Aplicação Dinâmica de Políticas Estáticas de Fluxo

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 today one of the most important techniques to find vulnerabilities in programs. Since its conception, this technique has evolved to achieve great precision and scalability. However, being static, information flow analysis has one shortcoming: to ensure soundness, it must conservatively refuse some programs which are well-known to be safe. Even exponential-time algorithms cannot change this scenario, as long as they remain static. To address this problem, in this paper we propose a combination of static analysis with dynamic checks to increase the precision of information flow techniques. Our dynamic checks let us rule out unsafe paths during the execution of the program while preserving secure flows of information. In this way, we can certify the safety of programs even in face of features which are difficult to analyze statically, such as subtyping polymorphism. To validate our ideas, we have materialized them into a tool, InspectorJ, which inserts instrumentation in a program to prevent the execution of unsafe paths.*

Resumo. *A análise de fluxo de informação é atualmente uma das técnicas mais importantes para descobrir vulnerabilidades em programas. Desde a sua concepção, esta técnica evoluiu ao ponto de alcançar grande precisão e escalabilidade. Entretanto, por ser estática, a análise de fluxo de informação é conservadora e, para garantir corretude, ela precisa invalidar alguns programas que são conhecidamente seguros. Enquanto a análise permanece totalmente estática, nem mesmo algoritmos exponenciais conseguem mudar esse cenário. Para abordar esse problema, propomos uma combinação de análise estática e checagem dinâmica para ampliar a precisão de técnicas de análise de fluxo de informação. Nossas checagens dinâmicas nos permitem detectar e descartar caminhos inseguros durante a execução do programa e preservar fluxos seguros de informação. Dessa forma, podemos garantir a segurança de programas mesmo diante de propriedades difíceis de serem analisadas estaticamente, como por exemplo polimorfismo de subtipagem. Para validar nosso trabalho, materializamos nossas ideias em uma ferramenta chamada InspectorJ, que insere instrumentação em um programa para prevenir a execução de caminhos inseguros.*

1. Introdução

Análises de fluxo de informação são uma das tecnologias mais importantes para encontrar vulnerabilidades de segurança em software. Devido à sua importância, desde o trabalho de

Denning e Denning [Denning and Denning 1977] nos anos setenta, muito tem sido feito para aumentar a precisão e a eficiência de tais técnicas. Dentre as implementações estado da arte atualmente em uso, encontram-se abordagens esparsas [Bruno Rodrigues 2016], dinâmicas [Nethercote and Seward 2007], sensíveis ao fluxo [Hunt and Sands 2006] e sensíveis ao contexto. O processo de implementação de técnicas de fluxo de informação tem tornado possível a sanitização de extensas bases de código, o que aumenta a confiança dos usuários na segurança de sistemas de software que eles usam.

Entretanto, apesar de todos os desenvolvimentos recentes, existe ainda um empecilho à implementação de análises de fluxo totalmente precisas: a dificuldade para analisar e entender automaticamente linguagens que usam polimorfismo de subtipagem acoplado com invocação dinâmica. Essas características, comuns em linguagens de programação orientadas a objeto, comprometem a precisão de análises de fluxo, pois as mesmas precisam rastrear todos os possíveis alvos de qualquer invocação de métodos. Para lidar com esse tipo de comportamento de execução, técnicas de fluxo geralmente recorrem a análises dinâmicas [Russo and Sabelfeld 2010]. Contudo, essa alternativa traz uma desvantagem significativa: mesmo abordagens sofisticadas podem prejudicar a performance de programas em até 35% [Clause et al. 2007].

Neste artigo, nós apresentamos uma solução para mitigar esse *overhead*. Projetamos, implementamos e testamos uma técnica que combina análise estática de fluxo de informação com checagem dinâmica, para possibilitar a validação de programas que, devido ao uso de polimorfismo de subtipagem, seriam considerados inseguros. Resumidamente, usamos uma técnica de análise estática para descobrir o maior número possível de caminhos seguros no programa, além de caminhos inseguros e caminhos que podem ser inseguros dependendo de algum valor conhecido apenas dinamicamente. A esses caminhos que não somos capazes de validar durante a primeira etapa, inserimos predicados de checagem dinâmica. Esses predicados são inseridos apenas em pontos de chamada que podem invocar métodos inseguros. Como mostraremos nas próximas seções, o número de chamadas desse tipo é extremamente limitado, o que leva nossa abordagem a adicionar um *overhead* muito pequeno aos programas validados. Na seção 2, apresentamos uma rápida visão geral dessas ideias, usando um exemplo para explicá-las ao leitor.

Para validar nosso trabalho, materializamos nossas ideias em uma ferramenta chamada InspectorJ, implementada no Soot, um framework para análise e otimização de bytecodes Java [Vallee-Rai et al. 1999]. A ferramenta analisa programas Java e insere instrumentação no código para prevenir a execução de caminhos inseguros. Nós configuramos nosso mecanismo de fluxo de informação para detectar um ataque conhecido como *command injection*. *Command injection* é um ataque cujo objetivo é executar comandos arbitrários no sistema operacional do cliente através de uma aplicação vulnerável. Este ataque é possível quando uma aplicação transfere dados inseguros fornecidos pelo usuário para um shell do sistema. Nossa análise foi capaz de detectar *command injection* em uma aplicação para o Flickr que está disponível em um diretório público no GitHub, que utiliza polimorfismo de subtipagem e invocação dinâmica. Detectamos o caminho inseguro e fomos capazes de instrumentar o código para previnir a execução de tal caminho, que atravessa as fronteiras de pelo menos 2 métodos[1]. Esse experimento nos dá embasamento

[1]O comprimento exato desse caminho, em número de invocações de método, depende do polimorfismo de subtipagem.


```
1  public static class T{
2      boolean comp(String password, String input){
3          int res = 0
4          for (int i=0; i < 8: ++i){
5              res = res | (password.charAt(i) - input.charAt(i));
6          }
7          return res == 0;
8      }
9  }
10
11 public static class U extends T {
12     boolean comp(String password, String input) {
13         for (int i = 0; i < 8; i++) {
14                 if (password.charAt(i) != input.charAt(i)) {
15                 return false;
16             }
17         }
18         return true;
19     }
20 }
21
22 public static void main(String args[]) {
23     T o = args.lenght % 2 == 0 ? new U() : new T();
24     o.comp(args[0], args[1]);
25 }
```

Figura 1. Exemplo de programa Java não isócrono que apresenta polimorfismo de subtipagem e invocação dinâmica.

para validar, ao menos preliminarmente, a seguinte conjectura: o número de pontos de chamada que não podem ser completamente desambiguados por análises estáticas é pequeno o suficiente para tornar a aplicação de checagem dinâmica barata o suficiente para ser usada na prática. Esse experimento e outros adicionais serão discutidos na seção 5.

2. Visão Geral

Este artigo descreve técnicas que melhoram a precisão de análises de fluxo de informação. Tais técnicas podem ser usadas para apontar, automaticamente, uma vasta gama de vulnerabilidades em software. A fim de exemplificar nossas ideias, escolhemos, dentre tais vulnerabilidades, um problema de segurança bem conhecido: canais laterais baseados em tempo. Ataques por variação de tempo têm como foco inferir 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. Nesta seção, usaremos o programa da Figura 1 como exemplo. O código é não isócrono, portanto é suscetível a ataques por variação de tempo. Além disso, ele apresenta polimorfismo de subtipagem e invocação dinâmica de métodos: duas características que tornam programas difíceis de analisar, e com as quais lidamos.

A Figura 1 apresenta um programa Java que possui duas classes, *T* e *U*, e um método *main*. A classe *U* possui uma função *comp* 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. Dessa forma, o método *comp* da classe *U* é inseguro.

Por sua vez, a classe *T* também possui um método *comp* de comparação de strings, com algumas diferenças. As strings não são comparadas diretamente, de forma que a chave *password* não alcança nenhum desvio condicional. É feita uma manipulação de bits com os caracteres das strings: uma variável *res* é inicializada com 0, e é feita uma operação de *OR* entre essa variável e a diferença entre os caracteres das duas strings. Se algum caractere for diferente, *res* será diferente de 0, e a função retorna *false*. Caso contrario, *res* será sempre 0, e a função retorna *true*. Outro ponto importante é que o método não realiza retornos antecipados ou tardios, uma vez que ele sempre percorre toda a string antes de retornar o resultado. Portanto, o método dessa classe é seguro.

Ainda na Figura 1 temos o método *main*. Uma variável *o* do tipo da classe *T* é definida, de forma que o seu objeto instanciado será definido apenas em tempo de execução, podendo ser do tipo da classe *T* ou da classe *U*. Isso ocorre pois o tipo do objeto instanciado depende do valor de *args.length*, em que *args* é recebida por parâmetro pela função, portanto definida apenas durante a execução. Após a definição de *o*, o objeto chama o método *comp*. Uma vez que a classe U estende a classe *T*, e as duas possuem o método *comp*, não é possível saber estaticamente qual método será chamado. Porém, como o método U é vulnerável, o programa será inseguro quando o objeto *o* for do tipo *U*, e será seguro quando o objeto for do tipo *T*.

Para garantir a segurança do código, e para não dizer que o mesmo possui uma falha independente do tipo de *o* para proteger o programa, devemos identificar quando este é inseguro. Para isso, rastreamos todos os possíveis caminhos de qualquer chamada de método, e identificamos o caminho contaminado que deve ser evitado. Assim, para impedir que o método seja executado caso o tipo de *o* seja *U*, inserimos guardas no código, que checam em tempo de execução se o objeto que chamará a função *comp* é do tipo da classe *U*. Em caso positivo, instrumentamos o código para lançar uma exceção antes da chamada de *comp*. O código resultante da instrumentação é mostrado na Figura 2. Dessa forma, uma vez instrumentado, o programa estará seguro, pois o método inseguro não será executado.

Essa combinação de análise estática com checagem dinâmica para aumentar a precisão da técnica de fluxo de informação cobre mais casos em que o programa seria considerado inseguro conservativamente. Considerando o exemplo da Figura 1, se fosse feita apenas uma análise estática, então o programa *inteiro* seria considerado inseguro, embora somente um método apresentasse uma potencial vulnerabilidade – independente do tipo do objeto instanciado. Supondo que a primeira linha do método *main* da Figura 1 fosse *T o = args[0] == "0"? new U() : new T();*. Nesse caso se *args[0]* fosse igual a 0 o tipo do objeto instanciado seria igual a *U*, tornando o código vulnerável. Para qualquer


```
1  public static void main(String args[]) {
2      T o = args.lenght %2 == 0 ? new U() : new T();
3
4      // BEGIN AUTOMATIC CODE:
5      if (o instanceof U)
6          throw new Error("C\'odigo Inseguro");
7      // END AUTOMATIC CODE.
8
9      o.comp(args[0], args[1]);
10 }
```

Figura 2. Instrumentação resultante da análise feita no programa da Figura 1.

outro valor de *args[0]* que fosse diferente de 0, o tipo do objeto instanciado seria igual a *T*, resultando em um código seguro[2]. Dessa forma, na maioria das vezes o programa não é vulnerável, mas seria considerado como tal caso utilizássemos apenas análise estática. Utilizando a checagem dinâmica juntamente com a análise estática, diminuímos o número de falsos positivos, enquanto mantemos o programa seguro.

3. A Análise Estática

Para detectar vulnerabilidades de segurança nos programas Java utilizando técnicas de análise estática de fluxo de informação, nossa abordagem segue três etapas. Primeiramente, analisamos o código estaticamente para rastrear o fluxo de informação do código, identificando dependências de dados e de controle entre as variáveis do programa. Em segundo lugar, criamos um grafo de dependências, a partir do rastreamento feito anteriormente, para representar e analisar como as variáveis estão relacionadas. Finalmente, realizamos uma análise de fluxo contaminado no grafo criado para identificar os caminhos seguros e inseguros e, assim, detectar uma vulnerabilidade de segurança, como por exemplo o ataque por variação de tempo. Essas etapas serão melhor explicadas a seguir.

3.1. Rastreando o fluxo

Para ter uma análise mais eficaz e identificar também as falhas que podem ser introduzidas pelo compilador, nossa análise trabalha em nível de compilador, analisando a representação intermediária do programa gerada pelo mesmo. Tal representação intermediária possui o formato de *Static Single Assignment form (SSA)* [Cytron et al. 1991], em que cada variável tem somente um ponto de definição no código intermediário. Além disso, se existe um desvio condicional, tal como *if else*, em que uma variável é definida nos dois caminhos possíveis, então ao final do desvio é inserida uma função *phi*. Funções *phi* são da forma $\mathsf{i1} = \phi(\mathsf{i0}, \mathsf{i2})$ e são usadas para unificar múltiplas definições da mesma variável — i0 e i2 — em um nome único, no caso i1. Outra característica dessa representação intermediária é que todas as instruções possuem no máximo três endereços. Essas duas características fazem com que novas variáveis que antes não existiam no código original sejam criadas nesse novo programa.

Em seguida, é feito um rastreamento do fluxo no novo programa gerado para identificar dependências de dados e de controle entre as variáveis do código, em que *dependência de dados* e *dependência de controle* são definidas como:

[2]Esse exemplo é bastante trivial e dificilmente aconteceria em um cenário real, mas nos auxilia a entender o problema.

Definição 3.1 (Dependência de Dados) *Uma variável v depende de dados de uma variável u se uma instrução que usa u for usada para definir v. Um exemplo seria* v=u+2. *Dessa forma, dizemos que existe um fluxo explícito de u para v.*

Definição 3.2 (Dependência de Controle) *Uma variável v é dependente de controle de uma variável u se a definição de v depende do valor de u. Um exemplo seria* se u então v=0. *Dessa forma, dizemos que existe um fluxo implícito de u para v.*

3.2. Grafo de Dependências

Para representar as relações de dependências do programa, criamos um *grafo de dependências*, definido como:

Definição 3.3 (Grafo de Dependências) *Um grafo de dependências é criado a partir da relação de dependência de dados e de controle entre as variáveis do programa. Cada vértice "n_v"do grafo representa uma variável "v"do programa e os "sorvedouros"de uma análise específica, que serão definidos posteriormente. Cada aresta "a_{uv}"representa uma dependência. Arestas sólidas representam dependências de dados e arestas não sólidas representam dependências de controle. Podemos assegurar que cada vértice corresponde a uma e somente uma variável no programa devido ao formato SSA do programa analisado.*

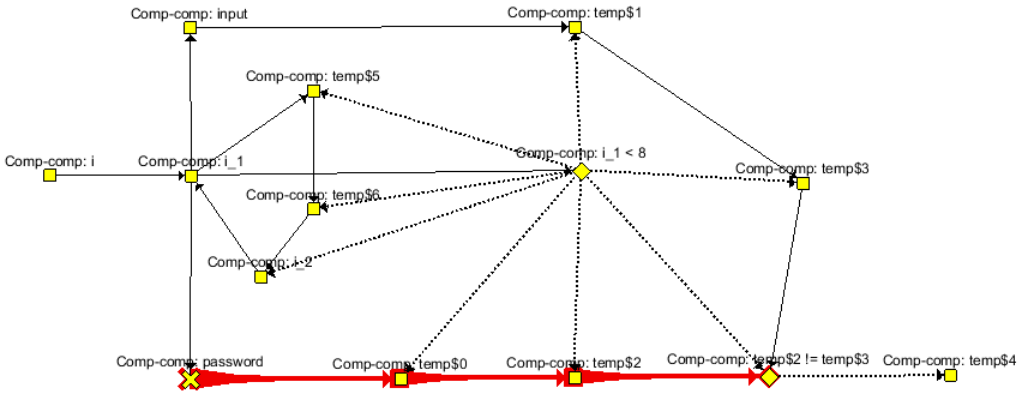

Figura 3. Grafo de dependências resultante para o método comp da classe U do exemplo da Figura 1.

O grafo nos permite visualizar como os dados trafegam dentro do programa, o que facilita sua análise e entendimento. A Figura 3 mostra o grafo produzido para o método *comp* da classe *U* da Figura 1. Uma variável é representada como um quadrado, um predicado de desvio condicional é um losango e a variável definida como fonte na análise é representada como um X. 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*. Uma aresta não sólida de *p* para *q* indica que o programa contém um teste condicional sob o predicado *p*, de forma que o valor a ser atribuído à variável *q* depende desse teste.

3.3. Análise de fluxo contaminado

Para detectar um determinado tipo de vulnerabilidade, como o ataque por variação de tempo explicado na seção 2, realizamos uma análise de fluxo contaminado no grafo criado. Definimos *análise de fluxo contaminado* como:

Definição 3.4 (Análise de Fluxo Contaminado) *A análise de fluxo contaminado recebe uma tripla T = (G, Fonte, Sorvedouro), em que G é o grafo de dependências definido anteriormente. Uma fonte é um dado fornecido pelo usuário, ou seja, é uma informação vinda do canal de entrada, que é passado para o programa. Um sorvedouro é um ponto sensível do programa, ou seja, são funções que realizam operações sensíveis (tal como escrever uma informação no canal de saída). Um caminho contaminado é um caminho no grafo de dependências entre a fonte e o sorvedouro. Assim, a análise de fluxo contaminado detecta se existe um caminho no grafo de dependências entre uma fonte e um sorvedouro. [Rimsa et al. 2011]*

Dizemos que um caminho é contaminado quando uma fonte alcança um sorvedouro pois a fonte é uma informação vinda do usuário, podendo conter dados maliciosos criados com o propósito de realizar algum ataque no sistema. Para cada tipo de vulnerabilidade a ser detectada, vamos definir uma fonte e sorvedouro específicos. Considerando o exemplo da Figura 1, em que queremos detectar se existe uma variação no tempo, a fonte seria a variável *password* do método *comp*, representada no grafo na Figura 3 como um X. Por sua vez, o sorvedouro, representado como um losango, seria o desvio condicional presente na linha 14. Como existe um caminho entre a fonte e o sorvedouro, identificamos esse caminho destacando-o de vermelho, como pode ser visto no grafo.

Para construir tais caminhos, a análise 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. Se um vértice marcado alcançar o sorvedouro, o caminho é contaminado. Os caminhos encontrados são identificados no grafo e reportados ao usuário. Cada vértice do grafo carrega o nome da variável que ele representa, além de conter o número da linha do programa Java da instrução que a definiu. O método e classe aos quais 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.

4. A checagem dinâmica

Acoplada à análise estática, realizamos uma checagem dinâmica para possibilitar a validação de programas que, devido ao uso de polimorfismo de subtipagem e invocação dinâmica, seriam considerados inseguros. Esta segunda parte da análise possui 2 etapas. Primeiramente, tratamos o polimorfismo de subtipagem a fim de descobrir todos os tipos que um determinado objeto poderia assumir dinamicamente. Depois, identificamos todas as invocações dinâmicas feitas e instrumentamos o código nestes pontos afim de evitar que código inseguro seja executado. Essas etapas serão melhor explicadas a seguir.

4.1. Análise de polimorfismo de subtipagem

Uma característica comum em linguagens de programação orientadas a objeto é o uso de polimorfismo de subtipagem com invocação dinâmica. O polimorfismo de subtipagem ocorre quando um determinado objeto é definido com um tipo que tem vários subtipos, em que o subtipo vai ser definido no momento da instanciação. Um exemplo disso é quando uma classe possui várias subclasses que a estendem, ou quando uma interface é implementada por várias classes. Assim, quando todas as subclasses de uma classe possuem um método com a mesma assinatura, que realiza operações diferentes para cada

subclasse, só saberemos qual método vai realmente executar em uma chamada se soubermos o subtipo do objeto. Entretanto, se a instanciação de um objeto é feita apenas dinamicamente, não é possível saber qual o tipo instanciado do objeto através apenas da realização de análise estática. Considere por exemplo o método main da Figura 1. A variável *o* é definida como do tipo da classe *T*, em que seu objeto instanciado será definido apenas em tempo de execução, podendo ser do tipo da classe *T* ou da classe *U* (como foi explicado na seção 2). No momento em que o objeto *o* chama o método *comp*, não sabemos se o método chamado será da classe *T*, que é seguro, ou da classe *U*, que é inseguro. Assim, para detectar e descartar caminhos inseguros durante a execução do programa e preservar fluxos seguros de informação, devemos rastrear todos os possíveis tipos que um objeto pode assumir, e assim rastrear todos os possíveis alvos de qualquer invocação de métodos.

Para isso, durante a fase de rastreamento de fluxo da análise estática, realizamos uma análise de polimorfismo de subtipagem. Para variáveis de tipos primitivos - como um *int* - que possuem apenas tipo estático, a análise é feita como descrita na seção 3. Para objetos que podem assumir mais de um tipo dinâmico, é necessário uma análise extra. Tipo estático e tipo dinâmico são definidos como:

Definição 4.1 (Tipo Estático) *O Tipo Estático de um objeto é o tipo usado em sua declaração. Tal tipo nunca muda durante a execução do programa, uma vez que ele é parte do texto daquele programa.*

Definição 4.2 (Tipo Dinâmico) *O Tipo Dinâmico de um objeto é o tipo com o qual ele é instanciado, e pode mudar sempre que o objeto recebe uma nova instância. Tal tipo pode mudar durante a execução do programa se o tipo a ser instanciado depender de algum valor definido apenas dinamicamente.*

No exemplo da Figura 1, no método *main*, o objeto *o* pode ter mais de um tipo dinâmico, uma vez que o tipo instanciado depende do tamanho da variável *args*, valor este conhecido apenas dinamicamente. Dessa forma, o tipo do objeto *o* é definido apenas em tempo de execução, e portanto o mesmo pode assumir tipos diferentes em cada execução do programa. Por sua vez, a variável *args* possui apenas tipo estático, uma vez que ela possui tipo *builtin*, que não podem ter subtipos em Java.

Para todos os objetos que possuem tipo dinâmico, fazemos uma análise de definições alcançáveis de cada um desses objetos para identificar todos os possíveis tipos que o objeto pode assumir durante a execução do programa, e armazenamos tais dados sobre cada objeto junto com o objeto em si para uso posterior. Quando encontramos um ponto de chamada de um método feita por um objeto *o*, salvamos a linha do programa Java em que a chamada foi feita. Além disso, analisamos o método de cada um dos tipos que o objeto *o* pode ser, descobrindo e criando todos os possíveis caminhos que o programa pode seguir, como descrito na seção anterior, para descobrir se algum deles é inseguro. Considerando ainda o exemplo da Figura 1, no momento em que o objeto *o* chama o método *comp*, identificamos que *o* pode ser do tipo da classe *T* ou *U*, e então analisamos o método *comp* das duas classes, criando caminhos no grafo de dependências para os dois métodos. Como identificamos que o método *comp* da classe *U* é inseguro, vamos destacar tal caminho no grafo. Entretanto, uma vez que o tipo do objeto *o* é dinâmico, não sabemos se o método inseguro executará ou não. Para proteger o programa sem perder precisão, partimos para a segunda etapa da nossa checagem dinâmica.

4.2. Instrumentação de código

Para todos os caminhos contaminados reportados no grafo de dependências, identificamos quais deles foram gerados a partir de uma invocação dinâmica de um método, como a invocação do método *comp*. Para cada método invocado, detectamos a qual classe ele pertence e inserimos uma guarda no código Java antes da chamada do método. Uma guarda é definida como:

Definição 4.3 (Guarda) *Uma Guarda é um predicado p(Cl, o, m), que permite a invocação do método m sobre o objeto o, sempre que o mesmo não for instância da classe Cl.*

No exemplo da Figura 1, a guarda inserida antes da chamada do método comp seria p(U,o,comp), significando que o método comp só será invocado sobre o objeto *o* se o objeto não for uma instância da classe U. Se o objeto *o* for uma instância da classe U, uma exceção será lançada informando que o código é inseguro, impedindo que o mesmo seja executado. A instrumentação resultante do exemplo da Figura 1 é mostrada na Figura 2. Dessa forma, essa instrumentação do programa previne a execução de caminhos inseguros, uma vez que uma guarda é inserida antes de uma invocação de um método detectado como inseguro. Tal instrumentação de código é feita diretamente em código Java, ao invés de ser feita no bytecode. A mesma é realizada dessa maneira para que o desenvolvedor que utilizou nossa ferramenta consiga ver e identificar o problema e o local em que a guarda foi inserida. Caso contrário, como no bytecode seria difícil identificar o código inserido, um usuário da ferramenta poderia questionar se o problema realmente existia no código ou se foi, erroneamente, gerado durante a instrumentação do mesmo.

5. Avaliação Experimental

Materializamos nossas ideias em uma ferramenta chamada InspectorJ, implementada no Soot, um framework para análise e otimização de bytecodes Java. Nós configuramos nosso mecanismo de fluxo de informação para detectar *command injection* e ataques por variação de tempo. Os experimentos foram realizados no sistema operacional Windows 7 Professional de 64 bits, com processador Intel Core i3 - 2120, com clock de 3.30GHZ e 8GB de memória RAM. Para validar a efetividade de nossas ideias, utilizamos a nossa ferramenta em uma aplicação para o Flickr, em que tal experimento será discutido como um estudo de caso, além de aplicar a ferramenta em uma série de *benchmarks*, para realizar uma análise qualitativa e uma análise de complexidade.

5.1. FlickrFaves - Estudo de Caso

O Flickr é um aplicativo online de gerenciamento e compartilhamento de fotos e vídeos com o mundo inteiro[3]. Por sua vez, FlickrFaves é uma pequena aplicação para o Flickr que permite que o usuário faça downloads em alta resolução de suas fotos e vídeos favoritos do Flickr. Tal aplicação está disponível em um diretório público no GitHub[4]. FlickrFaves é uma ferramenta multi-plataforma, escrita em Java. Dentre as classes implementadas em tal aplicação, existe uma classe que tem como finalidade detectar o sistema operacional em que a ferramenta está sendo utilizada. Isso permite que a *url* do Flickr

[3]https://www.flickr.com/

[4]https://github.com/magnusvk/FlickrFaves

```
1  public class BareBonesBrowserLaunch {
2      public static void openURL(String url) {
3          IBrowser browser = new BrowserFactory().getBrowser();
4          browser.launch(url);
5      }
6  }
7
8  public class BrowserFactory {
9      public IBrowser getBrowser(){
10         String osName = System.getProperty("os.name");
11         IBrowser browser = null;
12         if (osName.startsWith("Mac OS")) {
13             browser = new OSXBrowser();
14         } else if (osName.startsWith("Windows"))
15             browser = new WindowsBrowser();
16         else { // assume Unix or Linux
17             browser = new GenericBrowser();
18         }
19         return browser;
20     }
21 }
22
23 public class WindowsBrowser implements IBrowser {
24     public void launch(String url) throws Exception {
25         Runtime.getRuntime().exec("rundll32 url.dll,
               FileProtocolHandler " + url);
26     }
27 }
```

Figura 4. Código que detecta o sistema operacional em que o FlickrFaves está executando. Isso permite que a url passada como parâmetro seja aberta no browser específico de cada sistema operacional.

recebida pelo usuário seja aberta no *browser* específico de cada sistema operacional, possibilitando assim que a ferramenta seja multi-plataforma. As partes principais do código dessa funcionalidade são apresentadas na Figura 4, em que o código foi refatorado para se tornar mais modularizado, facilitando assim o entendimento da funcionalidade da classe.

A classe *BareBonesBrowserLaunch* possui um método *openURL* que recebe a *url* do Flickr que o usuário deseja abrir. Um objeto *browser* é declarado com o tipo estático *IBrowser*, que é uma interface que possui o método *launch*, implementada por três classes: *OSXBrowser*, *WindowsBrowser* e *GenericBrowser*, cada uma delas representando os sistemas operacionais OSX, Windows e Linux/Unix, respectivamente. O tipo dinâmico do objeto instanciado de *browser* será definido em tempo de execução através do método *getBrowser* da classe *BrowserFactory*. Após a definição de *browser*, o objeto chama o método *launch*, passando a *url* como parâmetro.

Por sua vez, o método *getBrowser* da classe *BrowserFactory* vai detectar, durante a execução do programa, qual o sistema operacional em que o programa está sendo executado, através da função do Java *System.getProperty("os.name")*. Assim, o tipo dinâmico que o objeto *browser* irá assumir dependerá do sistema operacional detectado. Uma vez que o objeto *browser* pode assumir um dos três tipos citados anteriormente, o método *launch* chamado na função *openURL* pode ser de qualquer uma das três classes. A princípio, isso não seria um problema, já que as classes *OSXBrowser* e *GenericBrowser* implementam o método *launch* de maneira segura. Entretanto, detectamos que o método da classe *WindowsBrowser* possui a vulnerabilidade de *command injection*.

Command injection é um ataque cujo objetivo é executar comandos arbitrários no


```
1  public class BareBonesBrowserLaunch {
2      public static void openURL(String url) {
3          IBrowser browser = new BrowserFactory().getBrowser();
4          // BEGIN AUTOMATIC CODE
5          if(browser instanceof WindowsBrowser){
6              throw new java.lang.Exception("Codigo Inseguro");
7          }
8          // END AUTOMATIC CODE
9          browser.launch(url);
10     }
11 }
```

Figura 5. Instrumentação resultante da análise feita no programa da Figura 4 da aplicação FlickrFaves.

sistema operacional do host através de uma aplicação vulnerável. Este ataque é possível quando uma aplicação transfere dados inseguros fornecidos pelo usuário para um shell do sistema. Neste ataque, os comandos do sistema operacional fornecidos pelo invasor geralmente são executados com os privilégios do aplicativo vulnerável. Uma vez que o método recebe a *url* do usuário e concatena a mesma com o parâmetro passado para o método *Runtime.getRuntime().exec()*, a *url* pode conter dados inseguros, tal como um comando, e o mesmo será executado assim que o método *exec* executar. Dessa forma, um usuário mal intencionado poderia utilizar essa aplicação no computador de uma pessoa que utiliza o sistema operacional Windows e fazer um ataque de *command injection* em tal sistema.

Uma vez que só é possível saber dinamicamente qual método *launch* irá executar, é necessário realizar uma análise estática acoplada com uma checagem dinâmica, para permitir que o programa execute quando for seguro e impedir que ele execute quando for inseguro. Utilizamos nossa ferramenta em tal programa, rastreamos todos os possíveis caminhos que poderiam existir e detectamos o caminho inseguro existente quando o sistema operacional utilizado era o Windows. Assim, fomos capazes de instrumentar o código para previnir a execução de tal caminho, inserindo uma guarda antes da chamada de *launch*. O código instrumentado resultante de tal análise é mostrado na Figura 5.

5.2. Análise Qualitativa

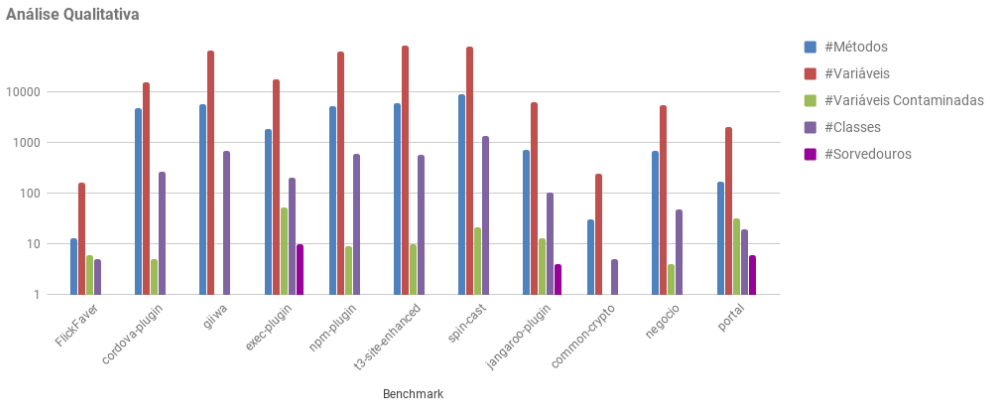

Figura 6. Gráfico resultante da análise qualitativa.

A Figura 6 mostra o resultado da aplicação de InspectorJ em 11 *benchmarks* diferentes que contêm vulnerabilidades já reportadas na literatura. Dentre os 11 *benchmarks*, os 9 primeiros são projetos de código aberto do GitHub e os 2 últimos são sistemas da Companhia de Tecnologia do Estado de Minas Gerais (PRODEMGE). InspectorJ foi configurado para detectar *command injection* e ataque por variação de tempo. A ferramenta corretamente reportou pelo menos uma das vulnerabilidades em 9 dos 11 *benchmarks* onde era esperado um problema, e não disparou qualquer aviso para os casos onde não era esperado uma vulnerabilidade. Podemos perceber que uma pequena parte de um programa já é suficiente para torná-lo inseguro: o *t3-site-enhanced*, por exemplo, possui 80390 variáveis, em que apenas 10 são contaminadas e tornam o programa inseguro. Dessa forma, inserir checagens dinâmicas para permitir que programas executem quando são seguros reduz a frequência com que programas são considerados vulneráveis.

5.3. Análise de Complexidade

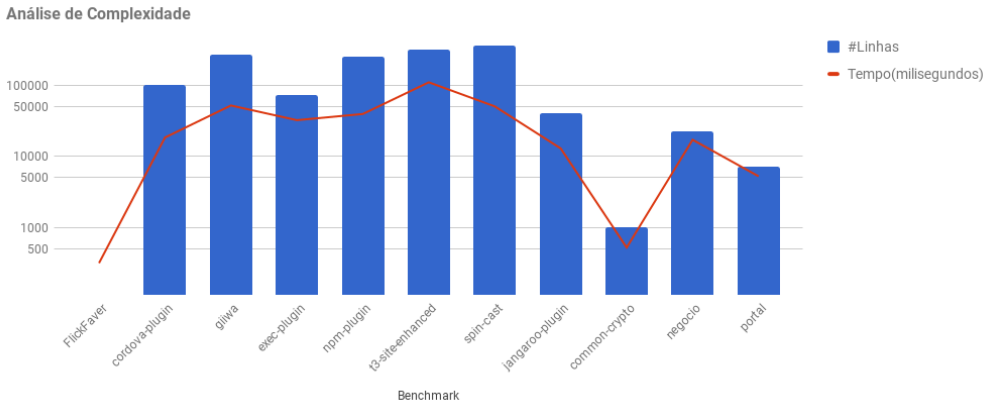

Figura 7. Gráfico resultante da análise de complexidade.

O resultado da análise de complexidade é mostrado na Figura 7. Medimos o tempo da nossa análise em todos os *benchmarks* citados anteriormente e percebemos que InspectorJ roda em tempo linear ao tamanho do programa. Além disso, a checagem dinâmica não aumentou o tempo que os programas demoram para executar, uma vez que a parte do código insegura era pequena em relação ao tamanho do programa inteiro.

6. Trabalhos Relacionados

Existem diversas técnicas de análise de fluxo de informação usadas para proteger programas orientados a objetos. Entretanto, este artigo é único na maneira em que ele combina análises estáticas e dinâmicas com tal propósito. A fim de salientar esse fato, esta seção descreve alguns dos resultados mais notáveis relacionados a técnicas de análise de fluxo de informação. Para tanto, serão discutidos trabalhos que buscaram combinar técnicas de proteção estáticas e dinâmicas. Enfatiza-se que nenhuma das técnicas discutidas a seguir utilizou tal combinação para permitir que mais programas orientados a objetos possam ser certificados como seguros.

Muitos pesquisadores diferentes combinaram análises dinâmicas e estáticas para rastrear o fluxo de informação durante a execução de programas. O propósito dessa

combinação é velocidade; porém, há trabalhos que, como este artigo, também buscaram aumentar a precisão da análise estática. Entretanto, ao contrário do que é feito neste trabalho, previamente a análise estática complementava a informação adquirida pela sua contrapartida dinâmica, e não vice-versa. Por exemplo, Zhang *et al.* [Zhang et al. 2011] propuseram uma ferramenta que instrumenta e executa programas, procurando por potenciais fluxos que podem ser controlados por um adversário. Durante essa espécie de execução monitorada, a ferramenta proposta por Zhang *et al.* também constrói o gráfico de fluxo de controle do programa. Essa estrutura é posteriormente dada a um analisador estático, que executa uma segunda rodada de verificações sobre as partes do programa que não foram atingidas pela análise dinâmica. Vogt *et al.* [Vogt et al. 2007] também usam análise estática para estender os resultados da análise dinâmica. Contudo, eles fazem isso durante a execução do programa. Ou seja, uma vez que a análise dinâmica identifica as regiões do programa que podem ser controladas por dados corrompidos, a ferramenta de Vogt *et al.* executa uma passada linear nessa região, marcando todas as variáveis definidas dentro dela. Caso essas variáveis sejam usadas posteriormente, o ambiente de execução poderá identificá-las como potencialmente contaminadas. Há também trabalhos que mostram como usar a análise dinâmica para eliminar falsos positivos produzidos pelo algoritmo estático. Como um exemplo, Balzarotti *et al.* [Balzarotti et al. 2008] projetaram *Saner*, uma ferramenta que usa análise dinâmica baseada em teste para verificar possíveis vulnerabilidades relatadas pela análise estática.

A análise estática também pode ser usada para melhorar o tempo de execução da técnica dinâmica, conforme mencionado anteriormente. Possivelmente, o trabalho mais influente nesta área pertence a Huang *et al.*; Trata-se de WebSSARI [Huang et al. 2004]. Esta ferramenta usa um sistema de tipos para identificar variáveis de programas que podem ser corrompidas por um usuário malicioso. A partir dessa identificação, WebSSARI instrumenta as funções sensíveis que lêem essas variáveis. Nosso trabalho é diferente do de Huang porque temos objetivos diferentes. Enquanto WebSSARI usa os resultados da análise estática para reduzir o overhead sobre a análise dinâmica, esse tipo de ganho de desempenho não é nosso objetivo. Nossa instrumentação é motivada pela existência de chamadas polimórficas em um programa orientado por objetos, e ocorre somente quando a análise estática falha em analisar o programa totalmente. A instrumentação de Huang *et al.* é o caso oposto: ela é usada como operação fim, sendo a análise estática uma forma de diminuir seu custo de execução. Além disso, Huang *et al.* não discutem como eles lidam com chamadas polimórficas de métodos. Em uma direção totalmente oposta ao trabalho de Huang ou ao nosso trabalho, mas ainda motivados por eficiência, Chebaro *et al.* [Chebaro et al. 2012] usaram análise estática para remover partes trivialmente seguras de um programa, a fim de que o mesmo pudesse ser testado exaustivamente. Em seguida, eles usam análise dinâmica baseada em testes (sobre o programa reduzido) para procurar por vulnerabilidades em suas partes inseguras.

7. Conclusão

Este artigo apresentou uma técnica de análise estática acoplada a checagem dinâmica para ampliar a precisão de técnicas de análise de fluxo de informação. Instrumentamos o código em pontos de chamada que podem invocar métodos inseguros. Tal checagem dinâmica nos permite detectar e descartar caminhos inseguros durante a execução do programa e preservar fluxos seguros de informação. Essa combinação garante a segurança

de programas mesmo diante de propriedades difíceis de serem analisadas estaticamente, como por exemplo polimorfismo de subtipagem.

Acredita-se que nosso trabalho seja único na maneira que combina análise estática com checagem dinâmica com o propósito de aumentar a precisão da análise para permitir que mais programas orientados a objetos possam ser certificados como seguros. Como trabalho futuro, pretendemos integrar nossa ferramenta à IDE Eclipse, que hoje é amplamente usada pelos desenvolvedores Java.

Referências

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

Bruno Rodrigues, D. A. e. F. M. Q. a. P. (2016). Sparse representation of implicit flows with applications to side-channel detection. In *Compiler Construction*, pages 1–20. ACM.

Chebaro, O., Kosmatov, N., Giorgetti, A., and Julliand, J. (2012). Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC*, pages 1284–1291. ACM.

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

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (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.

Huang, Y., Yu, F., Hang, C., Tsai, C., Lee, D., and Kuo, S. (2004). Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–52. ACM.

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

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

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.

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

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.

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

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

