InspectorJ: uma ferramenta de análise estática para detectar código não isócrono em programas Java.

Carina Capelão de Oliveira[1], **Glauco Gonçalves Cardoso**[2], **Fernando Magno Quintão Pereira**[1]

[1]Departamento de Ciência da Computação - UFMG
Avenida Antônio Carlos, 6627 - 31.270-010 - Belo Horizonte - MG - Brasil

[2]Companhia de Tecnologia do Estado de Minas Gerais - PRODEMGE
Belo Horizonte - MG - Brasil

{carina,fernando}@dcc.ufmg.br, glauco.cardoso@prodemge.gov.br

Abstract. *Information flow analysis is a compiler-based technique to reveal security vulnerabilities in software. Several tools implement this kind of analysis; however, none of them let users visualize the flow of information in low-level Java bytecodes. Low-level code analysis is very important for the discovery of side channels. To fill this omission, this paper presents InspectorJ, an open source Java bytecode analyzer. Our tool provides users with an API allowing them to produce rich graphs showing how information flows throughout the program. We have incorporated it into the production environment of Prodemge, the IT company of Minas Gerais' State Government. InspectorJ has pointed out side channels in one of Prodemge's internal products, leading to code patching.*

Resumo. *A análise de fluxo de informações é uma técnica que revela vulnerabilidades em software. Várias ferramentas implementam tal análise, porém nenhuma delas permite a visualização do fluxo de informações em* bytecodes *Java. A análise de código de baixo nível é importante, por exemplo, para a descoberta de canais laterais. Para preencher tal lacuna, este artigo apresenta InspectorJ, um analisador de* bytecodes *Java de código aberto. Nossa ferramenta provê usuários com uma API que lhes permite produzir gráficos mostrando como dados trafegam pelo programa. InspectorJ é usada na Prodemge, a Companhia de Tecnologia de Minas Gerais. Sua utilização revelou potenciais canais laterais em um dos sistemas da empresa, levando a atualizações de código.*

Link para video: `https://www.youtube.com/watch?v=zojXDGa7tBg`

1. Introdução

Para garantir a segurança de aplicações modernas e proteger o capital digital de empresas, é essencial utilizar algoritmos criptográficos robustos e consistentes. No entanto, apesar da crescente força que a criptografia possui, ainda é muito difícil garantir a total segurança dos dados de um programa, uma vez que a corretude de um algoritmo de criptografia não depende apenas do seu projeto abstrato, mas também da sua implementação concreta. Uma das principais vulnerabilidades que podem surgir nesse contexto é a introdução de canais laterais. Canais laterais são falhas de segurança no código de um programa que

permitem a um adversário descobrir informações sigilosas através da observação de comportamentos da implementação que não foram modelados pelo algoritmo. Existem vários tipos de ataques registrados pela literatura que são possíveis devido à presença dessas falhas. Pesquisadores mostraram que atacantes conseguem detectar pequenas diferenças em aspectos mensuráveis durante a execução do programa, como por exemplo seu tempo de execução [Kocher 1996], seu campo magnético [Quisquater and Samyde 2001], seu consumo de energia [Kocher et al. 1999], seus ruídos produzidos [Genkin et al. 2014], além da geração de calor dos processadores [Masti et al. 2015]. Outros tipos de ataques requerem acesso físico ao sistema para observar o comportamento de cache, recuperar dados residuais da memória ou para inserir falhas. Dessa forma, é importante verificar se a implementação real de um algoritmo criptográfico apresenta canais laterais para evitar que o adversário consiga inferir informações sigilosas sobre o algoritmo.

Dentre os ataques que podem ocorrer em algoritmos que apresentam canais laterais está o ataque de canal lateral baseado no tempo, que explora a variação de tempo de execução do programa. Essa variação ocorre quando informações sigilosas controlam o fluxo de informação do algoritmo. Códigos que apresentam essa característica são chamados de não isócronos. Desse modo, é necessário evitar que dados confidenciais afetem o resultado de desvios condicionais em um algoritmo, para garantir que essa variação de tempo não ocorra. Isso pode ser feito de forma manual pelo programador, o que é claramente uma tarefa árdua, pois exige um conhecimento profundo sobre as propriedades e estrutura da linguagem de programação utilizada. Ainda, uma verificação após a compilação do código é necessária, visto que o compilador utilizado pode inserir canais laterais em nível de linguagem intermediária devido a possíveis otimizações realizadas. Inspeções manuais nesse nível são muito difíceis, já que necessitam de análises complexas do código por profissionais que tenham conhecimento sobre a estrutura do compilador e sobre a representação intermediária gerada pelo mesmo.

Neste trabalho apresentamos uma ferramenta que realiza análises de fluxo de informação em programas Java para detectar automaticamente vulnerabilidades de segurança, com foco na detecção de algoritmos não isócronos. Isso permite que desenvolvedores lidem com o problema acima mencionado, a saber, a detecção de canais laterais baseados no tempo. Análises de fluxo de informação são uma das tecnologias mais importantes para encontrar vulnerabilidades de segurança em softwares. Devido a sua importância, desde o trabalho de [Denning and Denning 1977] nos anos setenta, muito tem sido feito para aumentar a precisão e a eficiência de tais técnicas. O processo de implementação dessas técnicas tem tornado possível a sanitização de extensas bases de código, o que aumenta a confiança dos usuários na segurança de sistemas de software. A solução deste trabalho busca detectar vulnerabilidades analisando-se a representação intermediária do programa. Primeiramente precisamos definir dois aspectos sobre o fluxo de informação: a fonte e o sorvedouro [Rimsa et al. 2011]. No caso da detecção de isocronia, a fonte é uma informação sigilosa e os sorvedouros são desvios condicionais. Um grafo de dependências é criado a partir da relação de dependências de dados e de controle entre as variáveis de um programa. A partir deste grafo, conseguimos detectar todos os caminhos que conectam a fonte aos sorvedouros, bem como destacá-los e reportá-los ao desenvolvedor do algoritmo, para que o mesmo modifique o código com o objetivo de eliminar as partes que tornam o algoritmo não isócrono.

Implementamos InspectorJ, uma ferramenta que suporta o desenvolvimento de código isócrono em Java. InspectorJ foi implementado como uma extensão de Soot [Vallee-Rai et al. 1999], um compilador de Java. InspectorJ possui três partes principais: (i) definição de fontes e sorvedouros: define qual informação (fonte) não pode chegar a um dado ponto do programa (sorvedouro); (ii) grafo de dependências: descreve as relações de dependência e controle entre as variáveis do programa; (iii) análise do fluxo de informação: utiliza o grafo para detectar se existe um caminho entre a fonte e o sorvedouro. No caso da não isocronia, a análise detecta se uma informação sigilosa alcança o predicado de um desvio condicional. A saída da ferramenta é um grafo interativo que destaca os caminhos em que a fonte alcança o sorvedouro para reportar a vulnerabilidade. Além disso, também é possível gerar um vídeo que percorre o grafo mostrando os caminhos destacados. Implementamos 21 casos de testes para auxiliar a construção da ferramenta. Esta foi testada no sistema de autenticação da PRODEMGE - a Companhia de Tecnologia do Estado de Minas Gerais. Sua utilização revelou potenciais canais laterais neste sistema, levando a refatorações e atualizações de código. InspectorJ está disponível online para download como uma ferramenta stand-alone.

2. Exemplo de Uso

A Figura 1a mostra um exemplo de programa Java que pode ser usado como entrada para InspectorJ. O código possui a vulnerabilidade de não isocronia que demonstra a capacidade da ferramenta de detectar se o algoritmo é ou não isócrono, além de expor algumas funcionalidades que a mesma provê. As seções a seguir explicam isso em maior profundidade.

```java
public class Comp {
  public boolean comp(String password, String input)
  {
    for (int i = 0; i < 8; i++) {
      if (password.charAt(i) != input.charAt(i))
      {
        return false;
      }
    }
  return true;
  }
}
```

```java
public class Comp2 {
  public boolean comp2(String password, String input)
  {
    int res = 0;
    for (int i = 0; i < 8; i++) {
      res = res | (password.charAt(i) - input.charAt(i));
    }
    return res != 0;
  }
}
```

Figura 1. a) A função ”comp”é um exemplo de programa Java não isócrono, uma vez que *password* alcança um desvio condicional; b) Já a função ”comp2”é isócrona, uma vez que *password* não alcança um desvio condicional.

2.1. Ataques por variação de tempo

Ataques por variação de tempo têm como foco descobrir informações confidenciais tais como uma chave criptográfica. Eles podem ocorrer quando esses dados sigilosos influenciam os predicados, que são estruturas condicionais das instruções de desvio de um programa. Tais estruturas definem as partes do código que serão executadas, controlando o tempo de execução do programa. O ataque se baseia em medir esse tempo para cada dado de entrada fornecido, o que permite ao adversário descobrir alguns ou até todos os bits da informação sigilosa.

A Figura 1a apresenta uma função de comparação de strings, que recebe uma chave *password* e uma entrada do usuário *input*. As strings são comparadas caractere a

caractere, e quando estes diferem, a função retorna. Um retorno antecipado indica que os caracteres iniciais são diferentes, enquanto um retorno tardio indica uma diferença nos últimos bits. Visto que o usuário controla o que será atribuído à variável *input*, esta pode estar contaminada com informações maliciosas a fim de controlar o fluxo do programa, afetando quais regiões do código serão executadas. Assim, variando em ordem lexicográfica o conteúdo de *input*, e medindo o tempo que a função demora para retornar, é possível aprender alguma informação sobre o conteúdo de *password*. Portanto, para evitar esse tipo de ataque, é necessário impedir que o dado secreto alcance os predicados do código, uma vez que um atacante consegue transformar um problema NP-Difícil (descobrir um *password*) em um problema polinomial, basicamente medindo os tempos de execução, quebrando assim o algoritmo criptográfico.

2.2. Fonte e sorvedouro

Além de um programa Java, também é necessário fornecer como entrada para a ferramenta a informação de quem são a fonte e os sorvedouros. Ou seja, qual informação não pode alcançar um dado ponto do programa. No nosso exemplo, em que queremos detectar a não isocronia, a fonte é a variável *password* e os sorvedouros são desvios condicionais.

Com as entradas fornecidas, o programa Java é transformado para Shimple — uma representação intermediária do compilador em formato de Static single assignment form (SSA) [Cytron et al. 1991] — em que todas as variáveis do programa só possuem uma definição. A partir do código Shimple, InspectorJ inicia as análises, criando o grafo de dependências.

2.3. Grafo de dependências

Um grafo de dependências representa as relações de dependências de dados e de controle entre as variáveis do programa, armazenando toda a informação relacionada aos fluxos explícitos e implícitos de um programa. Fluxos explícitos estão relacionados com as dependências de dados: se o código possui uma instrução que define a variável *a* e usa a variável *b*, tal como *a=b+1*, então existe um fluxo explícito de *b* para *a*. Por sua vez, fluxos implícitos estão relacionados com as dependências de controle: se o código possui um desvio condicional, como *if p==0 then a = b+1 else a = b-1*, então existe um fluxo implícito de *p* para *a*, pois o valor de *a* depende do valor de *p*.

InspectorJ analisa o código Shimple para detectar esses fluxos e cria o grafo. Os vértices representam as variáveis do programa, os predicados de desvios e métodos de saída padrão do Java. As arestas podem ser arestas de dados, que representam um fluxo explícito entre duas variáveis, ou arestas de controle, que representam um fluxo implícito entre um predicado e outra variável. No caso da instrução *a=b+1*, criamos uma aresta do vértice *b* para o vértice *a* para indicar que *a* depende de *b* e que o fluxo de informação flui de *b* para *a*.

2.4. Detecção de código não isócrono

A partir do grafo de dependências, InspectorJ é capaz de detectar se o código é não isócrono. Com tal propósito, InspectorJ pesquisa o grafo, buscando caminhos que levem de uma fonte até um sorvedouro. No nosso exemplo, a ferramenta busca por caminhos que conectem a fonte *password* a desvios condicionais.

Para construir tais caminhos, InspectorJ marca como contaminada a variável do programa que foi definida como a fonte. Todo vértice conectado com a fonte e as arestas que fazem essa ligação são também marcados. E assim sucessivamente: todo vértice conectado com um vértice contaminado é também marcado, e esta informação flui ao longo do grafo. Caso essa propagação alcance um desvio condicional, a fonte irá controlar o fluxo do programa, tornando-o suscetível a ataques de canal lateral baseados em tempo. Na Figura 1a, a variável *password* alcança o seguinte desvio: "*if (password.charAt(i) != input.charAt(i))*"; assim, o programa não é isócrono.

Os caminhos encontrados são identificados no grafo e reportados ao usuário. Cada vértice do grafo carrega o nome da variável que ele representa, mais o número da linha do programa Java da instrução que a definiu. O método e classe a qual a variável pertence também são mantidos. Assim, o desenvolvedor consegue facilmente identificar no seu código o ponto em que a fonte alcança o sorvedouro e corrigir o problema. Para tanto, é necessário alterar a implementação para quebrar o fluxo de dependências entre a fonte e o sorvedouro. A Figura 1b mostra um exemplo de como o programa da Figura 1a poderia ser modificado para torná-lo isócrono. Neste exemplo *password* não é comparada diretamente com *input*. É feita uma manipulação com o valor de cada caractere.

2.5. Saída

A saída da ferramenta é um grafo interativo que representa os padrões de dependências no programa, com os caminhos identificados. A Figura 2 mostra um exemplo de grafo. Variáveis são representadas por um quadrado, a fonte por um *X*, os desvios por um losango e variáveis de saída (que representam métodos de saída padrão do Java como *System.out.print*) por um triângulo. Esses métodos são considerados sorvedouros, se alcançados a partir de uma fonte, haverá um vazamento de informação. As arestas de dados são representadas por uma linha contínua, enquanto arestas de controle são representadas por uma linha tracejada. Além disso, variáveis de um dado método e classe são coloridas com a cor designada para tal método. Para identificar os caminhos, as arestas e vértices pertencentes ao mesmo são coloridas de vermelho. A saída para o exemplo de uso está mostrada na Figura 2. O grafo nos mostra que existe uma informação sigilosa, a variável *password* representada como *X*, e dois sorvedouros, o desvio condicional do *loop for* e o desvio condicional do *if*, ambos representados como losangos. A saída reporta que o programa da Figura 1a é não isócrono, uma vez que existe um caminho, indicado de vermelho, entre *password* e um desvio condicional. InspectorJ também disponibiliza o programa Shimple gerado para o programa Java passado como entrada.

3. Interface

InspectorJ é uma ferramenta stand-alone gratuita disponível online para download em `https://github.com/LAC-UFMG-PRODEMGE/inspectorj`. É baseada no framework Soot, um compilador que analisa bytecodes Java e aplicações Android. Soot recebe um programa como entrada, transforma-o em uma de suas representações intermediárias, realiza análises e otimizações, e retorna o programa, que pode ou não estar no seu formato original, dependendo da escolha do usuário.

Assim, a ferramenta está estruturada da seguinte forma: primeiro, ela recebe um programa Java e o transforma, via Soot, em um programa Shimple, gerando o arquivo

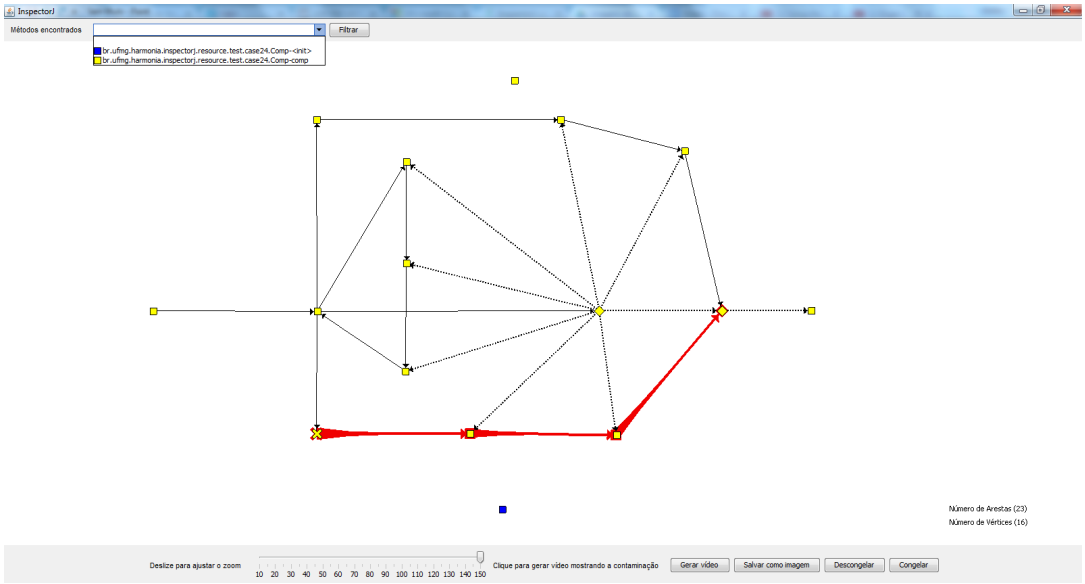

Figura 2. Grafo resultante da análise do Exemplo da Figura 1a. Os nodos representam as variáveis, e as arestas indicam fluxo de dados entre as variáveis. InspectorJ detecta um trecho não isócrono, uma vez que existe fluxo de dados entre uma informação privada e um condicional (caminho destacado no grafo).

`file.Shimple`. Neste, todas as variáveis possuem apenas uma definição e todas as intruções têm no máximo três endereços. InspectorJ realiza as análises sobre o arquivo Shimple e cria o grafo de dependências. A criação deste foi feita por meio da biblioteca GraphStream [Pigné et al. 2008], que tornou a saída da ferramenta interativa, com animações visualmente atraentes e interessantes sobre como a informação flui dentro do programa, facilitando o uso e entendimento da saída de InspectorJ.

3.1. Entrada

Antes de utilizar InspectorJ é necessário informar as entradas. Isto é feito através de um arquivo de propriedades em que o usuário pode digitar as informações necessárias. Deve-se informar o nome do método, assim como o nome da classe em que ele se encontra, que será o ponto de partida da análise; o nome da variável que será a fonte; o nome do método em que a fonte está, assim como o nome da classe em que o método se encontra; e os sorvedouros. Além disso, o usuário deve informar um *classpath*: o caminho onde a ferramenta irá procurar as classes que serão carregadas para análise. Também é necessário informar o nome do diretório em que as saídas da ferramenta serão salvas. A Figura 3 mostra como seria esse arquivo de propriedades para o exemplo da Figura 1a.

```
inicial.metodo= comp
inicial.classe= nomedomeupacoteaqui.Comp
fonte = password
fonte.metodo = comp
fonte.classe = nomedomeupacoteaqui.Comp
sorvedouro = printout,condicional
classpath = "caminho do local em que as classes que serao analisadas estao"
diretorio.saida = ".../saidaInspectorJ"
```

Figura 3. Exemplo de arquivo de configuração da ferramenta InspectorJ. Nesta figura, mostramos as configurações necessárias para analisar o programa da Figura 1a usando InspectorJ.

3.2. Saída

A Figura 2 mostra a interface de saída de InspectorJ para o programa da Figura 1a. Além do grafo, InspectorJ também disponibiliza o arquivo Shimple gerado durante a análise para facilitar ainda mais o entendimento. A ferramenta possui várias funcionalidades: **filtrar** o método que será representado no grafo sem mostrar como a informação flui das variáveis desse método para outros métodos, utilizando o botão "filtrar"na parte superior à esquerda; **alterar** o "zoom"no grafo para visualizar melhor determinada parte do mesmo, utilizando a barra que se encontra na parte inferior à esquerda, ou pela rolagem do mouse; **caminhar** pelo grafo utilizando as setas do teclado ou o mouse, e também **arrastar** o grafo com o mouse; **congelar** o grafo para arrastar somente um vértice, sem que todo o grafo se desloque, utilizando os botões "Congelar"e "Descongelar"na parte inferior da tela; **visualizar** o nome das variáveis, com o nome do seu método e a linha da instrução em que a mesma foi definida no programa Java, ao colocar o mouse sobre o vértice; **visualizar** as informações anteriores de todos os vértices do grafo, quando se usa um zoom maior que 50%; **visualizar** dados como número de vértices e arestas do grafo, disponíveis na parte inferior à direita; **salvar** uma imagem do grafo, através do botão "Salvar Imagem"na parte inferior; caso o programa seja não isócrono, **gerar** um vídeo que percorre o grafo mostrando o caminho, passando por todas as variáveis que pertencem a ele, através do botão "Gerar vídeo", também na parte inferior da tela.

4. Ferramentas relacionadas

A ferramenta mais próxima de InspectorJ é FlowTracker [Rodrigues et al. 2016]. Muitas das decisões de projeto de InspectorJ foram inspiradas naquela ferramenta. Contudo, há várias diferenças entre esses dois projetos, tanto do ponto de vista de engenharia, quanto do ponto de vista de usabilidade. Com relação ao primeiro aspecto, InspectorJ é implementado em Java, e analisa a representação intermediária do compilador Soot. Este compila bytecodes Java, o que inclui programas escritos em Java, Scala, Groovy e Closure, por exemplo. Já FlowTracker é implementada em C++, e analisa a representação intermediária de LLVM, o que inclui programas escritos em C, C++ e Rust. O fato de InspectorJ ser voltado para bytecodes Java torna a sua implementação muito diferente de FlowTracker: por um lado, não existe, no projeto de InspectorJ, a preocupação com acessos fora de memória alocada. Por outro lado, a invocação dinâmica de funções é um problema com o qual InspectorJ lida, mas que não existe em FlowTracker. Além disso, as duas ferramentas são muito diferentes, do ponto de vista de usabilidade. InspectorJ é utilizada como uma ferramenta stand-alone via uma interface gráfica. Já FlowTracker é utilizada via uma interface web (`http://cuda.dcc.ufmg.br/flowtracker/`), ou via linha de comando. Do ponto de vista teórico, InspectorJ apresenta uma funcionalidade que não está presente em FlowTracker, a possibilidade de inserir guardas no código analisado, a fim de aumentar a sua segurança.

Outra ferramenta similar foi implementada por [Lux and Starostin 2011], que também detecta vulnerabilidades de ataques por variação de tempo em programas Java. A principal diferença entre InspectorJ e a abordagem de Lux et al. é que eles analisam a linguagem de programação em alto nível, utilizando um algoritmo de inferência de tipos. Além disso, utilizam uma política de fluxo de informação que fornece uma orientação para a análise, em que é necessário especificar outras entidades de programa como detendo informações confidenciais se o conteúdo dessas entidades potencialmente

dependem de segredos ao executar o programa. Por sua vez, InspectorJ trabalha diretamente na representação intermediária do compilador, realizando a análise em um nível de abstração mais baixo. Trabalhamos com a idéia de dependência de dados e de controle, analisando diretamente o código e criando ligações entre as variáveis. Assim, o usuário deve informar apenas a variável que contém a informação secreta, de forma que outras variáveis recebam informações sigilosas caso possuam alguma ligação com a fonte.

Existem outras ferramentas de análise estática de código Java, que empregam técnicas similares à InpsectorJ: Sonarqube [Sonarqube 2008], Checkstyle [Burn 2017], Findbugs [Hovemeyer and Pugh 2004], PMD [Kleiber 2009] e JFlow [Myers 1999]. Entretanto, nenhuma delas lida com a questão de ataque por variação de tempo.

5. Considerações finais

Este artigo apresentou InspectorJ, uma ferramenta de análise estática que analisa fluxos de informação em programas Java para detectar automaticamente vulnerabilidades de segurança, com foco na detecção de algoritmos não isócronos. A ferramenta rastreia dependências de controle e de dados entre as variáveis do programa, e informa se uma informação sigilosa alcança um ponto indesejável do mesmo. Desenvolvedores Java podem utilizá-la para auxiliá-los no desenvolvimento de algoritmos isócronos. Desta forma, a utilização da ferramenta para automatizar a análise de software é efetiva e útil em um ambiente corporativo, uma vez que reduz o custo de verificar a corretude e a robustez de sistemas de softwares. Acredita-se que InspectorJ seja a primeira ferramenta capaz de certificar se um programa Java tem comportamento isócrono em nível de compilador, utilizando análise estática. Como trabalho futuro, pretendemos integrar nossa ferramenta à IDE Eclipse [Eclipse 2001], que hoje é amplamente usada pelos desenvolvedores Java.

Referências

Burn, O. (2017). Checkstyle v7.7. http://checkstyle.sourceforge.net/.

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

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

Eclipse (2001). The eclipse foundation open source community website. In *https://www.eclipse.org/*.

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

Hovemeyer, D. and Pugh, W. (2004). Finding bugs is easy. In *SIGPLAN*, pages 92–106, New York, NY, USA. ACM.

Kleiber, T. (2009). Pmd. https://pmd.github.io/.

Kocher, P., Jaffe, J., , and Jun, B. (1999). Differential power analysis. In *CRYPTO*, pages 388—397, Heidelberg, Alemanha. Springer.

Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, Heidelberg, Alemanha. Springer.

Lux, A. and Starostin, A. (2011). A tool for static detection of timing channels in java. In *Journal of Cryptographic Engineering*. Springer-Verlag.

Masti, R. J., Rai, D., Ranganathan, A., Muller, C., Thiele, L., and Capkun, S. (2015). Thermal covert channels on multi-core platforms. In *Security Symposium*, pages 865—880, Berkeley, CA, USA. USENIX.

Myers, A. C. (1999). Jflow: Practical mostly-static information flow control. In *POPL*, New York, NY, USA. ACM.

Pigné, Y., Dutot, A., Guinand, F., and Olivier, D. (2008). Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. In *ECCS*.

Quisquater, J.-J. and Samyde, D. (2001). Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *LNCS*, pages 200–210, Heidelberg, Alemanha. Springer.

Rimsa, A., Quintão Pereira, F. M., and d'Amorim, M. (2011). Tainted flow analysis on e-ssa-form programs. In *CC*, pages 122 – 141, Heidelberg, Alemanha. Springer.

Rodrigues, B., Quintão Pereira, F. M., and Aranha, D. F. (2016). Sparse representation of implicit flows with applications to side-channel detection. In *CC*, New York, NY, USA. ACM.

Sonarqube (2008). The leading platform for continuous code quality. In *https://www.sonarqube.org/*.

Vallee-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (1999). Soot - a java bytecode optimization framework. In *CASCON*. IBM Press.