FlowTracker - Detecção de Código Não Isócrono via Análise Estática de Fluxo.

Bruno R. Silva[1]**, Leonardo Ribeiro**[2]**, Diego Aranha**[3]**, Fernando M. Q. Pereira**[1]

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

[2]Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)

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

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

leonardofribeiro@gmail.com

Resumo. *FlowTracker é uma ferramenta disponível para uso online, capaz de analisar programas escritos em linguagem C ou C++ e reportar traços de instruções vulneráveis à ataques por análise de variação de tempo. Esses ataques são comuns em implementações de primitivas criptográficas e são possíveis devido a presença de canais laterais nas respectivas implementações. Um canal lateral ocorre quando bits da chave criptográfica ou de outra informação sigilosa acabam por influenciar a execução de desvios condicionais, alterando o fluxo de controle da aplicação e consequentemente variando o seu tempo de execução. Eles também ocorrem quando esses mesmo bits são utilizados como índices de acesso à memória, o que também pode afetar o tempo de execução devido às falhas de cache. FlowTracker foi testado em duas famosas e reconhecidas bibliotecas criptográficas: OpenSSL e NaCl. Foi possível validar a ausência de canais laterais em NaCl, e detectar centenas de trechos vulneráveis em OpenSSL.*

Vídeo disponível em https://youtu.be/e2AEZlimNas

1. Introdução

Algoritmos criptográficos robustos e consistentes são essenciais para garantir a segurança de aplicações modernas. Entretanto, a correção de um algoritmo criptográfico não está somente em seu projeto abstrato, mas também em sua implementação concreta [Fan et al. 2010, Kocher 1996, Standaert 2010]. Uma implementação é dita insegura quando ela apresenta *canais laterais*. Canais laterais são falhas de implementação que permitem a um adversário conhecer informações sigilosas de um algoritmo criptográfico. A literatura especializada em segurança já apresentou vários exemplos de tais ataques. Em alguns desses exemplos, adversários necessitam de acesso físico ao sistema, para nele inserir falhas [Biham and Shamir 1997] ou recuperar dados residuais da memória [Halderman et al. 2009]. Há casos onde o simples acesso a algum componente elétrico do hardware criptográfico, tal como os pinos das portas de I/O, pode ser suficiente para a recuperação de dados sigilosos [Genkin et al. 2014]. Porém, ataques menos invasivos também têm despertado o interesse da comunidade científica. A literatura da área descreve situações em que atacantes são capazes de detectar diferenças ínfimas em aspectos mensuráveis do comportamento de programas, tais como seu tempo

de execução [Kocher 1996], seu consumo de energia [Kocher et al. 1999], seu campo magnético [Quisquater and Samyde 2001] ou ruídos que ele produz [Genkin et al. 2014]. De posse de tais informações, adversários podem usá-las para inferir informação sigilosa do algoritmo criptográfico.

No contexto dos ataques que exploram a variação de tempo, garantir que informações sigilosas não influenciem o fluxo de controle da implementação é uma forma natural de eliminar canais laterais. Isso pode ser feito manualmente pelo programador ao evitar que dados oriundos das variáveis secretas controlem o resultado de desvios condicionais. Obviamente essa é uma tarefa árdua, que exige atenção e profundo conhecimento da linguagem de programação utilizada. Entretanto, mesmo um algoritmo implementado com tais cuidados necessita ser verificado após a compilação, pois otimizações no nível de linguagem intermediária podem inserir canais laterais. Verificações nesse nível usualmente requerem análises complexas de código por profissionais experientes, que tenham pleno conhecimento das características específicas da arquitetura de computador utilizada. Apesar de existirem ferramentas que dão suporte a essa verificação manual [Almeida et al. 2013, Chen et al. 2014], não existe atualmente técnica automática incorporada ao compilador capaz de garantir a isocronia de programas. Em outras palavras, compiladores de uso geral não proveem garantias de que eles não estão introduzindo vazamento de informação por variação de tempo no código que eles geram.

Neste trabalho, apresenta-se uma solução capaz de detectar um código não isócrono. Chamamos **não isócrono** o código que executa em tempo variável de acordo com os bits da informação que se deseja proteger. De forma análoga, chamamos de **isócrono** um programa criptográfico em que os bits da informação sigilosa não influenciam seu tempo de execução. Pode-se afirmar que um código isócrono é ausente de canais laterais e portanto não passível de ataques por variação de tempo. Diferentemente de outras abordagens descritas na literatura, que buscam detectar e reparar a ocorrência de canais laterais em uma determinada linguagem de programação [Lux and Starostin 2011], a solução aqui proposta atua sobre a representação intermediária do compilador. Além disso, a técnica proposta neste artigo é capaz de lidar com programas não estruturados, isto é, aqueles que utilizam instruções do tipo `goto`. Para tanto, um grafo de dependências é construído de acordo com a relação de dependências de dados e de controle entre as variáveis do programa. Tal grafo permite a detecção de todos os caminhos que conectam informações sigilosas a predicados de instruções de desvio. Esses caminhos são então reportados ao desenvolvedor da aplicação para que este tenha o devido cuidado de modificar o programa a fim de eliminar esses segmentos de código não isócronos.

A fim de validar as ideias apresentadas neste artigo, um detector de código não isócrono foi implementado sobre o compilador LLVM [Lattner and Adve 2004]. Essa implementação, batizada de *FlowTracker*, consiste em duas partes: (i) um grafo de dependências; e (ii) um detector de isocronia. O grafo de dependências descreve as relações entre as variáveis de um programa. O detector de isocronia usa esse grafo para verificar se informação sigilosa pode influenciar o tempo de execução do programa. Em caso afirmativo, cada traço estático vulnerável é reportado via um conjunto de arquivos de saída no formato `.png` e `.txt`. Esses módulos foram testados em duas famosas e reconhecidas bibliotecas criptográficas: *OpenSSL* e *NaCl* [Bernstein et al. 2012]. Foi possível validar a isocronia de *NaCl*, e detectar centenas de trechos não-isócronos em *OpenSSL*. *FlowTrac-*

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

Figura 1. Exemplo de código não isócrono.

ker está disponível para uso *online*[1], não necessitando a instalação de qualquer programa no computador do desenvolvedor. Entretanto, o código fonte da ferramenta está publicamente disponível para aqueles que desejarem instalação *offline*. O fato de todas essas detecções terem sido realizadas de forma totalmente automática indica que a técnica apresentada neste artigo é um mecanismo efetivo e útil para melhorar a qualidade de sistemas de criptografia.

2. Ataque por variação de tempo

Ataques por variação de tempo visam obter o conhecimento de informações sigilosas tais como uma chave criptográfica ou um número aleatório. Eles se baseiam na execução do programa com dados de entradas pré-determinados e medições do tempo de execução, permitindo ao adversário inferir com grande precisão alguns ou todos os bits da informação secreta. Tais ataques podem ocorrer quando esses dados sigilosos influenciam os predicados, isto é, as estruturas condicionais das instruções de desvio e portanto determinam quais partes do código serão executadas, afetando o tempo de execução do programa.

Para ilustrar esse problema, a Figura 1 a) apresenta uma função simples de comparação de *strings*. Essa função recebe uma senha pw que será comparada com uma entrada do usuário in. Visto que o usuário tem o total controle do que será definido como in, considera-se que essa variável pode estar contaminada com dados maliciosos a fim de ativar/desativar partes pré-determinadas do código fonte. Um adversário pode monitorar quanto tempo a função `compVar` leva para retornar. Um retorno antecipado indica que a comparação na linha 3 não obteve sucesso nos primeiros caracteres. Portanto, variando, em ordem lexicográfica, o conteúdo da *string* in, o adversário pode reduzir de exponencial para linear a complexidade de busca pelo conteúdo de pw.

Esse tipo de ataque é facilitado quando o usuário tem acesso ao código fonte da implementação, mas também é completamente viável mesmo sem a divulgação de detalhes da implementação, bastando o conhecimento prévio do algoritmo criptográfico utilizado, o qual é comumente de conhecimento público.

3. Detecção de código não isócrono

FlowTracker é capaz de detectar e reportar ao usuário o problema apontado na Figura 1. Para tanto, a primeira tarefa é a construção de um grafo de dependências que armazena toda informação relacionada aos fluxos explícitos e implícitos do programa, que estão definidos logo abaixo:

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

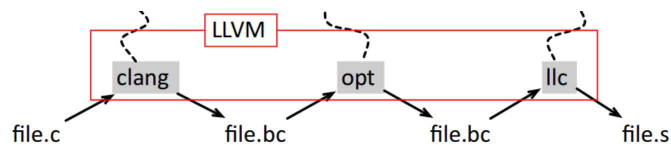

Figura 2. Exemplo de uso do compilador LLVM

1. Fluxos explícitos estão relacionados às dependências de dados. Se um programa contém uma instrução que define a variável v, e usa a variável u, tal como $v = u + 1$, então existe um fluxo explícito de informação de u para v.
2. Fluxos implícitos estão relacionados ao fluxo de controle do programa. Se o programa contém um desvio tal como $if\ p = 0\ then\ v = u + 1\ else\ v = u - 1$, então existe um fluxo implícito de informação de p para v, pois o valor atribuído ao último depende do primeiro.

FlowTracker detecta esses fluxos, cria um grafo de dependências do programa e procura nesse grafo caminhos entre informações sigilosas e instruções de desvio e índices de memória. No grafo de dependências, os vértices representam operandos, acessos à memória ou operações. As arestas são classificadas em arestas de dados, que representam um fluxo explícito entre dois vértices ou arestas de controle que representam fluxo implícito entre um predicado e um outro vértice qualquer.

Uma vez construído o grafo de dependências, *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 XML que será descrita na próxima Seção.

Cada caminho encontrado é reportado ao usuário que deverá por sua vez identificar tal traço de instruções 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. Vale mencionar que a escalabilidade de *FlowTracker* foi colocada à prova durante testes sobre a coleção de *benchmarks* SPEC CPU INT 2006 que contém programas como `GCC` com mais de 600 mil linhas de código. Nesse cenário, *FlowTracker* foi capaz de analisar todos esses programas em poucos segundos.

4. Exemplo de uso

FlowTracker é uma ferramenta implementada na forma de um módulo, também conhecido como *pass*[2], para o compilador industrial LLVM[Lattner and Adve 2004], bastante utilizado também em projetos de pesquisa envolvendo otimizações no nível do compilador. Basicamente LLVM corresponde à um *front end* e um conjunto de ferramentas a fim de gerar código compilado com possibilidade de realizar diversas otimizações.

Conforme mostra a Figura 2, utiliza-se o *front end clang* para transformar o programa C em *bytecodes* [3] LLVM - uma representação intermediária em formato *SSA - Static single assignment form* [Cytron et al. 1989] - gerando o arquivo *file.bc* que por sua vez

[2]Um *pass* é uma transformação ou análise que um compilador aplica sobre o programa.
[3]http://llvm.org/docs/LangRef.html

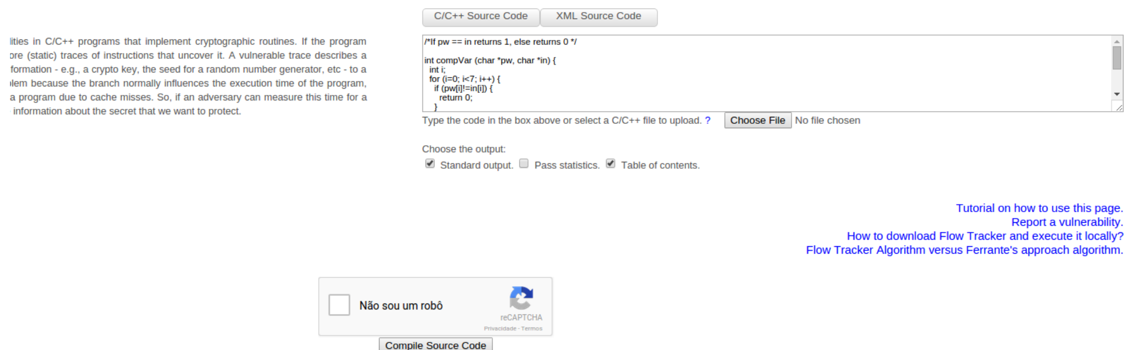

Figura 3. Interface Web de *FlowTracker*.

pode ser otimizado pelo programa *opt* também pertencente ao conjunto de ferramentas LLVM. *FlowTracker* é um analisador estático de código que é então carregado dinamicamente por *opt*. Uma vez que a representação intermediária é analisada por *FlowTracker* durante a execução de *opt* e concluído a não existência de canais laterais, pode-se então gerar o código final na linguagem assembly da arquitetura alvo, utilizando a ferramenta *llc* também disponível no conjunto de ferramentas do LLVM.

Entretanto, com o objetivo de facilitar ao máximo o uso de *FlowTracker* por aqueles sem muito conhecimento de como LLVM deve ser instalado e utilizado, desenvolveu-se uma interface web[4] que agrupa todas as etapas listadas acima, com exceção da geração final do código na linguagem alvo. Nesse caso, o usuário deverá apenas especificar um arquivo contendo o código fonte que será analisado e um arquivo XML contendo informações sobre quais dados do respectivo código, deverão ser considerados sigilosos por *FlowTracker*. O usuário também tem a possibilidade de digitar diretamente em caixa de texto, essas informações e o programa em linguagem C ou C++.

4.1. Entradas

A Figura 3 exibe a interface de usuário disponibilizada por FLow Tracker. Na aba *C/C++ Source Code* deve-se digitar o programa em linguagem C/C++. O usuário poderá optar por fazer o *upload* do arquivo com o código fonte clicando no botão *Choose File*. Na aba *XML Source Code*, o usuário poderá digitar as informações de quais dados serão considerados como sigilosos por *FlowTracker*. Obviamente também existe a opção de fazer *upload* de um arquivo `.xml` contendo tais informações.

O Formato do arquivo XML é descrito na Figura 4. Basicamente ele é composto de um preâmbulo fixo contendo as *tags* `<functions></functions>` e `<sources></sources>`. Para cada função que recebe ou gera uma informação considerada sigilosa tal como uma chave criptográfica ou um número aleatório, deve-se iniciar uma *tag* `<function></function>` contendo o nome da função. No exemplo da Figura 4 essa função é a `compVar`. Finalmente deve-se iniciar uma

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

```xml
<functions>
  <sources>
  <function>
    <name>compVar</name>
    <parameter>1</parameter>
  </function>
  </sources>
</functions>
```

Figura 4. Arquivo XML informando à *FlowTracker* o(s) dado(s) considerado(s) sigiloso(s).

tag `<parameter></parameter>`, contendo o valor 1 caso o primeiro parâmetro seja sigiloso, 2 caso o segundo e assim por diante. No caso de `compVar`, apenas o primeiro parâmetro (*pw*) deverá ser considerado sigiloso. Caso, o valor de retorno dessa função também seja considerado sigiloso, deve-se iniciar uma nova *tag* `<parameter></parameter>` informando o parâmetro 0.

4.2. Saídas

A Figura 5 mostra a saída de *FlowTracker* para as entradas correspondentes às Figuras 1 e 4. No sumário, percebe-se que foi encontrado: 1 informação sigilosa (parâmetro `pw` de `compVar`); 10 instruções de desvio e/ou índices de memória e 4 subgrafos vulneráveis. Cada subgrafo corresponde a um traço de instruções *assemby* correspondentes à linguagem intermediária de LLVM. Para cada traço, são gerados 3 arquivos de saída. Dois deles são mais adequados a usuários que conhecem um pouco da linguagem intermediária do compilador LLVM. São eles: `subgraphLLVM.dot` e `subgraphLines.dot`. Para visualizá-los pode-se usar qualquer ferramenta de visualização de arquivos `.dot` como `xdot` por exemplo. É possível também, clicar no ícone ao lado de cada arquivo para obter a versão em formato `.png`.

Tanto `subgraphLLVM.dot` quando `subgraphLines.dot` apresentam um subgrafo originado do grafo de dependências completo (arquivo `fullGraph.dot`), de forma que é possível visualizar 1 ou mais caminhos que conectam a informação sigilosa à uma instrução de desvio ou índice de memória. `subGraphLines.dot` apresenta como rótulo de cada vértice, a linha na qual ele é originado. Enquanto que `subGraphLLVM.dot` apresenta como rótulo de cada vértice, um *opcode* ou um operando. Arestas contínuas representam fluxo explícito e arestas tracejadas representam fluxo implícito.

Para cada subgrafo vulnerável é gerando também um terceiro arquivo de saída, chamado `subgraphASCIILines.txt`, o que informa em formato ASCII as linhas do programa C/C++ que correspondem ao respectivo subgrafo e representam um caminho no program entre informação sigilosa e instrução de desvio ou índice de memória. Neste caso, o usuário poderá localizar o traço vulnerável em seu código e corrigi-lo adequadamente à fim de eliminar o canal lateral. Basicamente, as correções se dão pela interrupção do caminho.

5. Trabalhos Relacionados

Lux *et al.* [Lux and Starostin 2011] implementou uma ferramenta que detecta vulnerabilidades de ataques por variação de tempo em programas Java. A principal diferença entre *FlowTracker* e a abordagem de Lux *et al*'s é o fato que eles operam na linguagem

Using LLVM 3.3 - Bytecode's size = 2872 bytes

Standard output

```
********* Flow Tracking Summary ***********

Secrets 1
Branch instructions  or memory indexes 10
Vulnerable Subgraphs: 4
```

Files

DOT files	TXT files
fullGraph.dot	subgraphASCIILines1.txt
subgraphLines3.dot	subgraphASCIILines3.txt
subgraphLines1.dot	subgraphASCIILines2.txt
subgraphLLVM3.dot	subgraphASCIILines0.txt
subgraphLLVM1.dot	
subgraphLines2.dot	
subgraphLLVM0.dot	
subgraphLines0.dot	
subgraphLLVM2.dot	

Figura 5. Exemplo de saída de *FlowTracker*.

de programação em alto nível, usando um conjunto de regra de inferência. Porém, a abordagem de *FlowTracker* tem algumas vantagens, porque ele trabalha diretamente na representação intermediária do compilador. Apesar de *FlowTracker* ser uma ferramenta para análise de código C e C++, seu algoritmo pode lidar com diferentes linguagens de programação bastando utilizar a ferramenta *llc* disponível na coleção de programas LLVM, que seja adequada à plataforma de *hardware* alvo. Além disso, otimizações do compilador podem ser realizadas antes da análise estática de *FlowTracker*, garantido que nenhum canal lateral possa ser introduzido no código executável. Finalmente, do ponto de vista de projeto, a abordagem de *FlowTracker* é substancialmente diferente de Lux *et al.*'s, porque pode-se analisar inclusive programas não estruturados, isto é, aqueles que utilizam instruções do tipo `goto`.

Existem também abordagens similares à de *FlowTracker*, mas que objetivam detectar outro tipo de canal lateral: variação de potência. Por exemplo, Moss et al. [Moss et al. 2012] apresenta um sistema de tipos que também opera na linguagem intermediária e detecta seguimentos de código vulneráveis à ataques por análise de variação de potência. Similarmente, Agosta *et al.* [Agosta et al. 2013], recentemente lançaram um passe LLVM que detecta quais instruções são dependentes de chaves criptográficas. As duas principais diferenças entre o trabalho de Agosta e *FlowTracker* são: (i) o fato que *FlowTracker* considera não somente o fluxo de dados, mas também o fluxo implícito de informação, isto é, aquele devido às dependências de controle; e (ii) o fato que *FlowTracker* está lidando com canal lateral relacionado a tempo e não potência.

6. Comentários finais

Acredita-se que *FlowTracker* seja a primeira ferramenta capaz de certificar que um programa possui um comportamento isócrono no nível do compilador. Ela é capaz de lidar com programas não estruturados pois foi concebida com um algoritmo simples e que executa em tempo linear, sendo capaz de representar em um grafo as relações de dependências de controle e de dados entre as variáveis do programa.

Referências

Agosta, G., Barenghi, A., Maggi, M., and Pelosi, G. (2013). Compiler-based Side Channel Vulnerability Analysis and Optimized Countermeasures Application. In *Proceedings of DAC*, New York, NY, USA. ACM.

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, 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.

Biham, E. and Shamir, A. (1997). Differential fault analysis of secret key cryptosystems. In *CRYPTO*, pages 513–525. Springer.

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.

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.

Fan, J., Guo, X., Mulder, E. D., Schaumont, P., Preneel, B., and Verbauwhede, I. (2010). State-of-the-art of secure ECC implementations: A survey on known side-channel attacks and countermeasures. In *HOST*, pages 76–87.

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

Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. (2009). Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98.

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

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

Lattner, C. and Adve, V. (2004). The llvm compiler framework and infrastructure tutorial. In *LCPC Mini Workshop on Compiler Research Infrastructures*.

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

Moss, A., Oswald, E., Page, D., and Tunstall, M. (2012). Compiler assisted masking. In *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 58–75. Springer.

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.

Standaert, F.-X. (2010). Introduction to Side-Channel Attacks Secure Integrated Circuits an d Systems. In *Integrated Circuits and Systems*, chapter 2, pages 27–42. Springer.