DETECÇÃO ESTÁTICA E DINÂMICA DA VULNERABILIDADE DE VAZAMENTO DE ENDEREÇOS

GABRIEL QUADROS SILVA

DETECÇÃO ESTÁTICA E DINÂMICA DA VULNERABILIDADE DE VAZAMENTO DE ENDEREÇOS

Dissertação 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 Mestre em Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira
Co-orientador: Mariza Andrade da Silva Bigonha

Belo Horizonte

Março de 2013

Quadros Silva, Gabriel
M1234x Detecção Estática e Dinâmica da Vulnerabilidade de Vazamento de Endereços / Gabriel Quadros Silva. — Belo Horizonte, 2013
xx, 46 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Fernando Magno Quintão PereiraCo-orientador: Mariza Andrade da Silva Bigonha

1. Vazamento de endereços. 2. Análise de fluxo de informação. 3. Semântica. 4. Sistema de tipos. 5. Análise dinâmica. 6. Análise estática. I. Título.

CDU 100.0*01.10

Agradecimentos

Agradeço à minha família, ao meu orientador Fernando, à minha coorientadora Mariza e aos meus colegas de laboratório Leonardo e Rafael que contribuíram para a realização desse trabalho, e, demais colegas.

Agradeço ao CNPq pela bolsa de estudos para conclusão do mestrado.

"Software have bugs. That is quite a known fact."
(Tyler Durden - Phrack 64-8)

Resumo

Uma vulnerabilidade de vazamento de endereço permite a um atacante descobrir onde um programa está carregado na memória. Apesar de parecer inofensiva, essa informação dá ao atacante meios para burlar dois mecanismos de proteção muito difundidos: *Address Space Layout Randomization* (ASLR) e *Data Execution Prevention* (DEP). Nessa dissertação de mestrado é mostrado, por meio de um exemplo, como explorar um vazamento de endereços para tomar o controle de um servidor remoto rodando em um sistema operacional protegido por ASLR e DEP. Em seguida, é apresentado um *framework* de instrumentação de código que usa um monitor dinâmico para prevenir o vazamento de endereços em tempo de execução. Finalmente, uma análise estática é usada para provar que partes do programa não precisam ser instrumentadas; consequentemente, reduzindo a sobrecarga de instrumentação. A implementação desse trabalho foi testada com um pacote de testes contendo 434 programas, que incluem o SPEC CPU 2006, e contém aproximadamente 2 milhões de linhas de código C. Foram explorados efetivamente 11 dos 19 avisos produzidos pela análise de fluxo. A combinação das análises estática e dinâmica fornece um modo prático para proteger softwares de vazamentos de endereços. Um programa completamente instrumentado é, em média, quase 500% mais lento que o programa original. Entretanto, a análise estática tem sido capaz de reduzir essa sobrecarga para menos que 3% em média.

Palavras-chave: Vazamento de endereços, Análise de fluxo de informação, Semântica, Sistema de tipos, Análise dinâmica, Análise estática.

Abstract

An address disclosure vulnerability allows an adversary to discover where a program is loaded in memory. Although seemingly harmless, this information gives the adversary the means to circumvent two widespread protection mechanisms: Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP). In this dissertation we show, via an example, how to explore an address leak to take control of a remote server running on an operating system protected by ASLR and DEP. We then present a code instrumentation framework that uses a dynamic monitor to prevent address leaks at runtime. Finally, we use a static analysis to prove that parts of the program do not need to be instrumented; hence, reducing the instrumentation overhead. We have tested our implementation on a benchmark suite with 434 programs, that includes SPEC CPU 2006, and gives us approximately 2 million lines of C. We have been able to effectively exploit 11, out of 19 warnings produced by our flow analysis. The combination of static and dynamic analyses provide us with a practical way to secure software against address disclosures. A fully instrumented program is, on average, almost 500% slower than the original benchmark. However, the static analysis has been able to reduce this overhead to less than 3% on average.

Keywords: Address disclosure, Information flow analysis; Semantics, Type system, Dynamic analysis, Static analysis.

Lista de Figuras

2.1 (a) Um programa que contém uma vulnerabilidade de *buffer overflow*. (b) Um exemplo esquemático de um *buffer overflow* na pilha. O endereço de retorno da `função` é desviado por uma entrada maliciosamente preparada para outro procedimento. 6
2.2 Um servidor de eco implementado em C. 8
2.3 Um programa em Python que explora os vazamentos de endereços no programa da Figura 2.2. 9

3.1 A sintaxe de *Angels*. 16
3.2 Exemplos da sintaxe de C mapeada para instruções de *Angels*. 17
3.3 A Semântica Operacional de *Angels*. 18
3.4 Um programa C traduzido para *Angels*. 19
3.5 A Semântica Operacional das instruções instrumentadas. 20
3.6 A instrumentação completa de um programa *Angels*. 21
3.7 Restrições que definem a análise de ponteiros. 24
3.8 Semântica Abstrata dos Operadores *meet* usados na *Engine* de Inferência de Tipos *Forward* e *Backward*. 25
3.9 A análise de inferência de tipos *forward*. 25
3.10 A análise de inferência de tipos *backward*. 26
3.11 A relação γ que instrumenta parcialmente programas *Angels*. 27
3.12 (a) Análise *Forward* aplicada no programa visto na Figura 3.4(a). (b) Análise *Backward*. (c) Programa parcialmente instrumentado. 28
3.13 Este programa C pode vazar informação. A análise estática não detecta essa vulnerabilidade, devido a semântica mal definida de acessos de fora dos limites de arranjos em C. 29
3.14 Na função `len`, a variável `w` tem dependência de controle na variável `s`, que carrega informação de endereço. A função `print_adr` contém um vazamento de endereço, mas a instrumentação não o detecta. 30

4.1 (Superior) Aumento no tamanho devido a instrumentação parcial. (Inferior) Aumento no tamanho devido a instrumentação completa. Cada ponto corresponde a um *benchmark*. Tamanhos são dados em número de instruções bytecode do LLVM. 34
4.2 (Superior) Tempo de execução dos programas parcialmente instrumentados (sec). (Inferior) Tempo de execução dos programas completamente instrumentados. Cada ponto corresponde a um *benchmark*. 36
4.3 Resultados estáticos para os programas no SPEC CPU 2006. P: número de *sinks*, isto é, `printf`'s no programa-alvo. I: número de instruções x86 produzidas para o programa (cerca de 1.7x maior que o número de bytecodes no programa). G: tamanho do grafo de dependência. $\gamma \rightarrow$: número de nós marcados como *maybe* pela análise *forward*. 37
4.4 (Superior) Tempo (sec) para executar a análise de fluxo contaminado (TA), e a análise de ponteiros (PA), comparado ao tamanho dos programas, dado em instruções bytecode do LLVM. Cada ponto no eixo x representa um *benchmark*. Foram considerados os 100 maiores *benchmarks*. (Inferior) Tempo da análise de ponteiros comparado ao tempo da análise de fluxo contaminado para os programas no SPEC CPU 2006. Números representam o tempo gasto pelas análises de ponteiros e fluxo, em segundos. 38

Lista de Tabelas

Sumário

Agradecimentos **vii**

Resumo **xi**

Abstract **xiii**

Lista de Figuras **xv**

Lista de Tabelas **xvii**

1 Introdução **1**

1.1 Contribuições da Dissertação 1

1.2 Organização da Dissertação 4

2 Revisão da Literatura **5**

2.1 Uma Breve Revisão sobre Exploração de Software 5

2.1.1 Um Exemplo de Vazamento de Endereços 8

2.2 Trabalhos Relacionados 10

2.2.1 Fluxo de Informação via Análises Dinâmicas 11

2.2.2 Fluxo de Informação via Análises Estáticas 12

2.2.3 A Combinação de Análises Estática e Dinâmica 12

2.3 Conclusão 13

3 Detecção Estática e Dinâmica de Vazamento de Endereços **15**

3.1 *Angels*: a Linguagem Protótipo 15

3.1.1 A Linguagem de Instrumentação 19

3.2 Reduzindo a Sobrecarga de Instrumentação via Análise Estática 23

3.2.1 Análise de Ponteiros 23

3.2.2 Um Sistema de Tipos para Detectar Estaticamente Vazamentos de Endereços 24

3.2.3 Instrumentação Parcial 26
3.2.4 Corretude da Instrumentação Parcial 27
3.2.5 Quando a Instrumentação Completa faz mais que a Análise Estática 29
3.2.6 Sensibilidade ao Controle 30
3.3 Conclusão 31

4 Experimentos Realizados **33**
4.1 O Impacto da Instrumentação no Tamanho do Código 33
4.2 O Impacto da Instrumentação no Tempo de Execução 35
4.3 Tempo de Execução das Análises Estáticas 35
4.4 A Precisão da Análise Estática 37
4.5 Conclusão 39

5 Conclusão e Trabalhos Futuros **41**
5.1 Trabalhos Futuros 41
5.2 Reprodutibilidade 42
5.3 Contribuições 42

Referências Bibliográficas **43**

Capítulo 1

Introdução

Sistemas operacionais modernos usam um mecanismo de proteção chamado *Address Space Layout Randomization* (ASLR) Bhatkar et al. [2003]; Shacham et al. [2004]. Essa técnica consiste em carregar os módulos binários que formam um programa executável em endereços diferentes a cada vez que o programa é executado. Essa medida de segurança protege o software de ataques bem conhecidos, como *return-to-libc* Shacham et al. [2004] e *return-oriented-programming* (ROP) Buchanan et al. [2008]; Shacham [2007]. Como é efetivo, e fácil de implementar, o ASLR está presente em virtualmente todo sistema operacional contemporâneo. Entretanto, esse tipo de proteção não é infalível.

Shacham et al. [2004] mostraram que métodos de ofuscação de endereços são suscetíveis a ataques de força bruta; não obstante, a ofuscação de endereços retarda a taxa de propagação de *worms* que dependem de vulnerabilidades de corrupção de memória substancialmente. Entretanto, um atacante pode ainda executar um ataque cirúrgico em um programa protegido contra ASLR. Nas palavras dos projetistas originais da técnica [Bhatkar et al., 2003, p.115], se "the program has a bug which allows an attacker to read the memory contents", então "the attacker can craft an attack that succeeds deterministically". O objetivo desse trabalho é descobrir e prevenir esse tipo de vulnerabilidade.

1.1 Contribuições da Dissertação

As contribuições dessa dissertação são:

1. **Descrição da vulnerabilidade de vazamento de endereços**

A primeira contribuição desse trabalho é a descrição da vulnerabilidade de vazamento de endereços. Embora esse problema tenha sido continuamente discutido em listas de e-mail e blogs por *experts* e entusiastas em segurança de software, até agora ele não tem sido atacado sob uma perspectiva acadêmica. Vazamentos de endereços são descritos informalmente na Seção 2.1, e é mostrado um exemplo real de *exploit* que depende do vazamento de endereços para burlar os mecanismos de proteção usados pelos sistemas operacionais modernos.

2. **Formalização do Problema**

   A formalização do problema é feita usando uma linguagem de programação que não apenas define vazamento de endereços, mas também embasa as técnicas que foram usadas para lidar com esse problema.

3. **Combinação das técnicas de análises estática e dinâmica para proteger programas contra vazamentos de endereços**

   As técnicas que foram usadas nesse trabalho, instrumentação e análise de fluxo de informação, não são novas; entretanto, esse é o primeiro trabalho a usá-las para prevenir vazamentos de endereços. Além disso, alguns aspectos da análise de fluxo desenvolvida, como sua divisão em um componentes *forward* e *backward*, e o uso pesado de análise de ponteiros, parecem ser únicos à esse trabalho.

4. **Projeto e implementação de um *framework* de instrumentação que converte automaticamente programas em um software que não pode conter vazamento de endereços**

   Em relação a análise dinâmica, foi projetado um *framework* de instrumentação que converte automaticamente um programa em um software que não pode conter vazamentos de endereços. Essa transformação consiste em instrumentar toda operação do programa que propaga dados, seja ela nos registradores ou na memória. Dessa forma, pode-se saber quais informações podem dar o conhecimento de um endereço para um atacante, e quais não. Se informação prejudicial atingir um ponto de saída que o atacante possa ler, o programa interrompe a execução. Assim, executando o programa instrumentado ao invés do programa original, o usuário torna muito mais difícil para o atacante executar *exploits* que requeiram informação de endereços para serem bem sucedidos.

5. **Desenvolvimento de análise de fluxo de informação que prova que partes do código não precisam ser instrumentadas**

Não é de surpreender que a instrumentação impõe uma sobrecarga pesada no tempo de execução do programa-alvo: o programa sanitizado pode ser tanto quanto 10 vezes mais lento que o código original. Para mitigar essa sobrecarga, foi desenvolvido uma análise de fluxo de informação que prova que partes do código não precisam ser instrumentadas.

6. **Incorporação do *framework* de instrumentação e da análise estática associada no compilador LLVM**

   Foi implementado um *framework* de instrumentação, e a análise estática associada no compilador LLVM Lattner & Adve [2004]. Assim, ao contrário de ferramentas de instrumentação binária Nethercote & Seward [2007]; Xu et al. [1999], atualmente os programas só podem ser instrumentados se houver acesso à seus códigos-fonte. Foi escolhido permanecer no nível de representação intermediária porque essa decisão torna mais fácil combinar a biblioteca de instrumentação com a análise estática. O preço a se pagar por essa escolha vem na forma de alertas falso-positivos que os programas instrumentados reportam dinamicamente: a informação que vem de bibliotecas desconhecidas é sempre marcada como perigosa. Pode-se reduzir os falso-positivos por meio da adição de funções conhecidamente inofensivas à uma *whitelist*[1]. Como uma prova de conceito, foi escolhido procurar por informações de endereço que escapam pela função `printf` da `libc`. Outras funções sensíveis podem ser facilmente adicionadas ao nosso *framework*. Como mostrado no Capítulo 4, a ferramenta desenvolvida foi usada para analisar uma suíte de testes que contém quase 2 milhões de linhas de código, e que inclui o SPEC CPU 2006. Programas instrumentados podem ser de 2% a 1,070% mais lentos que os programas originais, com uma média de *slowdown* de 76.88%. A análise estática contribui substancialmente para diminuir essa sobrecarga. Por apenas instrumentar as operações que a análise estática não foi capaz de provar serem seguras, obteve-se *slowdowns* variando de 0% até 22%, com uma média de 1.71%. Essa sobrecarga é baixa o bastante para justificar o uso do monitor dinâmico em código em produção. A análise estática também reportou 19 alertas na suíte de testes. A inspeção manual desses alertas mostrou que 11 deles eram vazamentos de endereços reais. Nesse capítulo também são descritos os trabalhos relacionados a pesquisa proposta nessa dissertação.

[1] Uma *whitelist* é uma lista de entidades aprovadas para acesso autorizado ou receber algum privilégio. No contexto desse trabalho, é uma lista de funções conhecidamente inofensivas que podem receber informações de endereços sem comprometer a segurança da aplicação.

1.2 Organização da Dissertação

O restante desse texto está organizado da seguinte forma: o Capítulo 2 apresenta uma breve revisão sobre exploração de software e apresenta a vulnerabilidade de vazamento de endereços, incluindo um exemplo real de programa vulnerável. Além disso, são discutidos os trabalhos relacionados.

O Capítulo 3 descreve a solução proposta nessa dissertação para detectar vazamentos de endereços em tempo de execução, os experimentos realizados e os resultados obtidos. Inicialmente é mostrado como combinar as análises estática e dinâmica para proteger programas contra vazamentos de endereços. A Seção 3.2 mostra que a análise estática aponta, de forma conservativa, quais instruções tem dependência de dados de informações de endereços. As operações inócuas que a análise estática identifica não são instrumentadas. A análise estática incorpora funcionalidades do estado-da-arte para alcançar boa precisão: ela é sensível a campos e fluxo, é inter-procedural, é sensível a objetos, ou seja, ela distingue um objeto de outro do mesmo tipo Hammer & Snelting [2009], ela modela a *heap* via análise de ponteiros, e alcança uma forma limitada de sensibilidade ao contexto via *function inlining*.

O Capítulo 4 apresenta os experimentos realizados para validar o trabalho desenvolvido, bem como os resultados obtidos. O Capítulo 5 apresenta a conclusão desse texto e os trabalhos futuros. Além disso, são apresentadas as contribuições desse trabalho na forma de artigos publicados em conferências da área.

Capítulo 2

Revisão da Literatura

Este capítulo apresenta uma revisão da literatura relacionada ao tema dessa dissertação. Especificamente, a Seção 2.1 apresenta uma breve revisão sobre exploração de software e a vulnerabilidade de vazamento de endereços, incluindo um exemplo real. A Seção 2.2 discute os trabalhos relacionados. Por fim, a Seção 2.3 conclui o capítulo.

2.1 Uma Breve Revisão sobre Exploração de Software

Um *buffer*, também chamado de arranjo ou vetor, é uma sequência de elementos armazenados na memória. Algumas linguagens de programação, como Java, Python e JavaScript são *fortemente tipadas*, o que significa que elas só permitem combinações de operações e operandos que preservem a declaração de tipo desses operandos. Como um exemplo, todas essas linguagens fornecem arranjos como estruturas de dados embutidas, e elas verificam se os índices estão dentro dos limites declarados desses arranjos. Existem outras linguagens, como C ou C++, que são *fracamente tipadas*. Elas permitem o uso de variáveis de formas não previstas pela declaração de tipo original dessas variáveis. C e C++ não verificam os limites de arranjos, por exemplo. Assim, pode-se declarar um arranjo com n células em qualquer uma dessas linguagens, e então ler a célula na posição $n + 1$. Essa decisão, motivada por questões de eficiência, Stroustrup [2007], é a razão por trás de um número incontável de *worms* e *vírus* que se espalham na Internet, Bhatkar et al. [2003].

Linguagens de programação normalmente usam três tipos de regiões de alocação de memória: estática, *heap* e pilha. Variáveis globais, constantes em tempo de execução, e quaisquer outros dados conhecidos em tempo de compilação geralmente permanecem

```
void function(char* str) {
   char buffer[16];
   strcpy(buffer,str);
}

void main() {
  ...
  char* evil_str = read_data();
  function(evil_str);
  ...
}
```

(a)

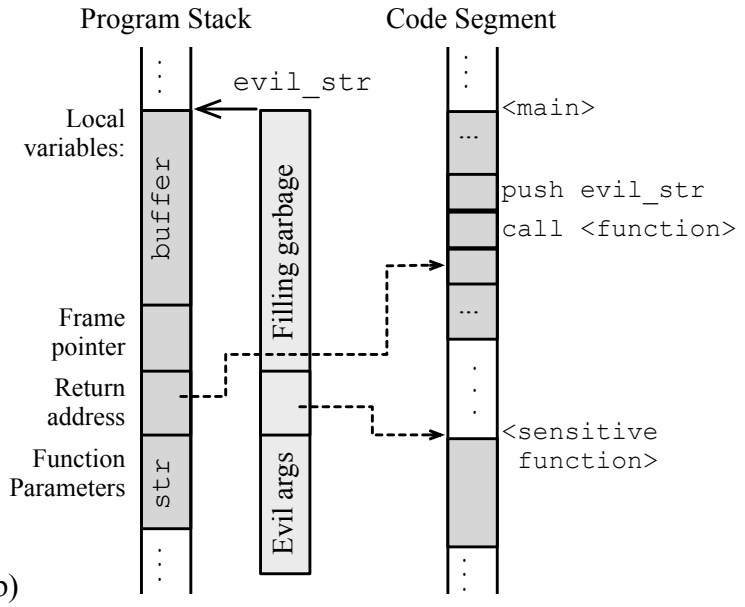

Figura 2.1. (a) Um programa que contém uma vulnerabilidade de *buffer overflow*. (b) Um exemplo esquemático de um *buffer overflow* na pilha. O endereço de retorno da `função` é desviado por uma entrada maliciosamente preparada para outro procedimento.

na área de alocação estática. Estruturas de dados criadas em tempo de execução, que ultrapassam o tempo de vida de funções onde elas foram criadas são colocadas na *heap*. Os registros de ativação de funções, que contêm, por exemplo, parâmetros, variáveis locais e endereço de retorno, são alocados na pilha. Em particular, uma vez que uma função é chamada, seu endereço de retorno é escrito em uma posição específica de seu registro de ativação. Depois que a função retorna, o programa retoma sua execução a partir desse endereço de retorno.

Um *buffer overflow* consiste em escrever em um *buffer* uma quantidade de dados

grande o bastante para ultrapassar seu limite superior; consequentemente, sobrescrevendo outros dados do programa ou usuário. O *overflow* pode acontecer na pilha, *heap* ou área estática. No cenário do *buffer overflow* na pilha, com a elaboração cuidadosa dos dados de entrada, pode-se sobrescrever o endereço de retorno no registro de ativação de uma função; dessa forma, desviando o fluxo de execução para outra área de código. Ataques à *buffer overflows* anteriormente incluiam o código que deveria ser executado no arranjo de entrada Levy [1996]. Entretanto, os sistemas operacionais modernos marcam as posições de memória que podem ser escritas como não-executável – uma proteção conhecida como *Read⊕Write* [Shacham et al., 2004, p.299], ou *Data Execution Prevention* (DEP). Por esse motivo, os atacantes tendem a desviar a execução para funções do sistema tais como `chmod` ou `sh`, se possível. Geralmente a *string* maliciosa contém também os argumentos que o atacante quer passar para a função sensível. A Figura 2.1 ilustra um exemplo de *buffer overflow* na pilha.

Uma vulnerabilidade de *buffer overflow* dá aos atacantes controle sobre o programa comprometido mesmo quando o sistema operacional não permite chamadas de função fora dos segmentos de memória alocados àquele programa. Os atacantes podem chamar funções da `libc`, por exemplo. Essa biblioteca, que é carregada em todo sistema UNIX, permite aos usuários executarem *fork* em processos e enviar pacotes através de uma rede, entre outras coisas. Tal ataque é chamado *return to* `libc`, Shacham et al. [2004]. *Return to* `libc` foi posteriormente generalizado para um tipo de ataque chamado *return-oriented-programming* (ROP) Shacham [2007]. Se um programa binário é grande o bastante, então é provável que ele contenha muitas sequências de bits que codificam instruções válidas. Hovav Shacham Shacham [2007] mostrou como derivar uma linguagem Turing completa a partir dessas sequências em uma máquina CISC, e Buchanan *et al.* Buchanan et al. [2008] generalizaram esse método para máquinas RISC.

Existem formas de prevenir esses tipos de ataques “return-to-known-code". O melhor mecanismo de defesa conhecido é a *ofuscação de endereços*, Bhatkar et al. [2003]. Um compilador pode randomizar a localização de funções dentro de um programa binário, ou o sistema operacional pode randomizar o endereço virtual de bibliotecas compartilhadas. Shacham et al. [2004] mostraram que esses métodos são suscetíveis à ataques de força bruta; apesar disso, a ofuscação de endereços diminui a taxa de propagação de *worms* que dependem substancialmente de vulnerabilidades de *buffer overflow*. A ofuscação de endereços não é, entretanto, o mecanismo de defesa definitivo. Se o programa alvo vaza algum de seus endereços internos por meio de um canal público, então os atacantes podem calcular os endereços de operações sensíveis que eles querem realizar. No resto dessa seção é mostrado, por meio de um exemplo, como explorar um

```
1 void *libc;
2 void process_input(char *inbuf, int len, int clientfd) {
3     char localbuf[40];
4     if (!strcmp(inbuf, "debug\n")) {
5         sprintf(localbuf, "localbuf %p\nsend() %p\n",
                  localbuf, dlsym(libc, "send"));
6     } else { memcpy(localbuf, inbuf, len); }
7     send(clientfd, localbuf, strlen(localbuf), 0);
8 }
9 int main() {
10    int sockfd, clientfd, c_len, len;
11    char inbuf[5001];
12    struct sockaddr_in myaddr, addr;
13    libc = dlopen("libc.so", RTLD_LAZY);
14    sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
15    myaddr.sin_family = AF_INET;
16    myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
17    myaddr.sin_port = htons(4000);
18    bind(sockfd, (struct sockaddr *)&myaddr, sizeof(myaddr));
19    listen(sockfd, 5);
20    c_len = sizeof(addr);
21    while (1) {
22        clientfd = accept(sockfd, (struct sockaddr *)&addr,
                  &c_len);
23        len = recv(clientfd, inbuf, 5000, 0);
24        inbuf[len] = '\0';
25        process_input(inbuf, len + 1, clientfd);
26        close(clientfd);
27    }
28    close(sockfd); dlclose(libc);
29    return 0;
30 }
```

Figura 2.2. Um servidor de eco implementado em C.

sistema protegido por ASLR e DEP.

2.1.1 Um Exemplo de Vazamento de Endereços

A vulnerabilidade de vazamento de endereços é ilustrada por meio do servidor de eco na Figura 2.2. O vazamento de informação nesse exemplo permite a exploração bem sucedida de um *buffer overflow* na pilha em uma máquina de 32 bits executando Ubuntu

```
1 import socket
2 import struct
3 c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4 c.connect(('localhost', 4000))
5 buf = "debug\n"
6 c.send(buf)
7 buf = c.recv(512)
8 leaked_stack_addr = int(buf[9:buf.find('\n')], 16)
9 leaked_send_addr = int(buf[27:buf.rfind('\n')], 16)
10 c.close()
11 c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12 c.connect(('localhost', 4000))
13 command = ('rm -f backpipe && mknod backpipe p && telnet'
       'localhost 8080 0<backpipe | /bin/bash 1>backpipe\x00')
14 command_addr = leaked_stack_addr + 64
15 system_addr = leaked_send_addr - 0x96dd0  # system()
16 system_ret_addr = system_addr - 0xa140 # exit()
17 buf = ('A' * 52 + struct.pack('I', system_addr) +
       struct.pack('I', system_ret_addr) +
       struct.pack('I', command_addr) + command)
18 c.send(buf)
19 c.close()
```

Figura 2.3. Um programa em Python que explora os vazamentos de endereços no programa da Figura 2.2.

11.10, um sistema operacional protegido por ASLR e DEP. Nesse exemplo, o programa vulnerável continua escutando por clientes na porta 4000, e quando um cliente se conecta, ele ecoa todo dado recebido. DEP dificulta um ataque de *buffer overflow* no estilo clássico de Levy Levy [1996], porque ele proíbe a execução de espaço de memória com permissão de escrita. Entretanto, pode-se usar um *buffer overflow* para desviar a execução de um programa para uma das funções do sistema operacional. Neste exemplo é aberto um terminal telnet no servidor. Este tipo de ataque requer que o adversário saiba o endereço de uma função sensível, por exemplo, `system` da `libc` nesse exemplo. Essa informação não está facilmente disponível em um sistema protegido por ASLR, a menos que o programa-alvo contenha um vazamento de endereço.

O vazamento de informação no exemplo dado ocorre na função `process_input`. Sempre que o servidor reconhece a *string* especial “debug"ele retorna dois endereços internos: a base de `localbuf`, que é um endereço da pilha, e o endereço de `send`, uma função de `libc`. Para construir o *exploit*, é usado o endereço de `send` para calcular o

endereço de `system` e `exit`, duas funções presentes na `libc`. Então é usado o endereço da pilha do ponteiro base de `localbuf` para encontrar o endereço dos argumentos para `system`. Um *script* Python que executa esse ataque é mostrado na Figura 2.3. Este *script* faz duas conexões para o servidor de eco. Na primeira conexão ele envia a *string* "debug"para receber os dois endereços vazados. Na segunda conexão ele envia os dados maliciosos para criar uma *shell connect-back*. Os dados maliciosos são compostos de: 52 A's para preencher a pilha até a posição do ponteiro de retorno; o endereço de `system`, calculado a partir do endereço vazado de `send`; o endereço de `exit`, também computado a partir do endereço de `send`; o endereço da *string* com o comando para criar a *shell connect-back*, calculado a partir do ponteiro base de `localbuf`; e finalmente a *string* contendo o comando para criar a *shell*. Sobrescrevendo o endereço de retorno de `process_input` com o endereço de `system`, ganha-se o controle da máquina remota. Chamando `exit` no fim do *exploit*, garante-se que o cliente termina sua execução silenciosamente depois de fornecer a *shell*.

2.2 Trabalhos Relacionados

A vulnerabilidade de vazamento de endereços é um problema bem conhecido, discutido em *blogs* e fóruns. Como um exemplo, Dion Blazakis explica como usar inferência de ponteiros para comprometer um compilador JIT de ActionScript [1]. Vazamentos de endereços são também mencionados por Fermin Serna como uma vulnerabilidade potencial [2], e o exemplo usado na Figura 3.13 foi baseado em um tutorial disponível online [3]. Apesar disso, a academia ainda não tem focado nesse problema, como pode ser inferido a partir da falta de publicações no campo. Entretanto, as técnicas que foram usadas neste trabalho já tem sido foco de muita pesquisa. Este trabalho se encaixa no *framework* de fluxo de informação proposto por Denning and Denning, Denning & Denning [1977]. Nas palavras de Hammer and Snelting, Hammer & Snelting [2009], o controle do fluxo de informação lida com dois problemas principais: *integridade* and *confidencialidade*. Integridade pede por garantias de que as computações não possam ser manipuladas de fora. Confidencialidade pede por garantias que dados sensíveis não possam escapar do programa. Em ataques contra a integridade, a informação contaminada flui de um atacante para as funções sensíveis; em ataques contra a confidencialidade a informação flui na direção oposta. Vazamentos de endereços são um problema de confidencialidade. Existem duas maneiras principais de rastrear o fluxo

[1] `http://www.semantiscope.com/research/BHDC2010`

[2] `The info leak era on software exploitation, slides disponíveis online.`

[3] `Pwn2Own-2010-Windows7-InternetExplorer8.pdf`

de informação em um programa: análises dinâmica e estática. Além disso, engenheiros podem combinar essas duas formas para obter resultados mais rápidos ou precisos. As seções seguintes descrevem esses dois tipos de abordagens de rastreamento.

2.2.1 Fluxo de Informação via Análises Dinâmicas

Na literatura são reconhecidas quatro categorias principais de análises dinâmicas: teste, emulação, verificações de hardware e instrumentação. Teste consiste em gerar entradas para um programa verificar se alguns eventos podem acontecer durante a execução real. Um exemplo de um trabalho muito influente nessa área é o sistema DART, que tenta, por meio de uma combinação de testes e execução simbólica, gerar entradas que cubram o programa inteiro Godefroid et al. [2005]. Emulação consiste em interpretar o programa, com o objetivo de verificar se algum comportamento acontece em tempo de execução. Um emulador bem conhecido é a ferramenta Valgrind Nethercote & Seward [2007], que suporta diferentes tipos de verificações que podem ser executadas nos programas em execução. O modo *taint* do Perl, usado para prevenir ataques de fluxo contaminado, também pode ser classificado como uma análise dinâmica baseada em emulação. Allen [2006]. O terceiro método, verificações de hardware, consiste em rastrear a propagação de informação no nível arquitetural, Suh et al. [2004].

Finalmente, instrumentação consiste em adicionar código extra ao programa para capturar eventos de interesse. Em outras palavras, esse método requer que o compilador modifique o programa. Tal programa executará diretamente em modo nativo, ao invés de ser interpretado. O *framework* de instrumentação que foi projetado e implementado no trabalho de dissertação descrito nesse texto se encaixa nessa última categoria. Instrumentação é usada geralmente em duas formas diferentes: no nível independente de máquina ou no nível binário. Este trabalho de dissertação usa uma biblioteca de instrumentação independente de máquina. Outras abordagens independentes de máquina incluem o *taint checker* de Xu Xu et al. [2006], e o *integer overflow checker* de Dietz *et al.* Dietz et al. [2012]. A literatura é abundante sobre trabalhos no nível binário. Para uma visão geral dessas abordagens recomenda-se a seção de trabalhos relacionados nos artigos de Clause Clause et al. [2007] e Newsome Newsome & Song [2005]. A principal vantagem da instrumentação binário é a habilidade para lidar precisamente com código de bibliotecas externas. As desvantagens incluem uma sobrecarga maior no tempo de execução, e a integração não-trivial com análises estáticas. Esta última deficiência, a difícil integração com análise estática, foi o fator chave que motivou a permanência no nível de representação intermediária do compilador neste trabalho.

2.2.2 Fluxo de Informação via Análises Estáticas

A maior parte da literatura relacionada ao fluxo de informação tem a análise estática como a ferramenta de escolha para rastrear informação Jovanovic et al. [2006]; Pistoia et al. [2005]; Rimsa et al. [2011]; Wassermann & Su [2007]; Xie & Aiken [2006]. A grande maioria desses trabalhos está preocupada com integridade, ou seja, "podem os dados maliciosos fluírem para alguma operação sensível do programa?". Contudo, existem também trabalhos que estão relacionados com a confidencialidade Hammer & Snelting [2009]; Russo & Sabelfeld [2010]. Existem muitas abordagens diferentes para descrever análises estáticas de fluxo de informação: sistemas de tipo Hammer & Snelting [2009]; Huang et al. [2004], fluxo de dados Jovanovic et al. [2006], interpretação abstrata Blanchet et al. [2003] e *program slicing* Rimsa et al. [2011]. Não obstante, todos esses artigos - incluindo este trabalho de dissertação - parecem depender de algoritmos similares que buscam caminhos perigosos no grafo de dependência do programa Ferrante et al. [1987].

Escolhemos formalizar a análise estática como um sistema de tipos, porque esse método torna fácil fornecer provas de corretude. A análise estática desenvolvida combina um componente *forward* e *backward* para obter resultados mais precisos. Pelo que se sabe, o único trabalho que também combina uma análise *forward* e *backward* é o de Rimsa *et al.*, Rimsa et al. [2011]. Entretanto, eles usam uma formulação de acessibilidade de grafos que não torna essa abordagem explícita. Além disso, enquanto que o trabalho de Rimsa *et al.* está preocupado com integridade - eles tentam proteger programas contra ataques de XSS - este trabalho de dissertação foca em confidencialidade.

2.2.3 A Combinação de Análises Estática e Dinâmica

Muitos pesquisadores diferentes têm combinado análises estática e dinâmica para rastrear informação ao longo dos caminhos dos programas. Essa combinação é motivada por desempenho ou precisão. No último caso, a análise estática complementa a informação que é adquirida pela sua contraparte dinâmica, ou a análise dinâmica elimina falso-positivos produzidos pelo algoritmo estático. Por exemplo, Zhang *et al.* Zhang et al. [2011] propuseram uma ferramenta que primeiro instrumenta o programa binário e executa-o, procurando por vulnerabilidades de fluxo contaminado. Durante essa execução protegida, a ferramenta também constrói o grafo do fluxo de informação do programa. Essa estrutura é dada depois para um analisador estático, que executa uma segunda passada de verificações no programa, verificando conservativamente as partes do programa que não foram alcançadas pelo fluxo de execução. Vogt *et al.* Vogt

et al. [2007] também usam análise estática para estender os resultados da análise dinâmica; entretanto, eles fazem isso *on the fly*. Isto é, uma vez que a análise dinâmica identifica regiões do programa que podem ser controladas pelos dados contaminados, a ferramenta de Vogt *et al.* executa uma passagem linear naquela região, marcando como contaminadas todas as variáveis definidas dentro dela. Em uma outra direção, Balzarotti *et al.* Balzarotti et al. [2008] projetou Saner, uma ferramenta que usa análise dinâmica baseada em testes para verificar as possíveis vulnerabilidades reportadas pela análise estática.

Análise estática pode também ser usada para melhorar o tempo de execução da técnica dinâmica. Possivelmente o trabalho mais influente nessa área é o WebSSARI de Huang *et al.*, Huang et al. [2004]. Essa ferramenta usa um sistema de tipos para identificar variáveis contaminadas do programa e insere sanitizadores em funções sensíveis que lêem essas variáveis. O trabalho apresentado nesse texto é diferente do trabalho de Huang em várias dimensões. Primeiro, este trabalho lida com confidencialidade, enquanto eles lidam com integridade. Segundo, o sistema de tipos desse trabalho é bidirecional, enquanto o deles é unidirecional (*forward*). Terceiro, eles instrumentam apenas as funções sensíveis, enquanto que este trabalho instrumenta o caminho contaminado inteiro, porque neste caso a noção de sanitizador não é bem definida. Em uma direção totalmente oposta ao trabalho de Huang e a este trabalho, mas ainda motivado pela eficiência, Chebaro *el al.* Chebaro et al. [2012] usaram uma análise estática para separar as partes seguras de um programa. Então eles usam uma análise dinâmica baseada em testes nos programas reduzidos para procurar por vulnerabilidades em suas partes inseguras.

2.3 Conclusão

Nesse capítulo foi apresentada uma breve revisão da literatura sobre exploração de software, com foco em vulnerabilidades de corrupção de memória, como o *buffer overflow*. Foi também apresentada uma descrição informal da vulnerabilidade de vazamento de endereços, juntamente com um exemplo real de programa vulnerável e o *exploit* correspondente. Além disso, discutiu-se vários trabalhos relacionados ao trabalho de dissertação descrito neste texto, fazendo as devidas comparações. Como foi discutido, o trabalho apresentado nessa dissertação é diferente dos trabalhos relacionados pois é pioneiro ao lidar com o problema do vazamento de endereços. Esse trabalho usa uma análise de tipos *backward* e *forward* para análise de fluxo de dados estática e combina análises dinâmica e estática para instrumentar caminhos contaminados inteiros para

detectar e prevenir as vulnerabilidades de vazamento de endereços.

Capítulo 3

Detecção Estática e Dinâmica de Vazamento de Endereços

Nesse capítulo é descrita a solução para detectar vazamentos de endereços em tempo de execução. Ele começa, na Seção 3.1, definindo uma linguagem contendo as construções de linguagens imperativas que desempenham um papel na vulnerabilidade de vazamento de endereços. Em cima dessa linguagem é definida uma *linguagem de instrumentação* na Seção 3.1.1. Programas implementados com a sintaxe de instrumentação podem rastrear o fluxo de informação em tempo de execução. A Seção 3.2 descreve um sistema de tipos que detecta vazamentos de endereço estaticamente. Esse sistema de tipos permite-nos reduzir a quantidade de instrumentação necessária para salvaguardar programas contra vazamentos de endereços. A Seção 3.3 conclui esse capítulo.

3.1 *Angels*: a Linguagem Protótipo

Essa seção define uma linguagem simples, chamada *Angels*, para explicar a abordagem usada para detecção dinâmica de vazamentos de endereços. *Angels* é uma linguagem *assembly-like*, cuja sintaxe é dada na Figura 3.1. Essa linguagem tem seis instruções que lidam com a computação de dados, e três instruções que alteram o fluxo do programa. As seis instruções relacionadas com dados representam construções típicas de programas C ou C++ reais, como a tabela ilustrada na Figura 3.2. Foi usado `adr` para modelar construções da linguagem que leem o endereço de uma variável, a saber, o operador *ampersand* (&) e funções de alocação de memória tais como `malloc`, `calloc` e `realloc`. Atribuições simples são representadas via a instrução `mov`. Operações binárias são representadas via a instrução `add`, que adiciona duas variáveis e coloca o resultado em uma terceira posição. Carregamentos e escritas na memória são mode-

(Variáveis)	::=	$\{v_1, v_2, \ldots\}$
(Instruções de dados)	::=	
– (Lê endereço)	|	$\texttt{adr}(v_1, v_2)$
– (Atribuição)	|	$\texttt{mov}(v_1, v_2)$
– (Adição binária)	|	$\texttt{add}(v_1, v_2, v_3)$
– (Armazena na memória)	|	$\texttt{stm}(v_0, v_1)$
– (Carrega da memória)	|	$\texttt{ldm}(v_1, v_0)$
– (Imprime)	|	$\texttt{out}(v)$
(Fluxo de controle)	::=	
– (Desvia se zero)	|	$\texttt{bzr}(v, l)$
– (Salto incondicional)	|	$\texttt{jmp}(l)$
– (Pára execução)	|	`end`
(Função ϕ - seletor)	|	$\texttt{phi}(v, \{v_1 : l_1, \ldots, v_k : l_k\})$

Figura 3.1. A sintaxe de *Angels*.

lados por `ldm` e `stm`. Finalmente, usou-se `out` para denotar qualquer instrução que dá informação para um usuário externo. Essa última instrução representa não apenas operações de impressão comuns, mas qualquer função que possa ser considerada como sensível. Por exemplo, um programa JavaScript geralmente depende de um conjunto de funções nativas para interagir com o navegador. Um usuário malicioso poderia usar essa interface para obter um endereço interno do interpretador JavaScript.

Este trabalho lida com programas na forma *Static Single Assignment* (SSA) Cytron et al. [1991]. Essa representação intermediária tem a propriedade chave que todo nome de variável tem apenas um ponto de definição no código do programa. Para garantir esta invariante, a representação intermediária SSA usa funções ϕ, uma notação especial que não existe nas linguagens *assembly* usuais. A Figura 3.1 mostra a sintaxe das funções ϕ. Essa representação não é necessária para completude computacional; entretanto, como será visto no Capítulo 4, ela simplifica a análise estática dos programas. Além disso, o formato SSA é usado pelo compilador *baseline*, LLVM, e muitos outros compiladores modernos, incluindo `gcc`. Assim, adotando essa representação diminui-se a lacuna entre o formalismo abstrato e sua implementação concreta.

A Figura 3.3 descreve a semântica operacional de pequenos passos de *Angels*. Essa formalização é similar a Schwartz *et al.*'s [Schwartz et al., 2010, Fig.5], ainda que tenha um nível muito mais baixo. Seja uma máquina abstrata uma tupla de cinco elementos $\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta \rangle$. Representa-se o programa P como um mapa que associa valores inteiros, chamados *labels*, com instruções. Seja `pc` o *contador do programa* atual, e seja `pc`' o contador do programa visto pela última instrução processada. É preciso manter

`v1 = &v2`	`adr`(v_1, v_2)
`v1 = (int*)` `malloc(sizeof(int))`	`adr`(v_1, v_2), v_2 é uma localização nova
`v1 = *v0`	`ldm`(v_1, v_0)
`*v0 = v1`	`stm`(v_0, v_1)
`*v1 = *v0`	`ldm`(v_2, v_0), v_2 é nova `stm`(v_1, v_2)
`v1 = v2 + v3`	`add`(v_1, v_2, v_3)
`*v = v1 + &v2`	`adr`(v_3, v_2), v_3 é nova `add`(v_4, v_1, v_3), v_4 é nova `stm`(v, v_4)
`f(v1, &v3)`, `f` é declarada como `f(int v2, int* v4);`	`mov`(v_2, v_1) `adr`(v_4, v_3)

Figura 3.2. Exemplos da sintaxe de C mapeada para instruções de *Angels*.

registro do contador do programa anterior para modelar funções ϕ. Essas instruções são usadas como variáveis multiplexadoras. Por exemplo, $v = \phi(v_1 : l_1, v_2 : l_2)$ copia v_1 para v se o controle vem de l_1, e copia v_2 para v se o controle vem de l_2. O contador de programa anterior aponta o caminho usado para alcançar a função ϕ.

A memória Σ é um ambiente que mapeia variáveis para valores inteiros, e Θ é um canal de saída. É usada a notação $f[a \mapsto b]$ para denotar a atualização da função f; isto é, $\lambda x.x = a \ ? \ b : f(x)$. Por simplicidade, não se distingue uma memória de variáveis locais, geralmente chamada de pilha, da memória de valores que vivem fora das funções que os criou, geralmente chamada *heap*. É usada uma função Δ para mapear os nomes de variáveis para seus endereços inteiros em Σ. O canal de saída é representado como uma lista Θ. Como pode ser visto na Regra [OutSem], a única instrução que pode manipular essa lista é a instrução `out`. Essa operação é denotada pelo operador $::$, como em ML e Ocaml.

Angels é uma linguagem de programação Turing Completa. Dado um excedente infinito de variáveis, pode-se implementar uma Máquina de Turing nela. A Figura 3.4 mostra um exemplo de um programa escrito em *Angels*. O programa na Figura 3.4(a) imprime o conteúdo de um arranjo, e então o endereço base do próprio arranjo. A Figura 3.4(b) mostra o programa *Angels* equivalente. Foi delineado o rótulo da primeira instrução presente em cada bloco básico deste exemplo. Foi também marcado em

$$[\textsc{AdrSem}]\quad \frac{\Delta(v_2) = n \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n]}{\langle \texttt{adr}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle} \qquad [\textsc{OutSem}]\quad \frac{\Sigma[\Delta(v)] = n \qquad \Theta' = n :: \Theta}{\langle \texttt{out}(v), \Sigma, \Theta\rangle \to \langle \Sigma, \Theta'\rangle}$$

$$[\textsc{EndSem}]\quad \frac{P[\texttt{pc}] = \texttt{end}}{\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta\rangle \to \langle \Sigma, \Theta\rangle} \qquad [\textsc{MovSem}]\quad \frac{\Sigma[\Delta(v_2)] = n \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n]}{\langle \texttt{mov}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle}$$

$$[\textsc{AddSem}]\quad \frac{\Sigma[\Delta(v_2)] = n_2 \qquad \Sigma[\Delta(v_3)] = n_3 \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n_2 + n_3]}{\langle \texttt{add}(v_1, v_2, v_3), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle}$$

$$[\textsc{StmSem}]\quad \frac{\Sigma[\Delta(v_0)] = x \qquad \Sigma[\Delta(v_1)] = n \qquad \Sigma' = \Sigma[x \mapsto n]}{\langle \texttt{stmem}(v_0, v_1), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle}$$

$$[\textsc{LdmSem}]\quad \frac{\Sigma[\Delta(v_0)] = x \qquad \sigma[x] = n \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n]}{\langle \texttt{ldmem}(v_1, v_0), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle}$$

$$[\textsc{JmpSem}]\quad \frac{P[\texttt{pc}] = \texttt{jmp}(l) \qquad \langle P, \texttt{pc}, l, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}{\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}$$

$$[\textsc{BzrSem}]\quad \frac{P[\texttt{pc}] = \texttt{bzr}(v, l) \qquad \Sigma[\Delta(v)] \neq 0 \qquad \langle P, \texttt{pc}, \texttt{pc} + 1, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}{\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}$$

$$[\textsc{BnzSem}]\quad \frac{P[\texttt{pc}] = \texttt{bzr}(v, l) \qquad \Sigma[\Delta(v)] = 0 \qquad \langle P, \texttt{pc}, l, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}{\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle}$$

$$[\textsc{PhiSem}]\quad \frac{\begin{array}{c} P[\texttt{pc}] = \texttt{phi}(v, \{v_1 : l_1, \ldots, v_k : l_k\}) \\ \texttt{pc}' = l_i \qquad \langle \texttt{mov}(v, v_i), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \qquad \langle P, \texttt{pc}, \texttt{pc} + 1, \Sigma', \Theta\rangle \to \langle \Sigma'', \Theta'\rangle \end{array}}{\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta'\rangle}$$

$$[\textsc{SeqSem}]\quad \frac{P[\texttt{pc}] \notin \{\texttt{bzr}, \texttt{end}, \texttt{phi}, \texttt{jmp}\} \qquad \langle P[\texttt{pc}], \Sigma, \Theta\rangle \to \langle \Sigma', \Theta'\rangle \qquad \langle P, \texttt{pc}, \texttt{pc} + 1, \Sigma', \Theta'\rangle \to \langle \Sigma'', \Theta''\rangle}{\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta''\rangle}$$

Figura 3.3. A Semântica Operacional de *Angels*.

cinza a memória inicial usada neste exemplo. *Angels* não tem instruções para carregar constantes em variáveis; entretanto, essa omissão é compensada iniciando os programas com uma memória não-vazia.

O programa na Figura 3.4 contém uma vulnerabilidade de vazamento de endereço, uma noção que foi introduzida na Definição 3.1.1. Um programa *Angels* P contém tal vulnerabilidade se um atacante pode reconstruir o mapa Δ para no mínimo uma variável que P usa. Note que não é assumido que P termina. Apenas requer-se que o programa imprima qualquer informação que um atacante possa ler. O exemplo contém uma vulnerabilidade de vazamento de endereço, porque ele imprime o endereço base do arranjo f. Assim, por meio da leitura do canal de saída, o atacante seria capaz de encontrar $\Delta(f)$.

Definição 3.1.1 A Vulnerabilidade de Vazamento de Endereços *"Um programa Angels P, tal qual $\langle P, 0, 0, \lambda x.0, []\rangle \to \langle \Sigma, \Theta\rangle$, onde $\lambda x.0$ é o ambiente que mapeia*

(a)

```
int main() {
    int i = 0, f[4];
    for (i = 0; i < 4; i++) {
        printf("%d\n", f[i]);
    }
    printf("%u\n", f);
}
```

(b)

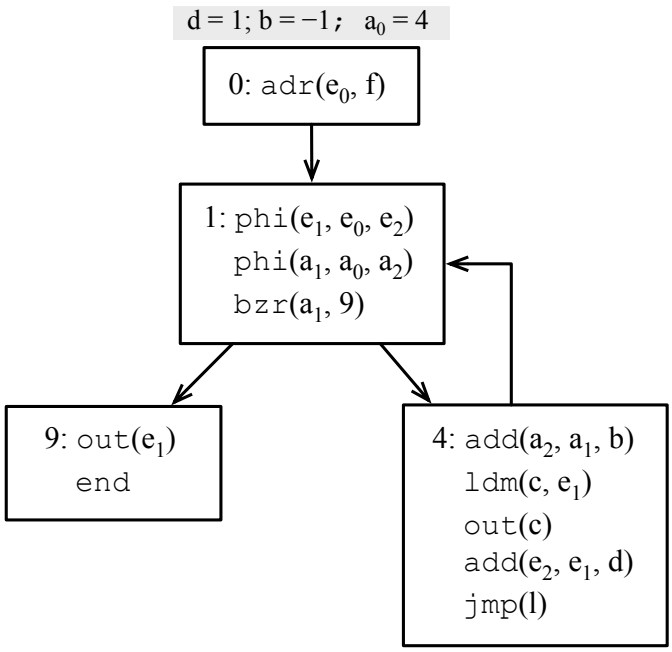

Figura 3.4. Um programa C traduzido para *Angels*.

variáveis para zero, e [] *é o canal de saída vazio, contém uma vulnerabilidade de vazamento de endereços se um atacante pode descobrir* $\Delta(v)$ *para algum* v *usado em* P*, dado o conhecimento de* Θ *mais o código fonte de* P*."*

3.1.1 A Linguagem de Instrumentação

Para proteger um programa contra vazamentos de endereços, ele é instrumentado. Em outras palavras, sua sequência original de instruções é substituída por outras instruções, que rastreiam o fluxo de valores em tempo de execução. É possível instrumentar um programa *Angels* usando apenas instruções *Angels*. Entretanto, para executar a instrumentação dessa forma, essa apresentação teria que ser feita de forma desnecessariamente complicada. Para evitar essa complexidade, definiu-se um segundo conjunto de instruções, cuja semântica é dada na Figura 3.5. O *framework* de instrumentação

$$[\textsc{AdrIns}]\quad \frac{\langle \texttt{adr}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \qquad \Delta(v_1) = n_1 \qquad \Sigma'' = \Sigma'[n_1 + D \mapsto \mathit{tainted}]}{\langle \texttt{sh\_adr}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

$$[\textsc{OutIns}]\quad \frac{\Delta(v) = n_v \qquad \Sigma[n_v + D \mapsto \mathit{clean}] \qquad \langle \texttt{out}(v), \Sigma, \Theta\rangle \to \langle \Sigma, \Theta'\rangle}{\langle \texttt{sh\_out}(v), \Sigma, \Theta\rangle \to \langle \Sigma, \Theta'\rangle}$$

$$[\textsc{MovIns}]\quad \frac{\langle \texttt{mov}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \qquad \Sigma'[\Delta(v_2) + D] = t \qquad \Sigma'' = \Sigma'[\Delta(v_1) + D \mapsto t]}{\langle \texttt{sh\_mov}(v_1, v_2), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

$$[\textsc{AddIns}]\quad \frac{\begin{array}{c}\langle \texttt{add}(v_1, v_2, v_3), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \\ \Sigma'[\Delta(v_2) + D] = t_2 \qquad \Sigma'[\Delta(v_3) + D] = t_3 \qquad t_1 = t_2 \sqcap_{sh} t_3 \qquad \Sigma'' = \Sigma[\Delta(v_1) + D \mapsto t]\end{array}}{\langle \texttt{sh\_add}(v_1, v_2, v_3), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

$$[\textsc{StmIns}]\quad \frac{\Sigma[\Delta(v_0)] = x \qquad \Sigma[\Delta(v_1)] = n \qquad \Sigma' = \Sigma[x \mapsto n] \qquad \Sigma'[\Delta(v_1) + D] = t \qquad \Sigma'' = \Sigma'[x + D \mapsto t]}{\langle \texttt{sh\_stm}(v_0, v_1), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

$$[\textsc{LdmIns}]\quad \frac{\Sigma[\Delta(v_0)] = x \qquad \Sigma[x] = n \qquad \Sigma' = \Sigma[\Delta(v_1) \mapsto n] \qquad \Sigma'[x + D] = t \qquad \Sigma'' = \Sigma'[\Delta(v_1) + D \mapsto t]}{\langle \texttt{sh\_ldm}(v_1, v_0), \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta\rangle}$$

$$[\textsc{PhiIns}]\quad \frac{\begin{array}{c}P[\texttt{pc}] = \texttt{sh\_phi}(v, \{v_1 : l_1, \ldots, v_k : l_k\}) \\ \texttt{pc}' = l_i \qquad \langle \texttt{sh\_mov}(v, v_i), \Sigma, \Theta\rangle \to \langle \Sigma', \Theta\rangle \qquad \langle P, \texttt{pc}, \texttt{pc} + 1, \Sigma', \Theta\rangle \to \langle \Sigma'', \Theta'\rangle\end{array}}{\langle P, \texttt{pc}', \texttt{pc}, \Sigma, \Theta\rangle \to \langle \Sigma'', \Theta'\rangle}$$

Figura 3.5. A Semântica Operacional das instruções instrumentadas.

define uma instrução *shadow* equivalente para `out` e para cada instrução que pode atualizar a memória Σ. Para armazenar os metadados produzidos pela instrumentação foi criada uma *memória shadow*. Para cada variável v, armazenada em $\Sigma[\Delta(v)]$, foi criada uma nova localização $\Sigma[\Delta(v) + D]$, onde D é um deslocamento que separa cada variável de sua *shadow*. Em outras palavras, informações sobre variáveis *shadow* e original são mantidas no mesmo *store*. Os valores *shadow* podem ser ligados a um de dois valores *clean* ou *tainted*. Programas instrumentados são avaliados pelas mesmas regras vistas na Figura 3.3, aumentadas com as seis novas regras dadas na Figura 3.5. É importante notar que os programas instrumentados podem terminar prematuramente. A Regra OutIns não progride se houver uma tentativa de imprimir uma variável cujo *shadow* está contaminado. O operador *meet* usado na definição de `sh_add`, $\sqcap_{sh}$ é tal que $n_1 \sqcap_{sh} n_2 = \mathit{tainted}$ sempre que um deles é *tainted*, e é *clean* caso contrário.

A Figura 3.6 define uma relação ι que converte um programa *Angels* comum em um programa instrumentado. Toda instrução que pode armazenar dados na memória é instrumentada; assim, essa relação é chamada de *framework de instrumentação completa*. Nota-se que as instruções que controlam o fluxo de execução não são instrumentadas. Elas não precisam ser instrumentadas porque nenhuma delas gera novos dados no *store*.

$$
\begin{array}{rcl}
\texttt{adr}(v_1, v_2) & \xrightarrow{\iota} & \texttt{sh\_adr}(v_1, v_2) \\
\texttt{out}(v) & \xrightarrow{\iota} & \texttt{sh\_out}(v) \\
\texttt{mov}(v_1, v_2) & \xrightarrow{\iota} & \texttt{sh\_mov}(v_1, v_2) \\
\texttt{add}(v_1, v_2, v_3) & \xrightarrow{\iota} & \texttt{sh\_add}(v_1, v_2, v_3) \\
\texttt{phi}(v, \{.., v_i : l_i, ..\}) & \xrightarrow{\iota} & \texttt{sh\_phi}(v, \{.., v_i : l_i, ..\}) \\
\texttt{stm}(v_0, v_1) & \xrightarrow{\iota} & \texttt{sh\_stm}(v_0, v_1) \\
\texttt{ldm}(v_0, v_1) & \xrightarrow{\iota} & \texttt{sh\_ldm}(v_0, v_1) \\
\texttt{bzr}(v, l) & \xrightarrow{\iota} & \texttt{bzr}(v, l) \\
\texttt{jmp}(l) & \xrightarrow{\iota} & \texttt{jmp}(l) \\
\texttt{end} & \xrightarrow{\iota} & \texttt{end}
\end{array}
$$

Figura 3.6. A instrumentação completa de um programa *Angels*.

3.1.1.1 Corretude da Instrumentação Completa

A instrumentação, isto é, a relação ι da Figura 3.6, não deveria modificar a semântica do programa original. Essa propriedade é provada no Teorema 3.1.2. Além disso, a relação ι deveria assegurar que um programa instrumentado não contém um vazamento de endereços. O Teorema 3.1.5 prova essa propriedade. Ambos teoremas requerem a noção de uma *trilha de execução*. A trilha de execução de um programa P é a sequência T de instruções que foram processadas durante a execução de P. A seguinte notação será usada: Se T é a trilha de execução de $\langle P, 0, 0, \lambda x.0, [] \rangle$, e $P \xrightarrow{\iota} P^\iota$, então T^ι é a trilha de execução de $\langle P^\iota, 0, 0, \lambda x.0, [] \rangle$. O resto desta seção se baseia em duas suposições: (i) toda variável é inicializada antes de ser usada, e (ii) não existe sobreposição entre a memória original e a memória *shadow*.

Teorema 3.1.2 *Se a variável v recebe o valor x na i-ésima instrução de T^ι, então a variável v recebe o mesmo valor x na i-ésima instrução de T.*

Prova: A prova é por indução no tamanho de T^ι. Assume-se que o teorema se mantém para as primeiras N instruções de T^ι. Executa-se uma análise de caso em cada regra na Figura 3.3, e na regra correspondente na Figura 3.5:

- ADRSEM/INS: v_1 recebe $\Sigma[\Delta(v_2)]$ em ambos os casos.
- OUTSEM/INS: a instrução `out` não define variáveis.
- MOVSEM/INS: $\Sigma[\Delta(v_1)]$ recebe $\Sigma[\Delta(v_2)]$ em ambos os casos. Pela indução, $\Sigma[\Delta(v_2)]$ é o mesmo em ambos os programas.

- StmSem/Ins: pela indução, $\Sigma[\Delta(v_0)]$ e $\Sigma[\Delta(v_1)]$ são os mesmos em ambos os programas. Por isso, $\Sigma[x \mapsto n]$ em ambos. Pela hipótese, $\Sigma[\Delta(v_1) + D]$ não foi modificado por atribuições no programa original, e uma atribuição para $\Sigma[x+D]$ não modifica qualquer posição de memória no programa original.

As outras regras são similares. □

Como um corolário direto do Teorema 3.1.2, tem-se que qualquer trilha de execução do programa instrumentado é um prefixo da trilha de execução no programa original:

Corolário 3.1.3 *T^ι é um prefixo de T.*

Prova: as duas regras que determinam o fluxo de controle, BzrSem e BnzSem são as mesmas para os programas original e instrumentado. Pelo Teorema 3.1.2, desvios sempre ocorrem nos mesmo valores em ambos os programas. □

O Lema 3.1.4 afirma que informação contaminada se propaga na memória *shadow* de acordo com as dependência de dados no programa original; Diz-se que uma variável v tem *dependência de dados* em uma variável u, em um programa P, se (i) v está definido por uma instrução que usa u, ou seja, $\texttt{mov}(v, u) \in P$; ou, (ii) $\Delta(v_0) = v$, e $\texttt{stm}(v_0, u) \in P$; ou, (iii) $\Delta(v_0) = u$, e $\texttt{ldm}(v, v_0) \in P$; ou (iv) v tem dependência de dados em algum x, e x tem dependência de dados em u.

Lema 3.1.4 *Seja P um programa angels, e P^ι seja tal que $P \xrightarrow{\iota} P^\iota$. Uma variável v tem dependência de dados em algum endereço na i-ésima instrução de T^ι, se, e somente se, $\Sigma[\Delta(v) + D]$ contem o valor* tainted.

Prova: A seguir prova-se a parte "se" do teorema. Procede-se pela indução no tamanho da trilha de execução T^ι, assumindo que a hipótese é verdadeira para todo prefixo T^ι_p de T^ι, tal que $|T^\iota_p| < N$. Executa-se uma análise de caso na instrução $I^\iota \in T^\iota$ que vêm depois de T^ι_p:

- $\texttt{sh\_adr}(v_1, v_2)$: a única variável modificada é v_1. Pela Regra AdrIns, $\Sigma[\Delta(v_1) + D] \mapsto \textit{tainted}$.

- $\texttt{sh\_out}(v)$: nenhuma variável é modificada.

- $\texttt{sh\_mov}(v_1, v_2)$: a hipótese se mantém para v_2, por indução. Pela definição, v_1 tem dependência de dados em v_2. O estado contaminado se propaga de v_2 para v_1 por MovIns.

- `sh_stm`(v_0, v_1): se $\Sigma[\Delta(v_0)] = x$, então a hipótese é verdadeira para x pela indução. Pela definição, v_0 tem dependência de dados em v_1. Pela Regra StmIns, o mesmo estado contaminado se propaga de x para v_0.

As outras regras são similares. □

Teorema 3.1.5 *Se P é um programa angels, e P^ι é tal que $P \xrightarrow{\iota} P^\iota$, então $\langle P^\iota, 0, 0, \lambda x.0, [] \rangle$ não contém uma vulnerabilidade de vazamento de endereços.*

Prova: De acordo com as premissas da Regra OutIns, informação é impressa no canal de saída somente se ela não está associada a um estado contaminado. Pelo Lema 3.1.4 apenas informação que tem dependência de dados em informação de endereço está associada com estados contaminados. □

3.2 Reduzindo a Sobrecarga de Instrumentação via Análise Estática

Como será mostrado empiricamente, um programa completamente instrumentado é consideravelmente mais lento que o programa original. Para diminuir essa sobrecarga, casou-se a instrumentação dinâmica com uma análise estática que detecta vazamentos de endereços. Essa análise estática é descrita como um sistema de tipos, que é resolvido via a combinação de uma máquina de propagação *forward* e *backward*.

3.2.1 Análise de Ponteiros

O sistema de tipos proposto é parametrizado por uma análise de ponteiros. A solução para a análise de ponteiros em *Angels* é um mapa Π de variáveis para fatos *points-to*, que obtém-se como o ponto fixo de um sistema de restrições dado na Figura 3.7. Um fato *points-to* é um conjunto de variáveis, de tal modo que se uma variável v pode carregar o endereço da variável x, então x está no conjunto *points-to* de v. Neste trabalho não é dado nenhum algoritmo para resolver o sistema de restrições na Figura 3.7, porque muitos deles já foram descritos na literatura. Na implementação, foi usada a *detecção de ciclos preguiçosa* de Hardekopf, Hardekopf & Lin [2007]. Esse algoritmo tem uma complexidade assintótica de $O(|V|^3)$ no pior caso, onde $|V|$ é o número de variáveis no programa de entrada. Entretanto, na prática o algoritmo é muito mais rápido, devido a heurísticas que encontram ciclos no grafo, como mostrado no Capítulo 4, Fig. 4.4.

Instrução	Restrição
$\texttt{adr}(v_1, v_2)$	$\{v_2\} \subseteq \Pi(v_1)$
$\texttt{mov}(v, v_1)$	$\Pi(v_1) \subseteq \Pi(v)$
$\texttt{stm}(v_0, v_1)$	$x \in \Pi(v_0) \Rightarrow \Pi(v_1) \subseteq \Pi(x)$
$\texttt{ldm}(v_1, v_0)$	$x \in \Pi(v_0) \Rightarrow \Pi(x) \subseteq \Pi(v_1)$

Figura 3.7. Restrições que definem a análise de ponteiros.

3.2.2 Um Sistema de Tipos para Detectar Estaticamente Vazamentos de Endereços

Essa seção descreve um sistema de tipos que anota cada variável do programa com um dos três tipos: *clean*, *maybe* ou *tainted*. O sistema de tipos é descrito por dois componentes, *forward* e *backward*. O componente *forward* primeiro anota variáveis com os tipos *clean* ou *maybe*. O componente *backward* então anota as variáveis *maybe* com os tipos *clean* ou *tainted*. Apenas variáveis que a análise *backwards* anota como *tainted* podem vazar informação de endereços. A máquina de inferência *forward* é definida pela relação $\xrightarrow{\gamma\rightarrow}$, dada na Figura 3.9. A máquina de inferência *backward* é definida pela relação $\xrightarrow{\gamma\leftarrow}$, dada na Figura 3.10. Essas análises usam os operadores *meet* definidos na Figura 3.8.

Como foi mencionado no início desta seção, este trabalho lida com programas na representação intermediária SSA. A principal vantagem de usar o formato SSA é que o estado abstrato de uma variável, por exemplo, *clean* ou *tainted*, é invariante ao longo de todo ponto do programa onde a variável existe. Assim, pode-se ligar essa informação diretamente à variável, ao invés de pares formados pelas variáveis e pontos no programa, como análises mais tradicionais fazem. Assim, a representação SSA permite executar a análise estática *esparsamente*. Como Choi *et al.* demonstraram duas décadas atrás Choi et al. [1991], análises esparsas são mais eficientes em termos de velocidade e memória.

A análise *forward* assume que toda variável é *limpa*. Ao encontrar aquelas variáveis que podem conter endereços, ela marca-as com o tipo *maybe*. Pela Regra AdrFwd, vê-se que `adr` é a única instrução que pode mudar o tipo de uma variável durante a análise *forward*. O objetivo da análise *backward*, por outro lado, é encontrar quais variáveis tipadas como *maybe* podem alcançar a instrução `out`. Da Regra OutBkw, pode-se ver que `out` é a única instrução que pode despejar o tipo *tainted* dentro do ambiente de tipos.

$\sqcap_{\rightarrow}$	*clean*	*maybe*
clean	*clean*	*maybe*
maybe	*maybe*	*maybe*

$\sqcap_{\leftarrow}$	*maybe*	*tainted*	*clean*
maybe	*maybe*	*tainted*	*clean*
tainted	*tainted*	*tainted*	*clean*
clean	*clean*	*clean*	*clean*

Figura 3.8. Semântica Abstrata dos Operadores *meet* usados na *Engine* de Inferência de Tipos *Forward* e *Backward*.

$$[\text{RF}] \qquad \langle \texttt{adr}(v_1, v_2), \Gamma, \Pi \rangle \xrightarrow{\gamma\rightarrow} \Gamma[v_1 \mapsto \mathit{maybe}]$$

$$[\text{MF}] \qquad \frac{\Gamma \vdash v_1 = t_1 \qquad \Gamma \vdash v_2 = t_2 \qquad t_1' = t_1 \sqcap_{\rightarrow} t_2}{\langle \texttt{mov}(v_1, v_2), \Gamma, \Pi \rangle \xrightarrow{\gamma\rightarrow} \Gamma[v \mapsto t_1']}$$

$$[\text{AF}] \qquad \frac{\Gamma \vdash v_2 = t_2 \qquad \Gamma \vdash v_3 = t_3 \qquad t_1 = t_2 \sqcap_{\rightarrow} t_3}{\langle \texttt{add}(v_1, v_2, v_3), \Gamma, \Pi \rangle \xrightarrow{\gamma\rightarrow} \Gamma[v_1 \mapsto t_1]}$$

$$[\text{PF}] \qquad \frac{\Gamma \vdash v_1 = t_1 \ldots \Gamma \vdash v_k = t_k \qquad t = t_1 \sqcap_{\rightarrow} \ldots \sqcap_{\rightarrow} t_k}{\langle \texttt{phi}(v, \{v_1, \ldots, v_k\}), \Gamma, \Pi \rangle \xrightarrow{\gamma\rightarrow} \Gamma[v \mapsto t]}$$

$$[\text{SF}] \qquad \frac{\Gamma \vdash v_1 = t_1 \qquad \Gamma \vdash x = t_x \qquad t_x' = t_1 \sqcap_{\rightarrow} t_x}{\langle \texttt{stm}(v_0, v_1), \Gamma, \Pi \rangle \xrightarrow{\gamma\rightarrow} \Gamma[x \mapsto t_x]}, x \in \Pi[v_0]$$

$$[\text{LF}] \qquad \frac{\Gamma \vdash v_1 = t_1 \qquad \Gamma \vdash x = t_x \qquad t_1' = t_1 \sqcap_{\rightarrow} t_x}{\langle \texttt{ldm}(v_0, v_1), \Gamma, \Pi \rangle \xrightarrow{\gamma\rightarrow} \Gamma[v_1 \mapsto t_1']}, x \in \Pi[v_0]$$

Figura 3.9. A análise de inferência de tipos *forward*.

Ambos os algoritmos de propagação *forward* e *backward* tem implementações $O(|V|^2)$. Antes de executar o algoritmo *forward*, temos $\Gamma[v] = \mathit{clean}$ para toda variável v. O tipo de uma variável pode mudar apenas uma vez, se tornando *maybe*; assim, o número total de vezes que uma variável precisa ser analisada é limitada superiormente por $O(1)$. A complexidade quadrática é alcançada devido ao uso de fatos *points-to*, que fazem com que as Regras SF e LF tenham complexidade $O(|V|)$. Obtém-se a complexidade $O(|V|^2)$ da análise *backward* de forma similar: o tipo de cada variável

$$[\textsc{Ob}] \quad \frac{\Gamma \vdash v = t \qquad t' = \mathit{tainted} \sqcap_{\leftarrow} t}{\langle \texttt{out}(v), \Gamma, \Pi \rangle \xrightarrow{\gamma\leftarrow} \Gamma[v \mapsto t']}$$

$$[\textsc{Mb}] \quad \frac{\Gamma \vdash v_1 = t_1 \qquad \Gamma \vdash v_2 = t_2 \qquad t_2' = t_1 \sqcap_{\leftarrow} t_2}{\langle \texttt{mov}(v_1, v_2), \Gamma, \Pi \rangle \xrightarrow{\gamma\leftarrow} \Gamma[v_2 \mapsto t_2']}$$

$$[\textsc{Ab}] \quad \frac{\begin{array}{c}\Gamma \vdash v_2 = t_2 \\ \Gamma \vdash v_3 = t_3 \qquad t_2' = t_1 \sqcap_{\leftarrow} t_2 \qquad t_3' = t_1 \sqcap_{\leftarrow} t_3\end{array}}{\langle \texttt{add}(v_1, v_2), \Gamma, \Pi \rangle \xrightarrow{\gamma\leftarrow} (\Gamma[v_2 \mapsto t_2])[v_3 \mapsto t_3']}$$

$$[\textsc{Pb}] \quad \frac{\Gamma \vdash v = t \qquad \Gamma \vdash v_i = t_i \qquad t_i' = t \sqcap_{\leftarrow} t_i}{\langle \texttt{phi}(v, \{\ldots, v_i, \ldots\}), \Gamma, \Pi \rangle \xrightarrow{\gamma\leftarrow} \Gamma[v_i \mapsto t_i']}$$

$$[\textsc{Sb}] \quad \frac{\Gamma \vdash x = t_x \qquad \Gamma \vdash v_1 = t_1 \qquad t_1' = t_x \sqcap_{\leftarrow} t_1}{\langle \texttt{stm}(v_0, v_1), \Gamma, \Pi \rangle \xrightarrow{\gamma\leftarrow} \Gamma[v_1 \mapsto t_1']}, x \in \Pi[v_0]$$

$$[\textsc{Lb}] \quad \frac{\Gamma \vdash x = t_x \qquad \Gamma \vdash v_1 = t_1 \qquad t_x' = t_x \sqcap_{\leftarrow} t_1}{\langle \texttt{ldm}(v_1, v_0), \Gamma, \Pi \rangle \xrightarrow{\gamma\leftarrow} \Gamma[x \mapsto t_x']}, x \in \Pi[v_0]$$

Figura 3.10. A análise de inferência de tipos *backward*.

pode mudar apenas uma vez, de *maybe* para *tainted*, e os fatos *points-to* contribuem com outro componente linear para o processamento de *loads* e *stores*.

3.2.3 Instrumentação Parcial

Dados os resultados do sistema de tipos, pode-se reduzir a quantidade de código instrumentado no programa-alvo que a relação ι da Figura 3.6 insere. Com tal propósito, definiu-se um um novo *framework* de instrumentação como a relação γ, que é dada na Figura 3.11. Ao contrário do algoritmo na Figura 3.6, dessa vez altera-se uma instrução apenas se (i) ela define uma variável que tem o tipo *tainted*; ou (ii) ela é a operação de saída, e usa uma variável com o tipo *tainted*. O Teorema 3.2.3 prova que a instrumentação parcial está correta. Por uma questão de espaço, omitiu-se da Figura 3.11 as regras que não mudam a instrução-alvo. Uma instrução pode permanecer inalterada no programa instrumentado, seja porque essa instrução foi considerada segura pela análise estática, ou porque ela não cria valores na memória.

A Figura 3.12 mostra os resultados da aplicação da análise estática no programa visto na Figura 3.4(b). A Figura 3.12(a) mostra o grafo de dependência que o programa-alvo induz. Cada variável neste grafo foi anotada com os resultados da máquina de

$$\frac{\Gamma \vdash v_1 = \mathit{tainted}}{\langle \texttt{adr}(v_1, v_2), \Gamma, \Pi\rangle \xrightarrow{\gamma} \texttt{sh\_adr}(v_1, v_2)}$$

$$\frac{\Gamma \vdash v = \mathit{tainted}}{\langle \texttt{out}(v), \Gamma, \Pi\rangle \xrightarrow{\gamma} \texttt{sh\_out}(v)}$$

$$\frac{\Gamma \vdash v_1 = \mathit{tainted}}{\langle \texttt{mov}(v_1, v_2), \Gamma, \Pi\rangle \xrightarrow{\gamma} \texttt{sh\_mov}(v_1, v_2)}$$

$$\frac{\Gamma \vdash v_1 = \mathit{tainted}}{\langle \texttt{add}(v_1, v_2, v_2), \Gamma, \Pi\rangle \xrightarrow{\gamma} \texttt{sh\_add}(v_1, v_2, v_3)}$$

$$\frac{\exists x \in \Pi[v_0], \Gamma \vdash x = \mathit{tainted}}{\langle \texttt{stm}(v_0, v_1), \Gamma, \Pi\rangle \xrightarrow{\gamma} \texttt{sh\_stm}(v_0, v_1)}$$

$$\frac{\Gamma \vdash v_1 = \mathit{tainted}}{\langle \texttt{ldm}(v_1, v_0), \Gamma, \Pi\rangle \xrightarrow{\gamma} \texttt{sh\_ldm}(v_1, v_0)}$$

Figura 3.11. A relação γ que instrumenta parcialmente programas *Angels*.

inferência *forward*. Assim, uma variável pode ter recebido o tipo *clean* (C) ou *maybe* (M). A atribuição final, como encontrada pela anotação *backward* é mostrada na Figura 3.12(b). Nesse caso, as variáveis e_0, e_1 e e_2 foram marcadas *tainted* (T). O *framework* de instrumentação parcial muda as instruções para ou defini-las ou imprimi-las, como vista na Figura 3.12(c).

3.2.4 Corretude da Instrumentação Parcial

É apresentada uma prova que programas parcialmente instrumentados não contém vazamentos de endereços. Essa prova consiste em mostrar que qualquer *cadeia de dependência* entre uma fonte de informação de endereços e a operação de saída contém apenas instruções instrumentadas.

Lema 3.2.1 *Apenas variáveis que tem dependência de dados em um endereço tem o tipo* maybe.

Prova: Apenas a Regra Rf liga o tipo *maybe* a uma variável. Uma análise de caso em cada uma das outras regras mostra que os tipos só se propagam de uma variável v_1 para outra variável v_2 se v_2 tem dependência de dados em v_1. □

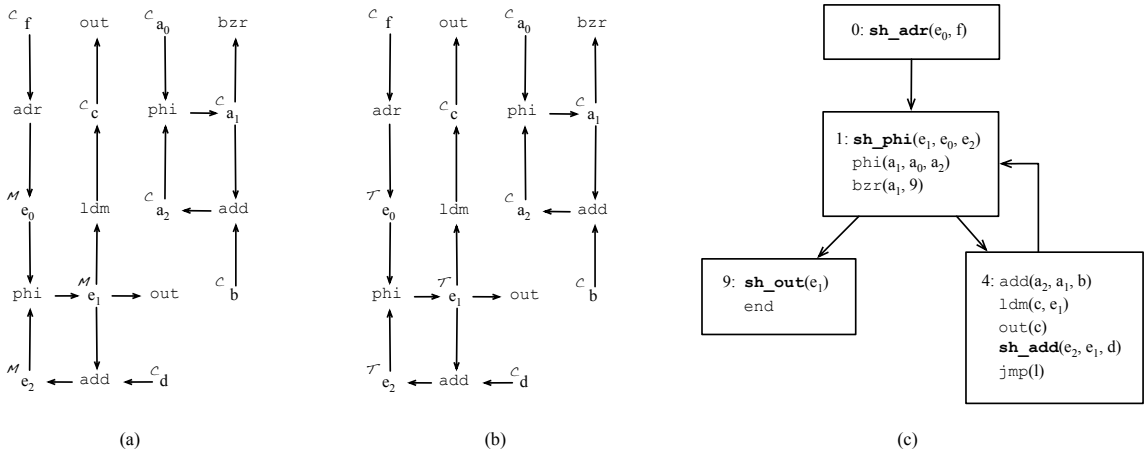

Figura 3.12. (a) Análise *Forward* aplicada no programa visto na Figura 3.4(a). (b) Análise *Backward*. (c) Programa parcialmente instrumentado.

Lema 3.2.2 *Uma variável v tem o tipo* tainted *se (i) ela tem dependência de dados em algum endereço, e (ii) existe* $\texttt{out}(u)$ *tal que u tem dependência de dados em v.*

Prova: Apenas a Regra Ob liga o tipo *tainted* a uma variável v. Para provar (i), pela definição de *tainted* (Fig. 3.8(inferior)), temos que *tainted* $\sqcap_{\leftarrow}$ t = *tainted* se t = *maybe*. Pelo Lema 3.2.1, v terá o tipo *maybe* se ela tem dependência de dados em algum endereço. Para provar (ii), uma análise de caso em cada uma das outras regras mostra que os tipos apenas se propagam de uma variável v_1 para outra variável v_2 se v_1 tem dependência de dados em v_2. □

Teorema 3.2.3 *Se P é um programa angels, e P^γ é tal que $P \xrightarrow{\gamma} P^\gamma$, então $\langle P^\gamma, 0, 0, \lambda x.0, [] \rangle$ não contém um vazamento de endereço.*

Prova: Dos Lemas 3.2.1 e 3.2.2, se uma variável é parte de uma cadeia de dependência entre uma informação de endereço e uma instrução de saída, então ela tem o tipo *tainted*. Pelas regras na Figura 3.11, pode-se observar que toda instrução que define uma variável *tainted* é instrumentada. Assim, qualquer instrução que está em uma cadeia de dependência entre uma informação de endereço e a operação de saída é instrumentada. Conclui-se aplicando o Lema 3.1.4 na fatia instrumentada do programa. □

```
1 void fun1(char *s, int N) {
2     int *p = s;
3     char buf1[] = "hello";
4     char buf2[40];
5     strncpy(buf2, s, N);
6     printf("%s\n", buf1);
7 }
```

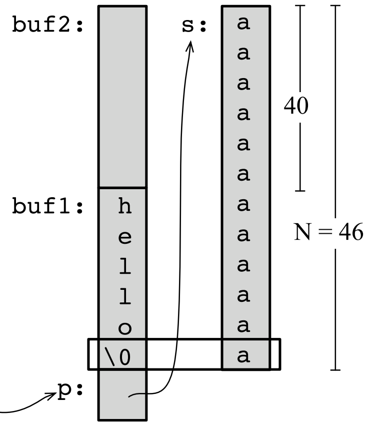

Figura 3.13. Este programa C pode vazar informação. A análise estática não detecta essa vulnerabilidade, devido a semântica mal definida de acessos de fora dos limites de arranjos em C.

3.2.5 Quando a Instrumentação Completa faz mais que a Análise Estática

A sobrecarga imposta pela instrumentação completa, de acordo com os experimentos que são mostrados no Capítulo 4, pode ser muito alta para software em produção. Todavia, a instrumentação completa não sofre das mesmas limitações que existem na análise estática. Essas limitações não são uma deficiência da análise deste trabalho em particular; pelo contrário, elas estão presentes em qualquer análise estática que assuma que o programa-alvo é bem definido. Se o programa que é analisado contém comportamento não-determinístico, então a análise estática pode se basear em suposições erradas. Comportamento não-determinístico pode estar presente em programas C e C++, devido ao sistema de tipos inerentemente inseguro usado nessas linguagens. O programa na Figura 3.13 ilustra esse comportamento.

O programa na Figura 3.13 contém uma vulnerabilidade de *buffer overflow*. Se `N` é maior que o tamanho de `buf2`, então escritas de memória fora dos limites acontecerão na linha 5. Um atacante pode usar essa vulnerabilidade para sobrescrever o *byte* nulo que marca o fim da *string* armazenada em `buf1`. Se essa sobrescrita ocorre, então a função `printf` na linha 6 irá relevar o conteúdo da variável local `p`. Neste exemplo, `p` armazena o endereço do *buffer* `s`. A análise estática não pode detectar essa vulnerabilidade, porque ela depende de uma análise de ponteiros que assume que a memória apontada por `buf1` e `buf2` é disjunta. Entretanto, a ausência de verificações de limite dinâmicas em C quebra essa suposição. Por outro lado, nossa biblioteca de

```
1 void len(char* s) {
2   unsigned w = 0;
3   while(*s != '\0') {
4     w++;
5     s++;
6   }
7   printf("%u\n", w);
8 }
```

```
1 void print_adr() {
2   int s = (int)&s;
3   unsigned x = 0;
4   unsigned m = ~(~0U>>1);
5   while(m) {
6     if (s & m)
7       x |= m;
8     m >>= 1;
9   }
10  printf("%u\n", x);
11 }
```

Figura 3.14. Na função `len`, a variável `w` tem dependência de controle na variável `s`, que carrega informação de endereço. A função `print_adr` contém um vazamento de endereço, mas a instrumentação não o detecta.

instrumentação completa é capaz de detectar esse tipo de vazamento de informação.

3.2.6 Sensibilidade ao Controle

As regras de instrumentação dinâmica mostradas na Figura 3.5 não são *sensíveis ao controle*. Aquelas regras apenas consideram dependências de dados, e não dependências de controle. Para a semântica formal de um monitor dinâmico sensível ao controle, recomenda-se o trabalho de Russo e Sabelfeld [Russo & Sabelfeld, 2010, Figs.9-12]. Foi implementado uma análise dinâmica insensível ao controle para manter a abordagem prática. Se fosse feito de outro modo, então seria necessário lidar com uma alta taxa de falso-positivos, porque muitos laços são controlados por endereços, por exemplo, iteradores em C++, ou ponteiros em C. A Figura 3.14 mostra um exemplo: a variável `w`, na função `len`, tem dependência de controle na informação de endereço armazenada na variável `s`. Entretanto, um atacante não pode inferir qualquer valor de ponteiro a partir da saída produzida pela função `len`.

Pode-se lidar com as dependências de controle mudando a representação da forma SSA para a forma *Gated Static Single Assignment* (GSSA) Ottenstein et al. [1990]. Entretanto, optou-se por não fazê-lo na ferramenta de produção; assim, a implementação é *unsound*. A Função `print_adr`, na Figura 3.14, ilustra uma vulnerabilidade de vazamento de endereço que a ferramenta não detecta. O valor na variável `s` – um endereço – é copiado para a variável `x`, bit-a-bit. Como `x` não é dependente de dados em `s`, as regras na Figura 3.14 não irão marcá-la com o estado contaminado, e um vazamento

não-reportado ocorrerá na linha 10. Todavia, enquanto que a função `print_adr` é improvável de existir em código real, a função `len` é comum o bastante para justificar nossa decisão.

3.3 Conclusão

Esse capítulo descreveu uma solução para detectar vazamentos de endereços em tempo de execução. Foi definida uma linguagem contendo as construções de linguagens imperativas que desempenham um papel na vulnerabilidade de vazamento de endereços. Em cima dessa linguagem foi definida uma *linguagem de instrumentação*. Programas implementados com a sintaxe de instrumentação podem rastrear o fluxo de informação em tempo de execução. Foi descrito um sistema de tipos que detecta vazamentos de endereço estaticamente. Esse sistema de tipos permite-nos reduzir a quantidade de instrumentação necessária para salvaguardar programas contra vazamentos de endereços. A combinação das análises estática e dinâmica permite a produção de código seguro que sofre pouca sobrecarga em tempo de execução.

Capítulo 4

Experimentos Realizados

Esse capítulo apresenta os experimentos realizados para validar o trabalho desenvolvido. As Seções 4.1 e 4.2 discutem o impacto da instrumentação no tamanho e tempo de execução dos programas, respectivamente. A Seção 4.3 apresenta os resultados do desempenho da análise de fluxo contaminado e a implementação da análise de ponteiros. Os experimentos de precisão da análise estática são apresentados na Seção 4.4. Por fim, a Seção 4.5 conclui o capítulo.

Todos os algoritmos descritos nesse trabalho foram implementados no compilador LLVM Lattner & Adve [2004]. Nesse capítulo a implementação é avaliada empiricamente. Os números mostrados nesse capítulo foram obtidos em um processador AMD Athlon(tm) II P340 Dual-Core, com um *clock* de 2.2GHz, 2GB de RAM, executando Linux Ubuntu, versão 12.04.1 LTS. A implementação foi testada em uma suíte de 434 programas, que incluem a suíte completa de testes do LLVM, mais a coleção de *benchmark* do SPEC CPU 2006. No total, foram analisadas quase 2 milhões de linhas de C, que dá 4.3 milhões de instruções na representação intermediária do LLVM (bytecodes).

4.1 O Impacto da Instrumentação no Tamanho do Código

A Figura 4.1 mostra o impacto da biblioteca de instrumentação no tamanho dos 30 maiores *benchmarks* na suíte de testes. Como esperado, a instrumentação completa de um programa aumenta o tamanho do programa instrumentado significativamente, como pode ser visto na parte inferior da Figura 4.1. Considerando a suíte de testes inteira, com 434 programas, o código dos programas instrumentados é 2.46 vezes maior que o código dos programas originais. Considerando apenas os 30 *benchmarks* usados

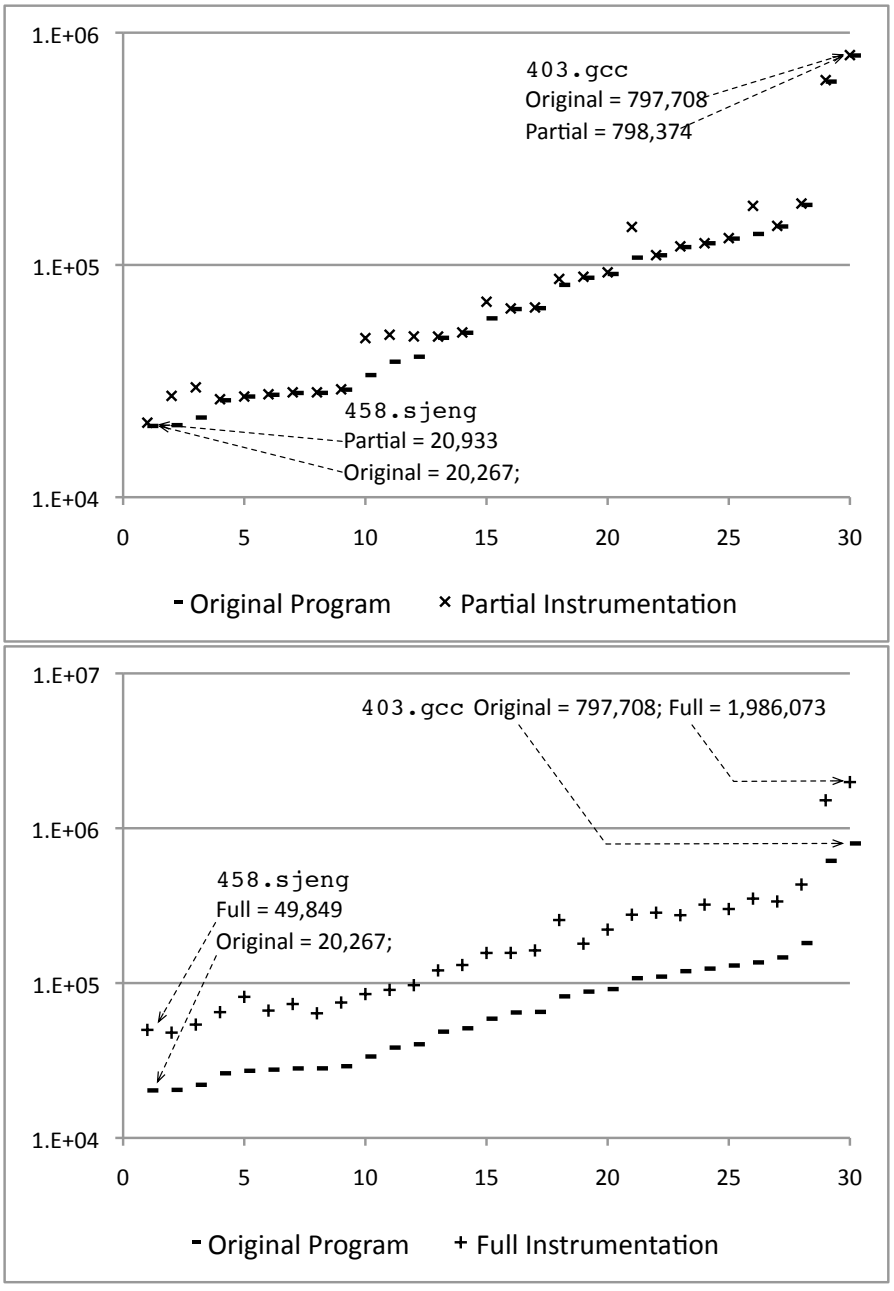

Figura 4.1. (Superior) Aumento no tamanho devido a instrumentação parcial. (Inferior) Aumento no tamanho devido a instrumentação completa. Cada ponto corresponde a um *benchmark*. Tamanhos são dados em número de instruções bytecode do LLVM.

na Figura 4.1 (inferior), então o código aumenta 4.47 vezes.

A parte superior da Figura 4.1 mostra como a análise estática reduz o impacto da instrumentação no tamanho do código. Por meio dos 434 *benchmarks*, observou-

se uma média de expansão de código de apenas 5.17%. Considerando apenas os 30 maiores *benchmarks*, os quais são mostrados na Figura 4.1 (Superior), obtém-se um número similar: um aumento de 5.06%. A instrumentação parcial aumenta também o código dos programas que a análise de fluxo marcou como completamente seguros, por causa das funções da biblioteca que são ligadas com o programa para lidar com os vazamentos. A principal conclusão que pode-se obter dos dois gráficos na Figura 4.1 é que a análise estática é essencial para evitar a explosão de código ao instrumentar programas para prevenir vazamentos de endereços.

4.2 O Impacto da Instrumentação no Tempo de Execução

A Figura 4.2 compara o tempo de execução dos programas originais e instrumentados. A Figura 4.2(inferior) mostra o impacto no tempo de execução da instrumentação completa. Considerando os 30 *benchmarks* vistos na Figura 2.1, observa-se um *slowdown* de 76.88% dos programas completamente instrumentados. Como esperado, a análise estática reduz esse *slowdown* substancialmente. A Figura 4.2(superior) compara o tempo de execução dos programas originais e parcialmente instrumentados. Em média, os 30 programas parcialmente instrumentados são 1.71% mais lentos que suas versões não modificadas.

A Figura 4.3 mostra números estáticos que a análise de fluxo produz para os programas no SPEC CPU 2006. Não existem resultados para `400.perlbench`, porque a distribuição do LLVM utilizada não o compila. O tamanho do grafo de dependência é aproximadamente o número de *bytecodes* na representação intermediária do programa, não contando desvios e nós ϕ. A análise *forward*, por exemplo, as regras na Figura 3.9, marcam cerca de 20-40% dos vértices em tais grafos como possivelmente vazante. A análise *backward*, por outro lado, foi capaz de marcar todo nó no grafo de dependência, exceto `bzip2`, como inócuo. Esse programa - `bzip2` - tem dois caminhos de fontes de informação de endereços para *sinks*.

4.3 Tempo de Execução das Análises Estáticas

A Figura 4.4(Superior) mostra o tempo para executar a análise de fluxo contaminado e a implementação da análise de ponteiros para os 100 maiores *benchmarks* na suíte de teste. A análise de fluxo é linear no tamanho do grafo de dependência do programa. Por isso, ela pode ser quadrática no número de instruções no programa. Entretanto, essa

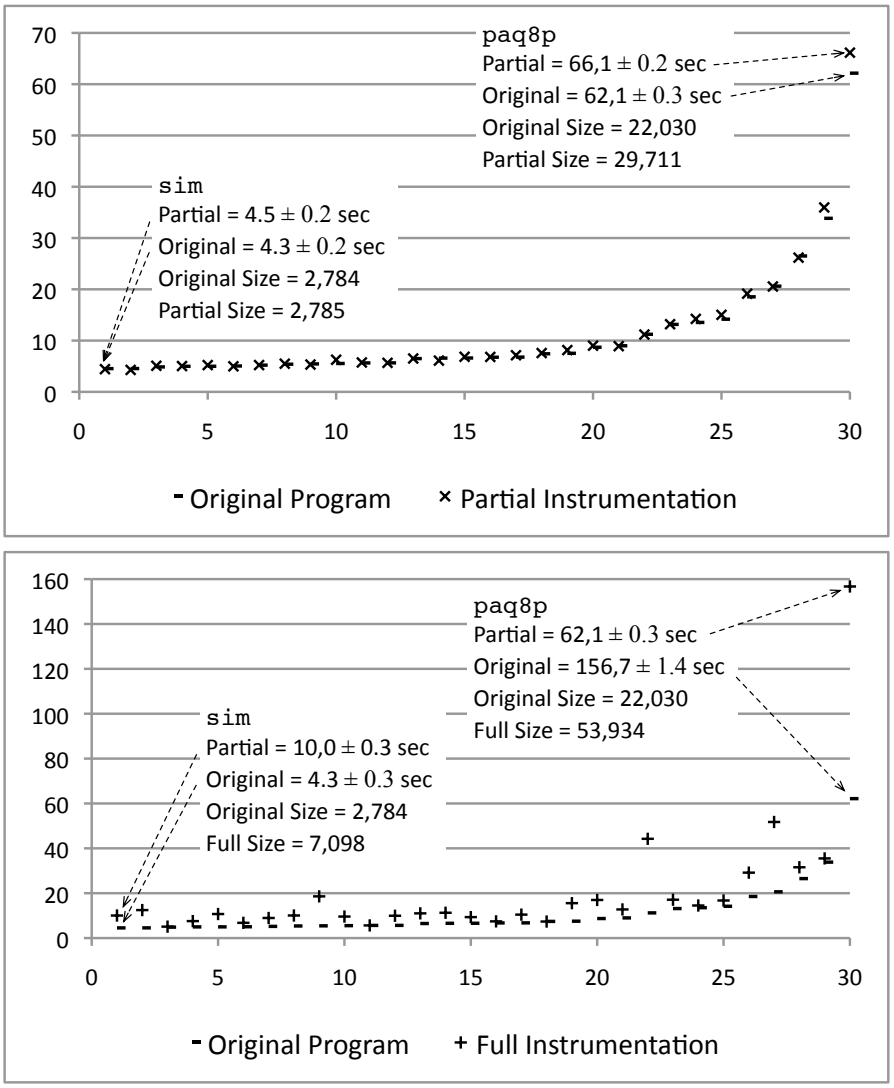

Figura 4.2. (Superior) Tempo de execução dos programas parcialmente instrumentados (sec). (Inferior) Tempo de execução dos programas completamente instrumentados. Cada ponto corresponde a um *benchmark*.

análise é linear na prática, porque esses grafos de dependência tendem a ser esparsos. O tempo para executar a análise de fluxo foi plotado junto com o tamanho dos programas para fornecer uma indicação visual de que a análise é linear na prática. O coeficiente de determinação entre essas duas quantidades, tamanho dos programas e tempo para executar a análise de fluxo, é 0.92, indicando uma forte correlação linear.

Foi também plotado na Figura 4.4(superior) o tempo para executar a análise de ponteiros, para fornecer uma comparação entre a análise de fluxo contaminado e outro algoritmo bem conhecido, Hardekopf & Lin [2007]. Em média, a análise de

Benchmark	I	P	G	$\gamma \rightarrow$
`mcf`	4,725	21	1,114	249
`libquantum`	16,297	25	3,253	770
`lbm`	13,724	10	3,991	613
`astar`	19,243	27	5,303	1,327
`bzip2`	38,831	23	13,150	1,995
`milc`	44,236	548	17,911	5,605
`sjeng`	54,051	191	21,282	4,140
`hmmer`	114,136	50	29,981	6,220
`namd`	100,76	87	58,300	12,028
`soplex`	136,364	0	79,424	31,077
`h264ref`	271,627	86	106,421	14,359
`omnetpp`	203,201	336	114,329	26,147
`gobmk`	308,475	163	130,640	27,005
`dealII`	1,381,408	0	221,305	134,145
`xalancbmk`	1,314,772	0	774,633	212,377
`gcc`	1,419,456	41	805,637	119,532

Figura 4.3. Resultados estáticos para os programas no SPEC CPU 2006. P: número de *sinks*, isto é, `printf`'s no programa-alvo. I: número de instruções x86 produzidas para o programa (cerca de 1.7x maior que o número de bytecodes no programa). G: tamanho do grafo de dependência. $\gamma \rightarrow$: número de nós marcados como *maybe* pela análise *forward*.

ponteiros é 3.4 vezes mais rápida que a análise de fluxo contaminado. A detecção de ciclos preguiçosa de Hardekopf é a característica chave por trás do bom desempenho da implementação de análise de ponteiros. A Figura 4.4(inferior) compara o tempo para executar ambas as análises nos programas do SPEC CPU 2006. Nesse caso, a análise de ponteiros é 3.73 vezes mais rápida que a análise de fluxo de informação.

4.4 A Precisão da Análise Estática

Uma questão importante que precisa ser respondida é quantos alertas produzidos pela análise de fluxo de informação são vulnerabilidades reais. Foram obtidos apenas dois alertas para a suíte de testes inteiro do SPE CPU 2006. Uma inspeção manual desses dois alertas revelou que eles eram vulnerabilidades reais. As funções `spec_fread` (em `bzip2/spec.c:187`), e `spec_fwrite` (em `bzip2/spec.c:262`) imprimem o endereço de *buffers* que elas recebem como parâmetros. Como esse código é executado apenas em modo de depuração, o monitor dinâmico não detectou vazamentos em ambos os casos. Além disso, a análise foi executada na suíte de testes completa do LLVM, obtendo 19 alertas. Foi possível recuperar informação de endereços em 11 casos, que dá uma taxa de acertos de 11/19, no universo de

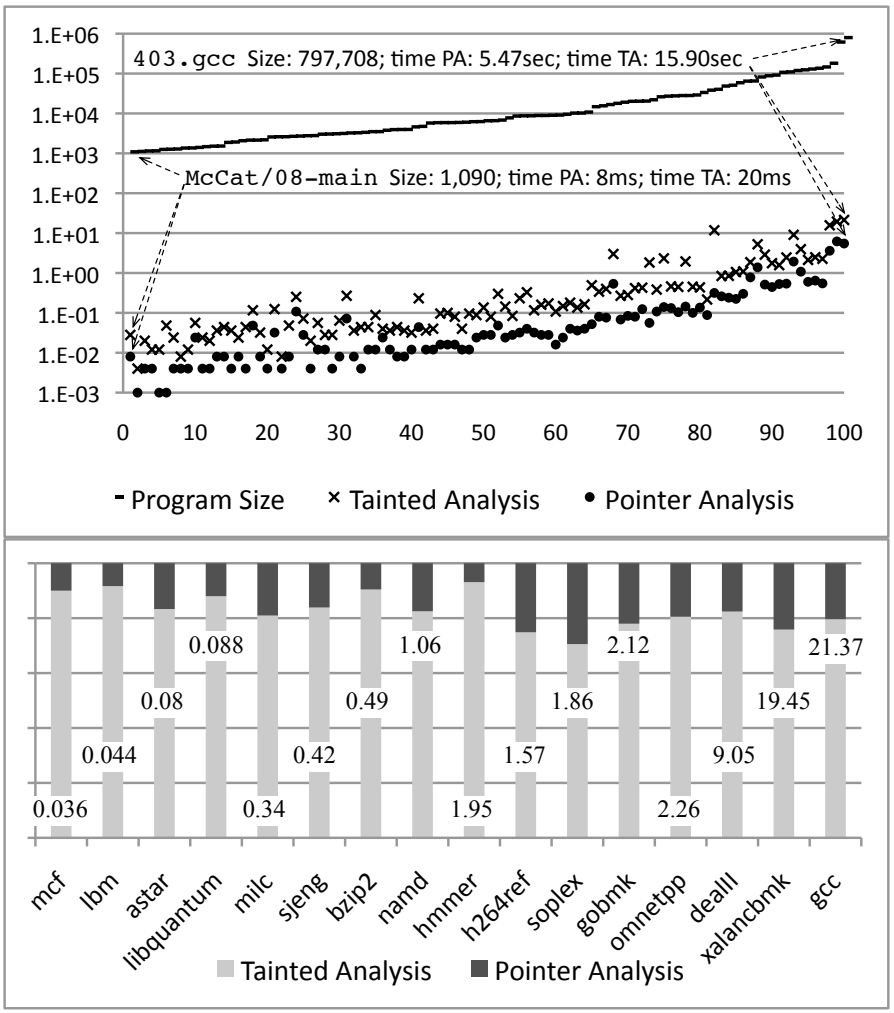

Figura 4.4. (Superior) Tempo (sec) para executar a análise de fluxo contaminado (TA), e a análise de ponteiros (PA), comparado ao tamanho dos programas, dado em instruções bytecode do LLVM. Cada ponto no eixo x representa um *benchmark*. Foram considerados os 100 maiores *benchmarks*. (Inferior) Tempo da análise de ponteiros comparado ao tempo da análise de fluxo contaminado para os programas no SPEC CPU 2006. Números representam o tempo gasto pelas análises de ponteiros e fluxo, em segundos.

434 programas que foram explorados. Os falso-positivos ocorrem devido a dois padrões. O primeiro padrão é a subtração entre ponteiros. Como um exemplo, a análise de fluxo reportou um alerta em `Shootout-C++/moments.cc:81` devido ao comando `printf("n:%d", v.end() - v.begin())`, onde `v.end()` e `v.begin()` são ponteiros. Não foi possível inventar uma forma de recuperar a informação de endereço a partir desse código. O segundo padrão foi a comparação entre ponteiros e `NULL`. Em `2003-05-07-VarArgs.c:59` obteve-se um alerta devido ao comando `printf("LargeS`

{0x%d}", ls.ptr != 0), onde ls.ptr é um ponteiro. A análise também foi testada em uma aplicação maior: a implementação da máquina SpiderMonkey usada para executar programas JavaScript na versão 16.0.1 do navegador Firefox. Foi encontrado um potencial vazamento em jscntxtinlines.h:119. Nesse caso, o comando printf("*** Compartment mismatch %p vs. %p", (void*) c1, (void*) c2) imprime dois endereços.

Os caminhos nos grafos de dependência entre os nós que produzem informação de endereços e os nós que consumem-na, por exemplo, printf, tendem a ser pequenos. O caminho mais longo que foi observados tem 21 *hops*, e ele representa um falso-positivo detectado em SIBsim4. O grafo de dependência desse *benchmark* tem 1.653 nós. O caminho mais curto encontrado tem apenas um *hop*. Ele é uma vulnerabilidade real encontrada em MallocBench/gs, cujo grafo de dependência tem 6.870 nós.

4.5 Conclusão

Esse capítulo apresentou os experimentos realizados para validar o trabalho desenvolvido. Foi discutido o impacto das formas de instrumentação completa e parcial no tamanho e tempo de execução dos programas. Também foram apresentados os resultados de desempenho e precisão da análise estática de fluxo contaminado e da implementação da análise de ponteiros.

Capítulo 5

Conclusão e Trabalhos Futuros

Este trabalho discutiu o problema de vazamento de endereços. Essa vulnerabilidade permite que atacantes burlem o *Address Space Layout Randomization*, um mecanismo de proteção usado em sistemas operacionais populares. Foi mostrado via um exemplo como vazamentos de endereços podem ser explorados na prática, e formalizada a vulnerabilidade via uma linguagem simples, mas Turing completa. Além de introduzir o problema, foi mostrado como tais vulnerabilidades podem ser descobertas estaticamente, e prevenidas dinamicamente. A combinação das análises estática e dinâmica permite a produção de código seguro que sofre pouca sobrecarga em tempo de execução. Além disso, o sistema de tipos criado foi usado para detectar vazamentos de endereços reais em programas muito grandes.

5.1 Trabalhos Futuros

Planeja-se estender esse trabalho em duas direções distintas. Primeiramente, o propósito principal por trás do *framework* de instrumentação é suportar a detecção de vazamentos de endereços via geração de testes automática. Assim, atualmente está sendo trabalhada a integração de nossa biblioteca de instrumentação com um gerador de testes publicamente disponível. Em segundo lugar, apesar de ser capaz de instrumentar corretamente programas grandes, o *framework* de instrumentação é um artefato de pesquisa. Consequentemente, alguns códigos parcialmente instrumentados são ainda muito mais lentos que os programas originais. Para diminuir essa sobrecarga, planeja-se usar otimizações tal como o *code coalescing* de Qin Qin et al. [2006], uma técnica que tem sido desenvolvida para melhorar o tempo de execução de *frameworks* de instrumentação binária.

5.2 Reprodutibilidade

Todos os *benchmarks* usados no Capítulo 4, além do SPEC CPU 2006, estão publicamente disponíveis no site do LLVM. O compilador LLVM é *open source*. Nosso código e demais materiais sobre esse trabalho, incluindo um interpretador Prolog de *Angels*, está disponível no site anônimo: `http://code.google.com/p/addr-leaks/`.

5.3 Contribuições

Além das contribuições enumeradas na Seção 1.1, foram publicados dois artigos e apresentada uma das ferramentas em um evento entre 2011 e 2012. São eles:

- Gabriel Quadros Silva and Fernando Magno Quintão Pereira. *Static Detection of Address Leaks.* The Eleventh Brazilian Symposium on Computer Security (SBSeg). 2011.
- Gabriel Quadros Silva, Rafael M. Souza and Fernando Magno Quintão Pereira. *Dynamic Detection of Address Leaks.* The Twelfth Brazilian Symposium on Computer Security (SBSeg). 2012.
- Gabriel Quadros Silva and Fernando Magno Quintão Pereira. *Static Analysis Tool to Detect Address Leaks.* CBSoft - Sessão de Ferramentas. 2012.
- Gabriel Quadros Silva, Rafael M. Souza and Fernando Magno Quintão Pereira. *The Address Disclosure Problem.* A ser submetido.

Referências Bibliográficas

Allen, J. (2006). *Perl*. Perlsec, 5.8.8 edição.

Balzarotti, D.; Cova, M.; Felmetsger, V.; Jovanovic, N.; Kirda, E.; Kruegel, C. & Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. Em *SP*, pp. 387--401. IEEE Computer Society.

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.

Blanchet, B.; Cousot, P.; Cousot, R.; Feret, J.; Mauborgne, L.; Miné, A.; Monniaux, D. & Rival, X. (2003). A static analyzer for large safety-critical software. Em *PLDI*, pp. 196--207. ACM.

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.

Chebaro, O.; Kosmatov, N.; Giorgetti, A. & Julliand, J. (2012). Program slicing enhances a verification technique combining static and dynamic analysis. Em *SAC*, pp. 1284--1291. ACM.

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

Clause, J.; Li, W. & Orso, A. (2007). Dytan: a generic dynamic taint analysis framework. Em *ISSTA*, pp. 196--206. ACM.

Cytron, R.; Ferrante, J.; Rosen, B. K.; Wegman, M. N. & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.

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

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

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

Godefroid, P.; Klarlund, N. & Sen, K. (2005). Dart: directed automated random testing. Em *PLDI*, pp. 213--223. ACM.

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.

Hardekopf, B. & Lin, C. (2007). The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. Em *PLDI*, pp. 290–299. 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. 40–52. ACM.

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

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

Levy, E. (1996). Smashing the stack for fun and profit. *Phrack*, 7(49).

Nethercote, N. & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. Em *PLDI*, pp. 89--100. ACM.

Newsome, J. & Song, D. X. (2005). Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. Em *NDSS*. USENIX.

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.

Pistoia, M.; Flynn, R. J.; Koved, L. & Sreedhar, V. C. (2005). Interprocedural analysis for privileged code placement and tainted variable detection. Em *ECOOP*, pp. 362--386.

Qin, F.; Wang, C.; Li, Z.; Kim, H.-s.; Zhou, Y. & Wu, Y. (2006). LIFT: A low-overhead practical information flow tracking system for detecting security attacks. Em *MICRO*, pp. 135--148. IEEE.

Rimsa, A. A.; D'Amorim, M. & Pereira, F. M. Q. (2011). Tainted flow analysis on e-SSA-form programs. Em *CC*, pp. 124--143. Springer.

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

Schwartz, E. J.; Avgerinos, T. & Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). Em *Security and Privacy*, pp. 317--331. IEEE.

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.

Stroustrup, B. (2007). Evolving a language in and for the real world: C++ 1991-2006. Em *HOPL*, pp. 1–59. ACM.

Suh, G.; Lee, J. & Devadas, S. (2004). Secure program execution via dynamic information flow tracking. Em *ASPLOS*, pp. 85--96. ACM.

Vogt, P.; Nentwich, F.; Jovanovic, N.; Kirda, E.; Krügel, C. & Vigna, G. (2007). Cross site scripting prevention with dynamic data tainting and static analysis. Em *NDSS*.

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

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

Xu, W.; Bhatkar, S. & Sekar, R. (2006). Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. Em *USENIX Security*. USENIX.

Xu, Z.; Miller, B. P. & Naim, O. (1999). Dynamic instrumentation of threaded applications. Em *PPoPP*, pp. 49--59. ACM.

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.