. Em *SAS*, pp. 332--348. Springer.

Sreedhar, V. C.; ching Ju, R. D.; Gillies, D. M. & Santhanam, V. (1999). Translating out of static single assignment form. Em *SAS*, pp. 194–210. Springer-Verlag.

Sreedhar, V. C. & Gao, G. R. (1995). A linear time algorithm for placing ϕ-nodes. Em *POPL*, pp. 62--73. ACM.

Taghdiri, M.; Snelting, G. & Sinz, C. (2011). Information flow analysis via path condition refinement. Em *FAST*, pp. 65--79. Springer.

Tavares, A. L. C.; Boissinot, B.; Pereira, F. M. Q. & Rastello, F. (2014). Parameterized construction of program representations for sparse dataflow analyses. Em *Compiler Construction*, pp. 2--21. Springer.

Wassermann, G. & Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. Em *PLDI*, pp. 32--41. ACM.

Wasserrab, D.; Lohner, D. & Snelting, G. (2009). On PDG-based noninterference and its modular proof. Em *PLAS*, pp. 31--44. ACM.

Weiser, M. (1981). Program slicing. Em *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pp. 439--449, Piscataway, NJ, USA. IEEE Press.

Weiser, M. (1982). Programmers use slices when debugging. *Commun. ACM*, 25(7):446--452.

Weiser, M. D. (1979). *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. Tese de doutorado, Ann Arbor, MI, USA. AAI8007856.

Xie, Y. & Aiken, A. (2006). Static detection of security vulnerabilities in scripting languages. Em *USENIX-SS*. USENIX Association.

Yarom, Y. & Benger, N. (2014). Recovering openssl ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140. `http://eprint.iacr.org/`.

Zhang, R.; Huang, S.; Qi, Z. & Guan, H. (2011). Combining static and dynamic analysis to discover software vulnerabilities. Em *IMIS*, pp. 175--181. IEEE Computer Society.

Zhao, J.; Nagarakatte, S.; Martin, M. M. K. & Zdancewic, S. (2013). Formal verification of SSA-based optimizations for LLVM. Em *PLDI*, pp. 175–186. ACM.