Uma Representação Intermediária para a Detecção de Vazamentos Implícitos de Informação*

Bruno R. Silva[1], **Fernando M. Q. Pereira**[1], **Leonardo B. Oliveira**[1]

[1]Dep. de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Av. Antonio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brasil

{brunors, fernando, leonardo}@dcc.ufmg.br

Resumo. *O rastreamento preciso e eficiente do fluxo de informação em um programa é um dos grandes temas abordados dentro da segurança computacional. Esse tipo de análise detecta vulnerabilidades como vazamento de segredos ou ataques de fluxo contaminado. Um dos maiores desafios nesse campo é a detecção dos chamados* fluxos implícitos*. Informação trafega implicitamente pelo programa quando dados são influenciados pelo resultado de testes condicionais. Neste artigo propomos uma análise estática que detecta tais fluxos. Ao contrário de trabalhos anteriores, nós o fazemos de forma eficiente. Como testemunho desse pragmatismo, implementamos nossa análise em LLVM, um compilador de qualidade industrial. Fomos capazes de auditar programas cujos grafos de propagação de informação possuem mais de 16 milhões de arestas.*

Abstract. *Information flow tracking is one of the core areas in software security. One of the key challenges in this field is to detect the so called implicit flows. Such flows are caused by conditional tests, which give an adversary the means to discover information about the program by analyzing the paths that it did or did not take during execution. This paper proposes a static analysis that detects implicit information flows. Our analysis is sound and efficient. To demonstrate its effectiveness, we have implemented it in LLVM, an industrial-strength compiler. We have been able to use it to analyze programs having information flow chains with more than 16 million edges.*

1. Introdução

Um dos grandes desafios abordados no campo da segurança computacional é o rastreamento de fluxo de informação no corpo de programas. Por informação entende-se qualquer dado que o programa manipule. Se o adversário alimenta a aplicação sob ataque com dados maliciosos para tomar o controle de operações críticas, diz-se que esse programa é vulnerável a *ataques de fluxos contaminados*. Informação pode também trafegar na direção contrária. Se o programa permite que um adversário tome conhecimento sobre dados sigilosos a partir da leitura de suas funções de saída, então diz-se que esse código possui um *vazamento de informação*. Neste artigo voltamos nossa atenção para esse segundo tipo de vulnerabilidade.

A informação trafega no programa via relações de *dependência*. Essas relações existem em dois sabores: dados e controle. Se o programa contém uma instrução como

*Este artigo foi parcialmente financiado pela Intel e CAPES. Ele faz parte do projeto eCoSoC - Energy-Efficient Code Instrumentation to Secure SoCs Devices.

$v = f(v_1, v_2, \ldots, v_n)$, então há uma dependência de dados de cada $v_i, 1 \leq i \leq n$ para v. Por outro lado, se o valor de uma variável v depende de um teste condicional, então o programa contém uma dependência de controle entre v e p, o predicado que controla aquele teste. Por exemplo, a sequência `p = (a > b); v = p?0:1;` determina uma dependência de controle. Dependências de controle criam os chamados *fluxos implícitos de informação*. Nesse caso, `v` depende de `p`, ou, posto doutro modo, informação flui implicitamente de `p` para `v`. Enquanto o rastreamento de dependências de dado é simples, e tem sido feito com sucesso em diversas análises de fluxo contaminado [Jovanovic et al. 2006, Rimsa et al. 2012, Tripp et al. 2009], fluxos de controle ainda são um desafio.

Em 2006, Hunt e Sands apresentaram um sistema de tipos que é expressivo o suficiente para detectar vazamentos por fluxos implícitos de informação [Hunt and Sands 2006]. Em termos simples, esse sistema associa a cada valor usado no programa um tipo, normalmente H para dados de alta segurança, e L para dados de baixa. Se informação de tipo H pode ser usada em alguma função que um adversário consegue observar, então o programa não passa na verificação de tipos. O sistema de tipos de Hunt e Sands motivou diversos trabalhos na área de segurança computacional, como é possível observar pela sua alta taxa de citação. Esses trabalhos, contudo, permanecem na esfera da computação teórica. O objetivo deste artigo é mover tais ideias para a esfera prática, provendo a primeira descrição detalhada de uma implementação de um sistema de tipos sensível ao fluxo da execução do programa.

A ideia chave deste artigo é modificar o programa sob análise, para que um sistema de tipos insensível ao fluxo de execução obtenha as mesmas propriedades que a técnica proposta por Hunt e Sands. Uma vez construída a representação intermediária do programa, feito que realizamos a nível do compilador, podemos extrair dessa representação um *grafo de dependências*. Esse grafo possui um vértice para cada unidade de armazenamento usada no programa. Uma unidade de armazenamento é uma variável escalar ou um bloco de memória. Uma aresta de u para v indica que a informação em v depende da informação em u. Se construímos esse grafo a partir de um programa qualquer, então obtemos todas as dependências de dados que nele existam. Se o construímos a partir do programa transformado pela nossa técnica, então, além das dependências de dados, encontramos também as dependências de controle. Consequentemente, somos capazes de rastrear os dois tipos de dependências.

Para demonstrar a praticidade de nossa abordagem, nós a implementamos no compilador LLVM, e a usamos para detectar o vazamento de endereços [Quadros and Pereira 2011, Quadros and Pereira 2012]. LLVM é um compilador de qualidade industrial, mantido pela empresa Apple Inc., e usado extensivamente na academia, como plataforma de pesquisa. Fomos capazes, por exemplo, de analisar os *benchmarks* presentes em SPEC CPU 2006, gerando grafos de dependências com até 16 milhões de arestas. Nossa análise revelou-nos todos os vazamentos de endereços originalmente reportados por Quadros e Pereira [Quadros and Pereira 2011], e diversos outros, até então não conhecidos. Esses resultados são descritos em maiores detalhes na seção 4 deste trabalho.

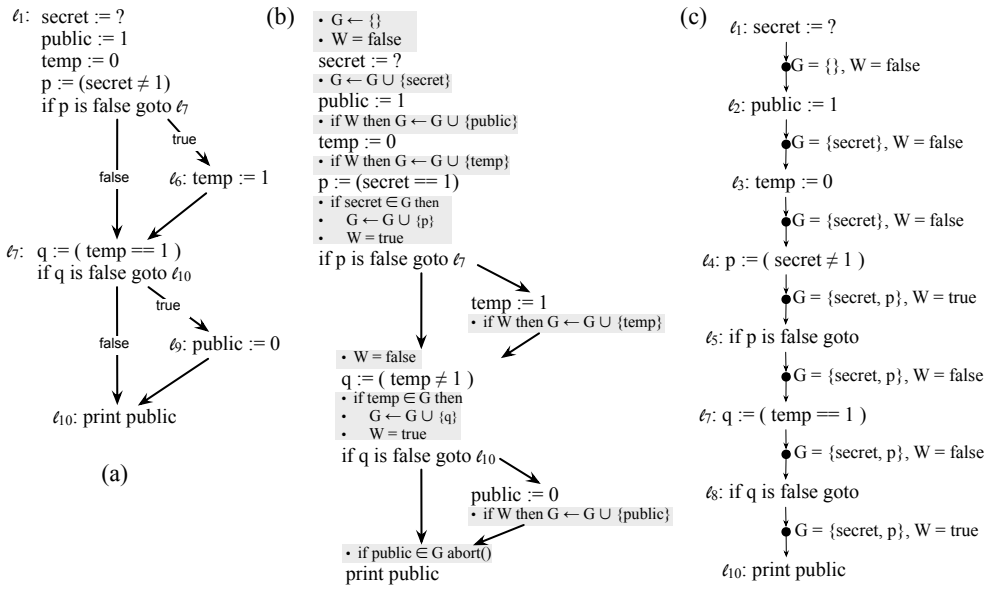

Figura 1. (a) Exemplo de programa que manipula informação sigilosa.

2. Detecção de Vazamento de Informação

Nesta seção definiremos vazamento de informação segundo um modelo simples, descrito informalmente. A seguir abordaremos duas formas de identificar programas passíveis de vazamento de informação. Finalmente, motivaremos o tipo de análise estática que defendemos nesse artigo. Nossa motivação vem do fato de análises dinâmicas, ao contrário de análises estáticas, não serem poderosas o suficiente para detectar todas as formas de vazamento de informação.

2.1. O Problema do Vazamento de Informação

Definimos o problema do vazamento de informação como um jogo, envolvendo um programa P, e uma variável sigilosa `secret`. P é escrito em uma linguagem muito simples, que suporta somente atribuições, operações relacionais ($<$, $\leq$, $>$, $\geq$, $\neq$ e $==$), a função `print`, desvios condicionais e desvios incondicionais. Uma vez que essas instruções possuem semântica óbvia, nós omitiremos a sua formalização em nome da brevidade. Em nosso modelo, adversários podem (i) ler o código fonte de P e (ii) observar todas as informações produzidas pela função `print`. O desafio é inferir qualquer informação acerca de `secret`. A figura 1 (a) contém um programa nessa linguagem. Antes de prosseguir em nossa exposição, convidamos o leitor a responder a seguinte pergunta: "É possível que um adversário conheça o valor de `secret` no programa visto na figura 1, apenas observando o valor de `public` no rótulo l_{10}?"

Existem duas formas de detecção de vazamento de informação: análise estática e análise dinâmica. O método dinâmico demanda a execução do código sob escrutínio. Análises estáticas não fazem tal exigência. Em vez disso, o analisador estático busca encontrar as relações de dependência de dados e controle aparentes pela sintaxe do programa auditado. Ambas as alternativas já foram usadas com grande sucesso, tanto pela academia quanto pela indústria, para a detecção de fuga de informação [Volpano et al. 1996].

Neste artigo nós nos ateremos à análise estática, porém, no que se segue mostraremos um exemplo de análise dinâmica. Valemo-nos, assim, de uma desvantagem dessa segunda alternativa para motivar a nossa implementação da primeira.

2.2. Monitores Puramente Dinâmicos são Inconsistentes

Análises dinâmicas são implementadas por *monitores*. Um monitor M acoplado a um programa P é um meta-programa que registra todas as mudanças de estado causadas por P. A literatura de segurança computacional contém vários exemplos de monitores dinâmicos [Clause et al. 2007, Nethercote and Seward 2007, Zhang et al. 2011]. Análises dinâmicas são mais precisas que abordagens estáticas, porém, sofrem de uma séria desvantagem: a *inconsistência*. Dizemos que uma análise é inconsistente quando ela permite a ocorrência de *falsos negativos*. Um falso negativo acontece quando o analisador dinâmico falha em abortar um programa que deixa escapar informação para o adversário. Um monitor é chamado puramente dinâmico quando ele pode somente registrar mudanças de estado que aconteceram durante a execução do programa. Existe um importante resultado em teoria da informação que diz que qualquer monitor puramente dinâmico é inerentemente falho devido à presença de *fluxos implícitos* de informação [Russo and Sabelfeld 2010].

A figura 1 (b) ilustra essa deficiência de monitores dinâmicos, e serve de motivação para o tipo de análise estática que defendemos neste trabalho. Essa figura mostra o código instrumentado do programa inicialmente visto na figura 1 (a). A instrumentação é a forma padrão de se criar monitores dinâmicos [Volpano 1999]. Uma vez que a instrumentação não influi na lógica do programa monitorado, nós a chamamos de *meta-programação*. Nosso monitor utiliza duas estruturas de dados para registrar mudanças de estado causadas durante a execução do programa. Temos um conjunto G de variáveis sigilosas, que podem conter informação sobre `secret`, e temos um observador booleano W, que rastreia dependências de controle. W é verdade quando o fluxo de execução do programa é desviado por um predicado que pode conter informação sigilosa.

Em nosso exemplo, esse monitor dinâmico falha em detectar o vazamento de informação mostrado na figura 1 (c). Nesse caso, muito embora o valor de `secret` não seja explicitamente copiado para nenhuma variável impressa por `print`, fluxos implícitos revelam informação. Observando o valor de `public` o adversário pode inferir que q é falso, desde que seja impresso o valor um. Se q é falso, o adversário enxerga que `temp` é diferente de um. A partir dessa informação, o adversário conclui que p é falso, e por conseguinte a variável `secret` foi carregada com o valor um. O monitor puramente dinâmico é incapaz de rastrear essa fuga de dados sigilosos, pois o adversário está descobrindo informação secreta a partir dos caminhos que o programa não trilhou, em vez daqueles pelos quais ele fluiu.

2.3. Fluxos Implícitos e Explícitos

A literatura [Russo and Sabelfeld 2010] classifica os fluxos de informação em duas categorias: implícitos e explícitos. Fluxos explícitos, também chamados dependências de dados, devem-se à movimentação de bits de uma posição de memória para outra. Conforme mencionamos na seção 1, se um programa possui uma instrução como $u = f(\ldots, v, \ldots)$, então existe um fluxo explícito de dados de v para u. O programa da figura 1 contém alguns fluxos desse tipo: “p := secret $\neq$ 1”e “q := temp $==$ 1”. Monitores puramente

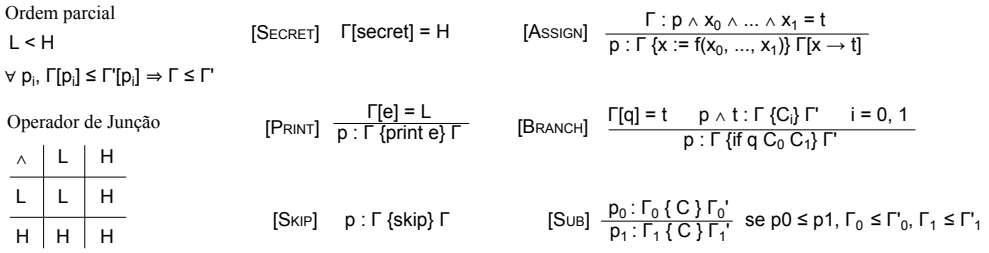

Figura 2. O sistema de tipos proposto por Hunt e Sands [Hunt and Sands 2006].

dinâmicos são suficientemente poderosos para capturar qualquer tipo de fluxo explícito de informação [Hamlen et al. 2006].

Fluxos implícitos existem devido a dependências de controle. Informação flui implicitamente de um predicado p, que controla um teste condicional, para toda variável atribuída no escopo desse teste. A informação sigilosa vaza no exemplo visto na figura 1 devido a um fluxo implícito de informação. Nesse caso, a variável `temp` é implicitamente dependente do predicado p. De modo similar, a variável `public` é implicitamente dependente de q.

Uma vez que a sintaxe de instruções *assembly* não evidencia esse tipo de fluxo, a sua detecção é difícil. Russo e Sabelfeld mostraram, em 2010, que monitores puramente dinâmicos não são poderosos o suficiente para capturar fluxos implícitos. Existem, por outro lado, análises estáticas, como o sistema de tipos proposto por Hunt e Sands [Hunt and Sands 2006], que podem fazê-lo. No restante deste artigo nós mostraremos como esse sistema de tipos pode ser implementado em um compilador industrial. Nosso analisador estático é tão expressivo quanto o modelo teórico de Hunt e Sands. Ao contrário da teoria proposta naquele trabalho, por outro lado, nossa análise foi implementada em um compilador de uso industrial.

3. Rastreamento Consistente de Informação

Nesta seção apresentamos o sistema de tipos proposto por Hunt e Sands em 2006. A partir dessa descrição, mostramos na Seção 3.2 como podemos implementá-lo no nível de instruções *assembly*.

3.1. O Sistema de Tipos de Hunt e Sands

Em 2006, Hunt e Sands projetaram um sistema de tipos que determina, estaticamente, se um programa contém vulnerabilidades de vazamento de informação [Hunt and Sands 2006]. As regras presentes nesse sistema de tipos, adaptadas para a nossa linguagem de instruções *assembly*, podem ser vistas na figura 2. O objetivo desse sistema de regras de inferência é determinar o tipo $\Gamma[v]$ de cada variável v presente no programa. Os tipos possíveis são H e L. O primeiro tipo indica informação com alto grau de sigilo; o segundo indica informação não sigilosa. A verificação de tipos consiste em analisar o programa, a fim de determinar se sua sintaxe atende as regras vistas na figura 2.

Cada regra de tipagem vista na figura 2 descreve como uma instrução modifica Γ, a tabela que mapeia nomes de variáveis para tipos. A notação $\Gamma C \Gamma'$ indica que a

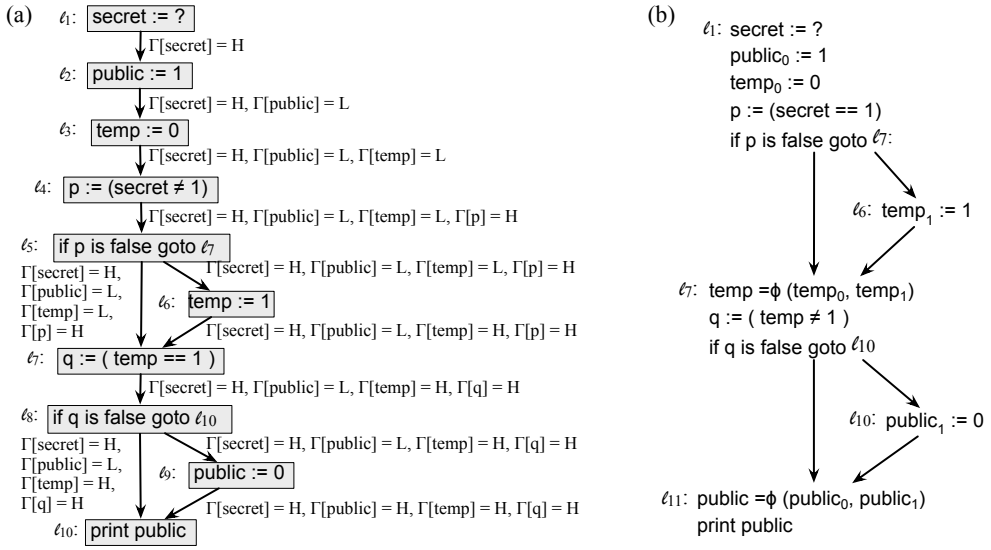

Figura 3. (a) Resultado da aplicação do sistema de tipos de Hunt e Sands no exemplo da figura 1. (b) Programa exemplo convertido para o formato SSA.

instrução C recebe uma tabela Γ e produz outra tabela, Γ'. A notação $\Gamma[v \mapsto t]$, vista na regra ASSIGN indica uma nova tabela Γ' que é igual a Γ para todas as variáveis, menos v. Para a variável v, temos que $\Gamma'[v] = t$. O operador de junção $\wedge$ indica como informação de tipos pode ser combinada. Segundo a sua definição, uma variável de tipo H, quando combinada com qualquer outra variável, produz informação também do tipo H. Assim, ao analisarmos uma instrução como $v = a + b$, temos que o tipo de v é H se o tipo de a ou o tipo de b for H.

Uma diferença entre o sistema de Hunt e Sands e outros sistemas de tipos descritos na literatura [1] é a presença de um predicado p nas regras de tipagem. Esse predicado guarda informação de tipagem criada por fluxos implícitos. Assim, uma instrução como if p then $x := 1$ else $x := 0$ levaria à avaliação das instruções $x := 0$ e $x := 1$ em um ambiente controlado pelo predicado p. A atualização desse predicado ocorre na segunda premissa da regra BRANCH. A notação $p : \Gamma C \Gamma'$ indica que as tabelas Γ e Γ' estão sendo avaliadas em um ambiente controlado pelo predicado p.

Um programa falha a verificação de tipos se ele possui uma instrução do tipo `print` v, sendo $\Gamma[v] = H$. O programa da figura 1 não passa na verificação de tipos. Para evidenciar esse fato, a figura 3 (a) mostra o resultado da aplicação das regras de tipagem para aquele programa. O predicado p possui tipo H, devido a uma dependência direta de `secret` no rótulo l_4. Essa dependência se propaga para `temp` em l_6 e então para q, em l_7. A sequência de propagações de l_8 para l_9 faz com que $\Gamma[\text{public}] = H$ nesse último rótulo. Temos então a junção de $\Gamma[\text{public}] = H$, vindo de l_9 e $\Gamma[\text{public}] = L$, proveniente de l_8. A definição do operador de junção na figura 2 termina por fazer com que $\Gamma[\text{public}] = H$ em l_{10}. Abrimos aqui um aparte para explicar a regra SUB na figura 2. Essa regra permite promover uma tabela de tipos a outra, mais restritiva. Assim, podemos fazer a junção de duas tabelas de tipos diferentes, como foi o caso nesse exemplo.

[1] Para uma descrição ampla de teoria de tipos, recomendamos a obra de Benjamin Pierce [Pierce 2004]

3.2. A implementação concreta do sistema de tipagem

A ideia chave proposta neste artigo é transforma o sistema de tipagem de Hunt e Sands em um problema de busca em grafos. Para tanto, definimos o grafo $G = (V, D \cup C, O)$ de dependências de um programa da seguinte forma:

- Existe um vértice $n_v \in V$ para cada variável v presente no programa.
- Existe um vértice $n_o \in O$ para cada ocorrência da função `print` no programa.
- Temos uma aresta $(n_u, n_v) \in D$ para cada dependência de dados entre duas variáveis, u e v.
- Temos uma aresta $(n_v, n_o) \in D$ se v é uma variável usada em uma função `print` o.
- Temos uma aresta $(n_p, n_v) \in C$ entre p, o predicado que controla um desvio condicional, e toda variável v definida na *região de influência desse desvio*.
- Temos uma aresta $(n_p, n_o) \in C$ entre p, o predicado que controla um desvio, e cada função `print`, isto é, o, usada na região de influência de p.

Essa definição de grafo de dependências demanda a noção da região de influência de um predicado. A região de influência de um predicado p, que controla um desvio condicional posicionado em um rótulo l_p do programa é um conjunto de rótulos. Esses rótulos incluem l_p, o seu *pós-dominador* l_d, e todos os demais rótulos entre esses dois nós. Diz-se que um rótulo l_d pós-domina um rótulo l_p se qualquer caminho no programa, entre l_p e sua última instrução, passa por l_d. Por exemplo, na figura 3 (a), temos que o pós-dominador de l_5 é l_7. Portanto, a região de influência de l_5 é $\{l_5, l_6, l_7\}$.

A necessidade de variáveis com estados invariantes. Nosso maior desafio é fazer com que o grafo de dependências seja uma representação adequada do problema de vazamento de informação. Para tanto, precisamos garantir que o estado de uma variável, isto é, seu tipo, seja invariante em qualquer parte do texto do programa. Essa propriedade não existe, em geral. Por exemplo, no programa visto na figura 3 (a), temos que $\Gamma[\texttt{temp}] = L$ entre l_3 e l_4. Porém, $\Gamma[\texttt{temp}] = H$ entre l_6 e l_7. Fenômeno similar ocorre com a variável `public`, pois ela é definida com o tipo L em l_2, e com o tipo H em l_9. Fortuitamente, existe uma representação de código, popular entre compiladores modernos, que nos permite lidar com essa dificuldade.

A fim de obter estados invariantes para cada nome de variável definido no programa, nós recorremos a uma representação intermediária de código chamada *Formato de Atribuição Estática Única*, ou SSA [2]. Essa forma de representar programas, inventada no final da década de 80 [Cytron et al. 1989], é hoje chave para dezenas de otimizações de código. Esse formato possui a propriedade de que cada nome de variável é definido em somente um ponto no texto do programa. Uma vez que o tipo de uma variável é determinado pelas operações e contexto usados no ponto de sua definição, o formato SSA se presta perfeitamente aos nossos propósitos.

A figura 3 (b) mostra nosso exemplo corrente convertido para a representação SSA. Esse novo programa possui duas particularidades. Primeiro, variáveis antes definidas múltiplas vezes foram renomeadas. Segundo, temos instruções especiais, denominadas funções ϕ unindo várias definições em um único nome. Essas instruções, uma abstração notacional, funcionam como multiplexadores. Por exemplo, se o fluxo

[2]Do inglês *Static Single Assignment* form

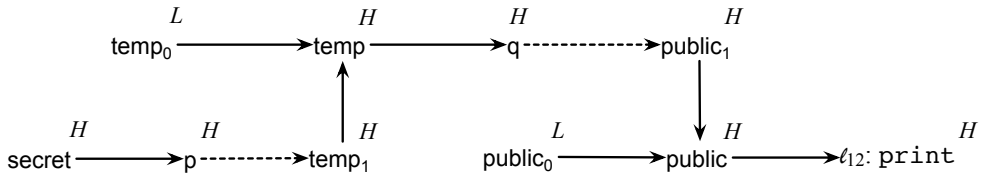

Figura 4. Grafo de fluxo de informação criado para o programa visto na figura 3 (b). O tipo de cada variável é mostrado ao lado do nó que a representa.

de execução do programa alcança l_{11} vindo de l_9, na figura 3 (b), então a função public $:= \phi(\text{public}_0, \text{public}_1)$ copia a variável public_0 para public.

A figura 4 mostra o grafo de fluxo criado para o programa visto na figura 3 (b). Arestas representando fluxos explícitos são sólidas, enquanto arestas representando fluxos implícitos são tracejadas. Junto a cada variável nós mostramos o seu tipo, L ou H. O tipo de uma variável é H se ela for alcançável a partir de `secret`. Desse modo, o grafo descreve um programa vulnerável se existe um caminho entre `secret` e alguma função `print`. Em nosso exemplo, esse é o caso. Note que esse caminho vulnerável existe porque estamos levando dependências de controle, isto é, fluxos implícitos, em consideração.

Análise Inter-Procedural. Há alguns detalhes de nossa implementação que acreditamos merecer destaque. O primeiro deles é a forma como lidamos com funções; o segundo é a maneira como atravessamos os grafos de fluxo. Um programa normalmente é dividido em muitas funções independentes. Porém, uma análise como a nossa precisa ser capaz de enxergar o programa por inteiro. Assim, a nossa análise de fluxo de informações é *inter-procedural*. Para alcançar a inter-proceduralidade, nosso grafo de fluxos possui as seguintes arestas ligando variáveis definidas em funções diferentes. Para cada invocação $x = f(y)$, sendo f uma função declarada como $f(a)\{\ldots, \text{ret } b\}$, criamos as seguintes arestas:

- Uma aresta de dependência de dados (n_y, n_a), para cada parâmetro real y copiado para um parâmetro formal a.
- Uma aresta de dependência de dados (n_b, n_x), entre a variável b que a função retorna, e a variável x que espera o valor de retorno.

Adicionalmente, se a invocação de f acontece dentro da região de influência de algum predicado p, então temos arestas de controle extras:

- uma aresta (n_p, n_v) é criada, denotando uma dependência entre p e v, em que v é qualquer variável definida no escopo de f.

A Busca por Caminhos Vulneráveis. O segundo detalhe de implementação que julgamos merecer explicação é a forma como encontramos e reportarmos caminhos vulneráveis no programa. Para cada par formado por uma fonte de informação sigilosa f, e um canal que o adversário pode ler c, nós reportamos todos os caminhos, dentro do programa, entre f e c. Um caminho é uma sequência de de nós no grafo de fluxos. Cada um desses nós possui uma indicação para a linha, no programa fonte, que lhe deu origem. Uma vez que reportamos todos os caminhos entre f e c, nossa análise encontra, para cada um desses pares, um subgrafo $G' \subseteq G$. G' é formado por todos os nós entre f e c.

Encontramos o subgrafo G' via duas buscas em profundidade. A primeira dessas buscas parte de f, marcando todos os nós alcançáveis a partir daquele vértice. Cada um desses nós passa a fazer parte de um conjunto S_f. A segunda busca parte de c, e marca todos os nós *inversamente* alcançáveis, definindo um conjunto S_c. Em outras palavras, essa segunda busca atravessa o grafo de fluxos no sentido inverso da direção das arestas. O subgrafo vulnerável é formado pelo conjunto interseção $S_f \cap S_c$. Uma vez que realizamos essas buscas para cada par $(f \times c)$, a complexidade de nossa análise é $O(|G| \times f \times c)$. Essa é uma complexidade relativamente alta. Por outro lado, se estivéssemos somente interessados em nós vulneráveis de G, então poderíamos reduzir essa complexidade para $O(|G|)$. Nessa definição mais simples do problema de vazamento de informação, uma vez visitado um vértice, ele pode ser memorizado, não mais havendo necessidade de posteriores visitas.

4. Resultados Experimentais

A partir das idéias discutidas neste trabalho, nós projetamos e implementamos uma ferramenta de detecção de vazamento de informação, a qual apelidamos `FlowTracking`. `FlowTracking` possui licença pública, e está disponível para *download*. A implementação que usamos foi construída sobre o compilador LLVM [Lattner and Adve 2004], versão 3.3, disponível em Junho de 2013. Todos os testes que mostraremos nesta seção foram feitos em uma máquina Intel Xeon, com 133 GB de memória RAM, e *clock* de 2.4 GHz. O sistema operacional usado foi Linux Ubuntu, versão 12.04.2 LTS. Nós fomos capazes de testar nossa estratégia nos 445 programas disponíveis no arcabouço de testes de LLVM. Nesta seção mostraremos resultados somente para os *benchmarks* presentes em SPEC CPU 2006, uma vez que essa é uma coleção amplamente adotada no teste de *software* básico.

O Problema do Vazamento de Endereços. Em nossos experimentos decidimos usar o problema de *vazamentos de endereços* como um caso de uso. Consideramos como fontes as funções de `libc` que alocam espaço em memória, tais como `malloc`, `calloc` e `realloc`. Consideramos "saídas"quaisquer funções que escrevam dados em canais públicos, como `printf`, `putc` e `fputc`. Nosso arcabouço pode ser facilmente configurado para lidar com outros tipos de fontes, e outros tipos de sorvedouros de informação. Antes de analisarmos nossos resultados experimentais, explicaremos o por quê de endereços serem informação sigilosa.

Sistemas operacionais modernos utilizam um mecanismo de proteção denominado *embaralhamento aleatório de endereços* [Bhatkar et al. 2003, Shacham et al. 2004] (ASLR) [3]. Segundo essa técnica, o endereço base onde um programa é carregado pode mudar de uma ativação para outra. Esse mecanismo de proteção diminui a efetividade de uma forma de ataque de *software* conhecida como *programação orientada ao retorno* [Shacham 2007] (ROP) [4]. Ataques do tipo ROP buscam divergir o endereço de retorno de um função para algum código de implementação conhecida. Adversários usualmente escolhem como alvos funções de sistema implementadas na biblioteca `libc`, quando explorando sistemas da família UNIX. Sistemas protegidos via ASLR diminuem a efetividade desse tipo de ataque por que forçam o adversário a supor um endereço correto entre milhões de possibilidades. Ainda que ataques exaustivos possam, em teoria, ser

[3]Tradução livre da expressão inglesa *Address Space Layout Randomization*.

[4]Do inglês *Return Oriented Programming*.

benchmark	Instruções	Vértices	Arestas de Dados	Arestas de Controle	Total de Arestas	Fontes	Saídas
mcf	2.556	$3,8K$	$6K$	$10,1K$	$16,1K$	4	26
libquantum	6.435	$9,8K$	$15,1K$	$27,8K$	$42,9K$	19	50
astar	8.646	$13,2K$	$20,5K$	$4,5K$	$6,5K$	22	27
omnetpp	91.146	$156,4K$	$239,2K$	$3,4M$	$3,6M$	3	65
hmmer	67.528	$107,4K$	$150K$	$580,4K$	$730,4K$	45	451
xalancbmk	588.502	$990,2K$	$1,4M$	$2,1M$	$3,5M$	0	14
sjeng	30.012	$43,8K$	$62,6K$	$1,3M$	$1,4M$	10	224
bzip2	17.324	$27,7K$	$41,5K$	$436,3K$	$477,9K$	5	84
perlbench	283.594	$434,9K$	$607,5K$	$15,5M$	$16,1M$	8	10
gobmk	144.267	$227,6K$	$332,6K$	$2,1M$	$2,4M$	22	471
h264ref	143.804	$234,7K$	$340K$	$1,1M$	$1,4M$	169	220

Figura 5. Estatísticas relacionadas aos grafos de dependência construídos para cada benchmark.

feitos, em geral uma suposição errada tende a terminar o programa sob ataque. Por outro lado, se adversários conseguem inferir um endereço interno do programa alvo, então eles podem realizar um ataque do tipo ROP com total acurácia. Assim, consideramos que um programa possui um vazamento de informação se ele deixa escapar um endereço para algum canal de leitura pública.

Escala das Buscas Realizadas. A figura 5 mostra informações sobre os grafos de fluxo que encontramos. Esses números nos dão uma idéia sobre a escala das buscas feitas para encontrar vazamentos de informações. Em geral, os grafos de fluxo contêm 1.6 vértices para cada instrução presente no programa *assembly*. Existem mais vértices que instruções por que algumas instruções são removidas devido a otimizações de código, após nossa análise de fluxo de informação. Os grafos que levam em consideração somente fluxos explícitos, isto é, dependências de dados, são muito esparsos. Temos em média 1.4 arestas para cada vértice. Esse resultado é esperado, uma vez que o número de definições e usos de variáveis em um programa tende a ser semelhante. Por outro lado, os grafos que levam em consideração fluxos implícitos de informação são muito mais densos. Neste caso, temos cerca de 40 arestas para cada vértice. Conforme podemos observar na figura 5, temos poucas fontes, isto é, operações que geram endereços. Funções de saída são uma ordem de magnitude mais comuns que fontes de informação. E tanto fontes quanto sorvedouros correspondem a uma parcela ínfima dos grafos de fluxo: menos que 1% dos vértices recebem algum desses papéis. A principal conclusão que tiramos dessas observações é que fluxos implícitos impactam de forma não negligenciável a escalabilidade de análises de vazamento de informações. Para corroborar essa conclusão, a figura 6 mostra a percentagem de arestas de dados e arestas de controle nos grafos de fluxo.

Quantidade de Avisos Reportados. A figura 7 nos mostra a quantidade de avisos que reportamos, com e sem considerar fluxos implícitos de informação. Considerando somente fluxos explícitos de dados, obtivemos 8,942 avisos para os programas em SPEC Cint 2006. Um aviso é um par (fonte, sorvedouro), seguido de todos os caminhos, no código fonte do programa – linhas de código C – que levam da fonte ou sorvedouro. Ao levar fluxos implícitos em consideração, obtivemos 28,755 avisos. A partir desse resul-

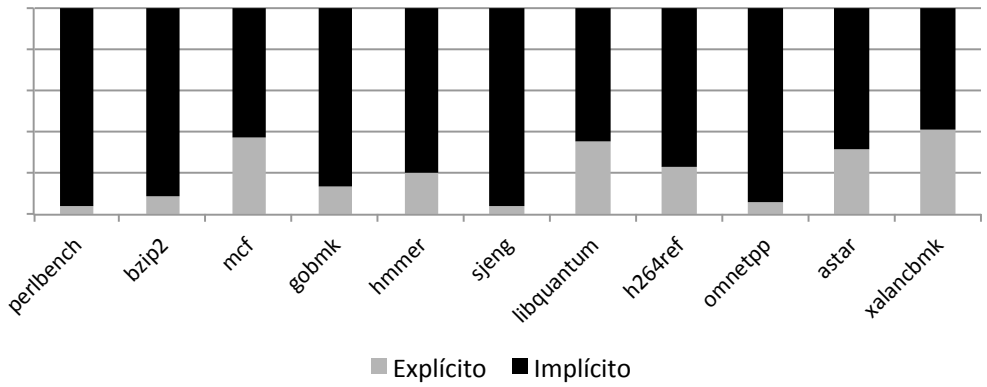

Figura 6. Distribuição das arestas de controle e de dados nos grafos de fluxo.

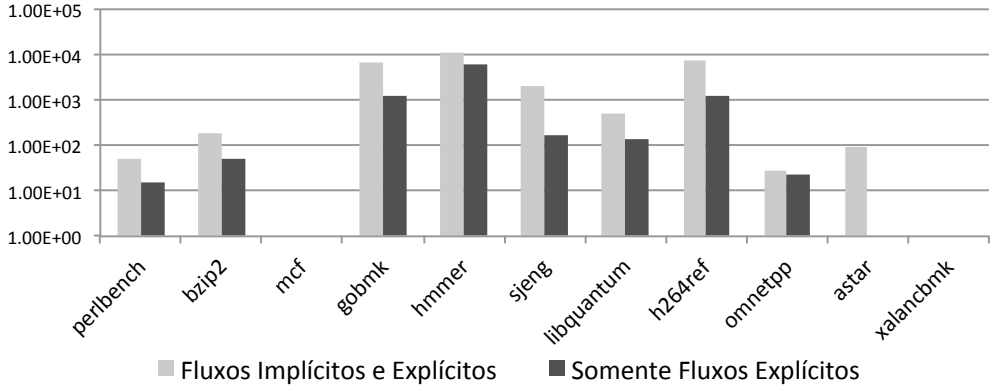

Figura 7. Número de avisos de vazamento de informação. O gráfico está usando escala logarítmica.

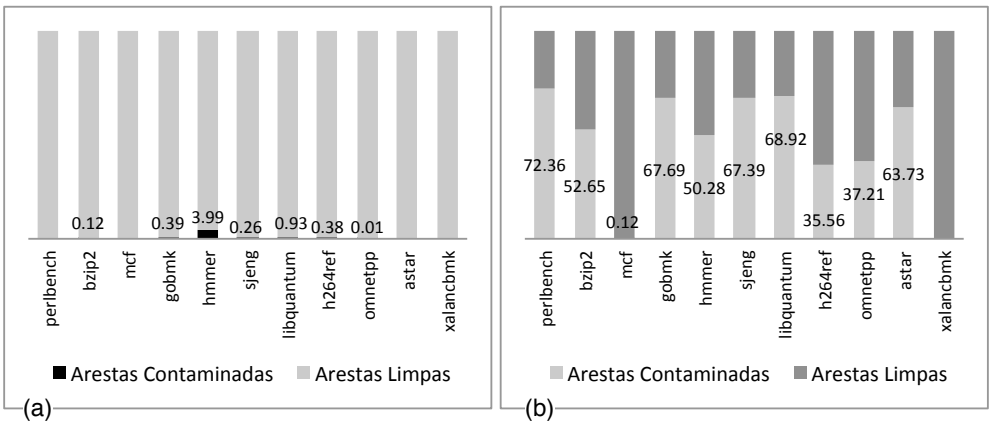

Figura 8. Percentual de arestas limpas com relação às arestas vulneráveis. (a) Somente fluxos explícitos. (b) Fluxos explícitos e implícitos.

tado, concluímos que análises que não consideram fluxos implícitos podem apresentar uma alta taxa de falsos negativos, deixando de reportar vulnerabilidades reais.

Uma das mais importantes perguntas que respondemos neste trabalho é: "qual a percentagem do grafo de fluxos que faz parte de algum caminho por onde vaza informação." Essa pergunta é relevante porque uma das serventias de nossa análise é diminuir a instrumentação necessária para guardar programas contra vazamento de informação. Partes do programa que não fazem parte de qualquer caminho vulnerável não precisam ser guardadas. A figura 8 mostra que ao considerarmos somente fluxos explícitos, temos de guardar, em média, menos que 1% do programa. Por outro lado, levando fluxos implícitos em consideração, temos de guardar, em média, 47% do programa.

5. Trabalhos Relacionados

Existe uma literatura extensa sobre o problema da detecção de vazamento de informação. Alguns pesquisadores buscam descobrir vazamentos de informação via técnicas de análise estática. Outros recorrem à análise dinâmica. E há também os grupos que usam uma combinação de ambos os métodos. Essas técnicas, em geral, não são equivalentes. Algumas abordagens são mais conservadores e reportam que programas seguros são vulneráveis. Outras são mais liberais, e deixam de reportar vulnerabilidades reais. Para um apanhado geral sobre a área, recomendamos o trabalho de Hammer e Snelting [Hammer and Snelting 2009]. Nós acreditamos que nosso trabalho difere de pesquisas anteriores porque ele é uma implementação concreta de um sistema de tipagem expressivo o suficiente para detectar fluxos implícitos de informação. Esse tipo de análise, até a presente data, nunca foi aplicada a programas muito grandes. Pudemos, pela primeira vez na literatura da área, perceber a escala e complexidade desse tipo de análise.

As primeiras análises estáticas propostas para detectar vazamento de informação levavam em consideração somente dependências de dados [Denning and Denning 1977]. Essas análises sofrem de falsos negativos, deixando de reportar vulnerabilidades devido a fluxos implícitos de informação. Monitores puramente dinâmicos são ainda mais permissivos que esse tipo de análise estática. Esses monitores, como os propostos por Sabelfeld e Russo [Sabelfeld and Russo 2009], são isentos de falsos negativos: todo aviso é uma vulnerabilidade, de acordo com um modelo semântico da linguagem de programação usada. Entretanto, conforme explicamos na seção 2.2, monitores puramente dinâmicos são incapazes de identificar todo vazamento de informação.

A análise estática proposta por Hunt e Sands [Hunt and Sands 2006] descobre vazamentos tanto devido a fluxos implícitos quanto explícitos. Contudo, neste trabalho pudemos observamos que, em situações práticas, a sua taxa de falsos positivos é muito alta. Em outras palavras, esse tipo de análise tende a indicar vulnerabilidades em programas que, em realidade, são seguros. Russo e Sabelfeld propuseram, em 2010, um tipo de monitor dinâmico que reporta as mesmas vulnerabilidades que o sistema de tipos de Hunt e Sands [Russo and Sabelfeld 2010]. Esse tipo de monitor é de difícil implementação, e nunca chegou a ser testado em programas reais.

O Problema de Vazamento de Endereços. A motivação para nossa seção de experimentos foi o problema de vazamento de endereços, estudado por Quadros e Pereira [Quadros and Pereira 2011, Quadros et al. 2012]. Esse problema ainda é pouco discutido na academia, porém ele é bem conhecido por profissionais de segurança e aficionados. A título de exemplo, Dion Blzakis explica como usar a inferência de ponteiros

para comprometer um compilador de ActionScript [5]. A inferência de ponteiros também é mencionada por Fermin Serna como uma vulnerabilidade potencial [6].

6. Conclusão

Esse trabalho apresentou a primeira implementação do sistema de tipos de Hunt e Sands em um compilador de uso industrial. Implementações anteriores eram protótipos, incapazes de lidar com programas reais. A nossa técnica é simples e eficiente. Sua elegância deve-se à ideia de reduzir aquele sistema de tipos a um problema de busca em grafos. Para tanto, vale-mo-nos de uma representação intermediária de programas conhecida como formato de atribuição estática única. Resultados experimentais mostraram que nossas ideias são efetivas, pois pudemos analisar programas com milhões de instruções *assembly* em tempo hábil. Nosso passo seguinte nessa linha de pesquisa é a instrumentação parcial de programas. Cenas desse próximo capítulo são descritas a seguir.

Instrumentação parcial de programas: A principal vantagem de um sistema de detecção de fluxo de informação como o que apresentamos neste trabalho é sua consistência. Se nossa análise informa que uma parte do programa é livre de vazamento de informação, então sabemos que esse código não deixa conhecimento escapar para um adversário de forma explícita ou implícita. Isso posto, nossa técnica é mais útil devido aos vazamentos que não reporta, do que devido aos avisos que gera. A título de exemplo, um método bem conhecido de rastreamento de informação consiste na instrumentação de código que discutimos na seção 2.2. O programa instrumentado tem todas as transferências de dados que produz registradas durante sua execução. Essa estratégia, obviamente, é muito custosa em termos de desempenho, podendo diminuir a velocidade do programa instrumentado em até 100x [Zhang et al. 2011]. Nós clamamos, neste artigo, que somente a parte considerada vulnerável por nossa análise precisa ser instrumentada. Estamos atualmente trabalhando na implementação desse tipo de instrumentador parcial, e os resultados preliminares mostram-se animadores.

Reproducibilidade: a nossa ferramenta de detecção de vazamento de informação está disponível em repositório público: `https://code.google.com/p/ecosoc/` Esse repositório também contem um breve tutorial sobre como usá-la.

Referências

Bhatkar, E., Duvarney, D. C., and Sekar, R. (2003). Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, pages 105–120.

Clause, J., Li, W., and Orso, A. (2007). Dytan: a generic dynamic taint analysis framework. In *ISSTA*, pages 196–206. ACM.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An efficient method of computing static single assignment form. In *POPL*, pages 25–35.

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

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

[6] *The info leak era on software exploitation* – conjunto de slides disponíveis on-line.

Hamlen, K. W., Morrisett, G., and Schneider, F. B. (2006). Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205.

Hammer, C. and 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.

Hunt, S. and Sands, D. (2006). On flow-sensitive security types. In *POPL*, pages 79–90. ACM.

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

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

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM.

Pierce, B. C. (2004). *Types and Programming Languages*. MIT Press, 1st edition.

Quadros, G. S. and Pereira, F. M. Q. (2011). Static detection of address leaks. In *SBSeg*, pages 23–37.

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

Quadros, G. S., Souza, R. M., and Pereira, F. M. Q. (2012). Dynamic detection of address leaks. In *SBSeg*, pages 61–75.

Rimsa, A. A., D'Amorim, M., Pereira, F. M. Q., and Bigonha, R. (2012). Efficient static checker for tainted variable attacks. *Science of Computer Programming*, 13(2):2–24.

Russo, A. and Sabelfeld, A. (2010). Dynamic vs. static flow-sensitive security analysis. In *CSF*, pages 186–199. IEEE Computer Society.

Sabelfeld, A. and Russo, A. (2009). From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conference*.

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

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *CSS*, pages 298–307. ACM.

Tripp, O., Pistoia, M., Fink, S., Sridharan, M., and Weisman, O. (2009). TAJ: Effective taint analysis of web applications. In *PLDI*, pages 87–97. ACM.

Volpano, D., Irvine, C., and Smith, G. (1996). A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187.

Volpano, D. M. (1999). Safety versus secrecy. In *SAS*, pages 303–311. Springer-Verlag.

Zhang, R., Huang, S., Qi, Z., and Guan, H. (2011). Combining static and dynamic analysis to discover software vulnerabilities. In *IMIS*, pages 175–181. IEEE Computer Society.

