Uma Técnica de Análise Estática para Detecção de Canais Laterais Baseados em Tempo

Bruno R. Silva[1], **Diego Aranha**[2], **Fernando M. Q. Pereira**[1]

[1]Dep. de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

[2]Inst. de Computação - Universidade Estadual de Campinas (UNICAMP)

{brunors, fernando}@dcc.ufmg.br, dfaranha@ic.unicamp.br

Abstract. *A time-based side-channel is a vulnerability related to implementations of cryptographic systems which allow an adversary to obtain secret information through detailed observations of a program's execution time. Masking and type systems have been proposed as strategies to mitigate this problem. This article proposes an alternative approach focused on static information flow analysis. We have applied it in NaCl and portions of the OpenSSL library where it was possible to validate the good quality of NaCl and to report several vulnerable traces in OpenSSL.*

Resumo. *Canais laterais baseados em tempo são vulnerabilidades ligadas à implementação de sistemas criptográficos e que permitem ao adversário conhecer acerca de uma informação sigilosa através de minuciosas observações do tempo de execução do programa. Mascaramento e sistemas de tipos já foram propostos objetivando mitigar esse problema. Este artigo propõe uma alternativa focada em análise estática de fluxo de informação. Aplicou-se essa análise na biblioteca NaCl e em porções da OpenSSL onde foi possível validar a boa qualidade da primeira e reportar vários traços vulneráveis na segunda.*

1. Introdução

Nas últimas décadas, pesquisadores descobriram que a resiliência de um algoritmo criptográfico depende não somente de seu projeto abstrato, mas também de sua implementação concreta [Kocher 1996]. Em termos de projeto, um algoritmo criptográfico deve estar livre de vulnerabilidades teóricas que poderiam permitir a um adversário a chance de decifrar suas mensagens codificadas sem o devido direito a isso. Em termos de implementação, tal algoritmo deve estar livre de canais laterais. Um ataque por canal lateral busca coletar informações sigilosas relacionadas à chave ou estados internos secretos da implementação. Existem alguns tipos de ataque nesse sentido. Em alguns deles, adversários devem ter profundo acesso ao sistema criptográfico alvo, de forma que eles possam inserir falhas ou recuperar dados residuais da memória. Ataques menos invasivos também existem. Eles tentam detectar diferenças sensíveis no tempo de execução [Kocher 1996], consumo de potência [Kocher et al. 1999], variações no campo eletromagnético [Quisquater and Samyde 2001] ou ainda emanações acústicas [Genkin et al. 2014].

Canais laterais baseados em tempo permitem a um adversário monitorar pequenas flutuações no tempo de execução do algoritmo criptográfico. Essas variações são devidas

aos desvios condicionais, otimizações no nível de instruções, desempenho da hierarquia de memória ou latência de comunicação. Ataques por análise de variação de tempo podem ser devastadores contra essas implementações inseguras. Por exemplo, eles são muito efetivos contra implementações ruins de *square-and-multiply* do algoritmo RSA e Diffie-Hellman ou software baseado em tabela da implementação do algoritmo AES. Contrário à crença popular, mesmo o ruído das conexões de rede não é suficiente para dificultar o vazamento de informação por análise de tempo.

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

Neste artigo, propõe-se uma solução para esse problema na forma de uma análise estática de fluxo de informação para a detecção de canais laterais baseados em tempo. A técnica aqui descrita aponta fluxos de dados secretos para instruções de desvio ou indexação de memória. A análise de fluxo é incorporada no compilador e atua sobre a representação intermediária do programa. Portanto, é possível identificar também os canais laterais introduzidos pelo próprio compilador. Além disso, por atuar no nível de linguagem de montagem, essa abordagem é capaz de lidar com programas não estruturados, isto é, aqueles que fazem uso indiscriminado de instruções `goto`. O resultado desse esforço foi um algoritmo de rastreamento de fluxo de informação que é extremamente simples e cujo núcleo central pode ser descrito por 40 linhas de código SML apresentado na Seção 3.1. O algoritmo é equivalente ao famoso sistema de tipos sensível ao fluxo de Hunt e Sands [Hunt and Sands 2006], que permanece na esfera teórica e ao contrário da abordagem aqui apresentada, só é capaz de lidar com programas bem estruturados.

Para validar este trabalho, essa nova forma de rastreamento do fluxo de informação foi implementada no compilador LLVM, e está disponível como um serviço *on-line*. Isso permitiu a interação com usuários externos que testaram vários *benchmarks* e puderam apontar alguns problemas nesse serviço, contribuindo para o seu aprimoramento. Na Seção 4.1, pode-se visualizar os resultados experimentais obtidos por testes em implementações largamente usadas e contidas em duas bibliotecas criptográficas: NaCl [Bernstein et al. 2012] e OpenSSL[1]. A implementação sobre o compilador LLVM

[1] `https://www.openssl.org`

tem qualidade industrial e excelente escalabilidade, como será mostrado na Seção 4.2. Ela foi aplicada em todos os programas de inteiros de SPEC CPU 2006 sendo possível analisar mais de 2.4 milhões de instruções *Assembly* em menos de 340 segundos. A beleza dessa técnica advém da possibilidade de fácil adaptação para a detecção de outros tipos de vazamento de informação ou vulnerabilidade de fluxo contaminado, tais como vazamento de endereço e estouros de arranjo e de inteiro, como será demonstrado na Seção 4.3.

2. Visão Geral

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

Vazamento devido ao fluxo de controle. O programa na Figura 1 (a) contém um vazamento de informação baseado em tempo. Nesse exemplo, um adversário pode perceber quanto tempo leva para a função `isDiffVul1` retornar. Um retorno antecipado indica que o casamento na linha 4 falhou em um dos primeiros caracteres. Através da variação, em ordem lexicográfica, do conteúdo do arranjo `in`, o adversário pode reduzir de exponencial para linear a complexidade da busca pela senha.

Vazamento devido ao comportamento da memória *cache*. O programa na Figura 1 (b) é uma tentativa de remover o canal lateral baseado em tempo do programa apresentado na Figura 1 (a). A Função `isDiffVul2` usa uma tabela para verificar se os caracteres usados na senha `pw`, combinam com aqueles apresentados no arranjo de entrada *in*. Se todos os caracteres em ambas cadeias aparecem na mesma ordem, a função retorna verdade, por outro lado retorna falso. A senha `pw` não controla qualquer instruções de desvio na função `isDiffVul2`; porém, este código ainda apresenta um vazamento baseado no tempo de execução. Dados pertencentes à senha são usados para indexar memória na linha 6 do

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

(a)

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

(b)

Figura 1. (a) Programa no qual o fluxo de controle é controlado por informação sigilosa. (b) Programa que permite vazamento de informação devido ao comportamento da memória *cache*.

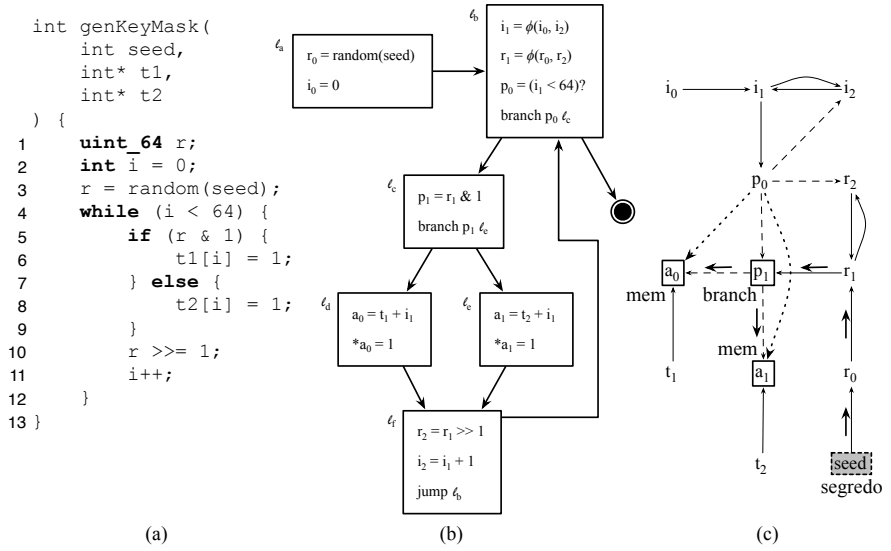


Figura 2. (a) Programa com informação sigilosa no argumento "seed". (b) Grafo de fluxo de controle do programa na forma *SSA*. (c) Grafo de dependências do programa. Linhas pontilhadas representam arestas de controle redundantes que o algoritmo da Seção 3.1 pode evitar inserir no grafo. Uma vulnerabilidade é qualquer caminho entre a informação sigilosa (`seed`) e um predicado que controla um desvio (p_1) ou uma variável que indexa a memória (a_0 ou a_1).

exemplo. Dependendo da distância relativa entre os caracteres de `pw`, algumas falhas de cache podem acontecer. Nesse caso, um adversário pode obter informação sobre quão espaçados estão os elementos alfanuméricos de `pw`. A praticidade desse tipo de ataque foi demonstrada em trabalhos anteriores [Bernstein 2004].

3. Rastreando Fluxo de Informação

A abordagem aqui descrita detecta vazamento de informação através do rastreamento das dependências de *dados* e de *controle* entre as variáveis que compõem um programa. Uma variável v é dependente de dados de uma variável u se v é definida por uma instrução que usa u. Em adição às dependências de dados, a literatura especializada em compiladores também reconhece dependências de *controle* entre variáveis. Uma variável v é *dependente de controle* de um predicado u se a atribuição de v depende do valor de u. O seguinte código esboça essa dependência: "*se u então v = 0*". Ambas dependências, de dados e de controle são transitivas e não podem ser circulares.

As relações de dependências de um programa são aqui representadas como um *grafo de dependências*. Esse grafo tem um vértice n_v para cada variável v no programa, e uma aresta de n_u para n_v se v depende de u. Assegurando que cada vértice corresponde a uma e somente uma variável no programa, foi usada uma representação chamada *Static Single Assignment* (SSA). Em um programa na forma SSA, cada variável tem somente um ponto de definição no código fonte. A Figura 2 (a) mostra um exemplo de programa e a Figura 2 (b) mostra o grafo de fluxo de controle do programa convertido para forma SSA. Instruções tais como $i_1 = \phi(i_0, i_2)$ são usadas para unificar múltiplas definições da mesma variável – i_0 and i_2 – em um nome único, i_1 no caso. Visto que os programas são processados em SSA, denomina-se o grafo de dependências como *Grafo SSA*.

A Figura 2 (c) mostra o grafo SSA produzido para o grafo de fluxo de controle visto na Figura 2. Arestas sólidas representam *dependências de dados*. Arestas não sólidas representam *dependências de controle*. Uma aresta sólida de n_u para n_v indica que o programa contém uma instrução que usa a variável u e define a variável v. Tal aresta existe, por exemplo, de *seed* para *r*, devido à atribuição na linha 3 na Figura 2 (a). Uma aresta não sólida de p para q indica que o programa contém um teste condicional sob o predicado p e dependendo do resultado desse teste, à variável q pode ser atribuído um valor. Continuando com o exemplo, o teste condicional na linha 5 da Figura 2 (a) origina uma aresta não sólida de p_1 para a_1 na Figura 2 (c).

Para construir o grafo SSA é necessário descobrir cada dependência de dados e de controle do programa. Descobrir dependências de dados é fácil: elas estão explicitadas na sintaxe do programa, bastando criar uma aresta a partir de cada variável no lado direito de uma instrução para a variável no lado esquerdo. A presença de ponteiros no programa complica esse problema superficialmente. Dependências de memória não estão explícitas na sintaxe do programa; porém, elas podem ser aproximadas com análise de ponteiros. Encontrar dependências de controle é um problema com solução menos óbvia, mas que será tratado na próxima Subseção.

3.1. Um Algoritmo Simples para Criar Arestas de Dependências de Controle

Duas das arestas de dependência de controle na Figura 2 (c), $p_0 \rightarrow a_0$ e $p_0 \rightarrow a_1$ são redundantes. Transitividade assegura o fato que a_0 depende de p_0, porque o grafo contém as arestas $p_0 \rightarrow p_1$ e $p_1 \rightarrow a_0$. Nesta subseção é apresentado um algoritmo que explora esse tipo de transitividade de forma a inserir no máximo uma aresta de dependência de controle por variável definida no programa.

A Figura 3 mostra o algoritmo para adicionar dependências de controle no grafo SSA. Ele está escrito em linguagem *Standard ML* (SML), uma linguagem de programação funcional. Os colchetes vazios, [], denotam uma lista vazia. Dois pontos duplos, :: são construtores de lista. Por exemplo, $(h :: t)$ é uma lista com cabeça em h, e cauda em t. O símbolo @ é uma concatenação de lista. O símbolo, _, representa qualquer padrão. As palavras chave "**fun** f . . . **and** g . . ."criam funções mutuamente recursivas: f pode chamar g, e g pode chamar f. Exceto para a definição ausente da função *immediate_post_dom*, que retorna o pós-dominador imediato[2] de um vértice. A Figura 3 contém o algoritmo completo. Em outras palavras, este programa pode ser testado em um interpretador SML.

Para ilustrar o algoritmo, foi definida uma linguagem simples nas linhas 1-7 da Figura 3. Blocos básicos são definidos como sequências de instruções que terminam com uma instrução *RET* (*return*), uma *BRZ* ou uma *JUMP*. Estas instruções são chamadas de *terminators*. Existem três tipos de instruções que definem novas variáveis, e foram usadas *strings* para representar variáveis. Um par $UNY(\text{“}v_1\text{”}, \text{“}v_2\text{”})$ é uma instrução unária genérica $v_1 = v_2$. Similarmente, a tripla $BIN(\text{“}v_1\text{”}, \text{“}v_2\text{”}, \text{“}v_3\text{”})$ representa uma instrução binária genérica $v_1 = v_2 \oplus v_3$. Finalmente a tripla $PHI(\text{“}v_1\text{”}, \text{“}v_2\text{”}, \text{“}v_3\text{”})$ representa a função $\phi\ v_1 = \phi(v_2, v_3)$. Desvios são triplas tais como $BRZ(\text{“}p\text{”}, \ell_1, \ell_2)$, onde p é o predicado que determina a saída do desvio, e ℓ_1 and ℓ_2 são as listas de instruções que repre-

[2]Um bloco Z pós-domina um bloco N se todos os caminhos a partir dele para o final do grafo de fluxo de controle devem passar por Z. Similarmente, o pós-dominador imediato de um bloco N é o pós-dominador de N que não pós-domina estritamente qualquer outro pós-dominador estrito de N.

sentam os possíveis alvos. Para o exemplo em questão não é necessária a semântica dessas instruções, bastando apenas a estrutura sintática dos programas. A árvore de dominância[3] nas linhas 9-11 da Figura 3 é uma coleção de dois tipos de vértices, desvios condicionais e saltos incondicionais. Eles são distinguidos por necessidade de tratar blocos que terminam em desvio condicional de forma especial: eles fornecem um novo predicado para ser empilhado pelo algoritmo - linha 13 e 14 da Figura 3.

O algoritmo percorre a árvore de dominância do programa a partir de sua raiz, empilhando predicados. A Função *vchild* visita os filhos de um vértice na árvore. A Função *vnode* visita o próprio vértice. Sempre que é encontrado um bloco que termina com um desvio, a função *push*, nas linhas 13-14, empilha seu predicado. Sempre que uma instrução i é visitada, cria-se uma aresta de controle entre o predicado no topo da pilha, e i. Esse passo é realizado pela função *link*, definida nas linhas 19-24 da Figura 3.

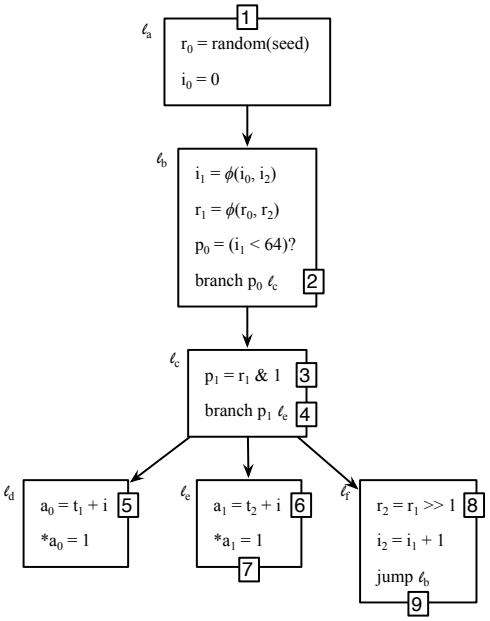
O algoritmo produz uma lista de arestas, $[(\text{“}p_0\text{”}, \text{“}p_1\text{”}), (\text{“}p_1\text{”}, \text{“}ans_2\text{”}), \ldots]$. *Link* é a função que cria as arestas. Ela recebe uma lista de instruções, e extrai arestas a partir dela. Usa-se uma *string* vazia, “”, para denotar o rótulo inicial, que é passada para o algoritmo quando ele inicia a travessia na árvore de dominância. Somente é necessário criar arestas quando o algoritmo visita qualquer operação unária, binária ou uma função ϕ. As outras instruções não necessitam de arestas, porque elas não definem novas variáveis. Além disso, uma vez encontradas, pode-se seguramente finalizar a rotina de *Link*, visto que (*BRZ, JMP, RET*) terminam um bloco básico.

A Figura 4 ilustra como o algoritmo funciona. Inicialmente é computada a árvore de dominância do programa visto na Figura 2 (a), com um custo linear sobre o tamanho do mesmo. Em seguida as arestas de controle observadas na Figura 2 (c) são criadas, exceto arestas $p_0 \rightarrow a_0$ e $p_0 \rightarrow a_1$ que não estão presente. Elas podem ser seguramente omitidas porque quando o algoritmo visita qualquer uma das duas instruções nas quais essas variáveis foram definidas, o topo da pilha contém o predicado p_1, e não o predicado p_0. Portanto, este algoritmo básico cria no máximo $O(N)$ arestas de controle no grafo SSA, onde N é a quantidade de variáveis definidas no programa.

[3]Vértice v *domina* vértice u em um grafo de fluxo de controle com um único ponto de entrada se cada caminho a partir de *start* para u passa através de v. Vértice v é o *dominador imediato* de u se qualquer outro vértice que domina u também domina v. Esta relação determina uma *árvore de dominância* única.

```
 1 datatype Instruction =
 2    UNY of string * string
 3  | BIN of string * string * string
 4  | PHI of string * string * string
 5  | BRZ of string * Instruction list * Instruction list
 6  | JMP of Instruction list
 7  | RET;
 8
 9 datatype DomTree =
10    BRANCH of Instruction list * string * DomTree list
11  | JUMP of Instruction list * DomTree list
12
13 fun push (BRANCH (_, p, _)) preds = (p :: preds)
14   | push (JUMP (_, _)) preds = preds
15
16 fun inspect (BRANCH (bb, _, children)) = (bb, children)
17   | inspect (JUMP (bb, children)) = (bb, children)
18
19 fun link [] _ = []
20   | link _ "" = []
21   | link ((UNY (a, _)) :: insts) label = (label, a) :: link insts label
22   | link ((BIN (a, _, _)) :: insts) label = (label, a) :: link insts label
23   | link ((PHI (a, _, _)) :: insts) label = (label, a) :: link insts label
24   | link _ _ = []
25
26 fun vchild [] _ _ = []
27   | vchild (n::ns) preds pdom =
28     vnode n preds pdom @ vchild ns preds pdom
29 and vnode n preds pdom =
30   let
31     val (bb, ns) = inspect n
32     fun top nil = "" | top (h::t) = h
33     fun pop nil = nil | pop (h::t) = t
34   in
35     if is_immediate_post_dom pdom n (top preds)
36     then vnode n (pop preds) pdom
37     else link bb (top preds) @ vchild ns (push n preds) pdom
38   end
```

Figura 3. O algoritmo básico que insere aresta de controle em um grafo SSA.



1) **Pilha**: vazia
2) Empilha "p_0"
   Pilha: [p_0]
3) Cria aresta de controle: "$p_0 \rightarrow p_1$"
   Pilha: [p_0]
4) Empilha "p_1"
   Pilha: [p_1, p_0]
5) Cria aresta de controle: "$p_1 \rightarrow a_0$"
   Pilha: [p_1, p_0]
6) Cria aresta de controle: "$p_1 \rightarrow a_1$"
   Pilha: [p_1, p_0]
7) Desempilha p_1
   Pilha: [p_0]
8) Cria aresta de controle: "$p_0 \rightarrow r_2$"
   Pilha: [p_0]
9) Desempilha p_0
   Pilha: []

Figura 4. Árvore de dominância do programa na Figura 2 (a), somado a alguns eventos chave que ocorrem quando executa-se o algoritmo básico nesta árvore.

3.2. Propriedades Estruturais do Algoritmo

Considerando que $\mathsf{dom}_{B'}$ e pdom_B denotam o conjunto de dominadores de B' e de pós-dominadores de B respectivamente, define-se *região de influência* como:

Definição 3.1 (Região de Influência) *A Região de Influência IR_B de um bloco B que termina com um desvio é um conjunto de blocos básicos. Ela contém o bloco B' se, e somente se: (i) $B \in \mathsf{dom}_{B'}$; (ii) $B' \notin \mathsf{pdom}_B$; (iii) não existe B" tal que B" $\in \mathsf{pdom}_B$ e B" $\in \mathsf{dom}_{B'}$.*

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

Definição 3.2 (Região de Influência Imediata) *A região de influência imediata IIR_B de um bloco B contém blocos B' se, e somente se: (i) $B' \in \mathsf{IR}_B$; e (ii) não existe bloco B'', B" $\neq B$, tal que $B' \in \mathsf{IIR}_{B''}$. Neste caso, B é a cabeça de IIR_B.*

O algoritmo da Figura 3 conecta um predicado p a uma variável definida dentro da região de influência imediata do bloco onde p é usado. Este fato é enunciado pelo Lema 3.4, que é um corolário do Lema 3.3:

Lema 3.3 *Se B é um dominador imediato de B' ($B = \mathsf{idom}_{B'}$) e B' pós-domina B ($B' \in \mathsf{pdom}_B$), então B' é o pós-dominador imediato de B ($B' = \mathsf{pdom}_B$).*

Lema 3.4 *A função `Visit` cria arestas de controle entre um predicado `pred` e uma variável v, se, e somente se, `pred` é usada em um desvio que finaliza um bloco B, e v é definida dentro de um bloco B', $B' \in \mathsf{IIR}_B$.*

O Lema 3.6 determina uma relação de equivalência total dentro da região de influência de um bloco básico. Esse Lema será necessário na prova do Teorema 3.7. A prova do Lema 3.6 requer o Lema 3.5, enunciado abaixo.

Lema 3.5 *Se B domina B' e $IR_B \cap IR_{B'} \neq \emptyset$, então $IR_B \supseteq IR_{B'}$.*

Lema 3.6 *O conjunto de regiões de influência imediata dentro da região de influência de um bloco B determina uma relação de equivalência total.*

Os Lemas 3.4 e 3.6 fornecem o arcabouço necessário para mostrar que o algoritmo da Figura 3 cria uma cadeia de de dependências transitivas que conectam um predicado com todas atribuições de variáveis que este predicado controla.

Teorema 3.7 *Se um bloco básico B termina em condicional "$\texttt{br}(p, \ell)$", então a função* `visit` *cria uma cadeia de dependências de controle conectando p à cada variável definida dentro de IR_B.*

4. Resultados Experimentais

O algoritmo da seção anterior foi materializado na forma de uma ferramenta chamada FlowTracker que foi incorporada ao compilador LLVM 3.3. FlowTracker cria o grafo SSA a partir de um código fonte de entrada em linguagem C/C++[4]. Uma vez construído o grafo SSA, *FlowTracker* inicia a busca por caminhos que conectam vértices que representam informações sigilosas à vértices sorvedouros, que representam predicados de instruções de desvio ou indexação de memória. Estas informações sigilosas devem ser informadas pelo usuário através de uma entrada para *FlowTracker* em formato *eXtensible Markup Language* (XML). Cada caminho encontrado por FlowTracker é reportado ao usuário que poderá identificar o traço de instruções correspondente em seu programa e corrigir o problema, basicamente modificando sua implementação a fim de quebrar a cadeia de dependências entre o sorvedouro e a informação sigilosa. *FlowTracker* executa em tempo linear sobre o tamanho do programa. A implementação de FlowTracker tem 4,771 linhas de código comentado C++. Este tamanho é devido à necessidade de manipular cada tipo individual de instruções da representação intermediária do LLVM. Essa manipulação corresponde às linhas 19-24 da Figura 3. Esta Seção serve para responder a 3 perguntas de investigação:

- PI1: Quão *efetivo* é FlowTracker para detectar vazamento de informação?
- PI2: Quão *escalável* é o algoritmo da Figura 3?
- PI3: Quão *adaptável* é FlowTracker para detectar outros tipos de vulnerabilidades de fluxo de informação?

4.1. PI1: Efetividade

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

[4]FlowTracker pode ser facilmente modificado para aceitar outras linguagens de programação, tais como Java ou C#

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

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

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

4.2. PI2: Escalabilidade

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

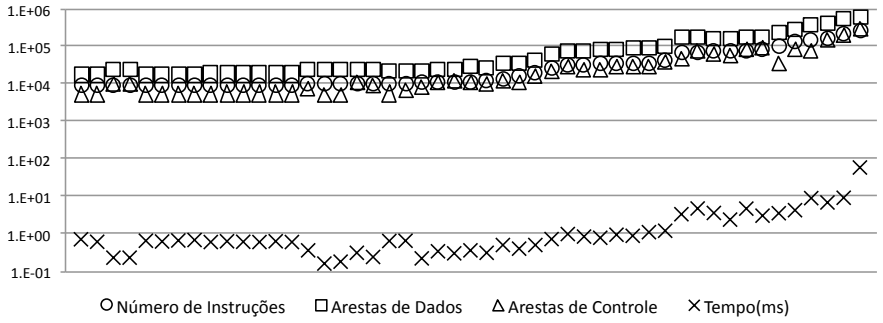


Figura 5. Tamanho dos programas (Número de Instruções) vs tamanho do grafo SSA (número de arestas) vs tempo (ms) para construir o grafo SSA.

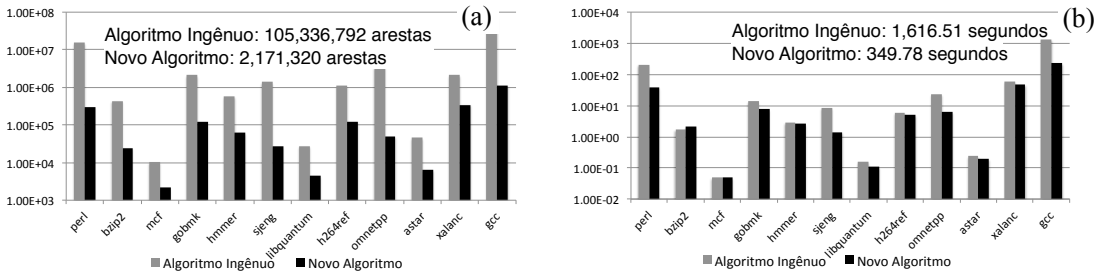


Figura 6. (a) Número de arestas inseridas com o algoritmo da Figura 3 vs sem a remoção de dependências transitivas (b) Tempo de execução do procedimento de construção do grafo, com e sem a remoção de dependências transitivas.

Cada núcleo tem uma *cache* de 6,144 KB. Essa máquina executava Linux Ubuntu 12.04.

A Tabela 1 mostra o tamanho do grafo SSA que foi produzido para a coleção de programas SPEC CPU 2006. É notável um número 1.5x maior de arestas de dependências de dados que variáveis no grafo SSA. Existem 2.5 variáveis por aresta de dependência de controle. Estes números indicam que o grafo é esparso, isto é, o número de arestas é linearmente proporcional ao número de vértices. A exploração da transitividade para remoção de dependências redundantes é essencial para assegurar essa propriedade. FlowTracker também foi aplicado nos 100 maiores programas da coleção de *benchmarks* de LLVM. A Figura 5 mostra esses números.

Para demonstrar a importância da remoção das dependências transitivas, foi considerado o comportamento de uma versão ingênua do algoritmo. Esta abordagem ingênua

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

Tabela 1. Como FlowTracker escala: a coleção de testes SPEC. *Vars*: número de variáveis em cada programa (número de vértices no grafo SSA). *Controle*: número de arestas de dependência de controle inseridas pelo Algoritmo da Figura 3. *Dados*: número de arestas de dependência de dados. *Tempo*: tempo para construir o grafo SSA.

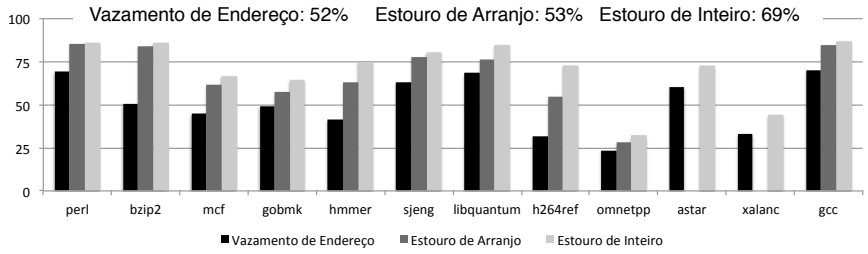


Figura 7. Porcentagem de grafo SSA "vulnerável" e que requer guardas para sanitizá-los.

cria uma aresta entre o predicado p e uma variável v sempre que v é controlada por p, não considerando transitividade. A Figura 6 compara ambas versões do algoritmo que detecta dependências de controle. A Figura 6 (a) mostra que o algoritmo proposto é duas ordens de magnitude mais frugal que essa versão ingênua. Essa lacuna cresce com o tamanho do *benchmark*. Por exemplo, o grafo SSA do menor *benchmark* SPEC, `mcf`, tem 3,940 vértices. Para esse *benchmark* foram criados 2,133 arestas de controle, enquanto a versão ingênua criou 10,241. O maior *benchmark*, `gcc`, fornece um grafo com 1,249,681 vértices. O algoritmo insere 1,112,889 arestas de controle no grafo, e a versão ingênua insere 78,522,510. Esta diferença no número de arestas de controle tem um impacto direto no tempo de execução do algoritmo, como mostra a Figura 6 (b). O algoritmo analisa o menor *benchmark* do conjunto, `mcf` em 0.05 segundos, e o maior, `gcc`, em 227 segundos. A versão ingênua toma 0.05 e 1,297 segundos, respectivamente. No total, o algoritmo leva 341.35 segundos para cobrir todo o SPEC CPU 2006, enquanto a versão ingênua necessita de 1,616.51 segundos.

4.3. PI3: Adaptabilidade

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

A Figura 7 mostra a porcentagem do grafo SSA dos *benchmarks* SPEC CPU 2006 que é considerada vulnerável. No caso de estouro de arranjo, considerou-se como fonte qualquer função de biblioteca cujo código não esteja disponível para o compilador. Sorvedouros são instruções de armazenamento na memória. Para estouro de inteiro foram consideradas como fonte as operações aritméticas que podem ser "arredondadas", isto é, podem gerar estouro aritmético. Como sorvedouro foram consideradas instruções de carga e armazenamento na memória, pois adversários podem usar estouro de inteiro para

habilitar estouro de arranjo. Estouros podem levar a falhas que são difíceis de encontrar. Como um exemplo, em 1996, o foguete Ariane 5 foi perdido devido a um estouro de inteiro – uma falha que custou mais de US$370 milhões [Dowson 1997]. Finalmente, para o problema de vazamento de endereço foi considerada como fonte qualquer operação que lê endereços de memória, e como sorvedouro qualquer função de biblioteca. Descoberta de endereços podem dar ao adversário, meios de contornar um mecanismo de proteção do sistema operacional chamado *Address Space Layout Randomization* (ASLR). Como um exemplo, Dionysus Blazakis explicou como usar informação de endereço para comprometer um interpretador *ActionScript* [Blazakis 2010].

Como a Figura 7 mostra, uma parte substancial de cada grafo SSA foi marcada como vulnerável. Isso é consequência de uma definição liberal de funções fonte. Note que a análise pode conter falsos positivos, isto é, um dado caminho dentro do programa nunca ser tomado dinamicamente. Cada vulnerabilidade demanda diferentes tipos de guarda. Estouros de inteiro podem ser evitados com instrumentação de Dietz *et al.* [Dietz et al. 2012]. Instruções de armazenamento que podem causar estouro de arranjo podem ser guardadas por ferramentas tais como *AddressSanitizer* [Serebryany et al. 2012]. E vazamentos de endereço pode ser prevenidos por uma adaptação da análise dinâmica de fluxo de dados de Chang *et al.* Ainda sobre essa última vulnerabilidade, foram inspecionados manualmente alguns grafos SSA, e encontrado vazamentos perigosos. Por exemplo, as funções `spec_fread` (em `bzip2/spec.c:187`), e `spec_fwrite` (em `bzip2/spec.c:262`) imprimem o endereço de arranjos que eles recebem como parâmetros; assim, eles dão ao adversário total conhecimento da informação secreta.

5. Trabalhos Relacionados

Uma breve história da vulnerabilidade de canal lateral baseado em tempo.

Canais laterais baseados no tempo são um problema bem conhecido. Muito do atual conhecimento sobre esse problema é devido à Kocher [Kocher 1996]. A conjectura de Kocher - que o tempo de execução pode ser usado para obter informação sobre dados sensíveis - foi demonstrada por vários pesquisadores. Dhem *et al.* [Dhem et al. 2000] mostrou como implementar um ataque por variação de tempo em uma implementação RSA executando em um *smart card*. Posteriormente, Brumley e Boneh mostraram que é possível recuperar informação sigilosa de uma implementação mesmo em face de ruídos introduzidos pela rede de comunicação. Eles foram capazes de recuperar a chave privada RSA de um sistema web com políticas de segurança baseadas na biblioteca OpenSSL. Neste trabalho a máquina adversária e o servidor web estavam localizados em prédios diferentes com três roteadores e vários *switches* entre eles. Poucos anos mas tarde, Brumley e Tuveri montaram uma recuperação completa da chave contra um servidor TLS que usava assinaturas ECDSA para autenticação. Para uma visão mais geral sobre o campo de ataques por variação de tempo, é recomendado um tutorial apresentado por Emmanuel Prouff na conferência CHES'13 [Prouff 2013].

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

Existem várias metodologias e linhas gerais para evitar canais laterais baseados em tempo. John Agat propôs um sistema de tipos para transformar um programa vulnerável em outro seguro. Ele realizou essa transformação pela inserção de instruções

inócuas nos blocos de desvio, para mitigar a diferença no tempo de execução de diferentes caminhos que podem ser tomados à partir de um teste condicional. De forma similar, Molnar *et al.* [Molnar et al. 2006] projetou um tradutor C fonte-fonte que detecta e conserta vazamentos baseados no fluxo de controle. Contrariamente ao trabalho aqui apresentado, as abordagens de Agat e de Molnar *et al.* não podem lidar com vazamentos baseados no comportamento da *cache*.

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

6. Conclusão

Este artigo apresentou uma técnica de análise estática que determina se o tempo de execução de um programa depende de informação sensível. A análise executa diretamente na representação intermediária enquanto o programa é compilado. Portanto, canais laterais não serão introduzidos pelo compilador durante a geração de código. A concretização dessa análise estática é uma ferramenta chamada FlowTracker que está disponível como um serviço *on-line*, e permitiu um alto nível de confiança sobre sua consistência.

Acredita-se que este artigo é o primeiro trabalho capaz de certificar que um programa tem comportamento isócrono no nível do compilador. É possível lidar com códigos não estruturados e foi projetado um algoritmo que rastreia dependências de controle entre variáveis de um programa. O algoritmo executa em tempo linear e cria no máximo uma aresta de controle entre cada variável e qualquer predicado no programa. Pelo que se sabe, nenhum outro algoritmo desse tipo na literatura assegura essa propriedade.

Referências

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

[Bernstein 2004] Bernstein, D. J. (2004). Cache-timing attacks on AES. URL: `http://cr.yp.to/papers.html#cachetiming`.

[Bernstein 2006] Bernstein, D. J. (2006). Curve25519: new diffie-hellman speed records. In *PKC*, pages 207–228. Springer.

[Bernstein et al. 2012] Bernstein, D. J., Lange, T., and Schwabe, P. (2012). The security impact of a new cryptographic library. In *Progress in Cryptology – LATINCRYPT*, pages 159–176. Springer.

[Blazakis 2010] Blazakis, D. (2010). Interpreter exploitation. In *WOOT*, pages 1–9. USENIX.

[Chen et al. 2014] Chen, Y.-F., Hsu, C.-H., Lin, H.-H., Schwabe, P., Tsai, M.-H., Wang, B.-Y., Yang, B.-Y., and Yang, S.-Y. (2014). Verifying Curve25519 software. In *Proceedings of CCS*, pages 299–309. ACM.

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

[Dietz et al. 2012] Dietz, W., Li, P., Regehr, J., and Adve, V. (2012). Understanding integer overflow in C/C++. In *ICSE*, pages 760–770. IEEE.

[Dowson 1997] Dowson, M. (1997). The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–.

[Genkin et al. 2014] Genkin, D., Shamir, A., and Tromer, E. (2014). RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO*, pages 444–461. Springer.

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

[Kocher et al. 1999] Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer.

[Kocher 1996] Kocher, P. C. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113. Springer.

[Lux and Starostin 2011] Lux, A. and Starostin, A. (2011). A tool for static detection of timing channels in java. *Journal of Cryptographic Engineering*, 1(4):303–313.

[Molnar et al. 2006] Molnar, D., Piotrowski, M., Schultz, D., and Wagner, D. (2006). The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of ICISC*, pages 156–168, Berlin, Heidelberg. Springer.

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

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

[Quisquater and Samyde 2001] Quisquater, J.-J. and Samyde, D. (2001). Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Smart Card Programming and Security*, pages 200–210. Springer.

[Serebryany et al. 2012] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: a fast address sanity checker. In *USENIX*, pages 28–28. USENIX Association.

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

