PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ALEXANDRE BRAGA DAMASCENO

TAINTJSEC: UM MÉTODO DE ANÁLISE ESTÁTICA DE MARCAÇÃO EM CÓDIGO *JAVASCRIPT* PARA DETECÇÃO DE VAZAMENTO DE DADOS SENSÍVEIS

Manaus - Amazonas
2017

ALEXANDRE BRAGA DAMASCENO

TAINTJSEC: UM MÉTODO DE ANÁLISE ESTÁTICA DE MARCAÇÃO EM CÓDIGO *JAVASCRIPT* PARA DETECÇÃO DE VAZAMENTO DE DADOS SENSÍVEIS

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas, como requisito parcial para a obtenção do grau de Mestre em Informática.

Orientador: Eduardo J. P. Souto

Manaus - Amazonas

2017

Damasceno, Alexandre Braga

D155t *TaintJSec*: Um Método de Análise Estática de Marcação em Código *JavaScript* para Detecção de Vazamento de Dados Sensíveis / Alexandre Braga Damasceno. — Manaus - Amazonas, 2017
xxi, 129 f. : il. color ; 31cm

Dissertação (mestrado) — Universidade Federal do Amazonas
Orientador: Eduardo J. P. Souto

1. Vazamento de Informação. 2. Dados Sensíveis. 3. JavaScript. 4. TaintJSec. 5. Análise Estática. I. Título.

CDU 004

FOLHA DE APROVAÇÃO

"TAINTJSEC: UM MÉTODO DE ANÁLISE ESTÁTICA DE MARCAÇÃO EM CÓDIGO JAVASCRIPT PARA DETECÇÃO DE VAZAMENTO DE DADOS SENSÍVEIS"

ALEXANDRE BRAGA DAMASCENO

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Eduardo James Pereira Souto - PRESIDENTE

Prof. Eduardo Luzeiro Feitosa - MEMBRO INTERNO

Prof. André Ricardo Abed Grégio - MEMBRO EXTERNO

Prof. Gilbert Breves Martins - MEMBRO EXTERNO

Manaus, 22 de Dezembro de 2017

Dedico esta dissertação aos meus pais,
Raimundo Nonato Damasceno e Sandra Maria Braga Damasceno,
que nunca mediram esforços para me dar uma boa educação
e me ensinaram, desde cedo, sobre o quão importante são os estudos em minha vida.

Agradecimentos

Primeiramente agradeço e louvo ao Deus de Abraão, Isaac e Jacó, que também é o meu Deus, por ter me dado força e coragem para enfrentar as tribulações que surgiram em minha vida durante o curso, mas não foram suficientes para impedir esta vitória.

Quero agradecer à minha família, meu porto seguro, que sempre me apoiou nos estudos e nunca me deixou esmorecer perante os desafios.

À minha amada Andressa D. Craveiro, por motivar-me e por ser tão compreensível nos momentos em que eu estive ausente por dedicação ao curso.

Ao meu orientador Prof. Dr. Eduardo J. P. Souto e ao coorientador Me. Thiago de Souza Rocha pelo conhecimento, companheirismo, auxílio e paciência que foram de suma importância durante o desenvolvimento deste trabalho.

Ao Prof. Dr. Eduardo L. Feitosa, presidente do Programa de Pós-Graduação em Informática (PPGI), por mostrar-se sempre pronto a ajudar o próximo.

Aos professores, Dr. Raimundo S. Barreto e Dr. Arilo C. D. Neto, pela confiança em mim depositada ao redigirem as cartas de recomendação para o curso.

Agradeço também ao Prof. Dr. David B. F. de Oliveira que durante minha tutoria mostrou ser uma pessoa de caráter ímpar, preocupado com o ser humano antes de qualquer outra coisa.

Agradeço aos colegas do laboratório ETSS (*Emerging Technologies and Systems Security*) pelo clima agradável de harmonia e descontração, onde sempre fui bem recebido e quase sempre tinha café quente.

Por fim, agradeço a todos aqueles que a seu modo contribuíram e me apoiaram para a conclusão deste trabalho.

Muito obrigado!

"Se permanecerdes em mim,
e as minhas palavras permanecerem em vós,
pedireis o que desejardes, e vos será concedido."
(Jesus Cristo)

Resumo

Javascript é uma das linguagens de programação mais utilizadas no mundo e continua expandindo-se gradativamente. Tal expansão deve-se à grande flexibilidade e dinamicidade que a linguagem possui, o que facilita bastante a criação de aplicações. Porém, essa mesma característica que a torna uma linguagem de sucesso é também o que torna difícil a análise estática do fluxo de execução, processo esse que visa identificar a presença de códigos maliciosos nas aplicações. Este trabalho apresenta o *TaintJSec*, uma nova abordagem que utiliza análise estática de marcação de código *JavaScript* para identificar e prevenir o vazamento de informação sensível. O diferencial do *TaintJSec* em relação aos outros trabalhos que utilizam análise estática de marcação é que ele consegue analisar o fluxo de códigos implícitos, acompanha a propagação do *taint tag* na execução da função *eval* e identifica o vazamento de informação em códigos ofuscados. Para validar a eficácia da abordagem, foram realizados testes de propagação do *taint tag*, divididos em 13 grupos de testes distintos. Em seguida, foram realizados testes para avaliar a propagação na execução da função *eval*. Por fim, a abordagem foi testada em um código malicioso, ofuscado por cinco ferramentas diferentes, específicas para tal finalidade. Os resultados obtidos demonstraram que a abordagem é eficaz na detecção do vazamento de informação e mais eficiente que outros métodos do estado da arte.

Palavras-chave: Vazamento de Informação, Dados Sensíveis, JavaScript.

Abstract

Javascript is one of the most used programming languages in the world and continues to expand gradually. Such success is due to the great flexibility and dynamicity that the language has, which greatly facilitates the creation of applications. However, this same characteristic that makes it a successful language is also what makes it difficult to analyze static execution flow, which aims to identify the presence of malicious code in applications. This work presents TaintJSec, a new approach that uses static code marking analysis to identify and prevent leakage of sensitive information in web applications. Unlike other works based on static analysis, TaintJSec is able to check the explicit and implicit code flow, accompanies the propagation of the taint tag in the execution of the eval function, and is able to identify information leakage in obfuscated codes. To validate the effectiveness of the approach, taint tag propagation tests were performed in a range of tests divided into 13 different test groups. Then, tests were performed to evaluate the propagation of the eval function. Finally, the approach was tested in a malicious code, obscured by five different tools, specific for that purpose. The results demonstrated that the approach is effective in detecting information leakage and more efficient than other methods of the state of the art.

Keywords: Data Leakage, Sensitive Data, JavaScript.

Sumário

Lista de Algoritmos

Lista de Códigos

Lista de Figuras

Lista de Quadros

Lista de Tabelas

Lista de Siglas

AJAX : *Asynchronous JAvascript and Xml*

ANTLR : *ANother Tool for Language Recognition*

ASP : *Active Server Pages*

AST : *Abstract Syntax Tree*

BYOD : *Bring Your Own Device*

CPU : *Central Processing Unit*

CSS : *Cascading Style Sheets*

DEX : *Dalvik EXecutable*

DOM : *Document Object Model*

DTD : *Document Type Definition*

HTML : *HyperText Markup Language*

HTTP : *Hypertext Transfer Protocol*

HTTPS : *Hypertext Transfer Protocol Secure*

IMEI : *International Mobile Equipment Identity*

IP : *Internet Protocol*

JNI : *Java Native Interface*

JSON : *JavaScript Object Notation*

MIME : *Multipurpose Internet Mail Extensions*

PHP : *PHP Hypertext Preprocessor*

PDF : *Portable Document Format*

POSIX : *Portable Operating System Interface for uniX*

RFC : *Request for Comments*

SGML : *Standard Generalized Markup Language*

SMS : *Short Message Service*

WWW : *World Wide Web*

Capítulo 1

Introdução

A linguagem *JavaScript* é atualmente uma das linguagens de programação mais populares e mais utilizadas no mundo para a criação de aplicações *web* [1, 2, 3, 4, 5]. Esse sucesso deve-se em parte à sua facilidade de manipulação, pois a linguagem possui uma sintaxe muito flexível, sendo possível atingir um mesmo objetivo utilizando diferentes formas de codificação. Essa flexibilidade é oriunda da tipagem fraca, onde uma variável não é obrigada a ser do mesmo tipo durante toda a execução do código, podendo iniciar como *integer*, por exemplo, e terminar como *string*, *function*, *array* ou outro tipo de objeto. A dinamicidade da linguagem também deve-se à possibilidade de inclusão de trechos de código em tempo de execução [6], permitindo que aplicações desenvolvidas em *JavaScript* executem rotinas de código e percorram fluxos de execução não explicitamente conhecidos.

Essa versatilidade, embora seja útil para o desenvolvimento de aplicações, acaba abrindo um leque de oportunidades para a execução de códigos maliciosos, que podem vir a comprometer os pilares da segurança da informação[1], como a confidencialidade e a integridade dos dados daqueles que utilizam a aplicação.

Para reduzir o risco de execução de código malicioso, os navegadores modernos fazem uso de um recurso que dificulta o carregamento de código executável a partir de outras fontes, denonimado de Política da Mesma Origem[2]. Por meio dessa política, uma aplicação pode ficar impossibilitada de carregar trechos de códigos de origens que não sejam do seu próprio domínio e até isolar a execução de códigos em diferentes *frames*. Porém, esse recurso somente assegura que códigos de terceiros não possam ser embutidos no código original das aplicações, mas não bloqueia a execução de códigos maliciosos provenientes da própria aplicação.

[1] Os pilares da segurança da informação são formados pela tríade CID (confidencialidade, integridade e disponibilidade)

[2] https://www.w3.org/Security/wiki/Same_Origin_Policy

1.1 Motivação e Descrição do Problema

Com a linguagem *JavaScript* evoluindo continuamente e ganhando novos campos de atuação, como o desenvolvimento de extensões para navegadores [7, 8], para aplicativos *desktop* [9, 10], para servidores [11] e para dispositivos móveis [12], o interesse pelas linguagens dinâmicas ganhou força, fazendo com que a comunidade de pesquisadores se voltasse para a criação de soluções que pudessem resolver problemas de segurança em aberto, tais como o vazamento de informação.

O vazamento de informação é a distribuição acidental ou não intencional de dados sigilosos para um destino não autorizado [13]. Esse problema ocorre com maior incidência no universo dos dispositivos móveis [14], onde é explorado por aplicações maliciosas que, uma vez instaladas nos dispositivos, capturam dados sigilosos para repassá-los a terceiros.

Diversos trabalhos foram realizados na tentativa de coibir o problema do vazamento de informação sensível. Pinto *et al.* [15] e Kuzuno *et al.* [16] verificam o conteúdo dos pacotes HTTP para checar a existência de informação sigilosa. Wang *et al.*[17] e Hsiao *et al.* [18] utilizam listas para permitir ou bloquear o acesso às fontes de informação sensível. Entretanto, a maioria dos trabalhos realiza análise no fluxo da informação, para observar os passos de funcionamento de uma aplicação e descobrir se uma informação sensível está sendo vazada [4, 19, 20, 21, 22, 23, 24, 25, 26, 27].

Contudo, diferente das linguagens estáticas, as quais são bem definidas e fortemente tipadas, a linguagem *JavaScript* torna-se um grande desafio para o desenvolvimento de abordagens que procuram analisar do fluxo da informação. Nesse contexto, os motivos que lhe renderam o título de linguagem de programação mais utilizada do mundo são os mesmos que fazem com que o fluxo da informação seja difícil de ser analisado.

Os trabalhos que procuram detectar o vazamento mediante análise de pacotes HTTP (*Hypertext Transfer Protocol*) não conseguem identificar o vazamento se os dados estiverem ofuscados. Os trabalhos que utilizam listas para permitir ou bloquear o acesso às fontes de informação sensível exigem que o usuário tenha conhecimento técnico o suficiente para decidir o que deve ser permitido e o que deve ser bloqueado. Enquanto os trabalhos que utilizam análise de fluxo esbarram na complexidade computacional do código *JavaScript* devido a sua versatilidade. Por esta razão, muitos desses trabalhos não têm suporte à função *eval*[3] ou qualquer outro método de criação dinâmica de código, como ilustra o Código 1.1.

```
1 <script>
2   var a = "g=n=>n";
3   var b = "g(n-1)";
4   var c = "n:1";
5   eval(a+'?'+b+'*'+c);
```

[3]É uma função nativa da linguagem *JavaScript* que permite a execução dinâmica de um *script* existente em uma *string*.

```
6   var saida = g(5);
7 </script>
```

Código 1.1. Exemplo de código *JavaScript* utilizando a função *eval*

O Código 1.1 apresenta um exemplo de código *JavaScript* que utiliza a função *eval* para criar uma função fatorial dinamicamente. Na linha 5 é possível observar uma expressão de concatenação de variáveis (a+'?'+b+'*'+c) para formar o argumento a ser enviado para a função. Essa concatenação resulta no valor literal 'g=n=>n?g(n-1)*n:1' que é o argumento enviado à função *eval*, a qual cria a função **g** em tempo de execução. Ao término da execução da função, a linha seguinte (linha 6) é processada, a qual executa uma operação de atribuição do resultado da função **g** (criada pela função *eval*) para a variável **saida**. O valor atribuído à variável **saida** é 120 que corresponde ao fatorial de 5.

A dificuldade para analisar a execução da função *eval* é o fator determinante para que muitos trabalhos de análise estática de código *JavaScript* não deem suporte à análise de criações dinâmicas de código.

É por esses motivos que este trabalho apresenta uma abordagem que utiliza a análise estática para identificar e prevenir o vazamento de informação sensível em aplicações *web*. O *TaintJSec* consegue verificar o fluxo de código explícito e implícito, acompanha a propagação do *taint tag* na execução da função *eval* e é capaz de identificar o vazamento de informação sensível em códigos ofuscados. A abordagem apresentada nesse trabalho não compromete o desempenho das aplicações por realizar unicamente a análise estática, sendo essa a menos onerosa de todas as técnicas de análise de fluxo existente no estado da arte.

1.2 Objetivos

1.2.1 Objetivo Geral

Aprimorar o processo de detecção de vazamento de dados em código *JavaScript* baseado na análise estática de fluxo de dados.

1.2.2 Objetivos Específicos

Para alcançar o objetivo geral desta pesquisa, os seguintes objetivos específicos devem ser alcançados:

1. Desenvolver um método para extrair e agrupar o código *JavaScript* presente na aplicação. É necessário extrair o código *JavaScript* para separá-lo de códigos de outras linguagem, tais como CSS e HTML. O código extraído e agrupado serve

de consulta e criação de representações intermediárias que são utilizadas durante a análise estática.

2. Desenvolver um método para criar uma árvore sintática abstrata de um código *JavaScript*. Tal método é utilizado para gerar uma representação intermédiária do código *JavaScript* da aplicação. Essa representação intermediária, no formato de árvore, representará o fluxo de execução do código *JavaScript*.

3. Desenvolver um analisador estático para percorrer a árvore sintática abstrata e realizar as operações contidas em cada nó da árvore.

4. Implantar um mecanismo de marcação de *tags* na estrutura dos registradores de objetos do código *JavaScript* (*taint marking*). Tal mecanismo é embutido no analisador estático, de modo que a marcação seja mais uma das operações realizadas pelo analisador.

1.3 Contribuições

Este trabalho fornece as seguintes contribuições:

1. Um analisador estático de código *JavaScript* capaz de identificar o vazamento de informação sensível em códigos implícitos e ofuscados.

2. Uma técnica denominada *taint position* para aumentar o controle da propagação do *taint tag* na análise da função *eval*.

1.4 Estrutura do Documento

O restante desta dissertação está organizada do seguinte modo:

- O Capítulo 2 expõe os conceitos básicos e fornece a fundamentação teórica necessária para que o leitor tenha um melhor entendimento sobre o problema tratado neste trabalho.

- O Capítulo 3 aborda os trabalhos relacionados e as principais abordagens para resolver o problema do vazamento de dados sensíveis. As abordagens são organizadas em três grupos e os trabalhos são apresentados de acordo com o grupo ao qual pertencem. O capítulo também apresenta uma análise comparativa das características e as limitações dos trabalhos relacionados.

- O Capítulo 4 apresenta o *TaintJSec* e descreve as três fases da abordagem. Também apresenta uma nova técnica, denominada *taint position*, para estender o processo de propagação do *taint tag* para o escopo da função nativa *eval*.
- O Capítulo 5 apresenta os testes e os resultados realizados para validar a eficácia do *TaintJSec* na detecção do vazamento de informação sensível. Também são descritas as três baterias de testes: testes de propagação, testes com a função *eval* e testes em códigos ofuscados.
- O Capítulo 6 conclui este documento apresentando as contribuições, as limitações da abordagem e os trabalhos futuros.

Capítulo 2

JavaScript e Desenvolvimento *Web*

Neste capítulo são abordados alguns conceitos básicos, com o propósito de prover ao leitor informações necessárias para que tenha um melhor entendimento sobre o contexto em que este trabalho está inserido. O capítulo inicia com uma breve descrição da linguagem *JavaScript* e conceitua os elementos da linguagem relevantes para os objetivos deste trabalho, como as bibliotecas *JavaScript* e a plataforma *Node.js*. Em seguida, o capítulo apresenta o problema do vazamento de informação e discorre sobre as técnicas de análise de marcação utilizadas para mitigar esse problema.

2.1 Linguagem de Programação *JavaScript*

JavaScript é uma linguagem de programação comumente utilizada no desenvolvimento *web*. Foi desenvolvida em 1995 pela *Netscape* como um meio para adicionar elementos dinâmicos e interativos para os *sites* [28]. Enquanto a *JavaScript* é influenciada pela linguagem Java, sua sintaxe é mais semelhante à linguagem C e baseada na *ECMAScript* [29], uma linguagem de *script* desenvolvida pela *Sun Microsystems*.

JavaScript foi originalmente criada para executar no lado do cliente (*client-side*), o que significa que o código-fonte foi projetado para ser processado pelo navegador *web* do cliente e não pelo servidor *web* [30, 31]. Por esse motivo, as funções de um código *JavaScript* podem ser executadas após uma página *web* ter sido carregada sem se comunicar com o servidor. Por exemplo, uma função pode verificar um formulário eletrônico antes deste ser enviado para certificar-se de que todos os campos obrigatórios foram preenchidos. Assim, o código *JavaScript* pode produzir uma mensagem de erro antes que qualquer informação seja realmente transmitida para o servidor [32].

Somente a partir do ano 2009, com a criação do *Node.js* [11] a linguagem *JavaScript* passou a ser cada vez mais utilizada no lado do servidor (*server-side*). O *Node.js* é um interpretador de código construído sobre o motor *JavaScript* V8 do navegador *Google*

Chrome para facilitar a construção de aplicações que executam no *server-side* [33].

Assim como as linguagens de *script* que são executadas no *server-side*, tais como PHP (*PHP Hypertext Preprocessor*) e ASP (*Active Server Pages*), o código *JavaScript* pode ser inserido em qualquer lugar dentro do HTML de uma página *web*. Essa inserção pode ser realizada de duas maneiras: *inline* ou por meio de um arquivo externo.

O código *inline* é inserido diretamente no código HTML e o conteúdo do código *JavaScript* fica inteiramente visível entre as *tags* <*script*> e </*script*>, como exemplifica o Código 2.1, cujo código *inline* está entre as linhas 6 e 15.

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8" />
5          <title>Exemplo</title>
6          <script>
7              function fatorial(n){
8                if(n<0)
9                  return -1;
10               else if(n==0)
11                 return 1;
12               else return n * fatorial(n-1);
13             }
14             var resultado = fatorial(4)
15         </script>
16     </head>
17 <body>
18 </body>
19 </html>
```

Código 2.1. Exemplo de *JavaScript inline*.

O código *JavaScript* inserido por um arquivo externo utiliza a *tag* <*script*> com o atributo *src* (source), cujo valor contém o caminho do código externo que geralmente é referenciado por um arquivo com a extensão .js, mas também pode apresentar outras extensões ou até mesmo nenhuma extensão, pois o documento externo que contém o código *JavaScript* precisa apenas informar ao navegador que o seu conteúdo é um código *JavaScript*, para ser interpretado como tal. Em PHP, por exemplo, essa informação é indicada pela instrução <?php header('Content-type:text/javascript'); ?>. Assim, o arquivo externo pode aparecer com a extensão diferente da tradicional .js e continuar sendo reconhecido pelo navegador como um arquivo *JavaScript*. Essa troca de extensão do arquivo externo é comumente utilizada quando deseja-se que o conteúdo seja gerado em tempo de execução, por meio de uma linguagem de programação dinâmica. A linha 6 do Código 2.2 apresenta um exemplo de inserção de código *JavaScript* externo.

```
1  <!DOCTYPE html>
2  <html>
3   <head>
4    <meta charset="utf-8" />
5    <title>Exemplo</title>
6    <script src="lib.js">
7    </script>
8   </head>
9  <body>
10 </body>
11 </html>
```

Código 2.2. Exemplo de *JavaScript* externo.

Indepentendemente se um código é *inline* ou externo, ele sempre é um código explícito, pois está declarado em meio à estrutura HTML. Os códigos explícitos são identificados simplesmente olhando o código-fonte da aplicação. Por exemplo, o Código 2.1 contém código *JavaScript* entre as linhas 6 e 15, enquanto o Código 2.2 contém código *JavaScript* entre as linhas 6 e 7.

Por outro lado, os códigos implícitos são difíceis de identificar visualmente, pois ficam ocultos na lógica de execução dos códigos explícitos. Geralmente, esse tipo de código é armazenado em variáveis, assumindo inicialmente valores literais (*strings*). O código implícito depende da execução de um código explícito para que seja transformado em um novo elemento *script* anexado ao DOM (*Document Object Model*). Dentre as funções mais utilizadas para anexar novos trechos de código ao DOM destacam-se a *document.write*, *appendChild* e a função *eval*.

O Código 2.3 apresenta um código HTML que contém código *JavaScript* com códigos implícitos em sua lógica de execução.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <script>
5       var implicito_1 = "<script>alert('teste1');</"+"script>";
6       document.write(implicito_1);
7
8       var implicito_2 = "<script src='lib.js'></"+"script>";
9       document.write(implicito_2);
10
11      var implicito_3 = document.createElement("script");
12      implicito_3.src = "lib.js";
13      document.body.appendChild(implicito_3);
14     </script>
15   </head>
16 </html>
```

Código 2.3. Trecho de código explícito contendo códigos implícitos.

É possível observar na linha 5 que a variável *implicito_1* é iniciada com um valor literal que não é um código *JavaScript* reconhecido pelo DOM, mas passa a ser considerado um novo trecho de código após a execução da função *document.write* da linha 6. O mesmo ocorre com a variável *implicito_2*, porém esta difere-se por não trazer consigo códigos *inline*, mas faz referência a um arquivo externo. Já a variável *implicito_3*, na linha 11, não é declarada com um valor literal, mas com um novo elemento do HTML que, inicialmente, não faz parte do DOM, vindo a fazer parte após a execução da função *appendChild*, na linha 13.

A Figura 2.1 apresenta o resultado da execução do Código 2.3, onde outras *tags* <*script*> foram dinamicamente criadas.

Figura 2.1. Novos códigos criados dinamicamente.

A Figura 2.1 apresenta em destaque três novos trechos de códigos *JavaScript* criados dinamicamente, em tempo de execução. Tais códigos estavam implícitos na lógica de execução dos códigos explícitos. Ao observar o Código 2.3 não é possível ver os novos trechos de código tal como apresentados na Figura 2.1, pois eles somente passam a existir após a execução dos códigos explícitos. Porém, ainda é possível ver que as variáveis *implicito_1* e *implicito_2* recebem blocos de código *JavaScript* em forma de *string*.

Essa visualização só é possível porque os valores atribuídos às variáveis *implicito_1*

e *implicito_2* estão em texto plano[1], mas caso estivessem ofuscados não seria possível identificar, apenas olhando para o código, que as variáveis estavam recebendo blocos de código implícito, como exemplifica o Código 2.4.

```
<!DOCTYPE html>
<html>
  <head>
    <script>
     var implicito_1 = "\x3C\x73\x63\x72\x69\x70\x74\x3E\x61\x6C\x65\x72\x74\x28
         \x27\x74\x65\x73\x74\x65\x31\x27\x29\x3B\x3C\x2F\x73\x63\x72\x69\x70\
         x74\x3E";
     document.write(implicito_1);

     var implicito_2 = "\x3C\x73\x63\x72\x69\x70\x74\x20\x73\x72\x63\x3D\x27\x6C
         \x69\x62\x2E\x6A\x73\x27\x3E\x3C\x2F\x73\x63\x72\x69\x70\x74\x3E";
     document.write(implicito_2);

     var implicito_3 = document.createElement("script");
     implicito_3.src = "lib.js";
     document.body.appendChild(implicito_3);
    </script>
  </head>
</html>
```

Código 2.4. Trecho de código explícito contendo códigos implícitos ofuscados.

O Código 2.4 é uma cópia Código 2.3, porém os valores atribuídos às variáveis *implicito_1* e *implicito_2*, que antes eram textos planos, foram substituídos por sua representação hexadecimal. Os valores que as variáveis recebem são os mesmos em ambos os códigos, a única diferença é que no Código 2.4 os valores estão ofuscados.

Código ofuscado é o código que passou por um processo de ofuscação, ou seja, um embaralhamento com o objetivo de transformar o código legível em um equivalente que é mais difícil de entender [34, 35].

Assim, é possível notar que a linguagem *JavaScript* é bastante flexível para permitir que um mesmo resultado possa ser obtido de diferentes maneiras. A possibilidade de criar códigos dinamicamente permite maior interatividade com a página *web*. Enquanto que criar códigos ofuscados é comumente usado por desenvolvedores de *software* para proteger a propriedade intelectual de seu código.

Contudo, a versatilidade da linguagem *JavaScript* também é explorada para a criação de *malwares*, que utilizam os recursos da linguagem para evadir sistemas de detecção baseados em assinaturas e praticar o vazamento de informação.

[1] Um texto simples com formato humanamente legível, sem qualquer embaralhamento proposital para dificultar a leitura.

2.2 Vazamento de Informação

Vazamento de informação é definido como uma distribuição acidental ou não intencional de dados sigilosos para um destino não autorizado [13].

Tal vazamento pode facilmente acarretar prejuízos ao dono da informação. Dependendo do tipo de informação vazada, o dano causado pode ser financeiro ou moral. Uma empresa, por exemplo, que possui muitas informações sigilosas, como dados financeiros, números de cartão de crédito e planos estratégicos, pode sofrer prejuízos caso esses dados sejam enviados à uma entidade não autorizada. Como foi o caso da empresa Sony no ano de 2014, onde *crackers* roubaram uma grande quantidade de informações sigilosas e divulgaram na Internet cinco filmes inéditos, causando um prejuízo financeiro estimado em US$ 300 milhões [36].

Os danos causados por vazamento de informação privilegiada podem ser classificados em duas categorias: diretos e indiretos.

Os danos diretos referem-se a danos tangíveis, ou seja, são danos mensuráveis, fáceis de medir ou estimar quantitativamente. Os danos indiretos, por outro lado, são muito mais difíceis de quantificar e possuem um impacto muito mais amplo em termos de custo, lugar e tempo [37]. Geralmente o dano indireto é identificado quando a empresa tem a sua imagem prejudicada junto aos seus clientes, fazendo com que novos negócios deixem de ser firmados devido à insegurança criada após um evento de vazamento de dados sigilosos. Como foi o caso da empresa Uber, que teve dados de 57 milhões de contas de usuários vazados no ano de 2016 quando *crackers* invadiram seus servidores. A empresa manteve segredo durante um ano, vindo a revelar sobre o vazamento no final do ano de 2017, quando havia acabado de fazer um acordo com a procuradoria-geral de Nova York sobre um processo envolvendo segurança de dados [38].

Com as pessoas, o dano indireto é identificado quando o vazamento de informação prejudica a moral ou a honra, fazendo com que a pessoa sinta-se desmoralizada, envergonhada ou prejudicada junto à sociedade.

O risco de ocorrer o vazamento de informação nas empresas está presente desde a perda de um malote de documentos impressos até uma invasão de *crackers* nos servidores. Porém, com o aumento da prática de BYOD (*Bring Your Own Device*) o risco de vazamento encontra-se também nos dispositivos das pessoas.

No BYOD, um funcionário pode levar seu *laptop*, *tablet* ou *smartphone* para a empresa e utilizá-lo como uma ferramenta de trabalho, acessando a rede coorporativa e tendo acesso a dados restritos [39]. Um funcionário com acesso a informações da empresa pode copiá-las para seu dispositivo ou para um repositório na nuvem, como o *Dropbox*[2] por exemplo. Esse dispositivo está sujeito a infecções de vírus, *spywares*, extravio e até

[2] http://www.dropbox.com

mesmo a instalação de aplicações maliciosas, as quais não são detectadas por sistemas antivírus.

Na tentativa de mitigar o problema do vazamento de informação em dispositivos, muitos trabalhos foram realizados. Em sua maioria, encontram-se os trabalhos que utilizam análise do fluxo de execução de aplicações instaladas nos dispositivos, a fim de detectar se uma aplicação está causando vazamento de dados sigilosos.

A seção 2.3 aborda sobre a técnica de análise de marcação em código, a qual é uma técnica que observa como os dados sensíveis transitam na aplicação.

2.3 Análise de Marcação (*Taint Analysis*)

A análise do fluxo da informação é utilizada por métodos que fazem a verificação dos passos de funcionamento de uma aplicação. A técnica consiste em marcar os dados que são identificados como dados sensíveis e observar por onde esses dados marcados (*tainted data*) são propagados. Permitindo assim identificar o momento que um *tainted data* propaga-se de um objeto para outro (*taint propagation*).

Essa verificação pode ser realizada dinamicamente em tempo de execução (análise dinâmica) ou pode ocorrer em modo estático durante o processo de compilação do código (análise estática).

2.3.1 Análise de Marcação Dinâmica

A análise dinâmica de código é uma técnica que faz suas verificações em tempo de execução [20, 40, 41], com o programa em funcionamento. Essa análise é realizada em um ambiente controlado e permite que o usuário interaja com a aplicação com segurança, pois parte da premissa de que um código malicioso pode ser ativado especificamente após certo evento ter ocorrido no sistema [42].

A análise funciona por meio de modificações realizadas no código original do programa, onde métodos utilizados para rastrear a informação são inseridos em pontos estratégicos do código. Assim, todos os dados que estiverem trafegando durante a execução do programa são analisados.

Ao acompanhar cada passo de um dado no fluxo da aplicação é possível agir com maior justiça, reduzindo os números de falsos positivos. Por outro lado, a sobrecarga imposta (*overhead*) para a execução da análise dinâmica é superior ao da análise estática [43].

2.3.2 Análise de Marcação Estática

A análise estática de código é um método utilizado para verificar o comportamento de um programa sem que seja necessário executá-lo [44]. A análise consiste em percorrer o código, tratando-o como um grafo dirigido, o qual representa o fluxo de execução do programa. Nesse grafo, a análise verifica a operação contida em cada vértice a fim de decidir qual aresta será utilizada para continuar o caminho pelo grafo.

Esse grafo é obtido por um processo de transformação do código-fonte em numa estrutura de dados, geralmente designada por uma árvore sintática abstrata (AST - *Abstract Syntax Tree*), usando um conjunto de regras predeterminado [45].

A árvore sintática abstrata ou simplesmente AST é uma representação sintática simplificada do código-fonte e, na maioria das vezes, é expressa em uma estrutura de dados da linguagem de programação utilizada para a implementação. É uma estrutura em formato de árvore, onde cada nó dessa árvore denota uma instrução que ocorre no código [46]. A sintaxe é "abstrata" (simplificada) por não representar todos os detalhes que aparecem na sintaxe real do código *JavaScript*. Por exemplo, o agrupamento de parênteses está implícito na estrutura da árvore e uma construção sintática como uma expressão *IF* pode ser apresentada como sendo um único nó com três ramificações. Isso distingue a AST de uma árvore de sintaxe concreta, tradicionalmente nomeada como árvore de análise ou árvore de *parser*, muitas vezes construída por um analisador durante o processo de compilação de códigos-fonte. Logo, a AST não tem necessidade de ter tanta informação quanto a árvore de *parser*.

A Figura 2.2 exemplifica um simples código em *JavaScript* e sua respectiva representação em formato de árvore sintática abstrata.

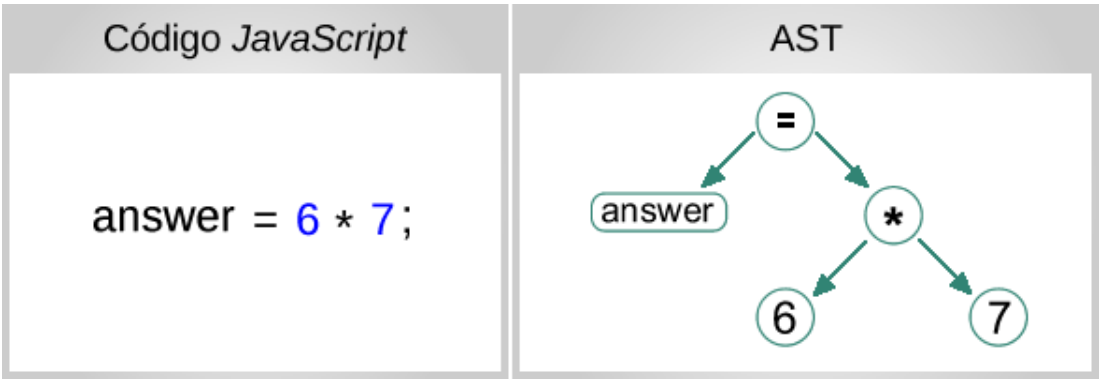


Figura 2.2. Exemplo de um código *JavaScript* com sua respectiva AST.

A AST é usada intensivamente por compiladores, durante a análise semântica de um código-fonte, onde o compilador verifica o uso correto dos elementos do programa e da linguagem, e depois de verificar a corretude, a AST serve como base para a criação de uma representação intermediária durante a compilação do código.

Além dos compiladores, a AST também é utilizada em outros campos de análise de código como, por exemplo, os trabalhos [47, 48, 49] que utilizam a AST como uma representação intermediária para verificar a existência de plágio entre dois ou mais códigos-fontes. O trabalho de Neamtiu *et al.* [50] usa a AST para verificar a evolução de um determinado código e apresenta as diferenças entre suas versões. Já os trabalhos de Blanc *et al.* [51] e Curtsinger *et al.* [52] utilizam a AST para detectar códigos maliciosos escondidos em trechos ofuscados de *JavaScript*.

Atualmente existem ferramentas voltadas para a realização do processo de transformação de código-fonte em AST como, por exemplo, o Esprima [53]. O Esprima é um módulo do *Node.js* utilizado para gerar estruturas de dados a partir de códigos *JavaScript*.

A Figura 2.3 apresenta um exemplo de um código *JavaScript* e a sua respectiva AST gerada pelo módulo Esprima.

Figura 2.3. Exemplo de AST gerada pelo módulo Esprima a partir de um código *JavaScript*.

A Figura 2.3 apresenta o código *JavaScript* **var answer = 6 * 7;** que, ao ser enviado ao módulo Esprima, resultou na sua representação em árvore sintática abstrata. A árvore é fornecida no formato JSON (*JavaScript Object Notation*) e cada um de seus

vértices possuem o atributo *type* para informar o tipo de operação executada em suas subárvores.

Por meio das informações contidas nos vértices da AST, a análise estática decide quais operações realizar e qual a próxima aresta que será utilizada para continuar a análise.

Quando uma operação de fornecimento de dados sensíveis é detectada, a análise marca o objeto (*taint marking*) que passa a conter o dado. Esse objeto, agora marcado pela análise, é continuamente observado para que os vértices, que contenham operações que interajam com esse objeto, sejam tratados como potenciais pontos de propagação de informação sensível (*taint source*). Todos os caminhos do grafo que terminam em uma operação de saída (*taint sink*) são considerados caminhos de vazamento de informação em potencial.

Em comparação com a análise dinâmica, a análise estática possui menor consumo de memória e, se utilizada em dispositivos móveis, consome menos bateria, pois não precisa acompanhar a sequência de fluxo de dados a cada interação durante todo o tempo em que o programa estiver em funcionamento [25]. Além disso, a análise estática permite percorrer todos os caminhos possíveis do grafo de execução de uma aplicação logo na primeira análise [41]. Diferente da análise dinâmica que necessita de um evento específico para analisar cada caminho do fluxo de execução. Ainda mais, a análise estática não necessita de intervenções no código-fonte original dos sistemas para que possa ser aplicada. Já a análise dinâmica necessita de alterações no código-fonte dos sistemas hospedeiros.

2.3.3 Termos da Análise de Marcação

Aqui são apresentados alguns termos da análise de marcação utilizados neste trabalho.

Taint Tag

É um atributo de valor booleano que indica se um objeto está marcado ou não. Quando o valor booleano é positivo, significa que o objeto está marcado.

Taint Data

É o nome dado a um objeto marcado com *taint tag* positivo. Os *tainted data* são objetos que contêm informações sensíveis.

Taint Marking

É o método de marcação dos objetos [27]. Cada abordagem que utiliza *Taint Analysis* cria seu próprio processo de *taint marking*.

Source

É um fornecedor de informações, que pode ser um método, função, objeto ou atributo de um objeto nativo que fornece alguma informação para o programa. Em *JavaScript*, por exemplo, são considerados *sources* os objetos como *window.navigator* e *window.history*, onde um fornece informações sobre o navegador do usuário e o outro fornece informações sobre o histórico de navegação, respectivamente.

Taint Source

É um fornecedor de informações, assim como o *source*, mas é um fornecedor marcado com *taint tag* positivo, por isso é denominado de *taint source*. Essa marcação é utilizada para diferenciar os fornecedores de dados sensíveis, *taint sources*, dos demais *sources* que não fornecem qualquer dado sensível [54].

Taint Sink

É um sumidouro também referenciado simplesmente de *sink*, que pode ser uma função, método ou objeto nativo que possui a capacidade de enviar um dado para fora dos domínios do programa [27, 54]. Em *JavaScript*, por exemplo, são considerados *sinks* o objeto *window.localStorage* e a função *window.postMessage()*, onde um é utilizado para armazenar dados no navegador do usuário e o outro serve para enviar mensagens para outros domínios, respectivamente. Durante a análise, o *sink* é sempre o último vértice do grafo de execução de código, antes que o vazamento de informação seja detectado, servindo como porta de saída para um *tainted data*.

Taint Propagation

É a propagação de um objeto com *taint tag* negativo por um *tainted data*. A propagação do *taint tag* ocorre em diversos momentos da execução do código, como em operações de atribuição, expressões binárias ou passagem de parâmetros, por exemplo.

2.4 Considerações Finais

Este capítulo forneceu conceitos necessários para um melhor entendimento sobre o problema tratado nesta pesquisa. A linguagem de programação *JavaScript* foi apresentada com uma breve descrição de suas principais características, tais como bibliotecas de desenvolvimento, código *inline*, código externo, código explícito e implícito. O problema do vazamento de informação foi descrito, informando os prejuízos causados por ele, que classificam-se em danos diretos e indiretos. A técnica de mitigação de vazamento de informação, conhecida como Análise de Marcação, foi explanada discorrendo sobre os dois tipos de análise de fluxo da informação existentes: análise dinâmica e análise estática.

Neste capítulo também foi realizada uma breve apresentação da plataforma de desenvolvimento *Node.js*, que utiliza a linguagem *JavaScript* para executar no lado do

servidor, e os módulos jsDom e Esprima, que têm função semelhante às bibliotecas *JavaScript* e são utilizados durante a implementação deste trabalho (Seção 4.4).

Capítulo 3

Trabalhos Relacionados

Este capítulo apresenta as principais abordagens para resolver o problema do vazamento de informação sensível.

A revisão sistemática da literatura restrita apenas à linguagem *JavaScript* não apresentou um número de resultados satisfatório, sendo necessário estendê-la também à linguagem Java, no ambiente *Android*. Também são apresentados trabalhos que realizam diferentes formas de análise estática de código *JavaScript*, uma vez que a abordagem *TaintJSec* possui como uma de suas contribuições sua própria forma de analisar o código.

Embora as abordagens sejam distintas, é possível agrupá-las em três grupos:

- Análise de Pacotes
- Lista de Permissão
- Análise do Fluxo da Informação

A seguir são apresentadas as características de cada grupo, com maior ênfase no grupo de análise de fluxo de informação, pois é o grupo ao qual pertence o trabalho apresentado nesta dissertação.

3.1 Análise de Pacotes

A análise de pacotes é utilizada por trabalhos que fazem verificação no conteúdo de pacotes HTTP, antes que esses sejam transmitidos pela internet. A verificação busca por dados sensíveis no campo de dados dos pacotes por meio de comparações entre as cadeias de caracteres.

O trabalho de B. Pinto *et al.* [15] verifica o conteúdo de pacotes antes que este seja cifrado para o protocolo HTTPS (*Hypertext Transfer Protocol Secure*). Para tanto, foi necessário realizar modificações nas bibliotecas nativas do sistema operacional. A

biblioteca *OpenSSL*[1] foi modificada para possibilitar a verificação de dados sensíveis nos campos do pacotes. A abordagem utiliza-se de duas regras de verificação: Cadeias de Caracteres Correspondentes e Expressões Regulares. A verificação de Cadeia de Caracteres Correspondentes tenta encontrar alguma informação sensível no conteúdo do pacote, em textos limpos, sem alterações. Devido possuir o maior custo de processamento, a verificação por Expressões Regulares só é utilizada quando a verificação por Cadeia de Caracteres não consegue encontrar nenhuma informação sensível. A abordagem permite que o usuário escolha o que fazer no exato momento em que um vazamento de informação é identificado, podendo negar o envio do pacote, habilitar o envio do pacote ou sanitizar a informação antes de permitir o envio. Para a última ação, a abordagem troca os caracteres da informação por quantidades iguais do caractere numérico zero, mantendo o tamanho da palavra, porém com o conteúdo repleto de 0's. A abordagem foi testada em um dispositivo móvel e possui como limitação a identificação de dados sensíveis que tenham sofrido algum tipo de ofuscação, como um processo de criptografia personalizado ou uma simples codificação em base 64, por exemplo.

Kuzuno *et al.* [16] desenvolveram uma abordagem de análise de pacotes que utiliza clusterização para detectar o vazamento de dados sensíveis. O agrupamento de dados do tráfico de rede é utilizado para criar assinaturas, onde pacotes contendo dados sensíveis formam um *cluster* de assinatura que é diferente do *cluster* formado pelos pacotes que não possuem qualquer informação sensível. A abordagem verifica a similaridade entre dois pacotes de uma aplicação, por meio da distância de informação nos pacotes, levando em consideração o *IP* (*Internet Protocol*) de destino, porta e outras informações do cabeçalho HTTP. Assim, é possível identificar qual pacote contém informação sensível daquele que não tem, mesmo que os dados do pacote estejam ofuscados.

[1] É uma biblioteca de criptografia para os protocolos *Transport Layer Security* (TLS) e *Secure Sockets Layer* (SSL). Maiores informações em https://www.openssl.org

3.2 Lista de Permissão

Lista de permissão é utilizada para que somente aqueles que nelas estiverem tenham acesso às informações sensíveis do usuário. Algumas listas possuem um maior nível de detalhamento a ponto de conseguirem controlar quais métodos nativos podem ser executados por cada aplicação. A vantagem de usar as listas é que o usuário pode controlar cada aplicação de modo personalizado, podendo restringir permissões atribuídas às aplicações durante sua instalação. A desvantagem é que, para fazer o uso correto das listas, o usuário precisa ter um nível de conhecimento intermediário para entender o que está acontecendo, precisa conhecer o que cada método do Android é encarregado de fazer. Caso contrário, estará restringindo métodos ou aplicações que não são maliciosas, fazendo com que as mesmas não tenham um bom funcionamento.

Wang *et al.* [17] desenvolveram o *ProtecteDroid*, uma abordagem que utiliza lista de permissão para controlar quais aplicações podem ter acesso aos dados sensíveis no sistema operacional *Android*. O nível de granularidade das permissões da lista é *All-or-None*, onde a aplicação tem acesso a todos os dados sensíveis que necessita ou todos os dados sensíveis solicitados retornam valores nulos, porém os *sources* estão disponíveis a qualquer aplicação. A abordagem é configurável e permite que o usuário possa informar quais aplicações podem acessar dados sensíveis. Todas as aplicações enviadas para a lista passam a receber valores nulos a partir dos *tainted sources*, tais como o número do telefone, ID do dispositivo, IMEI, coordenadas de localização geográfica, conteúdo de SMS, agenda de contatos, dados de redes *WiFi*, etc. Os autores informam que, a solução possui granularidade *All-or-None* para facilitar o uso para o usuário, uma vez que um nível de granularidade mais detalhado exigiria um nível de conhecimento técnico que a maioria dos usuários não possuem.

Hsiao *et al.* [18] desenvolveram o *PasDroid*, uma abordagem baseada no *TaintDroid* [27], porém estendendo o campo de controle das informações sensíveis para o usuário. O *PasDroid* é uma abordagem que faz uso de lista branca para controlar quais informações sensíveis que uma aplicação pode ter acesso. É possível que o usuário inclua novos dados para serem catalogados na lista, da mesma forma também pode criar grupos de tipos de informações sensíveis, podendo assim liberar todo um grupo de informações correlacionadas a uma determinada aplicação. Para capturar as mensagens que estão saindo do domínio da aplicação, o *PasDroid* usa a interface do *POSIX* (*Portable Operating System Interface for uniX*), capturando as mensagens de saída. Ao identificar que uma mensagem de saída contém uma informação sensível, é feita uma verificação na lista para checar se o aplicativo em questão tem autorização do usuário para fazer isso. Se tiver, a informação segue normalmente, caso contrário o usuário é informado sobre a possibilidade de vazamento de informação, possibilitando assim, que o usuário tome as

devidas medidas, impedindo ou habilitando a ação. Como limitações da abordagem, o *PasDroid* não leva em consideração os dados do usuários que são codificados em base 64, possui uma sobrecarga de processamento superior ao do *TaintDroid* e, exige que o usuário tenha bom conhecimento técnico para que possa avaliar que medidas tomar quando um vazamento de informação for detectado.

3.3 Análise do Fluxo de Informação

A análise de fluxo de informação é utilizada por trabalhos que fazem a verificação dos passos de funcionamento de um programa. A seguir são apresentados os trabalhos que fazem parte desse grupo.

Vineeth Kashyap *et al.* [19] propõem um interpretador abstrato de *JavaScript* (JSAI) que utiliza uma representação intermediária do código, denominada *notJS*, que corresponde a um produto reduzido de um número de subanálises essenciais que permitem diferentes configurações para a análise estática.

Por meio do *notJS* é gerada uma AST, a qual é utilizada para percorrer o código e realizar a análise de fluxo. Os autores validaram a AST por meio de uma comparação com a ferramenta comercial *SpiderMonkey* [55]. Para tanto, construíram manualmente 243 casos de testes e executaram todos no *SpiderMonkey* e depois no *notJS*, e então verificaram as saídas e compararam suas diferenças. O JSAI é implementado em linguagem *Scala* [56] versão 2.1 e é configurável, podendo executar com diferentes configurações de sensibilidade e profundidade da análise do contexto, permitindo fazer o balanceamento entre precisão e custo da análise. A avaliação dos resultados foi realizada usando *benchmarks* para medir o tempo de execução do JSAI com diferentes configurações de sensibilidade. Mas, diferente do *TaintJSec*, o JSAI não analisa a execução da função *eval* e de qualquer outra inserção de código dinâmico, como a criação de funções utilizando o objeto *new Function*, por exemplo.

O *JSNose* [20] é um abordagem que usa a análise estática em combinação com a análise dinâmica para "farejar" trechos de código *JavaScript* em um programa, procurando blocos de código como instrução de *switch* vazia, *catch* sem argumento e outras situações que servem de apoio ao desenvolvedor. A ideia é que este possa melhorar o código ou realizar *debug*. O *JSNose* é feito em linguagem Java e a análise estática é realizada utilizando o *Rhino* [57], que também realiza a análise sintática e fornece a AST, a qual é consumida em busca dos blocos de textos. Contudo, o *JSNose* não consegue analisar a execução da função *eval* e outras criações dinâmicas de código utilizando somente a análise estática, tal como o *TaintJSec* consegue. Por isso, o *JSNose* executa a análise dinâmica para complementar a lacunas deixadas pela análise estática. Para avaliar seus resultados, os autores organizaram 11 aplicações e analisaram cada uma delas,

enumerando a quantidade de situações encontradas. Como limitações da abordagem, os autores citam que o *JSNose* não checa todos as formas possíveis de criação de objetos, mas analisa somente objetos criados com *new* e a função *Object.create()*.

A ferramenta híbrida desenvolvida por Sora Bae [4] combina análise estática no código *JavaScript* com testes *concolic*[2] para gerar testes automáticos para programas que contêm campos de entrada de dados, como formulários que solicitam nome e data de nascimento, por exemplo, mas que não tratam os dados inseridos, sendo possível enviar qualquer valor. A ferramenta foi desenvolvida sob o *framework SAFE* [58, 59, 60] e faz uso do analisador estático desse *framework* para extrair informações sobre os campos de entrada de dados, para então, criar testes apropriados para cada tipo de campo. Com a ferramenta executando de modo simbólico, é possível detectar caminhos não percorridos no fluxo de código e a partir disso criar regras de entrada de dados que possam percorrer esses caminhos inicialmente não percorridos. Já a aplicação executando em modo concreto fornece valores de dados que podem ser utilizados tanto para os testes simbólicos quanto em testes concretos.

Jacques A. Pienaar *et al.* [21] desenvolveram uma extensão para o *Closure Compiler*[3], denominada *JSWhiz*, para detectar roubo de memória em códigos *JavaScript*. A abordagem utiliza análise estática para buscar por trechos de códigos com padrões previamente definidos que sinalizam o roubo de memória. Esse roubo de memória é caracterizado por uma variável ou um método que não é mais utilizado durante o fluxo de código, mas não é finalizado pelo Coletor de Lixo[4] (*Garbage Collector*). Isso acontece porque a variável possui um ou mais Escutadores de Eventos[5] (*Event Listener*) atrelados a ela, e mesmo que um valor nulo seja atribuído à variável o *Event Listeners* continua existindo, então o *Garbage Collector* não consegue liberar a memória. Essa ferramenta utiliza a AST para realizar a análise de código, porém os autores não informam se a estrutura de árvore sintática é gerada pelo *Closure Compiler* ou pelo *JSWhiz*. Todos os testes, assim como toda sua avaliação, foram feitos manualmente.

Yoonseok Ko *et al.* [22] apresentaram uma abordagem que faz uso do analisador estático do *SAFE* para realizar análise estática de larga escala em aplicações *JavaScript*, não sendo necessário verificar o código inteiro, mas somente alguns trechos. Esses trechos de código são selecionados por quem deseja realizar a análise, formando um subconjunto do código original. Então, o código inteiro é verificado inicialmente por um pré-analisador, o

[2] Palavra gerada pela união dos termos *concrete* e *symbolic*

[3] https://developers.google.com/closure/compiler/

[4] Processo usado para a automação do gerenciamento de memória. Com ele é possível recuperar uma área de memória inutilizada por um programa, o que pode evitar problemas de vazamento de memória, resultando no esgotamento da memória livre para alocação.

[5] Procedimento ou função de um programa que espera que um gatilho ocorra para que uma tarefa seja executada. São exemplos de gatilhos os cliques, os movimentos do mouse, o pressionamento de teclas, etc.

qual procura por trechos de códigos semelhantes aos previamente selecionados e para essa tarefa utiliza técnicas de aproximação. Como resultado, fornece a demarcação de todos os trechos semelhantes. Essa demarcação do código é, então, repassada ao analisador estático para ser utilizada como referência para analisar somente os trechos encontrados pelo pré-analisador. Foram criados dois pré-analisadores, ambos usando o analisador *WALA* [61], onde um analisador executa de modo mais rigoroso que o outro, mais exigente quanto à aproximação dos trechos de código previamente selecionados. Assim, o utilizador da abordagem pode escolher entre possuir um pre-analisador mais exigente ou um mais escalável. Para validar seu trabalho, os autores testaram a abordagem com cinco configurações diferentes. Essas configurações foram testadas em cinco *websites*, cinco bibliotecas *JavaScript* e cinco *benchmarks*. Os resultados foram apresentados em uma tabela que mostra o tempo que cada configuração levou para executar em cada uma das aplicações.

Vineeth Kashyap *et al.* [23] são autores de uma abordagem para fazer refinamento de análises estáticas no *JavaScript*. O refinamento serve para aumentar a precisão da análise evitando problemas com valores implícitos no código. A abordagem possui quatro tipos de refinamento: tipo de domínio abstrato, identificação de condições relevantes, filtragem de informações e propagação. Para criar a AST do código *JavaScript* os autores usaram o *Rhino* [57] e para realizar a análise estática utilizaram o *JSAI* [19]. Para avaliar a eficiência da abordagem utilizam *benchmarks*, os quais foram agrupados em três categorias: *Standard* – com o *SunSpider* [62] e o *Octane* [63], os quais são usados pelos navegadores para testar a performance das implementações *JavaScript*; *Opensrc* – para os programas *JavaScript* extraídos da *web* de diferentes projetos *open source*, com o *LINQ for JavaScript* [64] e o *Defensive JS*; *Emscripten* – para códigos gerados por máquinas por meio da compilação de programas C/C++ que usam o *Emscripten LLVM* [65]. Para cada categoria de *benchmark* a abordagem foi executada para avaliar a quantidade de nós criados pela AST gerada pelo *Rhino*, pois a quantidade de nós gerados é correlacionada com a performance da abordagem. Quanto mais nós possuir mais lenta será.

O trabalho de Daiping Liu *et al.* [24] utiliza a análise estática para buscar códigos *JavaScript* com ações maliciosas em documentos PDF (*Portable Document Format*). A abordagem realizada uma extração do código *JavaScript* presente no documento e insere métodos próprios no início e no fim de cada bloco de código encontrado. Assim, os autores procuram entender como é realizada a execução do código presente no PDF e procuram detectar chamadas de métodos que possam criar novos arquivos infectados, como *malwares* por exemplo.

O *TrustDroid* [25] é um analisador estático que foi criado como solução para o problema do vazamento de informação sigilosa em *smartphones* que usam o sistema operacional Android. O modelo assume que dispositivos móveis podem executar

aplicações maliciosas enquanto acessam informações coorporativas, por meio da crescente prática de *BYOD* (*Bring Your Own Device*), e assim acabam transferindo informações de cunho estratégico em modo furtivo. Embora o *TrustDroid* seja um analisador estático, é possível operá-lo no modo dinâmico, porém o elevado custo de processamento exigido pelos algoritmos e o consumo demasiado de bateria acabam criando uma sobrecarga de utilização de memória e CPU (*Central Processing Unit*) que tornam o modo dinâmico inviável. Em contrapartida, a análise estática do *TrustDroid* aproveita a máquina virtual *Dalvik*, que é a máquina presente no ambiente *Android*, para diminuir a sobrecarga. A abordagem realiza análise semântica no arquivo compilado para poder rastrear os dados no fluxo de execução da aplicação. O analisador semântico do *TrustDroid* foi construído a partir do gerador de analisadores de linguagem, o *ANTLR* (*ANother Tool for Language Recognition*), e do conjunto de símbolos *Dalvik Byte Code*. O analisador é responsável por processar a estrutura de um arquivo compilado *DEX* (*Dalvik EXecutable*) e apresentar o fluxo de execução da aplicação em uma estrutura de árvore. A Figura 3.1 apresenta um exemplo de um código-fonte de um programa qualquer, que ao ser compilado gera um arquivo *DEX*. Este arquivo, por sua vez, passa pelo analisador semântico que fornece a estrutura do *bytecode* em formato de árvore, conforme ilustra a Figura 3.2.

```
/* retrieve local phone number*/
      TelephonyManager phoneMgr=(TelephonyManager)
this.getSystemService(Context.TELEPHONY_SERVICE);
      String number=phoneMgr.getLine1Number();

      Trustdroid_test.sendinfo(number);

class Trustdroid_test
{
   public static void sendinfo(String phoneNumber) throws
UnknownHostException, IOException
   {
      Socket socket=new Socket("127.0.0.1",123);
      OutputStreamWriter writer=new OutputStreamWriter
(socker.getOutputStream());
      writer.write(phoneNumber);
   }
}
```

Figura 3.1. Exemplo de um trecho do código-fonte de um programa.

A árvore apresenta nós virtuais que servem para identificar qual a estrutura das subárvores, esses nós possuem o prefixo **I_**, como o nó **I_METHOD**, o qual indica que sua subárvore possui um método. São essas nomenclaturas presentes na estrutura da árvore que servem de base para o controle da análise estática. No *TrustDroid*, as regras do *taint propagation* dividem-se em cinco: regras de propagação primitiva, propagação por referência, propagação entre processos, propagação entre bibliotecas nativas e propagação em arquivos armazenados em disco. Os autores testaram a abordagem em um dispositivo

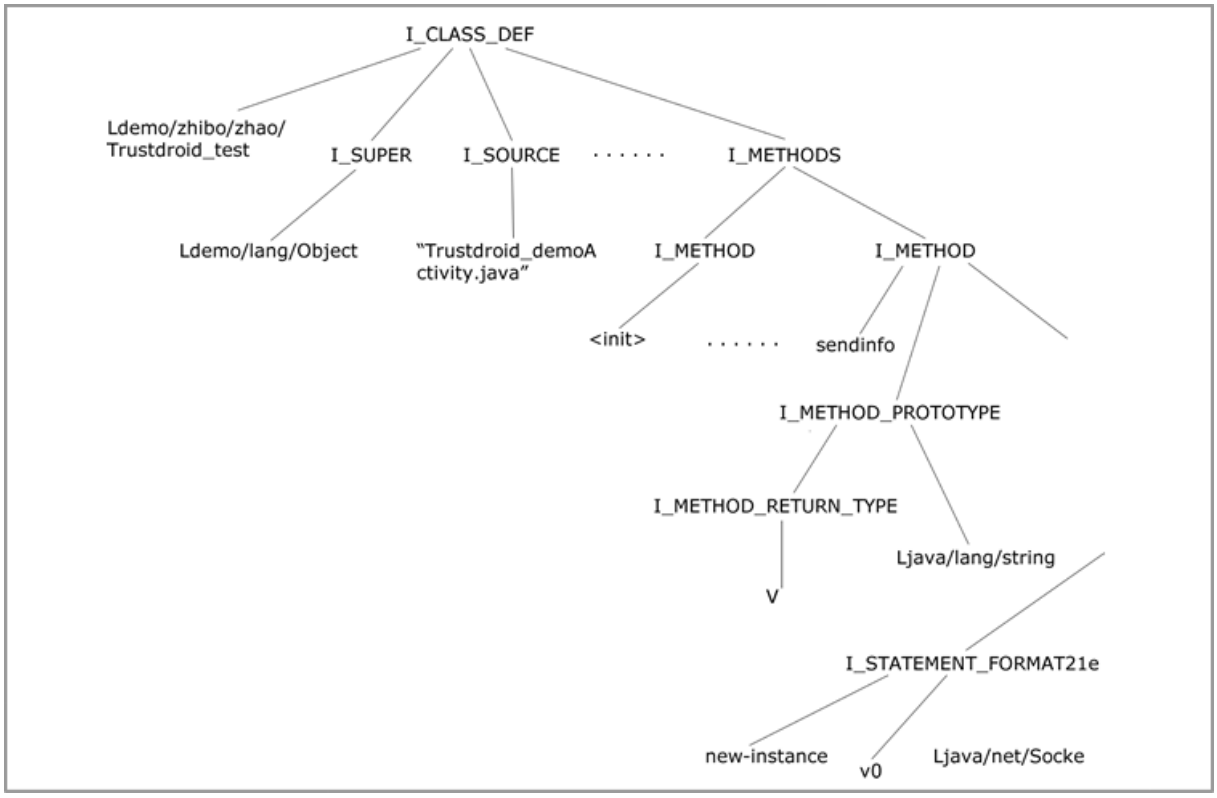


Figura 3.2. Estrutura de árvore gerada pelo código-fonte da classe *Trustdroid_test*.

Android e possui a limitação de não conseguir acompanhar o fluxo de aplicações que usem bibliotecas personalizadas por meio do JNI (*Java Native Interface*).

Um trabalho voltado para identificar código malicioso pelo lado do servidor (*server-side*) foi desenvolvido por Z. Yang *et al.* [26], denominado *LeakMiner*, cujo objetivo é identificar aplicações maliciosas nos centros de distribuição de aplicativos, antes que os aplicativos sejam disponibilizados para os usuários.

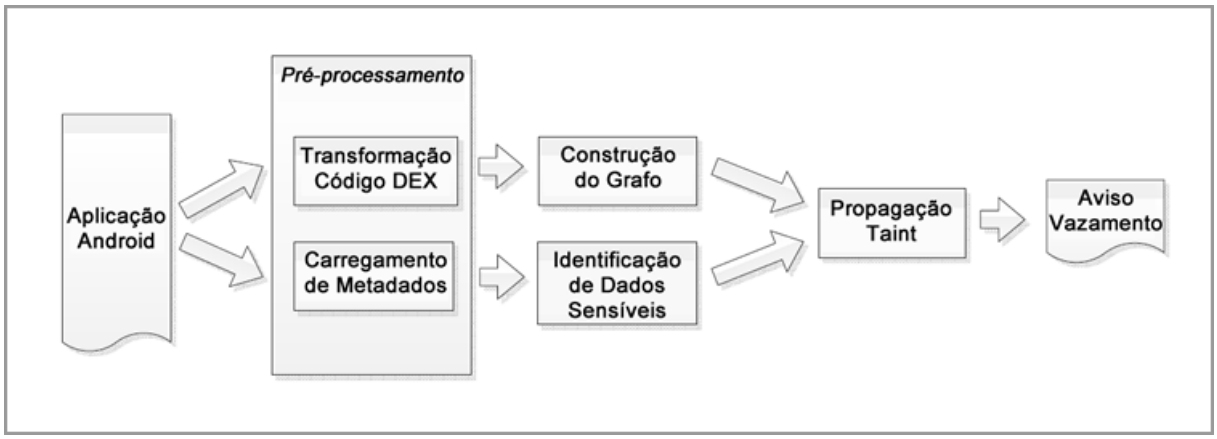


Figura 3.3. Arquitetura geral do *LeakMiner*.

A Figura 3.3 apresenta a estrutura do *LeakMiner*, que divide-se em três etapas para

fazer o processo de verificação do código de um aplicativo:

I. Pré-processamento: o *LeakMiner* transforma o arquivo DEX para *bytecode* JAVA, por meio de engenharia reversa. Nesse código JAVA é realizada análise estática para criar um grafo da execução da aplicação.

II. Identificação de dados sensíveis: é verificado quais permissões que o aplicativo possui, para isso é realizada a extração de metadados do arquivo *Manifest* da aplicação. São esses metadados que possibilitam ao *LeakMiner* a identificação de quais informações sensíveis a aplicação terá acesso.

III. Propagação do fluxo de informação: por fim, é realizada a análise de fluxo de execução da aplicação. Uma técnica de análise de fluxo da informação é aplicada para encontrar todas as instruções que provenientes de um *tainted source*. Se um *tainted data* é propagado para a rede ou armazenado no sistema local, então um caminho de vazamento é identificado e reportado ao usuário.

O *LeakMiner* [26] é considerado, pelos autores, a primeira abordagem *server-side* de análise estática para detecção de vazamento de informação sensível no sistema *Android*. A abordagem consegue interceptar o vazamento das seguintes informações sensíveis:

I. ID do dispositivo: é o número identificador dos *smartphones*. Por ser um número utilizado como autenticador de usuário em muitos aplicativos, então é considerado perigoso caso seja enviado em conjunto com outras informações sensíveis. A abordagem trata tanto o número IMEI (*International Mobile Equipment Identity*) quanto o ID de assinante como sendo ID do dispositivo. Essa informação sensível só é verificada quando o arquivo *Manifest* da aplicação possui a permissão *READ_PHONE_STATE*.

II. Localização: é a informação que contém a localização atual do dispositivo. Aplicações maliciosas podem usar essa informação para monitorar o deslocamento e as rotinas do dia a dia do usuário. Para que a aplicação tenha acesso a esse tipo de informação é preciso que ela solicite *ACCESS_FINE_LOCATION* e *ACCESS_COARSE_LOCATION*.

III. Número do celular: da mesma forma que o ID do dispositivo, o número do celular pode ser utilizado como autenticador o usuário. A abordagem trata tanto o número do celular quanto o número do cartão SIM (*Subscriber Identity Module*) como sendo informação sensível se a permissão *READ_PHONE_STATE* for utilizada pela aplicação.

IV. Agenda de contatos: a abordagem preocupa-se em acompanhar a informação da agenda de contatos quando uma aplicação faz uso da permissão *READ_CONTACT*.

V. Mensagem SMS: o *LeakMiner* assume que o vazamento de mensagem de texto SMS (*Short Message Service*) possui maior impacto ao usuário, pois todas as mensagens recebidas ou enviadas pelo usuário contêm muita informação pessoal. As aplicações só podem ter acesso às mensagens se a permissão *READ_SMS* for concedida.

VI. Calendário: muitos usuários guardam datas importantes ou agendam eventos

pessoais mantendo-os armazenados no calendário do *Android*, sendo um possível alvo de aplicações maliciosas, que podem capturar essas informações caso tenham a permissão *READ_CALENDAR*.

Essas informações podem ser carregadas mediante a execução de métodos nativos e não precisam obrigatoriamente estar no início do fluxo de execução de uma aplicação maliciosa, mas podem ser executados a qualquer momento. Por isso o *LeakMiner*, ao criar o grafo do fluxo de execução da informação em uma aplicação, define os *taint sources* para então realizar a análise estática percorrendo todos os possíveis caminhos que levem a um *taint sink*, encontrando assim um caminho de vazamento de informação em potencial. A abordagem foi testada em uma máquina Intel Xeon com o sistema operacional *Debian* e uma amostragem de 1750 aplicações, resultando na identificação de 145 aplicações maliciosas e 160 falsos positivos devido à insuficiência de informação de contexto, o que caracteriza uma limitação da abordagem. Aplicar análise sensível ao contexto poderia eliminar esses falsos positivos, mas deixaria o processo de análise mais lento.

Um outro trabalho denonimado *TaintDroid* [27] usa *taint analysis* em aplicações Android para detectar vazamento de informação. Para tal, são realizadas alterações no código nativo do sistema operacional para permitir que toda requisição de um aplicativo seja monitorada, para que seja possível identificar o momento em que um aplicativo acessa um *taint source* e realiza o carregamento de dados sensíveis. A marcação dos registradores é realizada em um banco de dados na memória principal, armazenando o número do registrador. Uma vez que a base de dados possua um ou mais registradores contendo *tainted data*, o *TaintDroid* passa a considerá-los como sendo *taint sources*, podendo atribuir dados sensíveis a outros registradores mediante funções de atribuição de variáveis. Portanto, a abordagem possui uma política de identificação de *taint propagation*, servindo tanto para identificar dados sensíveis, que estão sendo propagados para outros registradores, quanto para atualizar a sua base de dados.

A técnica de propagação utilizada na abordagem observa todas as funções de atribuição, chamadas e retorno de métodos de classes nativas e gravação de *tainted data* em arquivos físicos. Para operações de atribuição, o método assume que, se o registrador **A** é um *tainted data* e passa o seu valor ao registrador **B**, logo, o registrador **B** também será *tainted data*. Para as classes nativas, o método de propagação assume que a classe inteira é marcada com *taint tag* positivo se os seus campos de dados receberem um *tainted data*. Já os arquivos físicos são marcados com *taint tag* positivo se receberam escrita de um *tainted data*, e toda função de leitura que referencia um arquivo marcado é tambem marcada com o *flag*. Testes realizados com o *TaintDroid* utilizando 30 aplicativos mostraram que em média a sobrecarga de processamento do dispositivo é de 14%. Os resultados mostraram que 20 aplicações das 30 avaliadas possuíam uma ou mais ocorrências de uso indevido das informações dos usuários.

Patnaik *et al.* [66] desenvolveram uma ferramenta de análise estática de código *JavaScript* denominada *JSPrime*, para a detecção de código malicioso. Essa ferramenta utiliza um conjunto de *taint sources* e *sinks* e verifica o fluxo do código procurando encontrar um caminho de informação sensível a um *sink*. Assim como o *TaintJSec*, o *JSPrime* utiliza um código intermediário para realizar a verificação do fluxo de código. Porém, o analisador do *JSPrime* não executa verificação na função *eval*. Em vez disso, ele trata a função *eval* como um *sink* e qualquer chamada à função *eval*. Por exemplo, o código eval(“ ’ ’ ”) acusa um vazamento de informação. Por não verificar a função *eval* o *JSPrime* acusa falso positivo, como mostra a Figura 3.4.

Figura 3.4. Exemplo de execução do analisador *JSPrime*.

O falso positivo acusado pela ferramenta *JSPrime* não ocorreria se o escopo da função *eval* fosse analisado, como ocorre no *TaintJSec*.

Uma ferramenta denonimada *ScanJS* [67] foi desenvolvida por Paul Theriault para realizar análise estática no código *JavaScript* baseada em padrões de assinatura. Essa ferramenta é implementada em Node.js e utiliza o módulo Acorn[6] para criar uma representação intermediária do código *JavaScript* em formato de AST. Essa AST é percorrida em busca dos padrões de assinatura previamente definidos pela ferramenta e o resultado da análise é apresentado ao usuário. No resultado da análie, a ferramenta informa quais padrões foram encontrados e indica também a linha em que encontram-se no código original. A Figura 3.5 mostra a área de cadastro de novos padrões de assinatura da ferramenta *ScanJS*.

Por ser baseada em padrões, a ferramenta *ScanJS* consegue identificar os pontos de vulnerabilidade no código *JavaScript* a partir dos trechos de código que satisfaçam

[6] https://www.npmjs.com/package/acorn

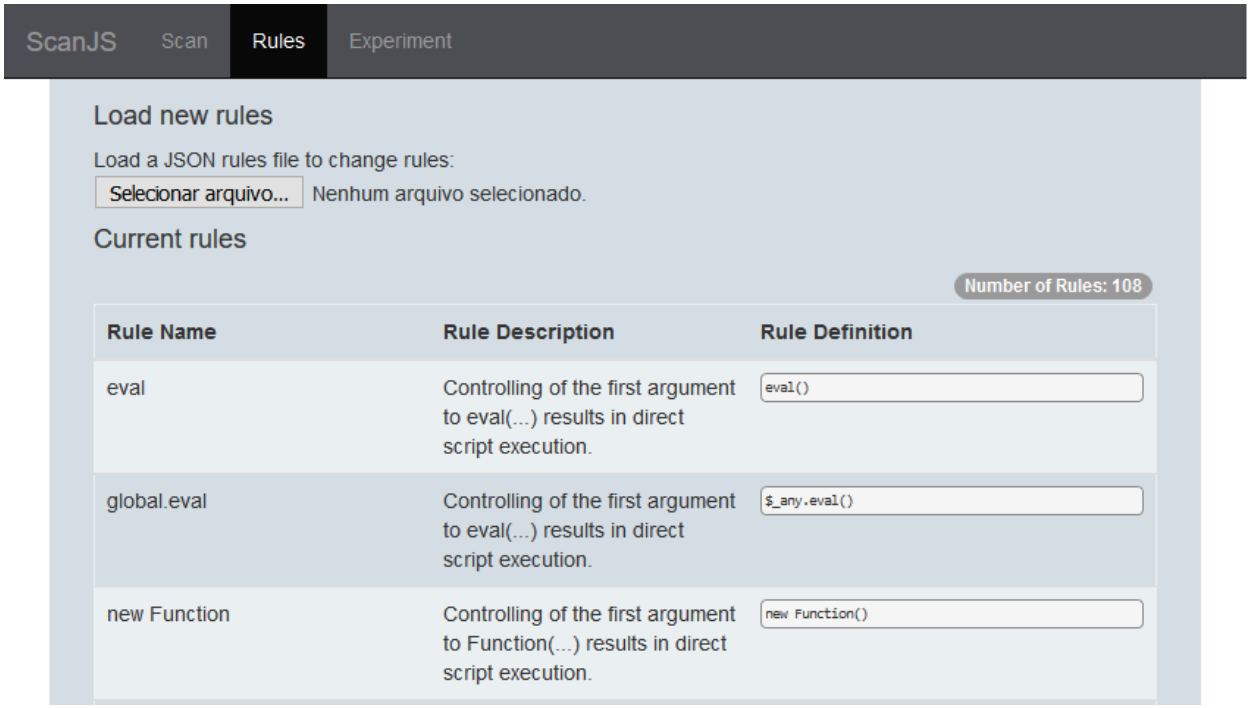


Figura 3.5. Área de cadastro de regras de assinaturas do analisador *ScanJS*.

as regras cadastradas. Isso significa que a ferramenta não é capaz de detectar códigos ofuscados que executem as mesmas funções e métodos presentes nas regras.

Monteiro *et al.* [68] criaram a ferramenta *JSpwn* para realizar análise estática no código *JavaScript*. *JSPwn* é uma versão modificada do *JSPrime* que une recursos do *Scanjs* para detectar vulnerabilidades. Com o mecanismo do *ScanJS* para detectar vulnerabilidades e o recurso de análise de fluxo de código do *JSprime*, a ferramenta tem a capacidade de detectar pontos de vulnerabilidades ao percorrer o código. Mas por utilizar um analisador de código de um trabalho existente, acaba herdando também suas limitações. Por isso, também não avalia o escopo da função *eval* e isso acarreta diversos falso positivo. A Figura 3.6 mostra um exemplo de código *JavaScript* sendo analisado pela ferramenta *JSPwn*.

Na Figura 3.6 é possível observar que o analisador não conseguiu identificar o vazamento na linha 4 e na linha 9. Isso aconteceu porque o analisador é baseado em regras de assinatura e a passagem de objetos não é contemplada. É possível observar na figura, que a ferramenta conseguiu detectar o uso da função *eval* na linha 7, mas não conseguiu detectar os objetos atribuídos para as variáveis na linha 2 e na linha 8, as quais são executadas logo em seguida sem qualquer detecção. Esse exemplo de código é detectado pelo *TaintJSec* por meio da propagação de *taint tag* e pelo método de analise de escopo da função *eval*.

Outros trabalhos que tinham como objetivo identificar o vazamento de informação

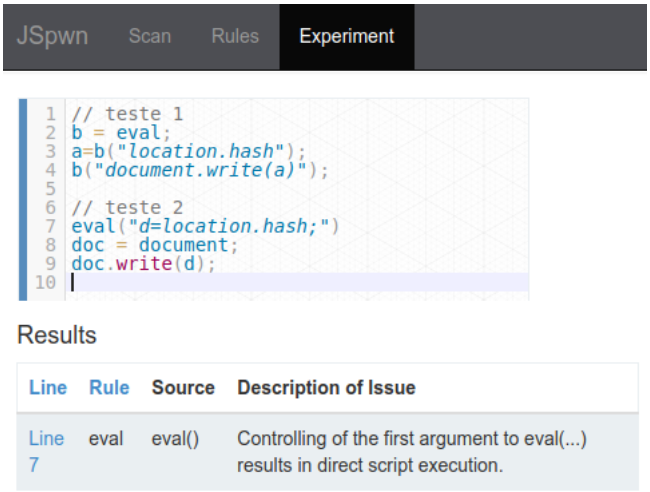


Figura 3.6. Exemplo de execução do analisador *JSpwn*.

sensível, resultaram em ferramentas como *JSLint*[7], Jalangi[8] e o *jsTaint*[9], porém todos eles apresentam limitações, seja na verificação da função *eval*, na criação de códigos dinâmicos ou na correta interpretação de códigos ofuscados.

[7] http://www.jslint.com
[8] http://people.eecs.berkeley.edu/~gongliang13/jalangi_ff/demo_integrated.htm
[9] https://github.com/idkwim/jsTaint

3.4 Discussão

De acordo com a Tabela 3.1 é possível observar que as abordagens preocupam-se em ausentar a participação do usuário durante sua execução. O que é compreensível, pois os usuários muitas vezes não possuem o conhecimento técnico necessário para auxiliar a abordagem nas decisões que devem ser realizadas. O *ProtecteDroid* e o *PasDroid*, por exemplo, necessitam da participação do usuário, porém sem exigir muito do mesmo, sendo necessário apenas uma escolha binária, onde o usuário informa se uma determinada aplicação pode ter acesso às informações sensíveis ou não.

Visto que as lojas oficiais de distribuição de aplicativos possuem seus próprios métodos de detecção de aplicações maliciosas e, que outros centros de distribuição não oficiais possuem pouca ou nenhuma rotina de análise de segurança, é possível observar na Tabela 3.1 que a maior parte dos trabalhos foram criados para executar no lado do cliente (*client-side*), onde não existe qualquer rotina nativa para detecção de vazamento de informação. Além disso, as abordagens mostram uma preferência para testes realizados no mundo real, talvez para ausentar qualquer limitação imposta pelos emuladores.

Assim como a maioria dos trabalhos, o *TaintJSec* é um método de análise estática que foi testado e aplicado no mundo real e executa em modo autônomo, mas diferencia-se dos demais métodos por poder ser executado tanto do lado do cliente quanto do servidor.

Tabela 3.1. Características dos trabalhos relacionados.

Autor	Tipo de Análise				Ambiente de Teste		Lado de Execução		Modo de Execução	
	DM	EM	LP	PH	Mundo Real	Emulador	Cliente	Servidor	Autônomo	Supervisionado
JSAI [19]		✓			✓		✓		✓	
JSNose [20]	✓	✓			✓		✓		✓	
Sora Bae [4]		✓			✓		✓		✓	
JSWhiz [21]		✓			✓		✓		✓	
Yoonseok Ko [22]		✓			✓		✓		✓	
Vineeth Kashyap [23]		✓			✓		✓		✓	
Daiping Liu [24]		✓			✓		✓		✓	
TaintDroid [27]	✓				✓		✓		✓	
TrustDroid [25]		✓			✓		✓		✓	
Leakminer [26]		✓				✓		✓	✓	
ProtecteDroid [17]			✓			✓	✓			✓
Nedwons Findings [15]				✓	✓		✓			✓
PAsDroid [18]			✓		✓		✓			✓
Kuzuno [16]				✓	✓		✓		✓	
TaintJSec		✓			✓		✓	✓	✓	

Legenda
DM = Análise Dinâmica com Marcação , EM = Análise Estática com Marcação ,
LP = Listas de Permissão , PH = Análise de Pacotes HTTP

Na Tabela 3.2 são apresentadas limitações e características dos trabalhos relacionados que realizam análise de fluxo no *JavaScript*.

É possível notar que muitos trabalhos aproveitam o analisador de algum trabalho predecessor, como é o caso do *JSNose* que utiliza o *Rhino* e os trabalhos de Sora Bae

Tabela 3.2. Limitações e características dos trabalhos relacionados com análise em *JavaScript*.

Autor	Suporta criação dinâmica de código?	Suporta função eval?	Possui analisador próprio?	Suporta código ofuscado?				
				JSCompress	Aaencode	JSFuck	JSCrambler	Packer
JSAI			✓	✓			✓	
JSNose	✓			✓			✓	
Sora Bae				✓			✓	
Yoonseok Ko		✓		✓			✓	
Vineeth Kashyap				✓			✓	
Daiping Liu	✓	✓	✓	✓	✓	✓	✓	✓
JSPrime	✓	✓	✓	✓			✓	✓
JSpwn	✓	✓						
ScanJS	✓	✓	✓					
JSLint	✓		✓	✓				
Jalangi	✓	✓	✓	✓		✓	✓	✓
jsTaint	✓		✓	✓				
TaintJSec	✓	✓	✓	✓	✓	✓	✓	✓

et al. [4] e Yoonseok Ko *et al.* [22] que utilizam o *framework* SAFE para realizar a análise estático do código. Como consequência, acabam herdando também as limitações do analisador, como é o caso do JSAI, que não é preparado para analisar códigos criados dinamicamente, e é utilizado como analisador estático no trabalho de refinamento de Vineeth Kashyap *et al.* [23], que apresenta as mesmas limitações. Também é possível observar que, embora suportem a função *eval*, alguns trabalhos não conseguem analisar os códigos quando estes apresentam-se ofuscados por ferramentas como o *Aaencode* e o *JSFuck*. Isso se deve ao método adotado para analisar o argumento enviado à função *eval*, o qual trata somente textos limpos, sem qualquer ofuscação. Tal limitação poderia ser superada se esses trabalhos utilizassem a técnica dos *tainted positions* criada por este trabalho e explicada mais adiante, no Capítulo 4.

Em comparação com o *TaintJSec*, os demais trabalhos apresentam limitações de detecção de vazamento de informação quando o código *JavaScript* utiliza funções de criação de código dinâmico, como a função eval. As ferramentas de ofuscação *Aaencode*, *JSFuck* e *Packer* usam técnicas de criação de código em tempo de execução, por isso métodos como o *JSNose* e o *JSPwn* não conseguem analisar o código ofuscado.

Capítulo 4

TaintJSec

Este capítulo apresenta o *TaintJSec*, uma abordagem de análise estática de marcação de código *JavaScript* para identificar e prevenir o vazamento de informação.

O *TaintJSec* realiza, antes de tudo, uma etapa de extração e agrupamento de todo o código *JavaScript* presente na aplicação. Em seguida, o código agrupado é enviado a um analisador, que a partir dele, constrói uma representação da estrutura do fluxo de sua execução em formato de árvore sintática abstrata (AST). Com a árvore sintática conhecida, é realizada a terceira etapa da abordagem, percorrendo as instruções do código e acompanhando o fluxo das informações sensíveis.

Logo, a abordagem pode ser dividida nas seguintes fases: identificação e extração de código; criação da árvore sintática abstrata; e análise do fluxo da informação, como apresentado na Figura 4.1.

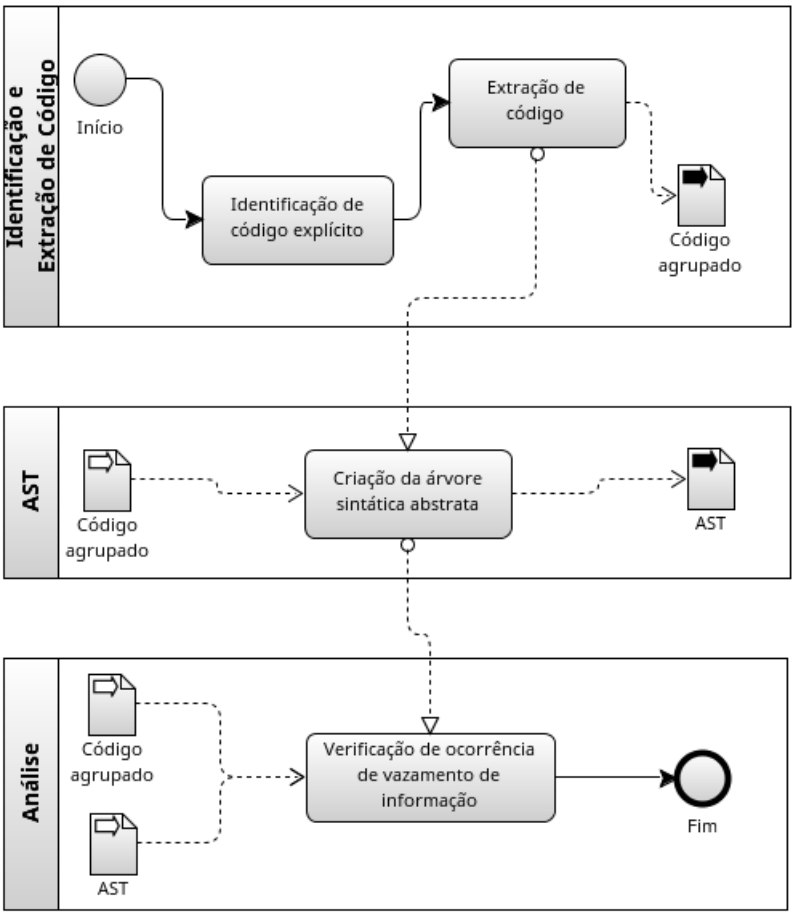


Figura 4.1. Fases da abordagem.

4.1 Identificação e Extração de Código

A primeira fase do *TaintJSec* tem o objetivo de identificar todos os códigos *JavaScript* (Figura 4.2a) que estão declarados explicitamente na aplicação, para que possa extraí-los, separando-os do restante do código HTML e, então, organizá-los de acordo com a ordem em que aparecem, sem alterar o fluxo de execução das instruções (Figura 4.2b). Não há distinção entre instruções síncronas e assíncronas, pois a abordagem trata todas as instruções como síncronas. Portanto, os diversos códigos *JavaScript* dispersos na aplicação, ao serem agrupados em um único arquivo, obedecem a ordem de interpretação em que eles aparecem no código HTML. Dessa forma, o fluxo da informação presente no código agrupado é igual ao fluxo original, presente na aplicação.

A tarefa de identificar todos os códigos *JavaScript* presentes em uma aplicação HTML merece atenção, pois o método escolhido para executar essa tarefa deve reconhecer falsas *tags script*. Durante a fase de identificação é preciso diferenciar legítimos códigos *JavaScript* daqueles que não devem ser levados em consideração como, por exemplo, os códigos presentes em conteúdo de comentários, códigos que localizam-se entre as *tags textarea* ou, ainda, códigos que estejam entre as *tags script* que contenham o atributo *src*.

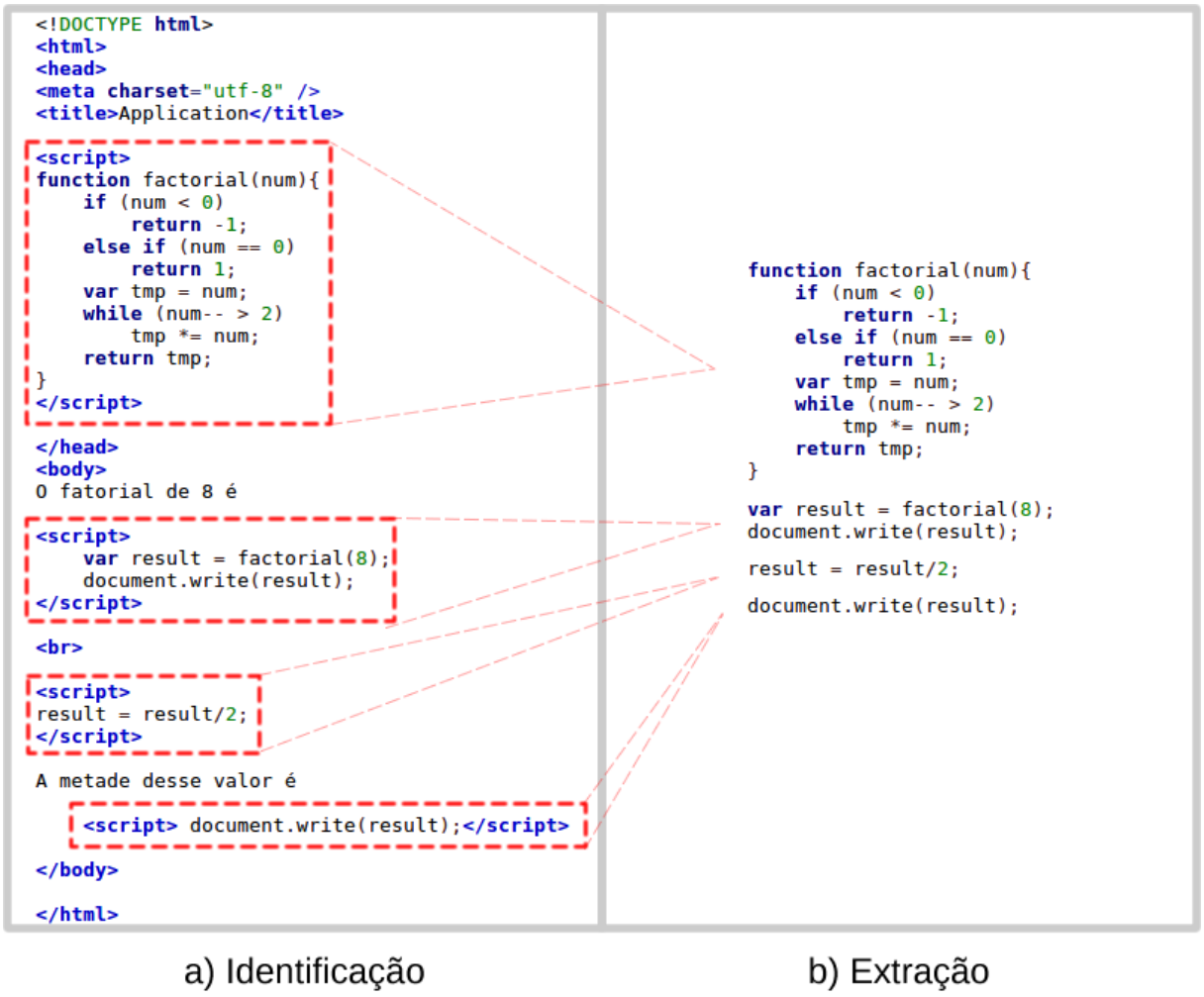


Figura 4.2. Etapas da fase de identificação e extração de códigos *JavaScript*.

O Código 4.1 apresenta uma estrutura HTML que não apresenta qualquer código *JavaScript* executável que possa ser extraído. Entre as linhas 4 e 6 consta um código *JavaScript* que contém uma função *alert*, porém a *tag* inicial contém o atributo *src* que anula todo o código presente entre as *tags* <*script*> e </*script*>. Somente o endereço do arquivo externo informado no valor do atributo *src* é levado em consideração, mas como o atributo não apresenta qualquer valor, então não há código para ser executado. Por essa razão, o trecho de código entre as linhas 4 e 6 não contém qualquer código *JavaScript* para extrair.

```
1 <!doctype html>
2 <html>
3  <head>
4   <script src>
5   alert("Todo código dentro dessa tag é ignorado devido o uso do atributo src na
        tag <script>.");
6   </script>
7   <!--
```

```
8   <script>
9   alert("Esse trecho não é um código JavaScript. É somente um comentário do có
        digo HTML.");
10  </script>
11  -->
12 </head>
13 <body>
14  <textarea>
15   <script>
16    alert("Isso é apenas conteúdo do objeto Textarea.");
17   </script>
18  </textarea>
19 </body>
20 </div>
21 </html>
```

Código 4.1. Exemplo de HTML sem código *JavaScript*.

Entre as linhas 8 e 10 do Código 4.1 é apresentado, aparentemente, outro código *JavaScript*, mas trata-se apenas do conteúdo de um comentário HTML. A *tag* de comentário HTML inicia na linha 7 e termina na linha 11. O conteúdo entre essas *tags* não é um código executável.

A linha 16 do Código 4.1 apresenta a função *alert* que está entre as *tags* <*script*> *e* </*script*> ambas nas linhas 15 e 17, respectivamente. Porém, este também não é um código *JavaScript* executável que possa ser extraído, pois é o conteúdo do elemento *Textarea* que inicia na linha 14 e termina da linha 18. Portanto o Código 4.1 não apresenta qualquer código *JavaScript* para ser identificado e extraído.

Quando um código válido é encontrado, todas as suas particulares devem ser preservadas. Nenhum comentário deve ser removido, assim como nenhuma tentativa de minificar[1] o código deve ser executada, pois até um simples caractere de espaço é essencial e deve ser mantido como consta no código original. Isso é importante para que a análise trabalhe em um código *JavaScript* idêntico ao presente na aplicação.

A primeira fase do *TaintJSec* termina quando todos os códigos *JavaScript* tiverem sido extraídos do HTML e devidamente organizados formando um só bloco de código, composto por todos os demais trechos obtidos separadamente.

4.2 Criação da Árvore Sintática Abstrata

A segunda fase da abordagem tem por objetivo criar a árvore sintática abstrata do código *JavaScript* extraído na fase anterior.

[1]Especialmente para *JavaScript*, minificar é remover os trechos desnecessários para que o código rode mais rápido.

No *TaintJSec*, o mecanismo de criação da AST possibilita a inserção de novas informações na árvore por meio de processamento subsequente. É possível adicionar o número da posição exata de um trecho de código *JavaScript* responsável pela criação de um nó específico da árvore, ou adicionar qualquer atributo de valor relevante com o propósito de enriquecer a AST com informações úteis que facilitem seu processamento na terceira fase da abordagem. Logo, a árvore sintática gerada deve ser uma estrutura flexível, que permita a inserção de novos atributos e a alteração dos dados presentes nos nós a qualquer momento da análise.

Com a AST criada, a segunda fase do *TaintJSec* chega ao fim. A abordagem passa a ter dois artefatos de saída: o bloco de código *JavaScript* criado na primeira fase da abordagem e a sua respectiva AST. Esses dois artefatos são necessários para a realização da análise estática no fluxo de execução.

4.3 Análise do Fluxo da Informação

A terceira fase da abordagem tem por objetivo a realização de uma análise no fluxo de execução do código *JavaScript* para identificar possíveis vazamentos de informação sensível. Para que a análise seja realizada são necessários dois artefatos de entrada: o código *JavaScript* e a sua respectiva representação em AST, ambos são artefatos de saída das fases anteriores. O código *JavaScript* serve de apoio para a análise, onde são feitas consultas no código com o propósito de obter o trecho real do código gerador da AST. Esse tipo de consulta é útil para tratar o problema do *eval* e detectar vazamento de informação sensível em códigos ofuscados, como veremos adiante.

A árvore sintática abstrata é o parâmetro de entrada para iniciar a terceira fase da abordagem, pois a análise consiste em percorrer os níveis da AST utilizando-se de estruturas para armazenar os resultados obtidos das operações encontradas durante o fluxo de execução. Essas operações são atividades comuns presentes na lógica do código *JavaScript*, como as operações de atribuição, cálculos de expressões matemáticas, chamadas de função, instruções condicionantes, etc.

Para facilitar a compreensão da análise do fluxo da informação, esta seção é subdividida para explanar separadamente cada parte que compõe a terceira fase da abordagem. A subseção 4.3.1 desenvolve acerca de como os registradores do *TaintJSec* são organizados. A subseção 4.3.2 mostra como a estrutura dos registradores é preparada para que a análise de marcação (*taint analysis*) seja empregada. A subseção 4.3.3 informa quais são os *taint sources* deste trabalho, assim como a subseção 4.3.4 informa quem são os *taint sinks*. Por fim, a subseção 4.3.5 discorre sobre as diferentes formas de propagação do *taint tag* e insere um novo termo denominado *taint position*.

4.3.1 Estrutura dos Registradores

Os registradores são estruturas que armazenam os dados obtidos durante a análise do fluxo de código. Esses registradores diferenciam o escopo ao qual pertencem, para que assim não ocorra ambiguidade entre os dados armazenados. Desta forma, um registrador *r1* não interfere nos dados de outro registrador *r2*, quando estes armazenarem nomes de variáveis idênticos, mas de escopos distintos. O Algoritmo 4.1 é utilizado para armazenar e atualizar os objetos de um determinado escopo, onde novos escopos são inseridos em um conjunto de escopos na medida em que a análise consome os níveis da AST. Caso o escopo já exista, então o objeto é inserido no conjunto de objetos do escopo (linha 7), de modo que nesse conjunto de objetos não é permitido a existência de dois objetos com o mesmo identificador. Logo, a inserção de um objeto com um identificador já inserido anteriormente sobrescreve o objeto mais antigo.

Algoritmo 4.1: Armazenamento e atualização de objetos em registradores.

Entrada: (S, u, k)
Saída: Conjunto de escopos atualizado
/* onde S é o conjunto de todos os escopos conhecidos, */
/* u é um escopo específico, e */
/* k é um objeto do escopo u */

```
1  início
2  |  i ← k.identificador
3  |  se u ∉ S então
4  |  |  S ← S ∪ {u}
5  |  |  σ(S, u) ← { i : k }
6  |  senão
7  |  |  σ(S, u, i) ← k
8  |  fim
9  fim
10 retorna S
```

A estrutura de registradores deve ser flexível o suficiente para permitir que novas informações sejam inseridas em todos os registradores ou somente em um registrador específico. Essa característica é útil para detectar se a função *eval* contém informação sensível em seus argumentos, bem como para propagar um novo atributo inserido pelo objeto *prototype*[2] a todas as instâncias de uma classe. Além de permitir a utilização da análise de marcação pelo *TaintJSec*.

De modo simplificado, os registradores do *TaintJSec* podem ser representados como uma estrutura dinâmica composta de vários objetos, onde o endereço de memória em que

[2]Em *JavaScript*, *prototype* é uma propriedade interna de todos os objetos. Modificações no *prototype* do objeto são propagadas a todos objetos através de encadeamento.

um registrador fica armazenado é obtido por meio do identificador do objeto *JavaScript* que ele armazena. A Figura 4.3 ilustra a abstração dessa representação.

Figura 4.3. Representação simplificada da estrutura de armazenamento de um registrador.

Para evitar ambiguidade nas informações armazenadas, cada escopo possui sua própria estrutura de registradores.

Uma visão simplificada da estrutura de armazenamento de escopos é apresentada na Figura 4.4, onde é possível observar que o conteúdo de cada escopo é composto por uma estrutura de registradores e referenciado por um identificador *id* único, que é gerado conforme os escopos vão sendo descobertos durante a execução da análise, respeitando a hierarquia em que aparecem. De tal maneira que dois escopos com identificadores **1.1** e **1.2**, por exemplo, são escopos adjacentes e também subescopos de um terceiro que possui identificador igual a **1**. Logo, a estrutura de armazenamento de escopos permite o reconhecimento de paridade por meio de seus identificadores.

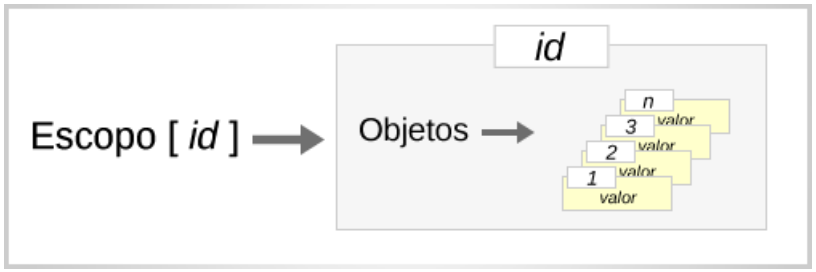


Figura 4.4. Representação simplificada da estrutura de armazenamento de escopos.

Para melhor entendimento do funcionamento dessas estruturas, o código **answer = 6 * 7** é utilizado como exemplo na Figura 4.5, a qual ilustra a estrutura gerada pelo código ao passar pela fase de análise.

A estrutura representa um escopo de identificador **1** que possui somente um objeto, cujo identificador e valor armazenados são *answer* e o número inteiro 42, respectivamente.

4.3.2 Marcação de Registradores

A marcação durante a análise tem o objetivo de rotular (ou etiquetar) os registradores que estiverem portando informação sensível, para que o algoritmo de detecção de vazamento

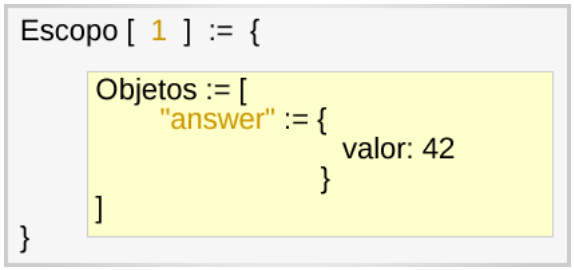


Figura 4.5. Exemplo da estrutura gerada por um caso específico.

de informação do *TaintJSec* possa identificá-los. Porém, para empregar a técnica de marcação é necessário que o *taint tag* esteja inserido na estrutura dos registradores.

Como dito previamente, um registrador de objeto é uma estrutura que contém informações relevantes, tais como o identificador (nome de variável ou função), o valor, o tipo e os atributos do objeto. Essa estrutura possui como característica a possibilidade de inserção de novas informações em qualquer momento da análise. Assim, o *taint tag* é inserido na estrutura dos registradores (Figura 4.6) e a análise passa a ser uma Análise de Marcação (*Taint Analysis*).

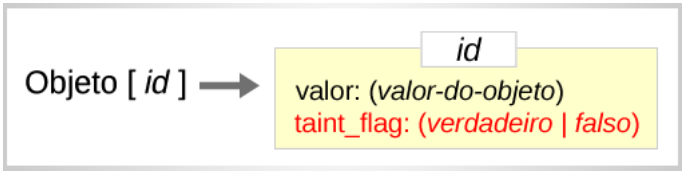


Figura 4.6. Representação simplificada da estrutura de armazenamento de um registrador contendo o *taint tag*.

Desse modo, com o *taint tag* na estrutura dos registradores é possível marcar um ou mais objetos, cujos valores possuam informações sensíveis. A *tag* possui valor booleano, onde *verdadeiro* e *falso* indicam objetos que estejam portando informação sensível (*tainted data*) e os objetos que não possuam informações sensíveis em seu valor, respectivamente. Uma informação sensível é o resultado da verificação de sua procedência, isto é, se a informação provém de um *taint source*, como explicado na próxima seção.

Para efeito de comparação, se o exemplo apresentado anteriormente possuísse o *taint tag*, sua estrutura seria como mostra a Figura 4.7 e o registrador de objeto com *id* igual a *answer* teria o *taint tag* com valor *falso*, indicando que esse objeto não contém uma informação sensível.

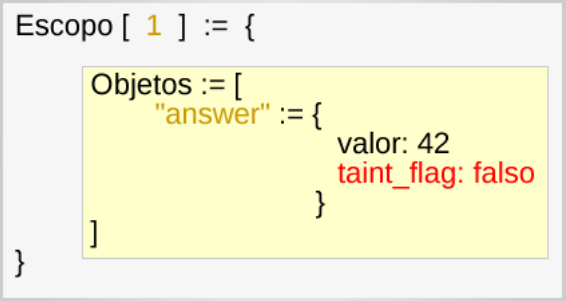


Figura 4.7. Exemplo do *taint tag* inserido na estrutura de um registrador.

4.3.3 *Taint Sources*

Os *taint sources*, no *TaintJSec*, são métodos ou funções presentes na linguagem *JavaScript*, que fornecem informação sensível. Neste trabalho, a informação sensível é definida como sendo toda informação que cause problemas de invasão de privacidade caso seja transmitida para destinos desconhecidos ou não autorizados. Por exemplo, as funções *getIP* e *getLocation* podem ser consideradas *taint sources*, pois fornecem o endereço *IP* do usuário e as coordenadas geográficas do mesmo.

Logo, é necessário listar todos os possíveis métodos ou funções que forneçam informações sensíveis e informá-los ao *TaintJSec*. Essa informação é armazenada para que sirva de consulta sempre que a análise do fluxo de execução encontrar uma operação de chamada de função ou método, como mostrado no Algoritmo 4.2. Desse modo, sempre que a análise identifica uma operação de chamada de função ou execução de método, a lista armazenada é consultada a fim de identificar se o objeto a ser executado é ou não um *taint source*.

Algoritmo 4.2: Verificação de *taint sources*.

Entrada: (T_{source}, η)
Saída: O valor *taint tag*
/* onde T_{source} é o conjunto de todos os taint sources, e */
/* η é um source */

```
1 início
2 |  se η ∈ T_source então
3 |  |  taint_tag ← 1
4 |  senão
5 |  |  taint_tag ← 0
6 |  fim
7 fim
8 retorna taint_tag
```

Todo objeto que armazena informações provenientes de um *taint source*, tem o valor

de seu *taint tag* alterado para *verdadeiro*, sinalizando que a partir daquele momento o objeto está portando uma informação sensível.

4.3.4 *Taint Sinks*

Os *taint sinks*, no *TaintJSec*, são todos os métodos ou funções presentes na linguagem *JavaScript*, que possibilitam o envio de informação sensível para um destino que esteja fora do domínio da aplicação, onde a aplicação não possua qualquer controle.

Algoritmo 4.3: Verificação de *taint sinks*.

```
1 Função TSINK (Θ, η)
      Saída: O valor booleano que indica se o elemento é um taint sink
      /* onde Θ é o conjunto de todos os taint sinks, e   */
      /* η é um elemento JavaScript                       */
2     início
3         taint_sink ← 0
4         se η ∈ Θ então
5             taint_sink ← 1
6         fim
7     fim
8     retorna taint_sink
```

Da mesma forma que os *taint sources*, é preciso criar o conjunto de todos os *taint sinks* existentes na aplicação. Então, esse conjunto é informado ao *TaintJSec*, que o armazena para que sirva de consulta cada vez que a análise identificar operações de atribuição ou de chamada de função. Essa consulta tem o objetivo de verificar se o objeto que estiver sendo executado é um *taint sink*, como mostrado no Algoritmo 4.3. Caso seja um *taint sink*, o vazamento de informação sensível ocorre se o dado que foi encaminhado ao *sink* for um *tainted data*, como mostra o Algoritmo 4.4.

Algoritmo 4.4: Detecção de vazamento de informação.

Entrada: (Θ, η, α)
Saída: O valor booleano que indica a detecção do vazamento de informação

```
/* onde Θ é o conjunto de todos os taint sinks, e      */
/* η é um elemento JavaScript, e                       */
/* α é um registrador                                  */
```

1 **início**
2 $\quad vazamento \leftarrow 0$
3 $\quad$ **se** $TSINK(\Theta, \eta)$ **então**
4 $\quad\quad vazamento \leftarrow \alpha.taint\_tag$
5 $\quad$ **fim**
6 **fim**
7 **retorna** *vazamento*

Alguns exemplos de *taint sinks* são os objetos *window.open* e *document.location.href*, pois podem enviar o conteúdo de um *tainted data* para domínios fora do controle da aplicação, como exemplifica o Código 4.2.

```
<script>
var dadosUsuario = {ip : getIP(),
                    localizacao : getLocation()}
var d = JSON.stringify(dadosUsuario);
function enviaDados1() {
    window.open( urlExterna + "?get=" + d );
}
function enviaDados2() {
    document.location.href = urlExterna + "?get=" + d;
}
enviaDados1();
enviaDados2();
</script>
```

Código 4.2. Código contendo objetos considerados *taint sinks* pelo *TaintJSec*.

4.3.5 Propagação do *Taint Tag*

Todo valor resultante de uma operação que envolve *tainted data* leva consigo o *taint tag* positivo. Esse valor é repassado a outros objetos de diferentes maneiras, podendo ser por uma simples atribuição de valor entre objetos ou por meio do envio a uma função em forma de parâmetro, por exemplo.

Desse modo, faz-se necessário criar um conjunto de regras de propagação do *taint tag* em meio às diversas operações no fluxo de execução do código *JavaScript*. Essas regras dividem-se em propagação por atribuição (4.3.5.1), propagação por expressão (4.3.5.2), propagação por passagem de parâmetro (4.3.5.3) e propagação por execução de função

nativa (4.3.5.4). Cada uma dessas regras é executada de acordo com o tipo de operação encontrada durante a análise da árvore sintática abstrata.

Assim, para cada tipo de operação *JavaScript* em que uma informação sensível pode ser enviada de um objeto para outro, existe uma regra de tratamento para que essa propagação seja refletida nos registradores. O Quadro 4.1 apresenta cada uma das regras de propagação e seu respectivo conjunto de operações. O modo como a propagação é realizada em cada uma das regras é explicado nas seções seguintes.

Quadro 4.1. Regras de propagação do *taint tag* e seus respectivos conjuntos de operações.

REGRAS	OPERAÇÕES
Atribuição	Expressão de atribuição Declaração de variável
Expressão	Expressão Binária Expressão Lógica
Passagem de parâmetro	Expressão de chamada de função declarada no código *JavaScript* (não nativa)
Função nativa	Expressão de chamada de função nativa. Por exemplo: *Array.push*, *eval* e *parseInt*

4.3.5.1 Propagação por Atribuição

A propagação por atribuição é realizada quando uma variável recebe o valor diretamente de algum *tainted source* ou de um *tainted data*.

Na atribuição, o registrador do objeto que está à esquerda da operação recebe a estrutura do registrador do objeto à direita, sem que haja qualquer tipo de verificação.

A Figura 4.8 exemplifica o acontecimento da propagação por atribução, onde o registrador da variável C (R_C) simplesmente recebe R_B ($R_C = R_B$).

Objetos do tipo vetor (*array*) possuem uma única *tag* para todos os elementos, de tal modo que basta a atribuição de um único *tainted data* para que todos os demais elementos do vetor sejam considerados dados sensíveis.

4.3.5.2 Propagação por Expressão

A propagação por expressão é realizada quando um ou mais elementos de uma expressão possuem *taint tag* positivo, fazendo com que o valor resultante da expressão também o tenha.

Código *JavaScript*	Registradores
A = 5;	R_A = { valor: 5, tainted: 0 }
B = getNome();	R_A = { valor: 5, tainted: 0 } R_B = { valor: "João", tainted: 1 }
C = B;	R_A = { valor: 5, tainted: 0 } R_B = { valor: "João", tainted: 1 } R_C = { valor: "João", tainted: 1 }

Figura 4.8. Exemplo de propagação por atribuição em um trecho de código *JavaScript*.

Durante o processo de cálculo de uma expressão, o valor da *tag* é propagado pela AST dos níveis inferiores para os níveis superiores. Logo, a raiz da AST possuirá a *tag* com o valor resultante de todas as disjunções ($A^{flag} \vee B^{flag}$) entre as tags dos níveis inferiores.

A Figura 4.9 apresenta inicialmente um exemplo de código *JavaScript*, onde um dos elementos da expressão é uma função *getNome*, definida como *tainted source*.

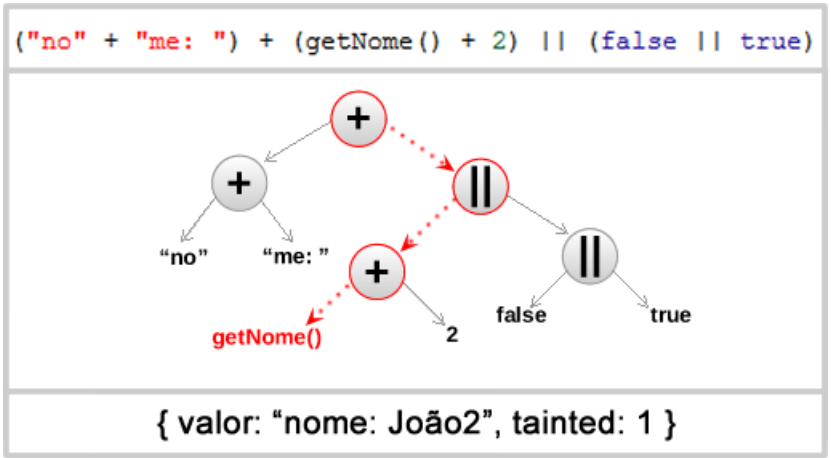


Figura 4.9. Exemplo de propagação por expressão em um trecho de código *JavaScript*.

Em seguida, é apresentada sua árvore de execução, onde o caminho de propagação do *taint tag* para os níveis superiores da árvore é destacado nas linhas vermelhas pontilhadas. Por fim, o objeto resultante da operação é apresentado na parte inferior da figura. Os passos de resolução da árvore são detalhados a seguir.

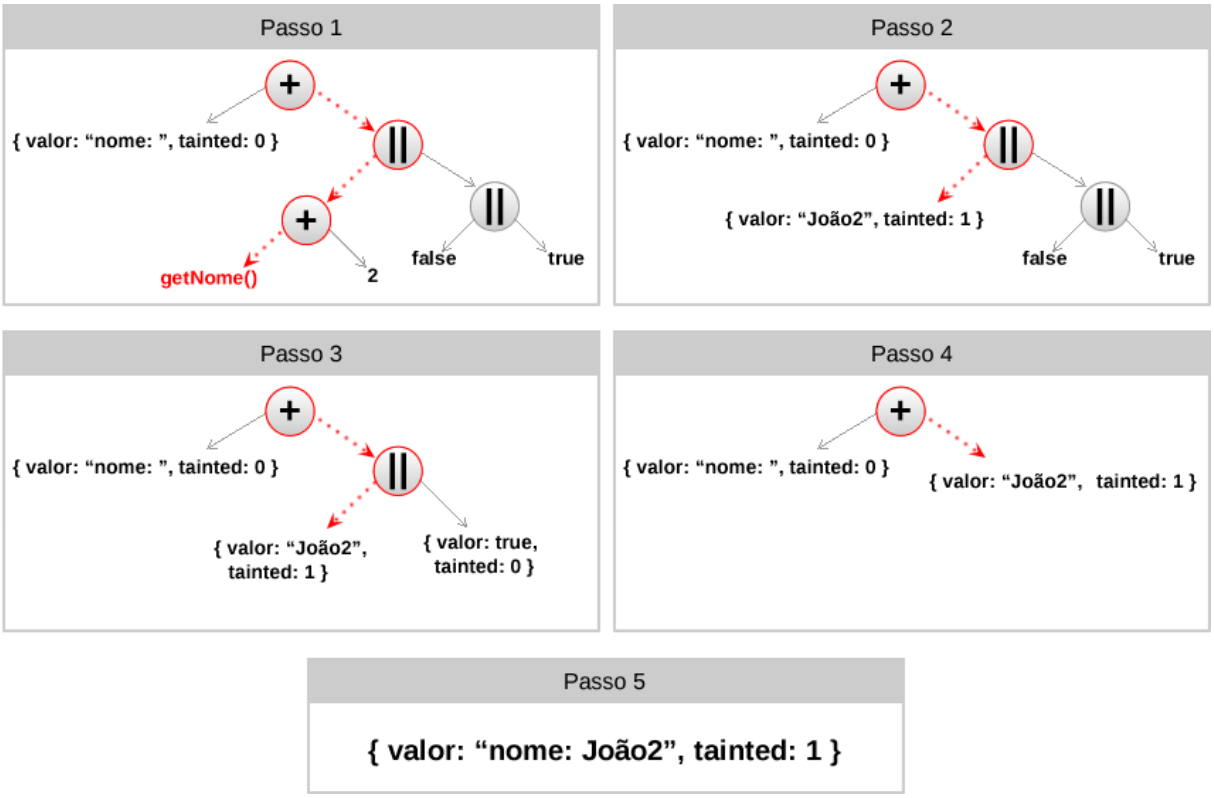


Figura 4.10. Exemplo dos passos da propagação do *taint tag* na AST.

A Figura 4.10 exemplifica como ocorre o processo de propagação do *taint tag* na AST. O processo inicia resolvendo primeiro as folhas da esquerda e, em seguida, as folhas da direita da árvore, como explicado nos passos abaixo:

1. É realizada uma operação de concatenação entre dois objetos literais (*strings*) **“no”** + **“me: ”** resultando em **“nome: ”** que não é um *tainted data*.

2. Ocorre a operação de concatenação entre o valor da função *getNome* e o número **2**. A função *getNome* é um *taint source*, portanto retorna o valor **“João”** marcado com *tag* positiva, indicando que trata-se de uma informação sensível. A operação de concatenação de **“João”** e o número **2**, resulta em **“João2”** e a disjunção $\text{“João”}^{tag} \vee 2^{tag}$ equivale a $(1 \vee 0) = 1$. Portanto, a *tag* resultante é positiva.

3. Ocorre a expressão lógica entre dois valores booleanos, **true || false**, que resulta no valor **true**, que não é um *tainted data*.

4. Ocorre a operação lógica **“João2”** || **true**, que resulta em **“João2”**. Como esse valor é um *tainted data*, então a disjunção $\text{“João2”}^{flag} \vee \text{true}^{flag}$ equivale a $(1 \vee 0) = 1$. Portanto, o *flag* resultante é positivo.

5. Ocorre a operação de concatenação **“nome: ”** + **“João2”**, que tem como resultado o valor literal **“nome: João2”**. A disjunção “nome: ”^{flag} $\vee$ “João2”^{flag} equivale a $(0 \vee 1) = 1$. Portanto, o *flag* resultante é positivo.

4.3.5.3 Propagação por Passagem de Parâmetro

A propagação por passagem de parâmetro ocorre quando pelo menos um dos argumentos enviados como parâmetro a uma função não-nativa[3] possui *taint tag* positivo.

Logo, após a chamada da função e antes da execução das operações de seu respectivo escopo é realizada uma verificação no valor dos parâmetros. No escopo da função, novos registradores recebem as estruturas enviadas pelos argumentos de entrada, de modo que, tratando-se de um *tainted data*, a estrutura enviada leva consigo o valor do *flag* positivo, o qual é recebido pelo registrador interno que, então, propaga-o durante o fluxo de execução.

A Figura 4.11 demonstra como é realizada a substituição das variáveis por seus respectivos registradores, para então realizar a propagação do *flag* no escopo da função.

Código *JavaScript*	Registradores
a = 5;	{ valor: 5, tainted: 0 }
b = getNome();	{ valor: "João", tainted: 1 }

funcao_qualquer (a, b);	{ func: funcao_qualquer,
	parametros: {
	{ valor: 5, tainted: 0 },
	{ valor: "João", tainted: 1 }
	}
	}

Figura 4.11. Exemplo de propagação por passagem de parâmetro em um trecho de código *JavaScript*.

O fato da propagação ocorrer do escopo externo para o escopo interno da função, não significa que o valor retornado por ela será obrigatoriamente um *tainted data*, o *flag* do objeto retornado pode ser positivo ou não, dependendo das operações internas da função.

4.3.5.4 Propagação por Função Nativa

A propagação por função nativa ocorre quando pelo menos um dos argumentos da função é um *tainted data*. Diferentemente da regra de passagem de parâmetro, a qual ocorre em chamadas de funções não-nativas onde o escopo da função é conhecido, a regra de propagação por função nativa não tem acesso ao escopo da função executada.

[3] Toda função declarada explicitamente no código *JavaScript* não é uma função nativa.

Para detectar o *tainted data* é realizada uma verificação nos argumentos enviados como parâmetros à função e, se pelo menos um deles contiver o atributo *taint tag* com valor positivo, então o valor do atributo *taint tag* do objeto retornado pela função também é considerado positivo. A Figura 4.12 mostra um exemplo de uma função nativa do objeto *window*, que converte uma *string* em sua representação na base 64, a propagação ocorre exclusivamente por tratar-se de um argumento considerado *tainted data*, uma vez que a análise não abrange as operações do escopo de funções nativas.

Código *JavaScript*	Registradores
a = getCPF();	{ valor: "69871267554", tainted: 1 }
b = btoa(a);	{ valor: "Njk4NzEyNjc1NTQ=", tainted: 1 }

Figura 4.12. Exemplo de propagação em função nativa.

Essa verificação de argumentos é realizada em todas as funções nativas, com exceção da função *eval*, a qual pode retornar objetos *tainted data* ou não, independentemente se o parâmetro é um *tainted data*. O valor retornado pela função *eval* depende exclusivamente da lógica contida em seu parâmetro literal.

4.3.5.5 Propagação na Função Eval (Caso especial de propagação por função nativa)

A função *eval* é uma função nativa utilizada para criar novos trechos de código em tempo de execução. Como exemplificado anteriormente no Código 1.1, muitos trabalhos que utilizam a análise estática em código *JavaScript* não oferecem suporte à função *eval* em razão das dificuldades encontradas durante a análise de sua execução. Como resultado, a função *eval* torna-se um problema por ser bastante utilizada em aplicações maliciosas que praticam o vazamento de dados sensíveis.

Para resolver esse problema, este trabalho propõe um novo conceito chamado *taint position*.

Definição 1: *Taint Position* é o nome dado ao número que identifica a posição de um *tainted data* contido em um valor literal.

Um *taint position* é criado após uma operação de concatenação entre dois objetos em que pelo menos um deles é um *tainted data*. O resultado dessa operação é armazenado no registrador de objetos junto com o *taint tag* positivo e um conjunto de *taint position* que indica a posição exata de um ou mais *tainted data* existentes no valor literal.

O conjunto de *taint position* é formado pela união do conjunto de *taint position* do elemento da esquerda com o conjunto do elemento da direita. Essa operação de união de conjuntos obedece às regras apresentadas abaixo:

(R1) O elemento da operação que não for *tainted data* fornece um conjunto de *taint position* vazio;

(R2) O *tainted data* do lado esquerdo da operação fornece um conjunto unitário contendo o *taint position* 1, quando este não possuir um conjunto de *taint position* no registrador;

(R3) O *tainted data* do lado esquerdo da operação fornece o conjunto de *taint position* de seu registrador, caso possua;

(R4) O *tainted data* do lado direito da operação fornece um conjunto unitário contendo o número resultante da operação (1 + *comprimento do elemento da esquerda*), quando não possuir um conjunto de *taint position* em seu registrador;

(R5) O *tainted data* do lado direito da operação que possui um conjunto X de *taint position* em seu registrador, fornece o conjunto imagem I(f), onde $f : X \to Y$ e $f(x) = x +$ *comprimento do elemento da esquerda.*

O Algoritmo 4.5 apresenta como as cinco regras acima são utilizadas para fornecer um conjunto de *taint position* resultante de uma operação de concatenação entre dois objetos.

Algoritmo 4.5: Criação do conjunto de *taint position* resultante de uma operação de concatenação entre dois objetos.

Entrada: (Reg^{esq}, Reg^{dir})
Saída: Conjunto de *taint position*
/* onde Reg^{esq} é o registrador do objeto da esquerda, e */
/* Reg^{dir} é o registrador do objeto da direita */

```
1  início
2      conj^esq ← { } ;                                              // (R1)
3      conj^dir ← { } ;                                              // (R1)
4
5      se Reg^esq.taint_flag é positivo então
6          se Reg^esq.taint_position existe então
7              conj^esq ← Reg^esq.taint_position ;                   // (R3)
8          senão
9              conj^esq ← { 1 } ;                                    // (R2)
10         fim
11     fim
12     se Reg^dir.taint_flag é positivo então
13         se Reg^dir.taint_position existe então
14             para cada i ∈ Reg^dir.taint_position faça
15                 conj^dir ← conj^dir ∪ { i + TAMANHO(Reg^esq.valor) } ;   // (R5)
16             fim
17         senão
18             conj^dir ← { 1 + TAMANHO(Reg^esq.valor) } ;           // (R4)
19         fim
20     fim
21 fim
22 S ← conj^esq ∪ conj^dir
23 retorna S
```

A Figura 4.13 exemplifica como ocorre a criação de um *taint position*, onde é possível observar que o código **var b = "texto" + a;** resulta em um *tainted data* contendo um conjunto de *taint position* com apenas um elemento, sendo ele o *taint position* 6. Isso ocorre porque a variável **a** é um *tainted data*, cujo valor inicia a partir do sexto caractere da literal resultante "texto127.0.0.1" da operação de concatenação (conforme R4).

O código **var c = a + b;** da Figura 4.13, apresenta uma operação de concatenação entre dois *tainted data* (**a** e **b**) que resulta em um *tainted data* com novos *taint positions*. Neste caso, o conjunto de *taint position* é obtido a partir da união do conjunto de *taint position* do elemento da esquerda da operação (variável **a**) com o conjunto de *taint position* do elemento da direita da operação (variável **b**). O elemento da esquerda obedece R2 e fornece o conjunto {1}, enquanto o elemento da direita, cujo registrador contém o *taint position* 6, obedece a R5 e fornece o conjunto {15}, pois **6 + 9 = 15**, onde o número 9 equivale ao comprimento (*length*) do valor de **a**. Como resultado, o conjunto de *taint*

Código *JavaScript*	Registradores
var a = getIP();	R_a = { valor: "127.0.0.1", tainted: 1 }
var b = "texto" + a;	R_b = { valor: "texto127.0.0.1", tainted: 1, taint_position: [6] }
var c = a + b;	R_c = { valor: "127.0.0.1texto127.0.0.1", tainted: 1, taint_position: [1,15] }

Figura 4.13. Exemplo de criação de *tainted position*.

position após a operação de concatenação é {1,15}.

Assim, a utilização de *taint position* como solução para o problema da função *eval* mostra-se eficaz, pois é possível propagar essa informação para o escopo da função *eval* e informar ao analisador a posição dos *tainted data* presentes no argumento da função.

No entanto, a obtenção do conjunto de *taint position* deve ocorrer durante o processo de reescrita do parâmetro de entrada da função *eval*. Esse processo consiste em reescrever o argumento enviado à função *eval*, da esquerda para a direita, substituindo todos os identificadores de variáveis pelos seus respectivos valores. Para cada variável encontrada, seu registrador é consultado para checar o valor do *taint tag*. Quando um *tainted data* é encontrado, o conjunto de *taint position* do elemento da direita da concatenação é obtido conforme as regras estabelecidas para a criação do conjunto de *taint position*, e serve de consulta durante a análise da execução da função *eval* (Figura 4.14). Esse processo ocorre até que não existam mais variáveis a serem substituídas, sobrando apenas uma única *string*.

Argumento Original	Argumento Reescrito	Taint Position
`eval("nome=\""+getNome()+"\"")`	`eval("nome=\"João\"")` 1 2 3 4 5 6 7 8 9 10 11	7

Figura 4.14. Exemplo do processo de reescrita do argumento da função *eval* e identificação dos *tainted positions*.

Com o argumento reescrito e os *tainted positions* conhecidos, é criada a árvore sintática abstrata do argumento, contendo o intervalo do código-fonte que originou cada nó da árvore. Todo nó da AST que contém um intervalo de código-fonte onde pelo menos um dos elementos do conjunto de *tainted position* está inserido resulta em um *tainted data*, cujo *flag* é propagado para os níveis superiores da árvore.

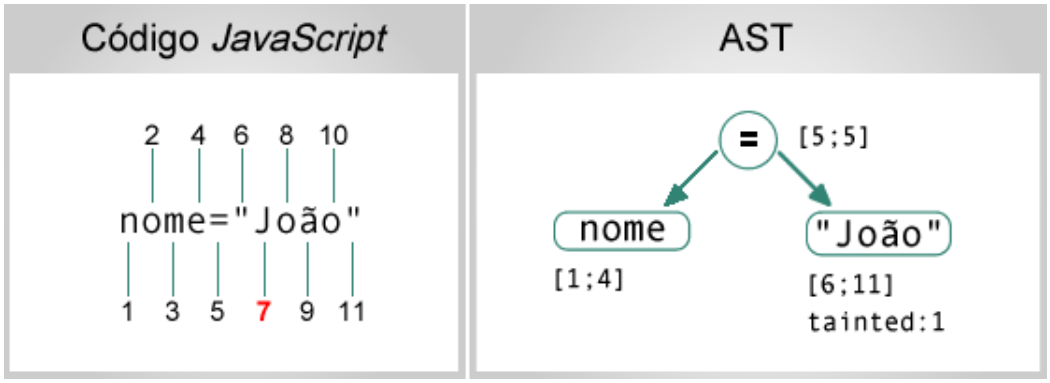


Figura 4.15. Exemplo de AST com os atributos de intervalo de código-fonte e o *taint tag* atribuído devido a existência de um *tainted position* no intervalo.

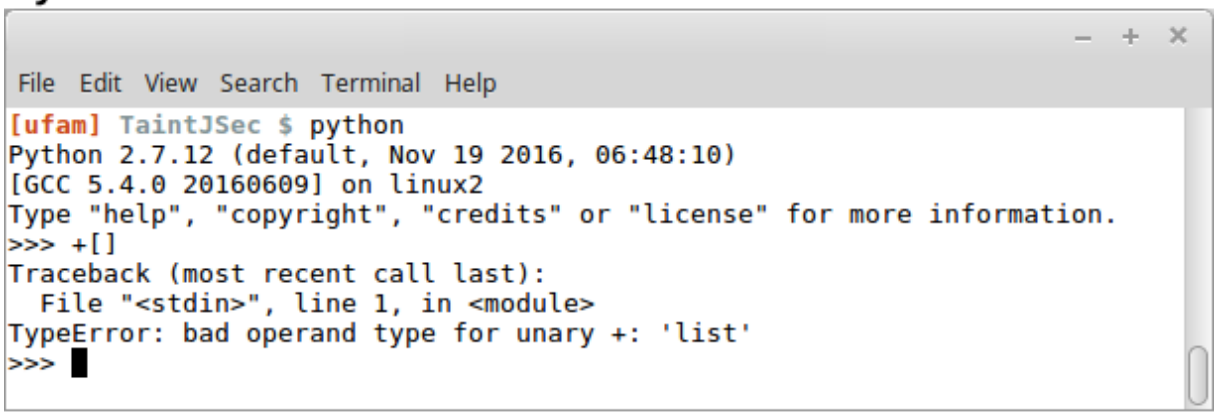
A Figura 4.15 apresenta a AST do argumento reescrito na Figura 4.14, cuja operação de atribuição resulta em um *tainted data*, pois o *tainted position* 7 está presente no intervalo de 6 a 11, atributo do nó à direita da árvore.

4.4 Implementação

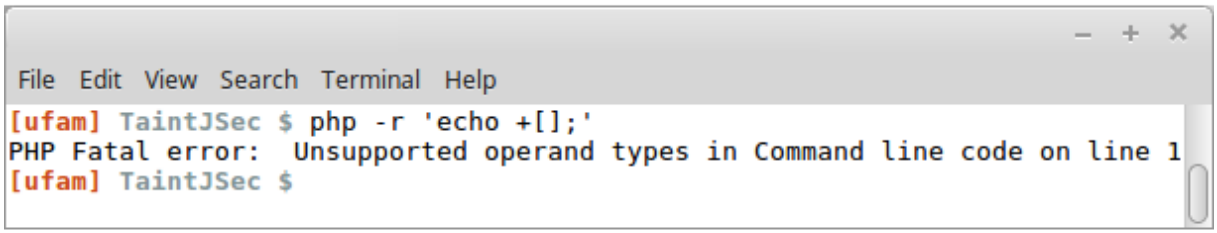
O *TaintJSec* foi implementado sob a plataforma Node.js, com o objetivo de aproveitar os recursos do microprocessador, tais como as funções nativas, tipagem de dados e toda dinamicidade disponível na linguagem.

Assim, recursos exclusivos da linguagem *JavaScript* não precisam ser implementados para que sejam reconhecidos pelo *TaintJSec*. A Figura 4.16, por exemplo, ilustra o resultado da expressão unária +[] sendo executada no *shell* do *Python*, PHP e *Node.js*, onde é possível observar que somente este último reconheceu a expressão.

Python



PHP



Node.js

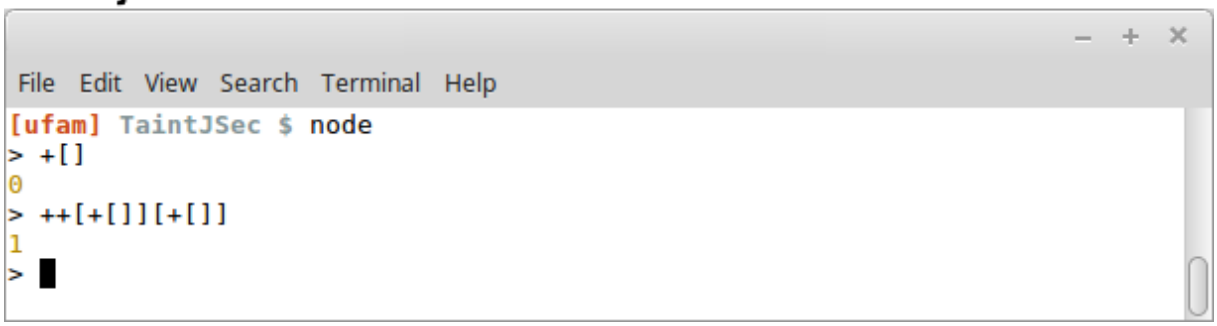


Figura 4.16. Testes de execução da expressão unária +[] no *shell* do *Python*, PHP e *Node.js*.

Além disso, a plataforma possibilita a utilização de módulos que exercem função de

apoio semelhante às bibliotecas *JavaScript*, os quais facilitam o desenvolvimento das fases da abordagem.

4.4.1 Primeira Fase: Identificação e Extração de Código

Na primeira fase da abordagem (seção 4.1) foi utilizado o módulo jsDom [69] para carregar o código HTML da página inicial da aplicação e gerar seu respectivo DOM. Assim, as buscas por trechos de código *JavaScript* foram realizadas no DOM em vez do código HTML.

O jsDom é um módulo do *Node.js* que possibilita acessar o DOM (*Document Object Model*) de uma estrutura HTML pelo lado do servidor. O jsDom carrega uma estrutura HTML e gera seu respectivo DOM da mesma forma que um navegador convencional geraria pelo lado do cliente. Também permite a adição de bibliotecas *JavaScript* para manipulação dos objetos do código HTML, como exemplifica o Código 4.3.

```
1 var jsdom = require("node-jsdom");
2 jsdom.env(
3   '<p><a class="the-link" href="https://github.com/tmpvar/jsdom">jsdom!</a></p>'
      ,
4   ["http://code.jquery.com/jquery.js"],
5   function (errors, window) {
6     console.log("contents of a.the-link:", window.$("a.the-link").text());
7   });
```

Código 4.3. Exemplo de uso do módulo jsDom, utilizando a biblioteca *jQuery* para manipular objetos do código HTML.

No exemplo apresentado pelo Código 4.3, o módulo jsDom é carregado no *Node.js* na linha 1 e atribuído à variável **jsdom**. O módulo inicia o carregamento do código HTML utilizando a função *env()*, como apresentado na linha 2. As linhas 3 e 4 apresentam os argumentos enviados à função *env()*, onde código HTML é o primeiro argumento e o segundo argumento é o endereço da biblioteca *jQuery* [70]. Assim, o módulo jsDom gera o DOM da estrutura HTML fornecida e permite que os elementos da estrutura possam ser manipulados pela biblioteca *jQuery*. A linha 6 apresenta seletor da biblioteca *jQuery*, **window.$**, que captura o texto presente entre as *tags* <*a*> e </*a*>. O resultado da execução do Código 4.3 apresenta a frase: "*contents of a.the-link: jsdom!*".

A biblioteca *jQuery* foi adicionada ao jsDom para facilitar a busca pelas *tags script* explicitamente declaradas, como mostra o Código 4.4.

```
1 function fnc_extrair_scripts(){
2   $("script").each(
3     function(){
4       var obj = $(this);
5       var src = obj.attr("src")?
6             obj.attr("src").trim():
```

```
7             null;
8       if (src!=null && src.length>0)
9         fnc_get_source_js(obj,1); // externo ...
10      else
11        fnc_get_source_js(obj,2); // inline ...
12    }
13  );
14}
```

Código 4.4. Função de extração de trechos scripts utilizando *jQuery*.

O seletor *$("script")* fornece o conjunto de *tags script* válidas, presente na página HTML previamente carregada, onde cada elemento do conjunto tem seu respectivo código *JavaScript* extraído pela função *fnc_get_source_js* e, posteriormente, concatenado ao código do elemento predecessor. Esse processo tem como resultado um único código *JavaScript* formado pela união de todos os demais fragmentos de código, e organizado de acordo com a ordem de leitura do HTML da página.

4.4.2 Segunda Fase: Árvore Sintática Abstrata

Na segunda fase da abordagem (seção 4.2) foi utilizado o módulo Esprima [53] para criar a árvore sintática abstrata do código gerado na fase anterior e, também para capturar trechos do código original a partir da AST. Além disso, esse módulo é utilizado para criar a AST de trechos de código *JavaScript* enviados como parâmetro para a função *eval*, onde são inseridos os *tainted positions*.

Por padrão, o módulo fornece uma AST minimalista, sem informação suficiente para que o analisador estático possa detectar vazamento de informação em códigos dinâmicos, como a função *eval*, por exemplo. No entanto, o Esprima possui *flags* de configuração, que servem para ativar recursos extras na AST gerada. Essas *flags* são apresentadas na Tabela 4.1.

Tabela 4.1. *Flags* de configuração do módulo Esprima.

Nome	Tipo	Padrão
jsx	Booleano	falso
range	Booleano	falso
loc	Booleano	falso
tolerant	Booleano	falso
tokens	Booleano	falso
comment	Booleano	falso

Como é possível observar na Tabela 4.1, todas as *flags* de configuração estão desativadas por padrão. Cada uma delas é responsável por inserir novas informações

na árvore sintática abstrata ou alterar o comportamento do Esprima durante a criação da AST. Cada uma dessas opções de configuração são apresentadas a seguir.

- **jsx** - É a *flag* que ativa o suporte à sintaxe JSX[4], que é uma extensão da sintaxe do *JavaScript* visualmente parecida com XML e HTML. O Código 4.5 apresenta um exemplo de um código *JavaScript* que utiliza a sintaxe JSX.

```
1 var mensagem = <div>
2   <p>Seja Bem-vindo</p>
3 </div>;
```

Código 4.5. Exemplo de código *JavaScript* com sintaxe JSX.

Caso a *flag* **jsx** não estivesse ativada no módulo Esprima, o código acima não seria suportado e ocorreriam erros durante o processo de criação da AST. Como essa sintaxe não faz parte das especificações oficiais do *ECMAScript*, então permaneceu desativada, pois não faz parte do escopo de pesquisa deste trabalho.

- **range** - Insere um atributo em cada nó da AST gerada, que informa a posição do código *JavaScript* que foi responsável por gerá-lo. Assim, é possível localizar por meio do atributo *range* o trecho de código contido em cada nó da AST, como exemplificado na Figura 4.17.

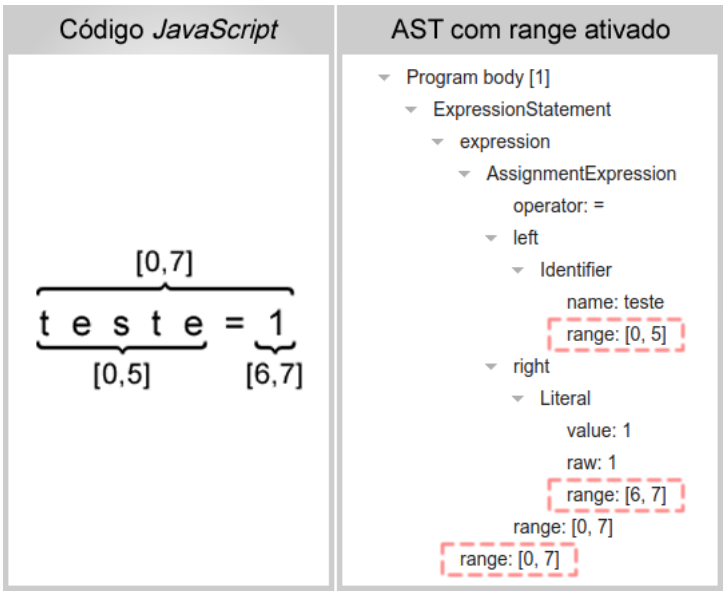


Figura 4.17. Exemplo de AST contendo o atributo *range*.

A Figura 4.17 mostra uma AST que contém o atributo *range* em cada um de seus nós, informando o intervalo do código *JavaScript* que eles representam.

[4] https://facebook.github.io/jsx/

Essa informação é útil para que o *TaintJSec* possa identificar trechos de informação sensível contidos em valores literais, aplicando a técnica de *taint position*. Por isso, essa *flag* foi ativada no Esprima.

- **loc** - Semelhante à *flag* **range**, essa opção também fornece a localização do trecho de código responsável por gerar os nós da árvore, mas a posição do trecho de código é informada pelo número da linha e da coluna.

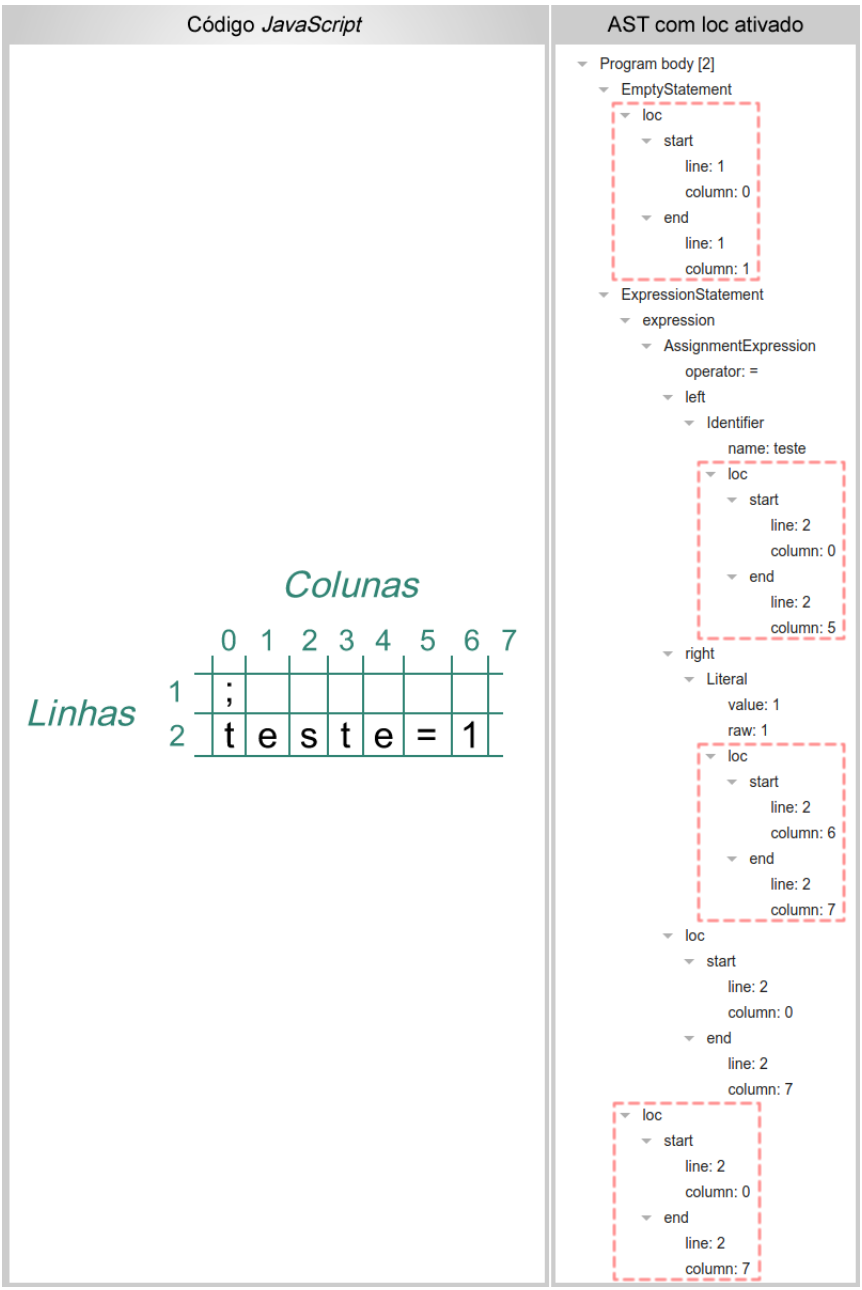


Figura 4.18. Exemplo de AST contendo o atributo *loc*.

Como essa *flag* é semelhante à **range**, diferenciando apenas por fornecer um meio diferente de encontrar a posição do código *JavaScript*, então ela não foi ativada.

- **tolerant** - Tolera alguns casos de erros de sintaxe. Com essa *flag* ativada, o módulo Esprima consegue gerar a AST mesmo que o código *JavaScript* apresente alguns erro de sintaxe, como por exemplo o uso de variáveis não declaradas em escopos que executam em modo estrito. Essa *flag* não foi ativada para uso no *TaintJSec*, pois o modo estrito não faz parte do escopo de pesquisa do *TaintJSec* por reduzir a versatilidade da linguagem, em especial a função *eval*.
- **tokens** - Coleta todos os *tokens* utilizados para gerar a AST. Esses *tokens* são utilizados pelo Esprima para criar a árvore sintática abstrata e, por padrão, não são apresentados na estrutura da árvore. Essa *flag* permaneceu desativada no *TaintJSec*.
- **comment** - Coleta todos os comentários de linha e de bloco. Por padrão, essa *flag* está desativada e o Esprima não insere na árvore os nós que representam os comentários presentes no código *JavaScript*. Essa *flag* permaneceu desativada no *TaintJSec*, pois o importante para este trabalho é manter os comentários no código *JavaScript* agrupado, que foi gerado na fase de identificação e extração de código, mas os comentários não precisam estar na AST.

Tabela 4.2. Configuração do módulo Esprima para a criação da AST utilizada no *TaintJSec*.

Nome	Valor
jsx	false
range	**true**
loc	false
tolerant	false
tokens	false
comment	false

Para obter uma AST satisfatória, com informações suficientes para que o analisador estático pudesse utilizá-la neste trabalho, foi preciso ativar o *flag* **range** e deixar os demais *flags* desativados, como apresentado na Tabela 4.2.

4.4.3 Terceira Fase: Análise do Fluxo da Informação

Na terceira fase da abordagem (seção 4.3) foi criada uma rotina de verificação para cada nó da AST, a qual executa uma série de operações de acordo com o tipo de nó informado pelo módulo Esprima (Quadro 4.2). Essas operações visam manter a correta computação e observância dos dados, adaptando-se de acordo com o tipo de interação entre os registradores.

O Quadro 4.2 apresenta todos os tipos de nós possíveis que a AST pode fornecer. Essa nomenclatura é fornecida pelo Esprima e cada nó da AST possui um atributo **type** que indica o seu tipo. Para cada tipo de nó, o analisador estático executa um conjunto de operações específicas, que servem para simular estaticamente o comportamente do código *JavaScript*.

Quadro 4.2. Tipos de operações de acordo com a nomenclatura do módulo Esprima.

Declarações	Expressões e Padrões
BlockStatement	ArrayExpression
BreakStatement	ArrowFunctionExpression
ContinueStatement	AssignmentExpression
DebuggerStatement	AwaitExpression
DoWhileStatement	BinaryExpression
EmptyStatement	CallExpression
ExpressionStatement	ClassExpression
ForStatement	ConditionalExpression
ForInStatement	FunctionExpression
ForOfStatement	Identifier
FunctionDeclaration	Literal
IfStatement	LogicalExpression
LabeledStatement	MemberExpression
ReturnStatement	MetaProperty
SwitchStatement	NewExpression
ThrowStatement	ObjectExpression
TryStatement	SequenceExpression
VariableDeclaration	Super
WhileStatement	TaggedTemplateExpression
WithStatement	ThisExpression
	UnaryExpression
	UpdateExpression
	YieldExpression

Para os nós que dependem da resolução de níveis inferiores da árvore, foi criada uma estrutura de pilha para armazená-los respeitando a ordem de interdependência entre os mesmos, de modo que as folhas da AST ficassem no topo da estrutura. Assim, cada nível da pilha representa um nó da AST, e conforme o nó presente no topo da pilha é desempilhado, o resultado de suas operações é propagado para o nível subsequente, satisfazendo a interdependência entre os nós.

A Figura 4.19 apresenta a árvore sintática abstrata do código **1==2?true:false** e ilustra os passos de movimentação da pilha durante a análise.

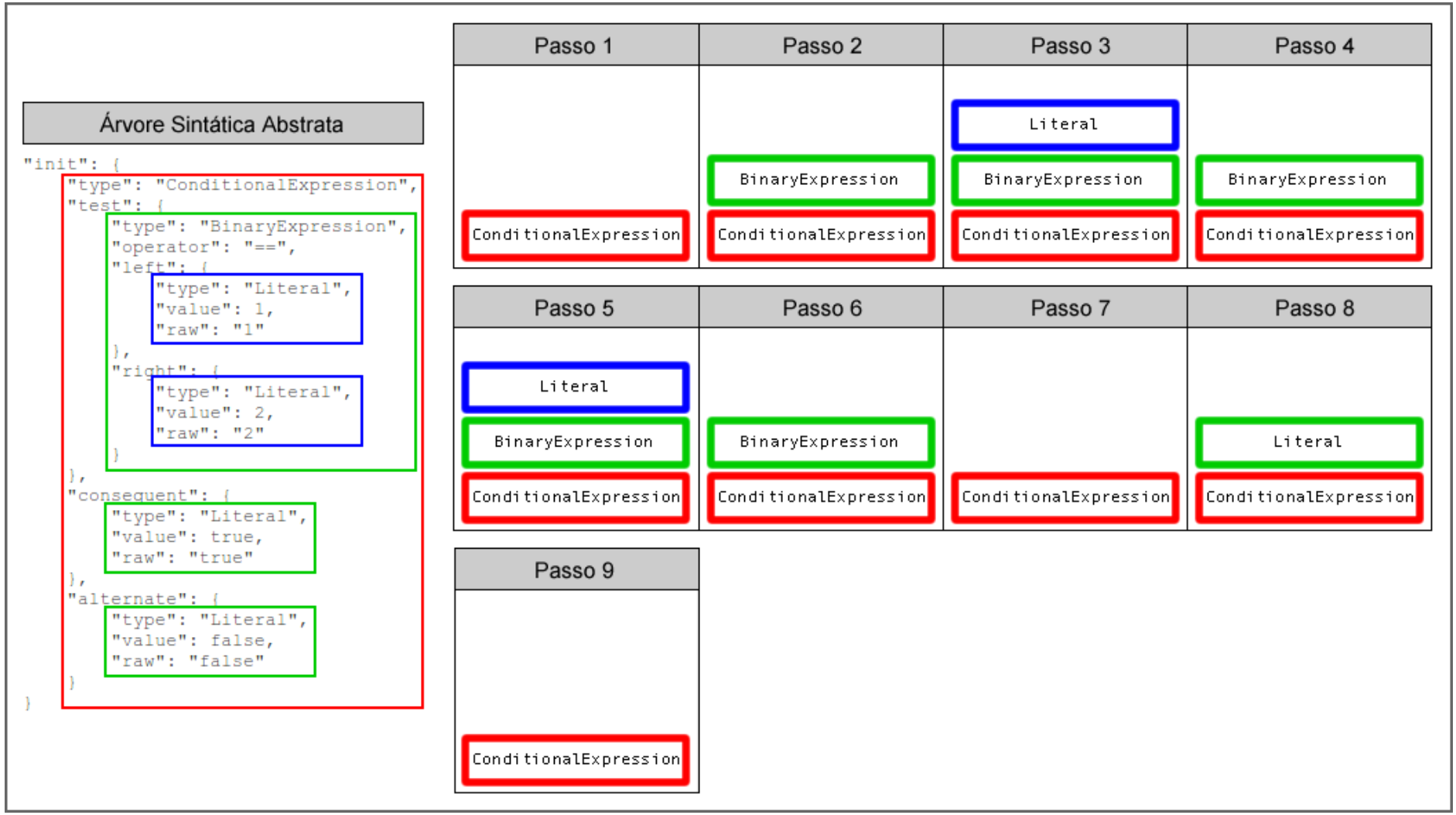


Figura 4.19. Exemplo do uso da pilha durante a análise.

No exemplo da Figura 4.19 o nó raiz da AST é do tipo *ConditionalExpression* (Expressão Condicional) e possui primeiramente uma fase de teste. Porém, no valor do atributo *test* existe um nó do tipo *BinaryExpression* que precisa ser calculado antes, já este por sua vez, depende da resolução dos atributos *left* e *right* que possuem ambos nós do tipo *Literal*. Os passos de movimentação da pilha são explicados abaixo:

1. A base da pilha. O nó do tipo *ConditionalExpression* é armazenado para aguardar a resolução do nó que contém o valor do atributo *test*.

2. O nó do tipo *BinaryExpression* (expressão binária) é empilhado, porém ele também precisa da resolução de outro nó para obter o valor do atributo *left*.

3. O nó do tipo *Literal* é empilhado e, por ser uma folha da AST e não depender de nenhum outro nó para que possa ser analisado, ele é calculado de imediato, retornando o valor de número inteiro 1 para o nó inferior, sendo desempilhado logo em seguida.

4. O valor do atributo *left* é obtido, porém é preciso obter o valor do atributo *right*.

5. O nó do tipo *Literal* é empilhado e, por ser uma folha da AST e não depender de nenhum outro nó para que possa ser analisado, ele é calculado de imediato, retornando o valor de número inteiro 2 para o nó inferior, sendo desempilhado logo em seguida.

6. As dependências do nó *BinaryExpression* foram satisfeitas e os valores dos atributos *left* e *right* foram obtidos. O teste (**1 == 2**) é realizado e seu resultado é enviado à base da pilha. O nó *BinaryExpression* é desempilhado.

7. Como o teste falhou, o atributo *consequent* não é verificado, mas em seu lugar o atributo *alternate*. Porém, para obter o valor de *alternate* é preciso analisar o nó do tipo *Literal* antes.

8. O nó do tipo *Literal* é empilhado e, por ser uma folha da árvore e não depender de nenhum outro nó para que possa ser analisado, ele é calculado de imediato, retornando o valor *booleano false* para o nó inferior e sendo desempilhado logo em seguida.

9. A base da pilha recebe o valor *false* e desempilha. Finalizando o processo da análise da AST.

O Algoritmo 4.6 apresenta a lógica utilizada para identificar o tipo do nó, assim como realizar operações específicas para cada um deles. Esse processo repete-se consumindo as

Algoritmo 4.6: Identificação e tratamento dos diferentes tipos de operações contidas em cada nó da AST.

```
1  Função COMPT (Δ, μ)
       Saída: A estrutura de um registrador contendo o valor resultante da
              operação atual
       /* onde Δ é a pilha, e                                          */
       /* μ é o nível atual da pilha                                   */
2      início
3          Noh ← Δ[μ]
4          reg ← {}
5          selecione Noh.tipo faça
6              caso 1 faça
7                  reg ← tracking_tipo_1;
8                  break;
9              fim
10             caso 2 faça
11                 reg ← tracking_tipo_2;
12                 break;
13             fim
14             ...
15             caso n faça
16                 reg ← tracking_tipo_n;
17             fim
18         fim
19     fim
20     retorna reg
```

folhas da árvore e propagando o resultado aos níveis superiores até chegar à sua raiz, caso nenhum vazamento de informação tenha sido identificado.

Para cada tipo de operação *JavaScript* especificada na AST foi criado um código para simular estaticamente o mesmo comportamento que a operação teria caso fosse executada dinamicamente.

Tainted Sources e *Tainted Sinks*

Os *tainted sources* e os *tainted sinks* foram organizados em estruturas de vetores, cujas posições foram preenchidas pela lista de funções ou métodos sensíveis.

Essa estrutura passou a ser consultada todas as vezes em que a árvore sintática abstrata apresentava um nó do tipo *MemberExpression* ou *CallExpression*, pois esses são os tipos responsáveis por identificar o nomes de métodos ou funções nativos, os quais podem fornecer dados sensíveis ou enviá-los para fontes não confiáveis.

Registradores

Para os registradores de objetos foi criada uma classe, a qual possuia três atributos: o valor, o *taint tag* e um atributo específico para objetos construídos a partir de instâncias de funções, quando essas possuem seus próprios atributos.

Cada registrador é uma instância dessa classe.

```
1 function Objeto(valor,tainted,varobj){
2   this.valor   = valor;
3   this.tainted = tainted==undefined ||
4                  tainted==null ? 0 : tainted;
5   this.varobj  = varobj==undefined ||
6                  varobj==null ? [] : varobj;
7 }
```

Código 4.6. Classe de registradores de objetos.

Escopos

Os escopos foram organizados em uma estrutura vetorial. Seu conteúdo é composto por um objeto com cinco atributos, sendo eles: o nome do escopo, o vetor de registradores de objetos que pertencem a esse escopo, os vetor de atributos, o vetor de variáveis de objetos e o vetor de construtores de funções declaradas no escopo.

A paridade entre os escopos é controlada pelo identificador de posição *id* de tal maneira que, dois escopos com identificadores **1.1** e **1.2**, por exemplo, são escopos adjacentes e também subescopos de um terceiro que possui identificador igual a **1**.

```
1 function fnc_cria_escopo(id, nome_escopo){
2   _escopos[id] =
3     {  nome: nome_escopo, // nome do escopo
4        objetos   :[],     // objetos do escopo
5        _this     :[],     // atributos do objeto
6        _varobj   :[],     // variáveis do objeto
7        _prototype:[]    // função utilizada para instanciar objetos
8      };
```

```
9 }
```

Código 4.7. Função para criar escopos.

Ao iniciar a análise de uma função explicitamente declarada, a função *fnc_cria_escopo* é invocada para criar o escopo da função a ser analisada. Esse mesmo escopo é destruído logo que a análise é finalizada.

Funções

As funções explicitamente declaradas no código *JavaScript* foram armazenadas na estrutura de registradores de objetos.

O registrador de uma função contém sua respectiva AST, informada pelo módulo Esprima, cuja raiz é formada com um nó do tipo *FunctionDeclaration*. Assim, quando o *TaintJSec* identifica a chamada de uma função não-nativa, um novo escopo é criado e a AST da função é fornecida pelo seu registrador. Ao final da análise, o escopo da função é destruído e, se a função possuir qualquer valor de retorno, este é enviado por meio de outro registrador.

Os construtores de funções não-nativas foram armazenados no atributo vetorial *_prototype* presente no escopo da declaração. Esse atributo tem a finalidade de fornecer o construtor para que seja utilizado em operações do tipo *NewExpression*, de tal maneira que as instâncias de uma função pudessem receber novos objetos inseridos no escopo da função instanciada por meio do *Prototype*, como exemplifica o Código 4.8.

```
1 function fnc_pessoa() {
2   this.primeiroNome = "Fulano";
3 }
4 var alguem = new fnc_pessoa();
5 fnc_pessoa.prototype.ultimoNome = "de Tal";
```

Código 4.8. Exemplo da utilização do *prototype* para inserir novos atributos em uma função/classe.

A linha 4 do Código 4.8 apresenta um exemplo de operação do tipo *NewExpression*, onde uma nova instância da função **fnc_pessoa** é criada. Nesse exemplo, o *TaintJSec* não executa a função **fnc_pessoa**, mas utiliza o seu construtor armazenado no registrador _escopos[1]._prototype[“fnc_pessoa”].

O construtor de uma função é obtido a partir da extração da declaração da função, no código *JavaScript* original (artefato de saída da primeira fase da abordagem). A posição exata do corpo da função consta no atributo *range* do nó *FunctionDeclaration* da AST. Foi utilizada uma *sandbox* para capturar todas as variáveis globais e dos escopos superiores para então utilizar o módulo *VM* [71] para compilar o código da função utilizando o

método vm.runInNewContext(code[, sandbox][, options]). Essa compilação não executa a função, mas fornece o seu construtor, o qual é armazenado no atributo *__prototype* do escopo em que a função foi declarada.

4.5 Considerações Finais

Este capítulo apresentou o *TaintJSec*, uma diferente forma de realizar análise estática de marcação em código *JavaScript* para prever o vazamento de informação.

Foram apresentadas as três fases da abordagem e cada uma foi explicada detalhadamente, sendo elas as seguintes fases: identificação e extração de código; criação da árvore sintática abstrata; e, análise do fluxo de informação.

Na primeira fase, a forma como é feita a identificação e a extração de código foi exemplificada. Foram apresentados pontos importantes para serem observados durante a fase de identificação de código para evitar a extração de falsos códigos *JavaScript*.

A segunda fase do *TaintJSec* informou os diferentes trabalhos no campo de análise de código que usam árvores sintáticas como um código intermediário e mostrou como AST é útil para o *TaintJSec*. A estrutura da árvore sintática e as informações que ela deve conter para que seja utilizada no analisador estático durante o processo de identificação de vazamento de informação foram abordadas.

Na terceira fase da abordagem foram apresentadas as estruturas dos registradores de objetos, utilizados para armazenar variáveis, funções, escopos e demais objetos do código *JavaScript*. Os algoritmos usados para manipular os objetos, verificar *taint sources*, *taint sinks* e detectar o vazamento de informação foram apresentados.

Os tipos de propagação do *taint tag* foram abordados, sendo eles: atribuição, expressão, passagem de parâmetro e função nativa.

Para realizar a análise estática na função nativa *eval* foi preciso criar um novo conceito denominado *taint position*, que é responsável por informar a posição de um *taint data* presente em um valor literal. O algoritmo usado para criar o conjunto de *taint position* foi apresentado junto com as cinco regras que a técnica utiliza para atualizar a posição dos dados sensíveis.

Este capítulo também apresentou como ocorreu a implementação do *TaintJSec* em cada uma das fases da análise, mostrando códigos e configurações utilizadas neste trabalho.

Capítulo 5

Testes e Resultados

Neste capítulo são apresentados os testes realizados para validar a eficácia do *TaintJSec* na detecção do vazamento de informação sensível. Este capítulo inicia com a apresentação do protocolo experimental, onde são descritos os passos realizados para a criação dos experimentos. Em seguida, são apresentados os testes e seus respectivos resultados.

5.1 Protocolo Experimental

Para montar a base de testes foram obtidos códigos *JavaScript* de quatro fontes distintas, sendo elas: códigos aleatórios gerados pela ferramenta *Eslump* [72]; códigos obtidos na base de testes do *framework* SAFE[1]; códigos da base de testes do *JSPrime*[2]; e, códigos extraídos de artigos publicados.

A utilização de uma ferramenta de geração automática e aleatória de códigos *JavaScript*, como a *Eslump*, foi motivada pela necessidade de diminuir a participação humana durante a criação de códigos para os testes. Dessa maneira, qualquer limitação humana para construir códigos *JavaScript* foi eliminada e os códigos gerados pela ferramenta mostram-se imparciais para a realização dos testes. Os códigos foram criados utilizando o comando **eslump −−source-type script** e somente aqueles validados pela função **esvalidate**[3] foram extraídos.

Os códigos utilizados para testar o *framework* SAFE [58, 60] e o *JSPrime* foram inseridos na base de testes do *TaintJSec* para que fossem usados para validar as operações realizadas pelo analisador estático.

Artigos publicados por Jihyeok Park *et al.* [59] e Alireza Gorji *et al.* [73] apresentam códigos *JavaScript* que os autores descreveram como difíceis de serem analisados por

[1] https://github.com/sukyoung/safe/tree/master/tests
[2] https://docs.google.com/document/d/17J2h43WbPX3sNTIjxr4GhzEGxKy6hvQJqedd3UQ11mY
[3] https://www.npmjs.com/package/esvalidate

trabalhos que usavam análise estática para detectar o vazamento de informação, como por exemplo, o Código 5.1, extraído do trabalho de Kannan [74].

```
1 a[secret] = 1;
2 for (int i = 0; i < a.length; i++) {
3   if (a[i] == 1) leak = i;
4 }
```

Código 5.1. Exemplo de um código *JavaScript* que usa informação sensível como chave de vetor para praticar o vazamento de informação.

O exemplo apresentado no Código 5.1 mostra uma tentativa de enganar os detectores de vazamento de informação. O código utiliza um *taint data* como chave de um elemento vetor (linha 1) para, logo em seguida, recuperar a informação sensível baseando-se no valor atribuído à posição referenciada pela chave (linha 3).

Esses códigos foram inseridos na base de testes do *TaintJSec* para avaliar a capacidade de detecção da abordagem.

Com a base de testes criada, foi realizada uma tarefa de identificação e exclusão de códigos similares. Essa tarefa foi necessária para evitar testes desnecessários, onde somente nomes de variáveis ou valores eram alterados em um código que utiliza a mesma lógica ou o mesmo recurso da linguagem. A Figura 5.1 exemplifica três códigos similares que usam função *JavaScript*.

Código 1	Código 2	Código 3
`function X(a){ return 1; }`	`function Y(b){ return "2"; }`	`function Z(c){ return null; }`

Figura 5.1. Exemplo de códigos *JavaScript* similares.

Todos os códigos da base tiveram seus respectivos códigos similares excluídos.

Após a tarefa de exclusão de similares sobraram 309 casos de testes, os quais foram agrupados de acordo com a operação *JavaScript* que executavam. Por exemplo, códigos que realizavam operação de execução de função foram agrupados formando um conjunto de testes para avaliar a correta computação de execuções de função. Outros códigos, que executavam operação de laço de repetição, foram agrupados formando um conjunto de testes para operações de laço de repetição, e assim por diante. Essa tarefa de agrupamento resultou em 14 conjuntos de testes, como apresentado no Quadro 5.1.

Quadro 5.1. Conjuntos de testes realizados para testar a propagação do *taint tag*.

Conjuntos	Testes	Nº de Operações
1	Atribuição Encadeada	3
2	Atribuição Simples	37
3	Condicional	22
4	Função Não-Nativa	50
5	Função Nativa	21
6	Função Nativa (eval)	10
7	Prototype	56
8	Laço de Repetição (do while)	11
9	Laço de Repetição (for)	9
10	Laço de Repetição (for in)	6
11	Laço de Repetição (while)	8
12	Switch	20
13	Try-Catch	26
14	Vetor	30

Os 14 conjuntos de testes foram utilizados para testar a qualidade computacional do analisador estático e a corretude da propagação do *taint tag*. Para validar os resultados obtidos em operações de expressões matemáticas ou execução de funções nativas ou qualquer interação entre os objetos do código, os testes foram executados no *SpiderMonkey*.

Foram organizadas três baterias de testes:

(1) **Testes de Propagação** - são testes que foram realizados para validar a correta propagação do *taint tag* nos diferentes tipos de operações *JavaScript*, tais como as operações de: atribuição; condicionais; laços de repetição; *switch*; *try-catch*; execução de funções; e, inserção de novos atributos em instâncias de objetos, por meio do *prototype*. Para essa bateria de testes foram utilizados todos os conjuntos, exceto o conjunto de testes para validar a propagação na execução da função *eval*, pois esse conjunto foi utilizado em outra bateria de testes.

(2) **Testes com a Função *eval*** - foram realizados para validar a eficácia da correta propagação do *taint tag* utilizando *taint position*. Nessa bateria de teste foi utilizado o conjunto de testes com 10 casos distintos de execução da função *eval* contendo operações como: concatenação de *taint data*; inclusão de objetos portadores de *taint data*; chamadas recursivas; e, operações encadeadas. Além dos casos de teste, também foi desenvolvida uma aplicação maliciosa, cujo código *JavaScript* apresenta uma combinação da função *eval* com técnicas de ofuscação

para *taint source* e *taint sink* com a finalidade de praticar o vazamento de informação.

(3) **Testes em Códigos Ofuscados** - realizados para avaliar a capacidade do *TaintJSec* em detectar vazamento de informação em códigos ofuscados por ferramentas específicas para tal finalidade. Foram realizados testes utilizando códigos ofuscados a partir de ferramentas como *JSCompress* [75], *Aaencode* [76], *JSFuck* [77], *JSCrambler* [78] e *Packer* [79].

A seguir, a Figura 5.2 apresenta uma visão geral do processo de criação dos experimentos utilizados para validar a proposta desta pesquisa.

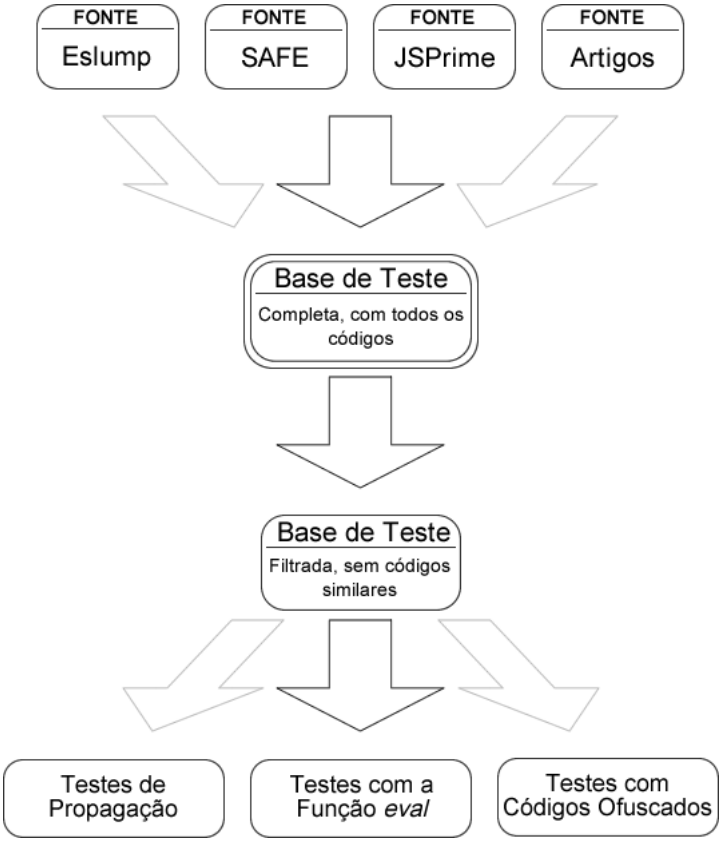


Figura 5.2. Visão geral do protocolo experimental.

As seções seguintes apresentam os testes realizados e os resultados obtidos. A seção 5.2 apresenta os testes de propagação do *taint tag*. A seção 5.3 mostra os testes com a função *eval*. Por fim, e a seção 5.4 apresenta os testes realizados com códigos ofuscados.

5.2 Testes de Propagação

Para validar o *taint propagation* foram realizados testes utilizando os conjuntos apresentados no Quadro 5.1 (exceto o conjunto 6, pois ele é utilizado nos testes da seção 5.3), onde cada grupo testou a corretude da computação de operações pertencentes às especificações da linguagem *JavaScript* (*ECMAScript* 6).

Uma função denominada *getNome* foi marcada como *taint source* para que o *taint tag* a propagação do *taint tag* pudesse ser visualizada nos resultados. Cada conjunto é apresentado seguido dos resultados obtidos pelo analisador estático, *TaintJSec*.

5.2.1 Atribuição

Para a operação de atribuição, os casos de testes dividem-se em operações encadeadas e simples. Nas atribuições encadeadas o valor é atribuído a mais de uma variável, onde as variáveis apresentam-se em sequência. Enquanto a atribuição simples realiza operações para uma única variável.

5.2.1.1 Atribuição Encadeada

```
1 atrserie_teste=atrserie_answer=++[++[+[]][+[]]][+[]];
2 var __atrserie_ab=atrserie_abc=++[+[]][0];
3 var ab=cd=getNome(); // tainted
```

A linha 1 do presente teste apresenta uma operação encadeada de duas variáveis globais. A variável **atrserie_teste** e a variável **atrserie_answer** recebem o valor **2** (resultante a expressão ++[++[+[ ]][+[ ]]][+[ ]]) que não é um *taint data*. A linha 3, por outro lado, apresenta uma operação de atribuição encadeada entre uma variável local e uma variável global, onde ambas recebem um *taint data* e, consequentemente, são marcadas com o *taint tag*.

Resultado 5.2. Teste de atribuição encadeada.

```
1  ------------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   __atrserie_ab = Variavel { valor: 1, tainted: 0}
6  ||   ab = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
7  ------------------------
8  ==================================
9  ------------------------
10 || Nome: Variaveis Globais
11 || Registradores Globais:
12 ||   atrserie_answer = Variavel { valor: 2, tainted: 0}
13 ||   atrserie_teste = Variavel { valor: 2, tainted: 0}
```

```
14 ||   atrserie_abc = Variavel { valor: 1, tainted: 0}
15 ||   cd = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
16 ----------------------
```

Como apresentado no Resultado 5.2, o analisador propagou corretamente a *taint tag* para as variáveis **ab** e **cd** (linhas 6 e 15, respectivamente) e manteve a separação de escopos.

5.2.1.2 Atribuição Simples

```
1  atrimed_a = 2;
2  atrimed_b = 5.2;
3  atrimed_c = atrimed_b % 2;
4  atrimed_d = 3;
5  atrimed_e = 2;
6  atrimed_f = atrimed_d << atrimed_e;
7  atrimed_g = 223 >> atrimed_d;
8  atrimed_h = 1345 >>> 3;
9  atrimed_i = atrimed_e & 3;
10 atrimed_j = atrimed_e ^ atrimed_c;
11 atrimed_k = atrimed_d | atrimed_a | 4;
12
13 var atrimed_var1 = "Teste String";
14 var atrimed_var2 = 4;
15
16 atrimed_l = 5;
17 atrimed_l += atrimed_a;   // Atribuição com adição
18 atrimed_m = 5;
19 atrimed_m -= atrimed_a;   // Atribuição com subtração
20 atrimed_n = 5;
21 atrimed_n *= atrimed_a;   // Atribuição com multiplicação
22 atrimed_o = 5;
23 atrimed_o /= atrimed_a;   // Atribuição com divisão
24 atrimed_p = 5;
25 atrimed_p %= atrimed_a;   // Atribuição com resto
26 atrimed_q = 5;
27 atrimed_q <<= atrimed_a;  // Atribuição com left shift
28 atrimed_r = 5;
29 atrimed_r >>= atrimed_a;  // Atribuição com right shift
30 atrimed_s = 5;
31 atrimed_s >>>= atrimed_a; // Atribuição com unsigned right shift
32 atrimed_t = 5;
33 atrimed_t &= atrimed_a;   // Atribuição com bitwise AND
34 atrimed_u = 5;
35 atrimed_u ^= atrimed_a;   // Atribuição com bitwise XOR
36 atrimed_v = 5;
37 atrimed_v |= atrimed_a;   // Atribuição com bitwise OR
38
39 atrimed_teste = {gu: parseInt(atrimed_var2+"_teste")};
```

```
40 atrimed_ver = atrimed_teste.gu + getNome();
```

O conjunto de atribuição simples é composto por operações de atribuição de valor para apenas uma variável. Os casos de testes cobrem todos os operadores de atribuição contidos nas especificações da linguagem, tais como operadores aritméticos e bit a bit. Para testar a propagação do *taint tag* a função marcada como *taint source* foi acrescentada na linha 40.

Resultado 5.3. Teste de atribuição simples

```
----------------------
|| Nível: 0
|| Nome: raiz
|| Registradores Locais:
||   atrimed_var1 = Variavel { valor: 'Teste String', tainted: 0}
||   atrimed_var2 = Variavel { valor: 4, tainted: 0}
----------------------
==================================
----------------------
|| Nome: Variaveis Globais
|| Registradores Globais:
||   atrimed_a = Variavel { valor: 2, tainted: 0}
||   atrimed_b = Variavel { valor: 5.2, tainted: 0}
||   atrimed_c = Variavel { valor: 1.2000000000000002, tainted: 0}
||   atrimed_d = Variavel { valor: 3, tainted: 0}
||   atrimed_e = Variavel { valor: 2, tainted: 0}
||   atrimed_f = Variavel { valor: 12, tainted: 0}
||   atrimed_g = Variavel { valor: 27, tainted: 0}
||   atrimed_h = Variavel { valor: 168, tainted: 0}
||   atrimed_i = Variavel { valor: 2, tainted: 0}
||   atrimed_j = Variavel { valor: 3, tainted: 0}
||   atrimed_k = Variavel { valor: 7, tainted: 0}
||   atrimed_l = Variavel { valor: 7, tainted: 0}
||   temp_0_44934226253843623 = Variavel { valor: 2, tainted: 0}
||   atrimed_m = Variavel { valor: 3, tainted: 0}
||   temp_0_49647817708097164 = Variavel { valor: 2, tainted: 0}
||   atrimed_n = Variavel { valor: 10, tainted: 0}
||   temp_0_11875163609911987 = Variavel { valor: 2, tainted: 0}
||   atrimed_o = Variavel { valor: 2.5, tainted: 0}
||   temp_0_1591466335119074 = Variavel { valor: 2, tainted: 0}
||   atrimed_p = Variavel { valor: 1, tainted: 0}
||   temp_0_2578797919261797 = Variavel { valor: 2, tainted: 0}
||   atrimed_q = Variavel { valor: 20, tainted: 0}
||   temp_0_9816224709577344 = Variavel { valor: 2, tainted: 0}
||   atrimed_r = Variavel { valor: 1, tainted: 0}
||   temp_0_34492666944778017 = Variavel { valor: 2, tainted: 0}
||   atrimed_s = Variavel { valor: 1, tainted: 0}
||   temp_0_9131041201487287 = Variavel { valor: 2, tainted: 0}
||   atrimed_t = Variavel { valor: 0, tainted: 0}
||   temp_0_6144391310204822 = Variavel { valor: 2, tainted: 0}
```

```
41 ||   atrimed_u = Variavel { valor: 7, tainted: 0}
42 ||   temp_0_03891477949971067 = Variavel { valor: 2, tainted: 0}
43 ||   atrimed_v = Variavel { valor: 7, tainted: 0}
44 ||   temp_0_8063319474736268 = Variavel { valor: 2, tainted: 0}
45 ||   atrimed_teste = Variavel { valor: { gu: Variavel { valor: 4, tainted: 0} },
       tainted: 0}
46 ||   atrimed_ver = Variavel { valor: '4Alexandre Damasceno', tainted: 1}
47 ----------------------
```

Os resultados obtidos demonstram que o analisador conseguiu calcular corretamente todas as operações de atribuição. O *taint tag* foi propagado corretamente para a variável global **atrimed_ver**, como consta na linha 46 do Resultado 5.3.

5.2.2 Condicional

As operações do conjunto condicional testaram a capacidade do analisador estático para escolher os vértices corretos da AST, de acordo com os resultados obtidos durante a análise. Todas as estruturas condicionais pertencentes à linguagem *JavaScript* foram estadas, como por exemplo as estruturas *If-ElseIf-Else* entre as linhas 4 e 12 e a estrutura de condicional ternário, na linha 39.

```
1  var cond_a = Math.PI;
2  cond_b = 10;
3
4  if( cond_a > cond_b){
5    cond_c = 5;
6  }
7  else if(cond_a == cond_b){
8    cond_c = 7;
9  }
10 else{
11   cond_c = getNome(); // tainted
12 }
13 //================
14 cond_d = new Array(1,2,15,1,181,48,48,"null"
       ,85,985,92,5,21,515,1,54854,3221,58,418,41,52,getNome); // tainted
15
16 cond_d2 = [1,2,15,1,181,48,48,"null"
       ,85,985,92,5,21,515,1,54854,3221,58,418,41,52,getNome]; // tainted
17
18 function teste456(){
19   this.teste=10;
20   return 2332;
21 }
22
23 cond_d3 = Array(1,2,15,1,181,48,48,"null"
       ,85,985,92,5,21,515,1,54854,3221,58,418,41,52,teste456); // não é tainted
24 cond_e = new cond_d3[cond_d3.length-1](); // cond_e não será tainted
25 cond_e2 = cond_d2[cond_d2.length-1](); // cond_e2 será tainted
26 cond_f = [1,2,15,1,181,48,48,85,985,92,5,21,515,1,54854,3221,58,418,41,52];
27
28 var cond_g = cond_f.length;
29
30 for(var cond_i=0;cond_i<cond_f.length;cond_i++){
31   if ( cond_g < cond_f[cond_i] ) {
32     cond_g = cond_f[cond_i];
33   }
34 }
35 //================
36 cond_h = 2.1;
37 cond_j = 2;
38
39 var cond_k = cond_h > cond_j ? "cond_h eh maior":"cond_h é menor";
```

```
40 //================
41 var cond_arvores = new Array("pau-brasil", "loureiro", "carvalho", "cedro", "sic
       ômoro");
42
43 delete cond_arvores[3];
44
45 if(3 in cond_arvores) {
46   cond_tamanho_arvore = 678;
47 }
48 else{
49   cond_tamanho_arvore = cond_arvores.length;
50 }
```

Os resultados obtidos da análise, de acordo com as estruturas de objetos apresentadas no Resultado 5.4, foram satisfatórios. A propagação ocorreu corretamente na variável **cond_c**, como consta na linha 28. A variável **cond_k**, embora não seja um *taint data*, também recebeu o valor correto em uma estrutura de condicional ternário, como mostra a linha 20.

Resultado 5.4. Resultado do teste de condicional

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   cond_a = Variavel { valor: 3.141592653589793, tainted: 0}
6  ||   teste456 = Variavel {
7    valor:
8     { estrutura:
9        { range: [Object],
10         type: 'FunctionDeclaration',
11         id: [Object],
12         params: [],
13         defaults: [],
14         body: [Object],
15         generator: false,
16         expression: false } },
17   tainted: 0}
18 ||   cond_g = Variavel { valor: 54854, tainted: 0}
19 ||   cond_i = Variavel { valor: 20, tainted: 0}
20 ||   cond_k = Variavel { valor: 'cond_h eh maior', tainted: 0}
21 ||   cond_arvores = Variavel { valor: [ 'pau-brasil', 'loureiro', 'carvalho', ,
       'sicômoro' ], tainted: 0}
22 ----------------------
23 ==================================
24 ----------------------
25 || Nome: Variaveis Globais
26 || Registradores Globais:
27 ||   cond_b = Variavel { valor: 10, tainted: 0}
28 ||   cond_c = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
```

```
29 ||   cond_d = Variavel { valor: [ 1,2,15,1,181,48,48,'null',85,985,92,5,21,515,
30                                   1,54854,3221,58,418,41,52,[Function: getNome]
                                        ],
31                           tainted: 1}
32 ||   cond_d2 = Variavel { valor: [ 1,2,15,1,181,48,48,'null',85,985,92,5,21,515,
33                                   1,54854,3221,58,418,41,52,[Function: getNome]
                                        ],
34                           tainted: 1}
35 ||   cond_d3 = Variavel { valor: [ 1,2,15,1,181,48,48,'null',85,985,92,5,21,515,
36                                   1,54854,3221,58,418,41,52,{estrutura: [Object
                                        ]} ],
37                           tainted: 0}
38 ||   cond_e = Variavel { valor: teste456 {
39                                  teste: Variavel { valor: 10, tainted: 0} },
                                       tainted: 0}
40 ||   cond_e2 = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
41 ||   cond_f = Variavel { valor: [ 1,2,15,1,181,48,48,85,985,92,5,21,515,1,54854,
42                                   3221,58,418,41,52 ], tainted: 0}
43 ||   cond_h = Variavel { valor: 2.1, tainted: 0}
44 ||   cond_j = Variavel { valor: 2, tainted: 0}
45 ||   cond_tamanho_arvore = Variavel { valor: 5, tainted: 0}
46 ----------------------
```

5.2.3 Função

O casos de teste para validar operações em funções, dividem-se em testes com função não-nativas e funções nativas. Os testes com funções não-nativas testaram a estrutura de armazenamento das funções, a separação dos escopos e a análise nas diferentes formas de declaração de função que a linguagem possui, tais como funções anônimas (linha 1), auto-executáveis (linha 11), criadas em tempo de execução (linhas 19 e 21) e função de seta (linha 40).

5.2.3.1 Função Não-Nativa

```
1  fnc_abc = function fnc_teste(x,y){
2    return (x+10)/y;
3  }
4
5  function fnc_teste_1(x){
6    return (x+10)/2;
7  }
8
9  fnc_x = fnc_abc;
10
11 (function(g){
12   global = g;
13 })(45);
14
15 fnc_aa = getNome() + "_"; // tainted
16
17 b = [1,2,fnc_aa];
18
19 fnc_teste_2 = new Function("x","var a=9;return fnc_aa + (a + 1 - 5 + x);");
20
21 eval("function fnc_fnceval(a){ this.xxx = '"+b[1]+"'; this.nome=fnc_aa; return '
       "+fnc_aa+"' ; /* "+fnc_aa+" */ }"); // tainted na função inteira
22
23 fnc_fnceval_string = fnc_fnceval + ''; // tainted
24
25 fnc_guther = new fnc_fnceval(3);
26
27 fnc_guther_3 = fnc_fnceval(); // tainted
28
29 fnc_teste = fnc_teste_1(10) + "_" + fnc_teste_2(2); // tainted
30
31 function fnc_teste_2(a,b,c){
32   fnc_param = c();  // não é tainted, mas é source tainted
33   abc = fnc_param;  // não é tainted, mas é source tainted
34   m = abc();    // tainted
35   return a + b;
36 }
```

```
37
38 fnc_result1 = fnc_teste_2(1,3,function aabc(){ return getNome; }); //
       fnc_result1 será 4
39
40 f=n=>n?f(n-1)*n:1  // exemplo de função fatorial
41 fnc_factorial_arrow = f(5); // 120
```

O Resultado 5.5 apresenta as estruturas geradas após a análise dos casos de teste de função não-nativas.

O registrador da função **fnc__fnceval** apresenta *taint tag* positivo, como mostra a linha 28 dos resultados. Isso se deve à propagação do *taint tag* da variável **fnc__aa** (linha 21 do teste) que foi concatenada ao argumento da função *eval*. O *taint tag* resultante da operação de concatenação com a função **fnc__fnceval** (linha 23 do teste) foi propagado para a variável **fnc__fnceval__string** como mostra a linha 73 dos resultados. O resultado obtido com o conjunto de função não-nativa foi satisfatório.

Resultado 5.5. Teste de função não-nativa

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   fnc_teste_1 = Variavel {
6    valor:
7     { estrutura:
8        { range: [Object],
9          type: 'FunctionDeclaration',
10         id: [Object],
11         params: [Object],
12         defaults: [],
13         body: [Object],
14         generator: false,
15         expression: false } },
16   tainted: 0}
17 ||   fnc_fnceval = Variavel {
18   valor:
19    { estrutura:
20       { range: [Object],
21         type: 'FunctionDeclaration',
22         id: [Object],
23         params: [Object],
24         defaults: [],
25         body: [Object],
26         generator: false,
27         expression: false } },
28   tainted: 1}
29 ||   fnc_teste_2 = Variavel {
30   valor:
31    { estrutura:
```

```
32        { range: [Object],
33          type: 'FunctionDeclaration',
34          id: [Object],
35          params: [Object],
36          defaults: [],
37          body: [Object],
38          generator: false,
39          expression: false } },
40    tainted: 0}
41  ----------------------
42  ==================================
43  ----------------------
44  || Nome: Variaveis Globais
45  || Registradores Globais:
46  ||    fnc_abc = Variavel {
47    valor:
48     { estrutura:
49        { range: [Object],
50          type: 'FunctionExpression',
51          id: [Object],
52          params: [Object],
53          defaults: [],
54          body: [Object],
55          generator: false,
56          expression: false } },
57    tainted: 0}
58  ||    fnc_x = Variavel {
59    valor:
60     { estrutura:
61        { range: [Object],
62          type: 'FunctionExpression',
63          id: [Object],
64          params: [Object],
65          defaults: [],
66          body: [Object],
67          generator: false,
68          expression: false } },
69    tainted: 0}
70  ||    global = Variavel { valor: 45, tainted: 0}
71  ||    fnc_aa = Variavel { valor: 'Alexandre Damasceno_', tainted: 1}
72  ||    b = Variavel { valor: [ 1, 2, 'Alexandre Damasceno_' ], tainted: 1}
73  ||    fnc_fnceval_string = Variavel {
74    valor: 'function fnc_fnceval(a){ this.xxx = \'2\'; this.nome=fnc_aa; return \'
          Alexandre Damasceno_\' ; /* Alexandre Damasceno_ */ }',
75    tainted: 1}
76  ||    fnc_guther = Variavel {
77    valor:
78     fnc_fnceval {
79       xxx: Variavel { valor: '2', tainted: 0},
80       nome:
```

```
81        Variavel { valor: 'Alexandre Damasceno_', tainted: 1} },
82    tainted: 0,
83    varobj: [ a: Variavel { valor: 3, tainted: 0} ] }
84  ||    xxx = Variavel { valor: '2', tainted: 0}
85  ||    nome = Variavel { valor: 'Alexandre Damasceno_', tainted: 1}
86  ||    fnc_guther_3 = Variavel { valor: 'Alexandre Damasceno_', tainted: 1}
87  ||    fnc_teste = Variavel { valor: '10_Alexandre Damasceno_7', tainted: 1}
88  ||    fnc_param = Variavel { valor: [Function: getNome], tainted: 0}
89  ||    abc = Variavel { valor: [Function: getNome], tainted: 0}
90  ||    m = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
91  ||    fnc_result1 = Variavel { valor: 4, tainted: 0}
92  ||    f = Variavel {
93    valor:
94     { estrutura:
95        { range: [Object],
96          type: 'ArrowFunctionExpression',
97          id: [Object],
98          params: [Object],
99          defaults: [],
100         body: [Object],
101         generator: false,
102         expression: true } },
103   tainted: 0}
104 ||    fnc_factorial_arrow = Variavel { valor: 120, tainted: 0}
105 -----------------------
```

5.2.3.2 Função Nativa

Os casos de teste de função nativa verificam a capacidade do *TaintJSec* em computador as funções existentes por padrão na linguagem *JavaScript*. Para avaliar a propagação do *taint tag* a função *getNome* foi inserida na linha 19.

```
1  fncnat_a = Math.pow(3,3);
2
3  fncnat_b = parseInt("2_teste",10);
4
5  fncnat_c = parseInt("3_tainted" + getNome(),10); // tainted
6
7  fncnat_d = parseFloat("2.5__");
8
9  fncnat_e = typeof "texto";
10
11 fncnat_f = typeof({});
12
13 fncnat_g = Boolean(1==2);
14
15 var fncnat_h = Array.prototype.slice.call([2,3]);
16
17 eval("fncnat_i=12345;fncnat_j=fncnat_i++;");
```

```
18
19 var op = new getNome(); // tainted
20
21 d = ((((''|c|eh|var|''+op.constructor.name+''||legal|d|trim|split||''))).split('|').
       join("-").split("--").join("|").replace(/\|/g,".") + " teste ").trim().
       substring(10,17);   // tainted
22
23 f = eval("eval('eval(d)')"); // tainted
```

O Resultado 5.6 apresenta o resultado da análise do *TaintJSec*, onde é possível notar que os valores obtidos do cálculo das funções nativas estão corretos, como por exemplo o cálculo exponencial da linha 1 do teste, onde a função nativa *Math.pow* calculou o cubo de 3 e atribuiu o resultado ao registrador da variável global **fncnat_a**. A variável local **op**, na linha 19 do teste, fez um acesso ao *taint source getNome* e por isso foi marcada com o *taint tag*, como mostra a linha 6 da saída do analisador. A variável **op** foi utilizada na linha 21 do teste e, por isso, propagou o *taint tag* para a variável **d**. Essa por sua vez, propagou o *taint tag* para a variável **f**, por meio de execuçcao da função *eval*. Essas variáveis foram marcadas com *taint tag* positivo, como é possível ver nas linhas 21 e 22 do resultado da análise.

Resultado 5.6. Resultado do teste de função nativa

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   fncnat_h = Variavel { valor: [ 2, 3 ], tainted: 0}
6  ||   op = Variavel { valor: getNome {}, tainted: 1}
7  ----------------------
8  ==================================
9  ----------------------
10 || Nome: Variaveis Globais
11 || Registradores Globais:
12 ||   fncnat_a = Variavel { valor: 27, tainted: 0}
13 ||   fncnat_b = Variavel { valor: 2, tainted: 0}
14 ||   fncnat_c = Variavel { valor: 3, tainted: 1}
15 ||   fncnat_d = Variavel { valor: 2.5, tainted: 0}
16 ||   fncnat_e = Variavel { valor: 'string', tainted: 0}
17 ||   fncnat_f = Variavel { valor: 'object', tainted: 0}
18 ||   fncnat_g = Variavel { valor: false, tainted: 0}
19 ||   fncnat_i = Variavel { valor: 12346, tainted: 0}
20 ||   fncnat_j = Variavel { valor: 12345, tainted: 0}
21 ||   d = Variavel { valor: 'getNome', tainted: 1}
22 ||   f = Variavel { valor: [Function: getNome], tainted: 1}
23 ----------------------
```

5.2.4 *Prototype*

Os casos de teste do conjunto *prototype* testaram a capacidade do analisador em propagar a inserção de novos atributos por meio do protótipo do objeto.

A função **prot__person2**, por exemplo, é declarada na linha 55 e possui somente um atributo denominado *firstName*. Logo em seguida, essa função é instanciada por duas variáveis, **prot__m** na linha 59 e **prot__n** na linha 60. Até esse momento do fluxo de execução, as variáveis **prot__m** e **prot__n** possuem somente o atributo *firstName*. Mas na linha 62 ocorre uma operação de inserção do atributo *lastName* no protótipo da função **prot__person2** que deve ser propagado para todas as suas instâncias. Logo, o novo atributo deve ser propagado para as duas variáveis.

```
1  x = 1;
2  function prot_teste(){
3    this.nome = "guther";
4    var ter1 = x;
5    this.fnc = function prot_teste2(){
6      this.nome = "guther2_"+ter1;
7      var ter2 = ++ter1;
8      this.fnc = function prot_teste3(){
9        this.nome = "guther3";
10     }
11     return "teste45_"+ter1;
12   }
13   return 56;
14 }
15
16 var prot_a = new prot_teste(); // Object { nome: "guther", fnc: prot_teste2() }
17
18 prot_a1 = prot_a.fnc(); // teste45_2  ... agora prot_a passou a ser Object {
       nome: "guther2_1", fnc: prot_teste3() }
19
20 teste = prot_a.nome; // guther2_1
21
22 prot_b = prot_a.fnc.toString().length;
23
24 prot_c = new prot_a.fnc(); // Object { nome: "guther3" }
25
26 prot_d = prot_c.nome; // guther3
27
28 prot_e = prot_a.constructor.toString(); // mostra a função prot_teste em string
29
30 var prot_f = prot_a.fnc;
31
32 prot_g = prot_a.constructor.name; // prot_teste
33
34 prot_h = new prot_f();
35
```

```
36 prot_i = prot_f.toString(); // mostra a função prot_teste3 em string
37
38 prot_j = prot_a.fnc.toString().toString().toString().toString(); // mostra a fun
       ção prot_teste3 em string
39
40
41 function prot_person(first,last) {
42   this.prot_firstName = first;
43   this.prot_lastName = last;
44   prot_gr = first + " " + last;
45   return 85;
46 }
47
48 prot_arg = 456;
49
50 prot_k = new prot_person("Alexandre","Damasceno");
51
52 prot_l = prot_person("John",prot_arg);
53
54
55 function prot_person2() {
56     this.firstName = "r e";
57 }
58
59 prot_m  = new prot_person2();
60 prot_n = new prot_person2();
61
62 prot_person2.prototype.lastName = " h t" + " u g";
63
64 prot_o = prot_m.firstName + prot_n.lastName; // "r e h t u g"
65
66 prot_p = prot_o.replace(/ +/g,"").split("").reverse().join(""); // "guther"
67
68 var prot_answer = 6 * 7;
69
70 function prot_fnc1(){
71   var abc = 10;
72   this.nome = prot_answer;
73   this.nome2 = 15;
74 }
75
76 function prot_fnc2(){
77   var abc = 10;
78   this.nome = prot_answer;
79 }
80
81 var prot_a1 = new prot_fnc1();
82
83 prot_fnc1.prototype.numero = 778;
84
```

```
85 var prot_numero = prot_a1.numero; // 778
86
87 prot_comparacao1 = prot_fnc2 == prot_a1.constructor; // false
88
89 prot_comparacao2 = prot_fnc1 == prot_a1.constructor; // true
90
91 prot_trecho_interno_fnc = prot_a1.constructor.toString().toString(); // função
       prot_fnc1 em string
92
93 prot_b1 = [1,2,3,4,5];
94
95 prot_b2 = prot_b1.toString().trim().toString().substring(0,7).split(",").reverse
       ().join("-"); // '4-3-2-1'
96
97 prot_c1 = getNome().toString() + " Braga"; // tainted
98
99 prot_vlength = prot_trecho_interno_fnc.split("10")[1].length;
```

Resultado 5.7. Resultado do teste de *prototype*

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   prot_teste = Variavel {
6    valor:
7     { estrutura:
8        { range: [Object],
9          type: 'FunctionDeclaration',
10         id: [Object],
11         params: [],
12         defaults: [],
13         body: [Object],
14         generator: false,
15         expression: false } },
16   tainted: 0}
17 ||   prot_a = Variavel {
18   valor:
19    prot_teste {
20      nome: Variavel { valor: 'guther2_1', tainted: 0, varobj: [Object] },
21      fnc: Variavel { valor: [Object], tainted: 0, varobj: [Object] } },
22   tainted: 0,
23   varobj: [ ter1: Variavel { valor: 2, tainted: 0} ] }
24 ||   prot_f = Variavel {
25   valor:
26    { estrutura:
27       { range: [Object],
28         type: 'FunctionDeclaration',
29         id: [Object],
30         params: [],
31         defaults: [],
```

```
32          body: [Object],
33          generator: false,
34          expression: false } },
35    tainted: 0,
36    varobj: [ ter1: Variavel { valor: 2, tainted: 0} ] }
37  ||    temp_0_6664122824145129 = Variavel {
38    valor: 'function prot_teste3(){\n\t\t\tthis.nome = "guther3";\n\t\t}',
39    tainted: 0}
40  ||    prot_person = Variavel {
41    valor:
42     { estrutura:
43        { range: [Object],
44          type: 'FunctionDeclaration',
45          id: [Object],
46          params: [Object],
47          defaults: [],
48          body: [Object],
49          generator: false,
50          expression: false } },
51    tainted: 0}
52  ||    prot_person2 = Variavel {
53    valor:
54     { estrutura:
55        { range: [Object],
56          type: 'FunctionDeclaration',
57          id: [Object],
58          params: [],
59          defaults: [],
60          body: [Object],
61          generator: false,
62          expression: false } },
63    tainted: 0}
64  ||    temp_0_9454113479327422 = Variavel { valor: 'rehtug', tainted: 0}
65  ||    temp_0_9275837368313093 = Variavel {
66    valor: [ 'g', 'u', 't', 'h', 'e', 'r' ],
67    tainted: 0}
68  ||    prot_answer = Variavel { valor: 42, tainted: 0}
69  ||    prot_fnc1 = Variavel {
70    valor:
71     { estrutura:
72        { range: [Object],
73          type: 'FunctionDeclaration',
74          id: [Object],
75          params: [],
76          defaults: [],
77          body: [Object],
78          generator: false,
79          expression: false } },
80    tainted: 0}
81  ||    prot_fnc2 = Variavel {
```

```
  valor:
   { estrutura:
      { range: [Object],
        type: 'FunctionDeclaration',
        id: [Object],
        params: [],
        defaults: [],
        body: [Object],
        generator: false,
        expression: false } },
  tainted: 0}
||   prot_a1 = Variavel {
  valor:
   prot_fnc1 {
     nome: Variavel { valor: 42, tainted: 0},
     nome2: Variavel { valor: 15, tainted: 0} },
  tainted: 0,
  varobj: [ abc: Variavel { valor: 10, tainted: 0} ] }
||   prot_numero = Variavel { valor: 778, tainted: 0,
  varobj: [ abc: Variavel { valor: 10, tainted: 0} ] }
||   temp_0_8462935304505241 = Variavel { valor: '1,2,3,4,5', tainted: 0}
||   temp_0_39906837389926375 = Variavel { valor: '1,2,3,4', tainted: 0}
||   temp_0_2655353333984558 = Variavel { valor: [ '4', '3', '2', '1' ],
    tainted: 0}
-----------------------
-----------------------
|| Nível: 0.0
|| Nome: prot_teste
|| _varobj: [ ter1: Variavel { valor: 2, tainted: 0} ]
|| _obj_container: prot_a
|| Registradores Locais:
||   prot_teste2 = Variavel {
  valor:
   { estrutura:
      { range: [Object],
        type: 'FunctionDeclaration',
        id: [Object],
        params: [],
        defaults: [],
        body: [Object],
        generator: false,
        expression: false } },
  tainted: 0}
||   ter1 = Variavel { valor: 2, tainted: 0}
-----------------------
-----------------------
|| Nível: 0.1
|| Nome: prot_teste
|| _varobj: [ ter1: Variavel { valor: 2, tainted: 0} ]
|| _obj_container: prot_a
```

```
|| Registradores Locais:
||   prot_teste3 = Variavel {
  valor:
   { estrutura:
      { range: [Object],
        type: 'FunctionDeclaration',
        id: [Object],
        params: [],
        defaults: [],
        body: [Object],
        generator: false,
        expression: false } },
  tainted: 0}
-----------------------
-----------------------
|| Nível: 0.2
|| Nome: prot_teste
|| _varobj: [ ter1: Variavel { valor: 2, tainted: 0} ]
|| _obj_container: prot_a
|| Registradores Locais:
||   prot_teste3 = Variavel {
  valor:
   { estrutura:
      { range: [Object],
        type: 'FunctionDeclaration',
        id: [Object],
        params: [],
        defaults: [],
        body: [Object],
        generator: false,
        expression: false } },
  tainted: 0}
-----------------------
==================================
-----------------------
|| Nome: Variaveis Globais
|| Registradores Globais:
||   x = Variavel { valor: 1, tainted: 0}
||   teste = Variavel {
  valor: 'guther2_1',
  tainted: 0,
  varobj: [ ter1: Variavel { valor: 2, tainted: 0} ] }
||   prot_b = Variavel { valor: 53, tainted: 0}
||   prot_c = Variavel {
  valor:
   prot_teste3 {
     nome: Variavel { valor: 'guther3', tainted: 0, varobj: [Object] } },
  tainted: 0,
  varobj: [ ter1: Variavel { valor: 2, tainted: 0} ] }
||   prot_d = Variavel { valor: 'guther3', tainted: 0,
```

```
181    varobj: [ ter1: Variavel { valor: 2, tainted: 0} ] }
182  ||   prot_e = Variavel {
183    valor: 'function prot_teste(){\n\tthis.nome = "guther";\n\tvar ter1 = x;\n\
           tthis.fnc = function prot_teste2(){\t\n\t\tthis.nome = "guther2_"+ter1;\n\
           t\tvar ter2 = ++ter1;\n\t\tthis.fnc = function prot_teste3(){\n\t\t\
           tthis.nome = "guther3";\n\t\t}\n\t\treturn "teste45_"+ter1;\n\t}\n\treturn
            56;\n}',
184    tainted: 0}
185  ||   prot_g = Variavel { valor: 'prot_teste', tainted: 0}
186  ||   prot_h = Variavel {
187    valor: prot_teste3 { nome: Variavel { valor: 'guther3', tainted: 0} },
188    tainted: 0,
189    varobj: [ ter1: Variavel { valor: 2, tainted: 0} ] }
190  ||   prot_i = Variavel {
191    valor: 'function prot_teste3(){\n\t\t\tthis.nome = "guther3";\n\t\t}',
192    tainted: 0}
193  ||   prot_j = Variavel {
194    valor: 'function prot_teste3(){\n\t\t\tthis.nome = "guther3";\n\t\t}',
195    tainted: 0}
196  ||   prot_arg = Variavel { valor: 456, tainted: 0}
197  ||   prot_gr = Variavel { valor: 'John 456', tainted: 0}
198  ||   prot_k = Variavel {
199    valor:
200     prot_person {
201       prot_firstName: Variavel { valor: 'Alexandre', tainted: 0},
202       prot_lastName: Variavel { valor: 'Damasceno', tainted: 0} },
203    tainted: 0,
204    varobj:
205     [ first: Variavel { valor: 'Alexandre', tainted: 0},
206       last: Variavel { valor: 'Damasceno', tainted: 0} ] }
207  ||   prot_firstName = Variavel { valor: 'John', tainted: 0}
208  ||   prot_lastName = Variavel { valor: 456, tainted: 0}
209  ||   prot_l = Variavel { valor: 85, tainted: 0}
210  ||   prot_m = Variavel {
211    valor:
212     prot_person2 {
213       firstName: Variavel { valor: 'r e', tainted: 0} },
214    tainted: 0}
215  ||   prot_n = Variavel {
216    valor:
217     prot_person2 {
218       firstName: Variavel { valor: 'r e', tainted: 0} },
219    tainted: 0}
220  ||   prot_o = Variavel { valor: 'r e h t u g', tainted: 0}
221  ||   prot_p = Variavel { valor: 'guther', tainted: 0}
222  ||   prot_comparacao1 = Variavel { valor: false, tainted: 0}
223  ||   prot_comparacao2 = Variavel { valor: true, tainted: 0}
224  ||   prot_trecho_interno_fnc = Variavel {
225    valor: 'function prot_fnc1(){\n\tvar abc = 10;\n\tthis.nome = prot_answer;\n\
           tthis.nome2 = 15;\n}',
```

```
226    tainted: 0}
227  ||   prot_b1 = Variavel { valor: [ 1, 2, 3, 4, 5 ], tainted: 0}
228  ||   prot_b2 = Variavel { valor: '4-3-2-1', tainted: 0}
229  ||   prot_c1 = Variavel {
230    valor: 'Alexandre Damasceno Braga',
231    tainted: 1}
232  ||   prot_vlength = Variavel { valor: 47, tainted: 0}
233  ----------------------
```

O Resultado 5.7 apresenta a saída obtida pelo *TaintJSec* ao término da análise.

É possível notar na linha 220 do resultado, que os atributos inseridos por meio do *prototype* dos objetos foram propagados para as instâncias, pois na linha 64 dos casos de testes existe uma operação de concatenação entre o atributo *firstName* (existente desde a declaração da função) e o atributo *lastName* (inserido pelo *prototype*) que resulta no valor **"r e h t u g"**.

Contudo, os atributos inseridos pelo *prototype* não aparecem no relatório do analisador estático. Como mostra as linhas 210 e 215 dos resultados do teste, onde as variáveis **prot__m** e **prot__n** apresentam somente o atributo *firstName*. Isso se deve a uma limitação da plataforma *Node.js*, que não apresenta os atributos inseridos por meio do *prototype* na função *console.log*.

Embora essa limitação exista, ela é meramente visual e não impacta nos resultados da análise, vindo apenas a prejudicar a leitura do relatório final gerado pelo *TaintJSec*.

5.2.5 Laço de Repetição

Os conjuntos para testar os laços de repetição foram organizados em casos de teste para operações *Do While*, *For*, *For In* e *While*. Cada conjunto testa a corretude computacional do analisador para percorrer os diferentes tipos de estruturas de laço contidas na AST.

5.2.5.1 *Do While*

Para avaliar a propagação do *taint tag*, a função *taint source getNome* foi inserida na linha 3.

```
1  var repdw_valor2 = {a:1,b:2,c:3};
2
3  var repdw_valor = ["guther","teste",11,getNome()]; // tainted
4
5  repdw_a = ++repdw_valor[repdw_valor2.b] + ++[+[]][+[]]; // 13
6
7  var repdw_teste = "";
8  repdw_i = 0;
9
10 do{
11   repdw_teste += "_" + repdw_i;
12   repdw_i++;
13
14   if(repdw_i==10/2);
15     leak = repdw_valor[repdw_valor.length-1]; // tainted
16
17 }
18 while(repdw_i < 10);
```

Os resultados obtidos após a análise são apresentados no Resultado 5.8, onde a estrutura da variável **repdw_valor** é apresentada com o *taint tag* positivo (linha 9), assim como as demais variáveis cujo valor foi atribuído após uma operação envolvendo o *taint data*, como é o caso das variáveis **repdw_a** e **leak**. Logo, a propagação ocorreu corretamente.

Dentro do escopo do laço de repetição, os valores obtidos nas operações estão corretos. A variável **leak**, por exemplo, recebeu o *taint tag* em uma operação dentro do laço de repetição (linha 15 do teste).

Assim, o resultado do teste mostrou que o analisador consegue analisar a estrutura *Do While* corretamente.

Resultado 5.8. Resultado do teste do laço de repetição *Do While*

```
1  ------------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   repdw_valor2 = Variavel { valor: { a: 1, b: 2, c: 3 }, tainted: 0}
6  ||   repdw_valor = Variavel {
7    valor: [ 'guther', 'teste', 12, 'Alexandre Damasceno' ],
8    tainted: 1}
9  ||   repdw_teste = Variavel { valor: '_0_1_2_3_4_5_6_7_8_9', tainted: 0}
10 ------------------------
11 ==================================
12 ------------------------
13 || Nome: Variaveis Globais
14 || Registradores Globais:
15 ||   repdw_a = Variavel { valor: 13, tainted: 1}
16 ||   repdw_i = Variavel { valor: 10, tainted: 0}
17 ||   leak = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
18 ------------------------
```

5.2.5.2 *For*

Para avaliar a propagação do *taint tag* no laço de repetição *For*, a função *getNome* foi criada em tempo de execução, a partir da união de valores literais (linha 9) que formam o identificador da função. Essas partes são concatenadas dentro do escopo do laço de repetição (linha 13) e, em seguida, o identificador da função *getNome* é criado pela função *eval*, na linha 15, e o valor da variável **fnc__union** é atualizado com o *taint data*.

```
1  var repf_valor = "";
2
3  for(repf_a=0; repf_a<5; repf_a++ ){
4    for(repf_i=0; repf_i<repf_a; repf_i++ ){
5      repf_valor = repf_valor + "_" + repf_i;  // _0_0_1_0_1_2_0_1_2_3
6    }
7  }
8
9  var fnc = ["ge","tN","om","e"];
10 var fnc_union = "";
11
12 for(var repf_j=0; repf_j<fnc.length; repf_j++ ){
13     fnc_union += fnc[repf_j];
14 }
15 fnc_union = eval(fnc_union); // tainted
```

Os resultados mostraram que a análise da estrutura do laço de repetição *For* está correta, pois o valores final da variável **repf__valor** está de acordo com o esperado, como é possível observar na linha 6 do Resultado 5.9.

A estrutura da variável **fnc__union**, na linha 8, contém a função *getNome* no valor do objeto e apresenta *taint tag* com valor positivo, indicando que a propagação ocorreu corretamente.

Resultado 5.9. Resultado do teste do laço de repetição *For*

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   repf_valor = Variavel { valor: '_0_0_1_0_1_2_0_1_2_3', tainted: 0}
6  ||   fnc = Variavel { valor: [ 'ge', 'tN', 'om', 'e' ], tainted: 0}
7  ||   fnc_union = Variavel { valor: [Function: getNome], tainted: 1}
8  ||   repf_j = Variavel { valor: 4, tainted: 0}
9  ----------------------
10 ====================================
11 ----------------------
12 || Nome: Variaveis Globais
13 || Registradores Globais:
14 ||   repf_a = Variavel { valor: 5, tainted: 0}
15 ||   repf_i = Variavel { valor: 4, tainted: 0}
16 ----------------------
```

5.2.5.3 *For In*

Para testar o analisador estático na estrutura de laço de repetição *For In*, a função *taint source getNome*, foi inserida nas linhas 3 e 4 para verificar o resultado da concatenação incremental, onde cada caractere do valor retornado pela função *getNome* é concatenado à variável **forin_teste** a cada repetição do laço.

```
1 var forin_teste = "";
2
3 for(var forin_i in getNome()){
4   forin_teste += (getNome())[forin_i];   // tainted
5 }
```

O Resultado 5.10 apresenta as estruturas de objetos geradas pelo analisador ao término do teste de laço de repetição *For In*.

Na linha 5 do Resultado 5.10 é possível observar o registrador da variável **forin_teste** com *taint tag* positivo, que foi propagada na linha 4 dos testes executados.

Resultado 5.10. Resultado do teste do laço de repetição *For In*

```
1 ----------------------
2 || Nível: 0
3 || Nome: raiz
4 || Registradores Locais:
5 ||   forin_teste = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
6 ||   forin_i = Variavel { valor: '18', tainted: 0}
7 ----------------------
```

5.2.5.4 *While*

No teste realizado com o laço de repetição *While*, o *taint source* foi inserido na linha 5. Cada caractere do valor retornado pelo *taint source* é concatenado com o valor literal "_" e, em seguida, atribuído à variável **repwh_teste**.

```
1 var repwh_teste = "";
2 repwh_i = 0;
3
4 while(repwh_i < getNome().length){
5   repwh_teste += "_" + getNome().substring(repwh_i,repwh_i+1); // tainted
6   repwh_i++;
7 }
```

O resultados mostram que o analisador estático foi eficaz em percorrer a AST com a estrutura do laço de repetição *While*. Os valores foram calculados corretamente e o *taint tag* foi propagado para o registrador da variável **repwh_teste**, como apresenta a linha 5 do Resultado 5.11.

Resultado 5.11. Resultado do teste do laço de repetição *While*

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   repwh_teste = Variavel { valor: '_A_l_e_x_a_n_d_r_e_ _D_a_m_a_s_c_e_n_o',
       tainted: 1}
6  ----------------------
7  ==================================
8  ----------------------
9  || Nome: Variaveis Globais
10 || Registradores Globais:
11 ||   repwh_i = Variavel { valor: 19, tainted: 0}
12 ----------------------
```

5.2.5.5 *Switch*

Para avaliar o capacidade de análise do *TaintJSec* em estrutura do tipo *Switch*, o *taint source* foi inserido na linha 14.

```
1  var swt_variavel = true;
2  swt_tr = [2,'e'];
3  swt_b = swt_tr[1];
4  var swt_tui = eval("'test'") + swt_b;
5
6  switch(swt_variavel) {
7    case eval("false"):
8      swt_resultado = 10;
9      break;
10   case true:
11     if(!(1==1))
12       swt_resultado = 100;
13     else
14       swt_resultado = getNome();    // tainted
15     break;
16   case swt_tui:
17     swt_resultado = 10090;
18     break;
19   default:
20     swt_resultado = 10000;
21     break;
22 }
```

Os resultados obtidos demonstram que o analisador foi capaz de percorrer a estrutura da AST corretamente. A *taint tag* foi propagada para a variável **swt_resultado** como apresenta o Resultado 5.12 na linha 14.

Resultado 5.12. Resultado do teste de *switch*

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   swt_variavel = Variavel { valor: true, tainted: 0}
6  ||   swt_tui = Variavel { valor: 'teste', tainted: 0}
7  ----------------------
8  ==================================
9  ----------------------
10 || Nome: Variaveis Globais
11 || Registradores Globais:
12 ||   swt_tr = Variavel { valor: [ 2, 'e' ], tainted: 0, varobj: [] }
13 ||   swt_b = Variavel { valor: 'e', tainted: 0, varobj: [] }
14 ||   swt_resultado = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
15 ----------------------
```

5.2.5.6 *Try-Catch*

No conjunto dos casos de teste *Try-Catch*, a propagação da informação sensível inica na linha 10 e ocorre novamente na linha 19. Os demais casos de teste foram realizados para testar a capacidade do analisar em percorrer estruturas *Try-Catch* aninhadas, como consta nas linhas 26 e 27.

```
1  tc_codigo = 101;
2
3  try {
4    if (tc_codigo < 10) {
5      throw "erro_1";
6      tc_tester = 88;
7    }
8    else {
9      if (tc_codigo > 100) {
10       throw getNome(); // taint propagation
11     }
12   }
13 }
14 catch (tc_erro) {
15   if (tc_erro == "erro_1") {
16     tc_a ='Codigo é menor que 10';
17   }
18   if (tc_erro == getNome()) {
19     tc_a = 'Codigo é maior que 100 - '+tc_erro; // tainted
20   }
21 }
22 finally{
23   var tc_t = 8;
24 }
25
26 try {
27   try {
28       throw "oops";
29   }
30   catch(tc_ay){
31       tc_b = tc_ay;
32       throw tc_b + 2;
33   }
34   finally{
35       throw tc_b++;
36       tc_teste = 10;  // essa operação não chega a ser analisada devido ao
             ThrowStatement
37   }
38 }
39 catch (tc_ex) {
40   tc_saida = delete tc_codigo;
41 }
```

O Resultado 5.13 apresenta as estruturas de objeto geradas pelo *TaintJSec*.

Os resultados demonstram que o analisador foi capaz de percorrer a estrutura da operação *Try-Catch*. A informação sensível foi atribuída à variável **tc_a**, e esta passou a ser um *taint data*, como informado na linha 13 do Resultado 5.13.

Resultado 5.13. Resultado do teste de *Try-Catch*

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   tc_t = Variavel { valor: 8, tainted: 0}
6  ----------------------
7  ==================================
8  ----------------------
9  || Nome: Variaveis Globais
10 || Registradores Globais:
11 ||   tc_a = Variavel {
12   valor: 'Codigo é maior que 100 - Alexandre Damasceno',
13   tainted: 1}
14 ||   tc_b = Variavel { valor: NaN, tainted: 0}
15 ||   tc_saida = Variavel { valor: true, tainted: 0}
16 ----------------------
```

5.2.5.7 Vetor

Este conjunto de teste tem o objetivo de avaliar a qualidade da análise do *TaintJSec* em operações que envolvam vetores.

Dentre os casos de teste deste conjunto, constam operações que utilizam as duas formas distintas de declaração de vetores (linhas 3 e 39), atribuição (linha 15) e recuperação (linha 13). Também foi utilizado um exemplo de vazamento de informação retirado do trabalho de Prakasam Kannan [74], onde a informação sensível é utilizada como chave de vetor (linha 48) e recuperada dentro do escopo de um laço de repetição (linha 52).

```
1  array_x = getNome(); // tainted
2
3  var array_a=[getNome(),[5,[3,8],7],3]; // tainted
4
5  var array_f = [1,"a",4,array_a[0]+1]; // tainted
6
7  array_f["'a'"] = 99;
8
9  array_dr = array_f["'a'"]; // tainted
10
11 array_b = {primeiro:1,"'segundo'":{a:2,b:{teste:"guther"}},terceiro:7};
12
13 array_c = array_a[1][1][1]; // tainted
14
15 array_a[1][0] = 10;
16
17 array_teste = Array(1);
18
19 array_teste.push("teste");
20
21 array_teste[1] += "_1" ;
22
23 array_a["2"] = [3,4,5];
24
25 array_a[2][0] = 1+1;
26
27 array_b.segundo = 100;
28
29 array_t  = array_b.segundo;
30
31 array_b["primeiro"] = 607;
32
33 array_p = array_b["'segundo'"].b["teste"];
34
35 array_s = array_b["'segundo'"].b.teste;
36
37 array_aa = 5;
38
```

```
39 array_op2 = new Array(1,array_aa,3);
40
41 var array_junto = array_op2.join();
42
43 array_op2.pop();
44
45 array_re = array_op2[1] + array_op2[array_op2.length-1];
46
47 var array_z = [];       // Exemplo de vazamento de informação
48 array_z[getNome()] = 1;   // retirado do artigo: Taint and Information
49               // Flow Analysis Using Sweet.js Macros
50 for(i in array_z){      // i tainted
51   if(array_z[i]==1)
52     b = i;         // taint propagation
53 }
54 c = i;             // taint propagation
```

Os resultados apresentados pelo *TaintJSec* foram satisfatórios. Nas estruturas de objetos apresentadas pelo Resultado 5.14 é possível observar que o *taint tag* propagou corretamente. O registrador da variável **array__a**, por exemplo, apresenta o *taint tag* positivo na linha 7. Tal propagação foi realizada na linha 3 dos testes, onde um *taint source* foi inserido em uma das posições do vetor.

A propagação do *taint tag* assim como as demais operações do conjunto de testes em vetores foram realizadas corretamente. Até mesmo o exemplo de vazamento de informação, retirado de artigo publicado, foi detectado e a variável que recebeu o valor sensível presente na chave do vetor passou a ser *taint data*, como mostra a linha 36 dos resultados.

Resultado 5.14. Resultado do teste com Vetor

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   array_a = Variavel {
6    valor: [ 'Alexandre Damasceno', [ 10, [Object], 7 ], [ 2, 4, 5 ] ],
7    tainted: 1}
8  ||   array_f = Variavel {
9    valor: [ 1, 'a', 4, 'Alexandre Damasceno1', '\'a\'': 99 ],
10   tainted: 1}
11 ||   array_junto = Variavel { valor: '1,5,3', tainted: 0}
12 ||   array_z = Variavel { valor: [ 'Alexandre Damasceno': 1 ], tainted: 0}
13 ----------------------
14 ==================================
15 ----------------------
16 || Nome: Variaveis Globais
17 || Registradores Globais:
18 ||   array_x = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
19 ||   array_dr = Variavel { valor: 99, tainted: 1}
```

```
||   array_b = Variavel {
  valor:
   { primeiro: Variavel { valor: 607, tainted: 0},
     '\'segundo\'': Variavel { valor: [Object], tainted: 0},
     terceiro: 7,
     segundo: Variavel { valor: 100, tainted: 0} },
  tainted: 0}
||   array_c = Variavel { valor: 8, tainted: 1}
||   array_teste = Variavel { valor: [ , 'teste_1' ], tainted: 0}
||   array_t = Variavel { valor: 100, tainted: 0}
||   array_p = Variavel { valor: 'guther', tainted: 0}
||   array_s = Variavel { valor: 'guther', tainted: 0}
||   array_aa = Variavel { valor: 5, tainted: 0}
||   array_op2 = Variavel { valor: [ 1, 5 ], tainted: 0}
||   array_re = Variavel { valor: 10, tainted: 0}
||   i = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
||   b = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
||   c = Variavel { valor: 'Alexandre Damasceno', tainted: 1}
||   a = Variavel { valor: 5, tainted: 0}
-----------------------
```

5.3 Testes com a Função *eval*

Para validar a eficácia de utilização de *taint position* para a correta propagação do *taint tag* no escopo de execução da função *eval*, foi utilizado o conjunto 6 do Quadro quadro:conjtestespropagacao. Este conjunto é constituído por 10 casos de testes, e cada caso apresenta uma forma distinta de envio de argumentos para a função *eval*. O Quadro 5.2 apresenta os 10 casos de teste.

Quadro 5.2. Testes de propagação do *taint tag* na função *eval*.

Código Base	
var eval_a0 = fnc_Tainted_Source();	
Nº do Teste	**Código**
1	eval_a1 = eval(eval_a0);
2	eval_a2 = eval(“eval_a0”);
3	eval_a3 = eval(“eval” + “_a0”);
4	eval_a4 = eval(“‘eval” + “_a0’”);
5	eval_a5 = eval(“eval(\”‘eval\“ + \”_a0’\“)”);
6	eval_a6 = eval(“eval(\”eval\“ + \”_a0\“)”);
7	eval_a7 = eval(“eval_”+“b7= ‘” + eval_a0 +,“’;”);
8	eval_a8 = eval(“eval_”+“b7= ‘” + eval_a0 + “’;’texto’ ”);
9	eval(“eval_a9=‘”+eval_a0+“’; eval_b9=‘”+“texto”+“’; eval_c9=eval_a9”);
10	eval_a10 = eval(“eval_b10=fnc_Tainted_Source;eval_c10=eval_b10()”);

Assumindo que a função *fnc_Tainted_Source* é um *taint source*, os resultados dos testes são apresentados no Quadro 5.3, onde é possível observar o resultado do *taint propagation* durante a análise da função *eval* utilizando a técnica dos *tainted positions*.

Quadro 5.3. Resultado dos testes de propagação do *taint tag* na função *eval*.

Código Base	
eval_a0 = { valor: 'Informação Sensível', tainted: 1 }	
Nº do Teste	**Resultado**
1	(Erro de Síntaxe)
2	eval_a2 = { valor: 'Informação Sensível', tainted: 1 }
3	eval_a3 = { valor: 'Informação Sensível', tainted: 1 }
4	eval_a4 = { valor: 'eval_a0', tainted: 0 }
5	eval_a5 = { valor: 'eval_a0', tainted: 0 }
6	eval_a6 = { valor: 'Informação Sensível', tainted: 1 }
7	eval_a7 = { valor: 'Informação Sensível', tainted: 1 } eval_b7 = { valor: 'Informação Sensível', tainted: 1 }
8	eval_a8 = { valor: 'texto', tainted: 0 }
9	eval_a9 = { valor: 'Informação Sensível', tainted: 1 } eval_b9 = { valor: 'texto', tainted: 0 } eval_c9 = { valor: 'Informação Sensível', tainted: 1 }
10	eval_a10 = { valor: 'Informação Sensível', tainted: 1 } eval_b10 = { valor: [Function: fnc_Tainted_Source], tainted: 1 } eval_c10 = { valor: 'Informação Sensível', tainted: 1 }

O resultado foi satisfatório, atingindo uma acurácia de 100% para os testes realizados para avaliar o método de detecção de propagação do *taint tag* na execução da função *eval*.

5.3.1 Função *eval* + Ofuscação de código + Propagação nos objetos do DOM

Em outro teste, uma aplicação maliciosa foi desenvolvida para realizar o vazamento de informação sensível utilizando a função *eval* somada à prática de ofuscação de código, em uma função *JavaScript* gerada dinamicamente.

Essa aplicação tinha como objetivo o vazamento de dados de geolocalização, enviando essa informação para uma página *web*. Essa página recebia os dados vazados e os armazenava para posterior consulta (Código 5.15).

```
1 <?php
2 $bd = "dados.txt";
3 if(isset($_GET["dados"])){
```

```
4     $dados = $_GET["dados"];
5     $abrir = fopen($bd,"a");
6     $sucesso = fwrite($abrir,$dados."\n\n");
7     fclose($abrir);
8     if($sucesso)
9       echo "Informação sensível capturada com sucesso.";
10  }
11  else{
12    if(file_exists($bd))
13      echo implode("<br>",file($bd));
14    else
15      echo "Nenhuma informação sensível capturada.";
16  }
17  ?>
```

Código 5.15. Código da página *getDados.php*.

Para realizar o vazamento de informação, a aplicação utiliza-se da função *eval* para criar dinamicamente uma função *Leakage* e inserir dados sensíveis em seu escopo como se fossem simples comentários, como apresenta o Código 5.16.

Mesmo que a função *Leakage* não seja um *source tainted* e nem retorne dados sensíveis, ela possui um comentário que contém informação sensível, a qual foi concatenada no argumento da função *eval*. Essa informação, mesmo estando em um comentário, é capturada e enviada a um *taint sink*, neste caso o atributo *src* do objeto *HTMLImageElement*.

```
1  eval("Leakage = function(){ return 10; /*" + getGeoLocation() + "*/ }");
2  pos = (Leakage + "")["split"]("/*")[1]["split"]("*/")[0];
3  (new Image())["src"] = "http://localhost/getDados.php?dados="+pos;
```

Código 5.16. Trecho do código da aplicação maliciosa.

O vazamento de informação ocorre quando o objeto *HTMLImageElement* procura carregar o *bytecode* da imagem (linha 3 do Código 5.16), então a página *getDados.php* é acessada e, consequentemente, a informação sensível é capturada.

O *TaintJSec* foi capaz de detectar o vazamento de informação utilizando os *tainted positions*.

5.4 Testes em Códigos Ofuscados

Em seguida, o Código 5.16 foi criptografado para avaliar a qualidade do *tracking* do *TaintJSec*.

Foram realizados testes utilizando códigos ofuscados a partir de ferramentas como *JSCompress* [75], *Aaencode* [76], *JSFuck* [77], *JSCrambler* [78] e *Packer* [79].

5.4.1 *JSCompress*

A ferramenta *JSCompress* realizou uma minificação no código original, resultando na Figura 5.3. Esse código não deveria apresentar qualquer comentário, devido ao processo de minificação, porém o *JSCompress* não detectou o comentário embutido no argumento da função *eval* e o manteve.

A minificação de código não foi obstáculo para a realização da propagação da *tag*, como é possível observar nas linhas 26, 27 e 28 do Resultado da Propagação a *tag* com o valor positivo.

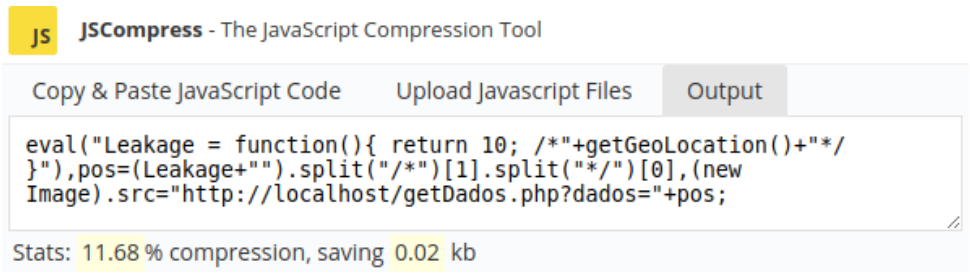


Figura 5.3. Código gerado pelo *JSCompress*.

Resultado da propagação do *taint tag* no *JSCompress*

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   temp_0_7113743252814397 = Variavel {
6    valor: 'function (){ return 10; /*[-3.0880814,-59.9659753]*/ }',
7    tainted: 1 }
8  ||   temp_0_5140872926833178 = Variavel { valor: '[-3.0880814,-59.9659753]*/ }',
       tainted: 1 }
9  ----------------------
10 ==================================
11 ----------------------
12 || Nome: Variaveis Globais
13 || Registradores Globais:
14 ||   Leakage = Variavel {
15   valor:
16    { estrutura:
```

```
17        { range: [Object],
18          type: 'FunctionExpression',
19          id: null,
20          params: [],
21          defaults: [],
22          body: [Object],
23          generator: false,
24          expression: false,
25          tainted_eval: 1 } },
26     tainted: 1 }
27 ||   pos = Variavel { valor: '[-3.0880814,-59.9659753]', tainted: 1 }
28 ||   2temp_0_6720708740652281 = Variavel { valor: HTMLImageElement {}, tainted:
       1 }
29 ----------------------
```

A Figura 5.4 apresenta o resultado da análise de detecção do vazamento de informação, sendo possível observar o alerta no momento da detecção, no nível 4 da AST gerada pelo código.

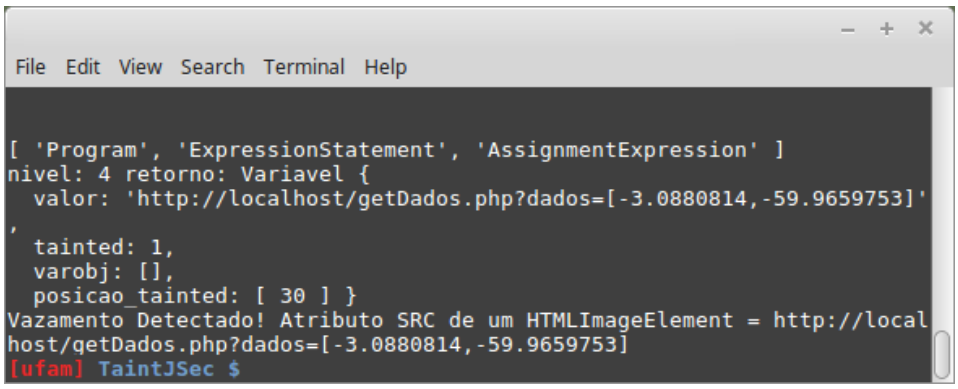


Figura 5.4. Detecção do vazamento de informação no código gerado pelo *JSCompress*.

5.4.2 *Aaencode*

Foi utilizado o *Aaencode* para criptografar o Código 5.16, como mostra a Figura 5.5, então o código gerado foi analisado pelo *TaintJSec*.

Figura 5.5. Código gerado pelo *Aaencode*.

Resultado da propagação do *taint tag* no *Aaencode*

```
----------------------
|| Nome: Variaveis Globais
|| Registradores Globais:
||   0 = Variavel {
  valor:
   { estrutura:
      { range: [Object],
        type: 'FunctionExpression',
        id: [Object],
        params: [],
        defaults: [],
        body: [Object],
        generator: false,
        expression: false } },
  tainted: 0 }
||   ﾟωﾟﾉ = Variavel { valor: undefined, tainted: 0 }
||   _ = Variavel { valor: 3, tainted: 0 }
||   ﾟｰﾟ = Variavel { valor: 4, tainted: 0, args_original: '' }
||   o = Variavel { valor: 3, tainted: 0 }
||   ﾟΘﾟ = Variavel { valor: 1, tainted: 0, args_original: '' }
||   c = Variavel { valor: 0, tainted: 0, args_original: '' }
```

```
22 ||   ﾟДﾟ = Variavel {
23   valor:
24    { 'ﾟΘﾟ': Variavel { valor: 'f', tainted: 0 },
25      'ﾟωﾟﾉ': Variavel { valor: 'a', tainted: 0 },
26      'ﾟｰﾟﾉ': Variavel { valor: 'd', tainted: 0 },
27      'ﾟДﾟﾉ': Variavel { valor: 'e', tainted: 0 },
28      c: Variavel { valor: 'c', tainted: 0 },
29      o: Variavel { valor: 'o', tainted: 0 },
30      _: Variavel { valor: [Function: Function], tainted: 0 },
31      return: Variavel { valor: '\\', tainted: 0 },
32      'ﾟΘﾟﾉ': Variavel { valor: 'b', tainted: 0 },
33      constructor: Variavel { valor: '"', tainted: 0 } },
34   tainted: 0 }
35 ||   ﾟoﾟ = Variavel { valor: 'constructor', tainted: 0,
36   args_original: '' }
37 ||   ﾟεﾟ = Variavel { valor: 'return', tainted: 0, args_original: '' }
38 ||   temp_0_9986435039518833 = Variavel { valor: 1, tainted: 0, args_original: '
     ' }
39 ||   oﾟｰﾟo = Variavel { valor: 'u', tainted: 0 }
40 ||   _0 = Variavel {
41   valor:
42    { estrutura:
43       { range: [Object],
44         type: 'FunctionExpression',
45         id: [Object],
46         params: [],
47         defaults: [],
48         body: [Object],
49         generator: false,
50         expression: false } },
51   tainted: 0 }
52 ||   Leakage = Variavel {
53   valor:
54    { estrutura:
55       { range: [Object],
56         type: 'FunctionExpression',
57         id: [Object],
58         params: [],
59         defaults: [],
60         body: [Object],
61         generator: false,
62         expression: false,
63         tainted_eval: 1 } },
64   tainted: 1 }
65 ||   pos = Variavel { valor: '[-3.0880814,-59.9659753]', tainted: 1 }
66 ||   temp_0_23334639449380967 = Variavel { valor: HTMLImageElement {}, tainted:
     1 }
67 ----------------------
```

É possível observar nas linhas 65 e 66 do resultado de propagação do código ofuscado

pelo *Aaencode* que o *taint tag* propagou normalmente para os demais objetos. Na linha 65 é possivel observar o registrador da variável ***pos*** marcado com *taint tag* positivo. O mesmo ocorre na linha 66 com o objeto *HTMLImageElement*.

Logo, a propagação da *tag* ocorreu mesmo com o código ofuscado e foi possível detectar o vazamento de informação, conforme apresenta a Figura 5.6.

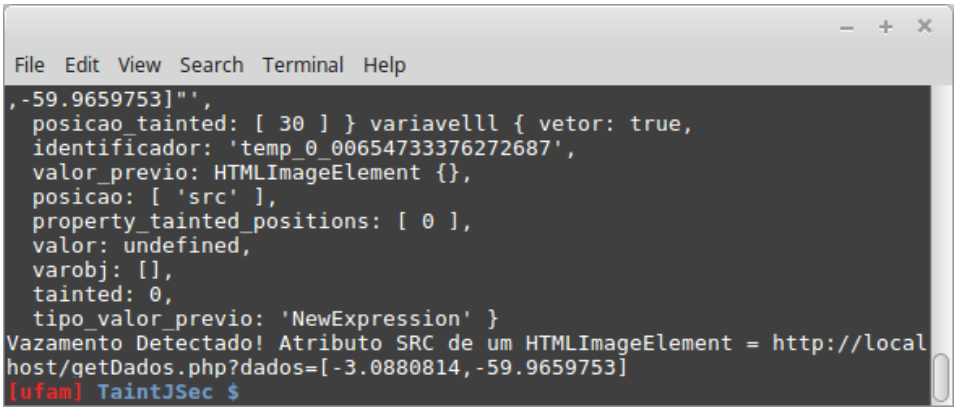


Figura 5.6. Detecção do vazamento de informação no código gerado pelo *Aaencode*.

5.4.3 *JSFuck*

Foi utilizado o *JSFuck* para criptografar o Código 5.16, como mostra a Figura 5.7, então o código gerado foi analisado pelo *TaintJSec*.

Figura 5.7. Código gerado pelo *JSFuck*.

Resultado da propagação do *taint tag* no *JSFuck*

```
----------------------
|| Nível: 0
|| Nome: raiz
|| Registradores Locais:
||   temp_0_9325194618336168 = Variavel {
  valor:
   { estrutura:
      { range: [Object],
        type: 'FunctionExpression',
        id: [Object],
        params: [],
        defaults: [],
```

```
13          body: [Object],
14          generator: false,
15          expression: false } }
16     }
17 ||   temp_0_2633719552855911 = Variavel { valor: 'false0', tainted: 0 }
18 ||   temp_0_82266519121584 = Variavel { valor: 'false0', tainted: 0 }
19 ||   temp_0_6684026363503319 = Variavel { valor: /false/, tainted: 0 }
20 ||   temp_0_33544993146969126 = Variavel { valor: '', tainted: 0 }
21 ||   temp_0_1213166194308728 = Variavel { valor: '', tainted: 0 }
22 ||   temp_0_5123075582114358 = Variavel { valor: 211, tainted: 0 }
23 ||   temp_0_7790923849919331 = Variavel { valor: 20, tainted: 0 }
24 ||   temp_0_7751559646379698 = Variavel { valor: '', tainted: 0 }
25 ||   temp_0_6757265285263963 = Variavel { valor: '', tainted: 0 }
26 ----------------------
27 ==================================
28 ----------------------
29 || Nome: Variaveis Globais
30 || Registradores Globais:
31 ||   Leakage = Variavel {
32   valor:
33    { estrutura:
34       { range: [Object],
35         type: 'FunctionExpression',
36         id: [Object],
37         params: [],
38         defaults: [],
39         body: [Object],
40         generator: false,
41         expression: false,
42         tainted_eval: 1 } },
43   tainted: 1,
44    }
45 ||   pos = Variavel { valor: '[-3.0880814,-59.9659753]', tainted: 1 }
46 ||   temp_0_18928080461112717 = Variavel { valor: HTMLImageElement {}, tainted:
      1 }
47 ----------------------
```

É possível observar nas linhas 45 e 46 do resultado de propagação do código ofuscado pelo *JSFuck* que o *taint tag* propagou normalmente para os demais objetos. Na linha 45 é possivel observar o registrador da variável ***pos*** marcado com *taint tag* positivo. O mesmo ocorre na linha 46 com o objeto *HTMLImageElement*.

Logo, a propagação da *tag* ocorreu mesmo com o código ofuscado e foi possível detectar o vazamento de informação, conforme apresenta a Figura 5.8.

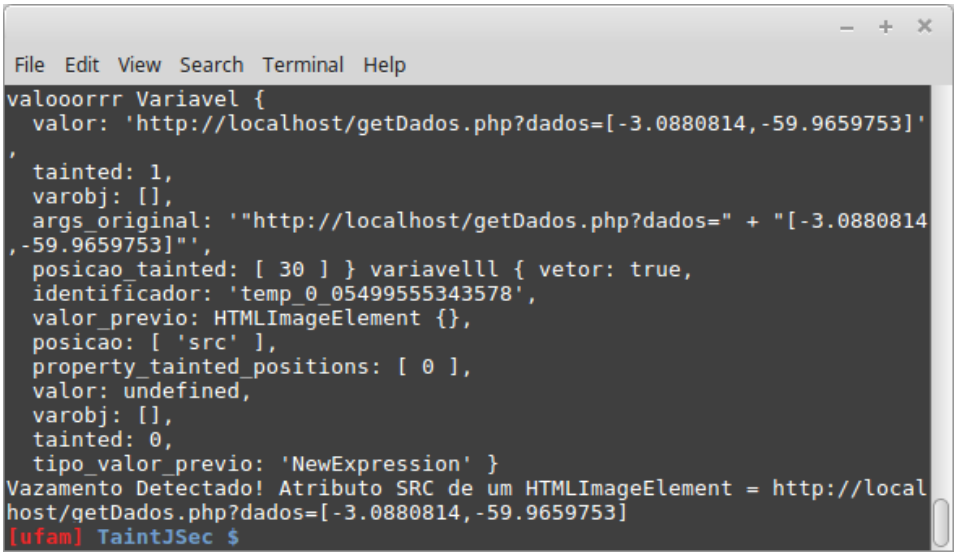


Figura 5.8. Detecção do vazamento de informação no código gerado pelo *JSFuck*.

5.4.4 *JSCrambler*

Foi utilizado o *JSCrambler* para criptografar o Código 5.16, como mostra a Figura 5.9, então o código gerado foi analisado pelo *TaintJSec*.

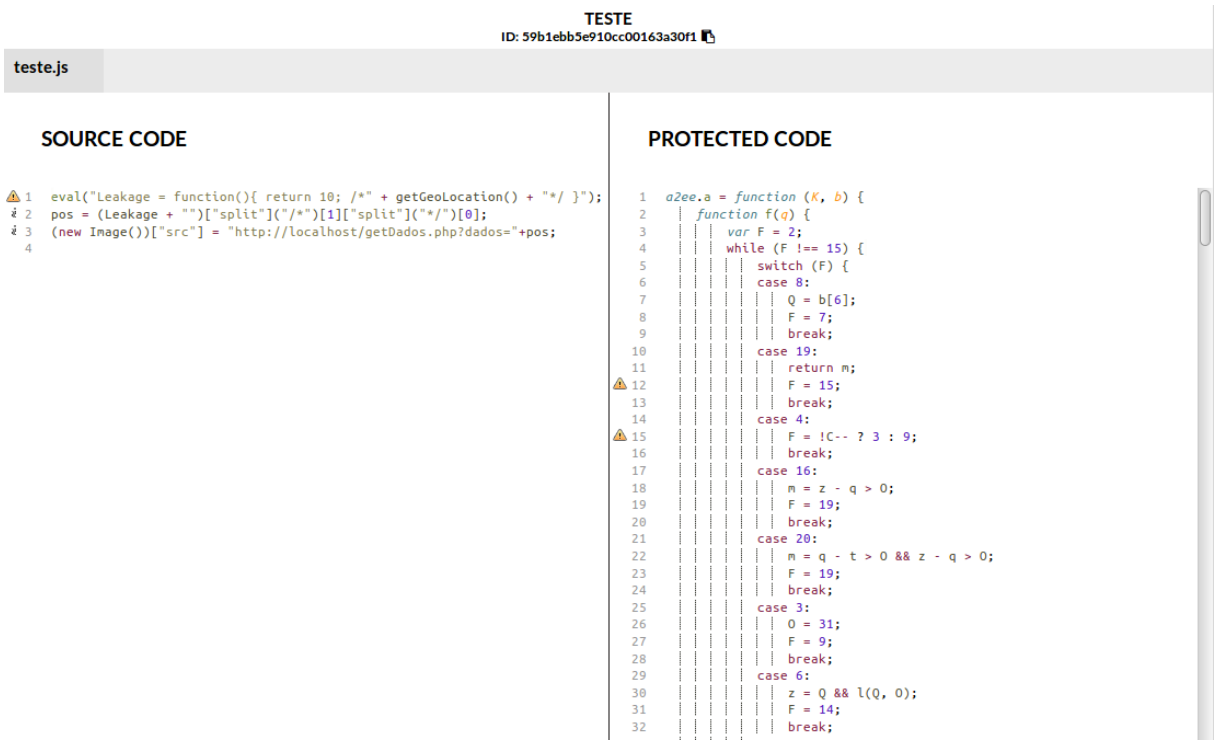


Figura 5.9. Código gerado pelo *JSCrambler*.

Resultado da propagação do *taint tag* no *JSCrambler*

```
----------------------
|| Nível: 0
|| Nome: raiz
|| Registradores Locais:
||   a2ee = Variavel {
  valor:
   { estrutura:
      { range: [Object],
        type: 'FunctionDeclaration',
        id: [Object],
        params: [],
        defaults: [],
        body: [Object],
        generator: false,
        expression: false },
     a: Variavel { valor: [Object], tainted: 0},
     C: Variavel { valor: [Object], tainted: 0},
     b: Variavel { valor: [Object], tainted: 0},
     n: Variavel { valor: [Object], tainted: 0},
     w: Variavel { valor: [Object], tainted: 0},
```

```
21      P: Variavel { valor: [Object], tainted: 0},
22      H: Variavel { valor: [Object], tainted: 0},
23      D: Variavel { valor: [Object], tainted: 0},
24      z: Variavel { valor: [Object], tainted: 0},
25      q: Variavel { valor: [Object], tainted: 0},
26      O: Variavel { valor: [Object], tainted: 0},
27      R: Variavel { valor: [Object], tainted: 0},
28      V: Variavel { valor: [Object], tainted: 0},
29      f: Variavel { valor: [Object], tainted: 0} },
30   tainted: 0}
31 ||   temp_0_13594414804006916 = Variavel {
32   valor: 'function (){ return 10; /*[-3.0880814,-59.9659753]*/ }',
33   tainted: 1}
34 ||   temp_0_9526427394422232 = Variavel { valor: '[-3.0880814,-59.9659753]*/ }',
       tainted: 1}
35 ----------------------
36 ==================================
37 ----------------------
38 || Nome: Variaveis Globais
39 || Registradores Globais:
40 ||   temp_0_9239368118878519 = Variavel { valor: 'D', tainted: 0}
41 ||   temp_0_4615788786109254 = Variavel { valor: 'a', tainted: 0}
42 ||   temp_0_5295130272124962 = Variavel { valor: 't', tainted: 0}
43 ||   temp_0_30889415425710354 = Variavel { valor: 'e', tainted: 0}
44 ||   temp_0_6037332225292458 = Variavel { valor: 'g', tainted: 0}
45 ||   temp_0_7995355004950915 = Variavel { valor: 'e', tainted: 0}
46 ||   temp_0_06361612749153633 = Variavel { valor: 't', tainted: 0}
47 ||   temp_0_7881605166150913 = Variavel { valor: 'T', tainted: 0}
48 ||   temp_0_03634676088582145 = Variavel { valor: 'i', tainted: 0}
49 ||   temp_0_8379682134551547 = Variavel { valor: 'm', tainted: 0}
50 ||   temp_0_360087520650036 = Variavel { valor: 'e', tainted: 0}
51 ||   temp_0_025186848311961052 = Variavel { valor: 'c', tainted: 0}
52 ||   temp_0_2811977517255182 = Variavel { valor: 'h', tainted: 0}
53 ||   temp_0_643917155565777 = Variavel { valor: 'a', tainted: 0}
54 ||   temp_0_07002137401964115 = Variavel { valor: 'r', tainted: 0}
55 ||   temp_0_16914927265105106 = Variavel { valor: 'A', tainted: 0}
56 ||   temp_0_8835132468672098 = Variavel { valor: 't', tainted: 0}
57 ||   temp_0_44022913204821124 = Variavel { valor: 't', tainted: 0}
58 ||   temp_0_37759461723017806 = Variavel { valor: 'o', tainted: 0}
59 ||   temp_0_3138235441878878 = Variavel { valor: 'S', tainted: 0}
60 ||   temp_0_29436043433945613 = Variavel { valor: 't', tainted: 0}
61 ||   temp_0_7736862073660966 = Variavel { valor: 'r', tainted: 0}
62 ||   temp_0_6938402010545426 = Variavel { valor: 'i', tainted: 0}
63 ||   temp_0_18186274263352686 = Variavel { valor: 'n', tainted: 0}
64 ||   temp_0_4458461474768367 = Variavel { valor: 'g', tainted: 0}
65 ||   temp_0_6373474465784135 = Variavel { valor: 'p', tainted: 0}
66 ||   temp_0_9045829635839095 = Variavel { valor: 'a', tainted: 0}
67 ||   temp_0_692755056882038 = Variavel { valor: 'r', tainted: 0}
68 ||   temp_0_3998741783465658 = Variavel { valor: 's', tainted: 0}
69 ||   temp_0_7636223986943256 = Variavel { valor: 'e', tainted: 0}
```

```
70 ||   temp_0_4602688270746973 = Variavel { valor: 'I', tainted: 0}
71 ||   temp_0_1377593961749255 = Variavel { valor: 'n', tainted: 0}
72 ||   temp_0_07437498347618465 = Variavel { valor: 't', tainted: 0}
73 ||   temp_0_03583006417833978 = Variavel { valor: 'l', tainted: 0}
74 ||   temp_0_5461145435583603 = Variavel { valor: 'e', tainted: 0}
75 ||   temp_0_5559148963788878 = Variavel { valor: 'n', tainted: 0}
76 ||   temp_0_1582237025492239 = Variavel { valor: 'g', tainted: 0}
77 ||   temp_0_24430862277791876 = Variavel { valor: 't', tainted: 0}
78 ||   temp_0_18796976879167593 = Variavel { valor: 'h', tainted: 0}
79 ||   temp_0_8837578511558091 = Variavel { valor: '1', tainted: 0}
80 ||   temp_0_5201395424910498 = Variavel { valor: 'n', tainted: 0}
81 ||   temp_0_9200617068469255 = Variavel { valor: 'm', tainted: 0}
82 ||   temp_0_17771213569681255 = Variavel { valor: 'a', tainted: 0}
83 ||   temp_0_686796733742566 = Variavel { valor: 't', tainted: 0}
84 ||   temp_0_297518540469065 = Variavel { valor: 'm', tainted: 0}
85 ||   temp_0_30981652108573665 = Variavel { valor: 't', tainted: 0}
86 ||   temp_0_2957813046427411 = Variavel { valor: '4', tainted: 0}
87 ||   temp_0_6500056869824056 = Variavel { valor: 'a', tainted: 0}
88 ||   Leakage = Variavel {
89   valor:
90    { estrutura:
91       { range: [Object],
92         type: 'FunctionExpression',
93         id: [Object],
94         params: [],
95         defaults: [],
96         body: [Object],
97         generator: false,
98         expression: false,
99         tainted_eval: 1 } },
100  tainted: 1}
101 ||   pos = Variavel { valor: '[-3.0880814,-59.9659753]', tainted: 1}
102 ||   temp_0_559489344676888 = Variavel { valor: HTMLImageElement {}, tainted: 1}
103 ----------------------
```

É possível observar nas linhas 101 e 102 do resultado de propagação do código ofuscado pelo *JSCrambler* que o *taint tag* propagou normalmente para os demais objetos. Na linha 101 é possivel observar o registrador da variável ***pos*** marcado com *taint tag* positivo. O mesmo ocorre na linha 102 com o objeto *HTMLImageElement*.

Logo, a propagação da *tag* ocorreu mesmo com o código ofuscado e foi possível detectar o vazamento de informação, conforme apresenta a Figura 5.10.

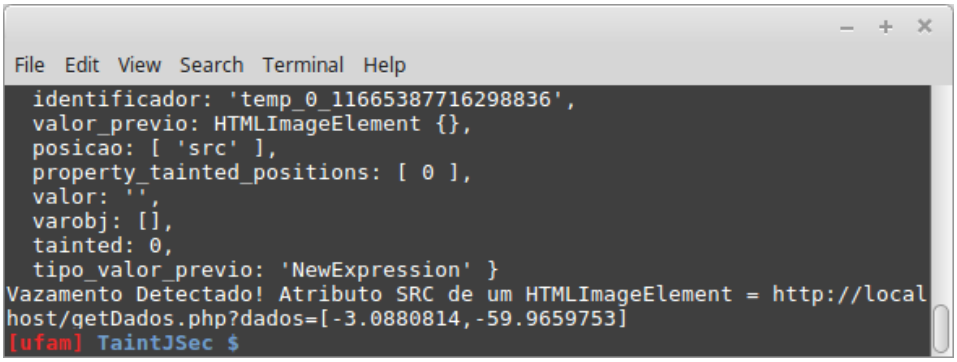


Figura 5.10. Detecção do vazamento de informação no código gerado pelo *JSCrambler*.

5.4.5 *Packer*

Foi utilizado o *Packer* para criptografar o Código 5.16, como mostra a Figura 5.11, então o código gerado foi analisado pelo *TaintJSec*.

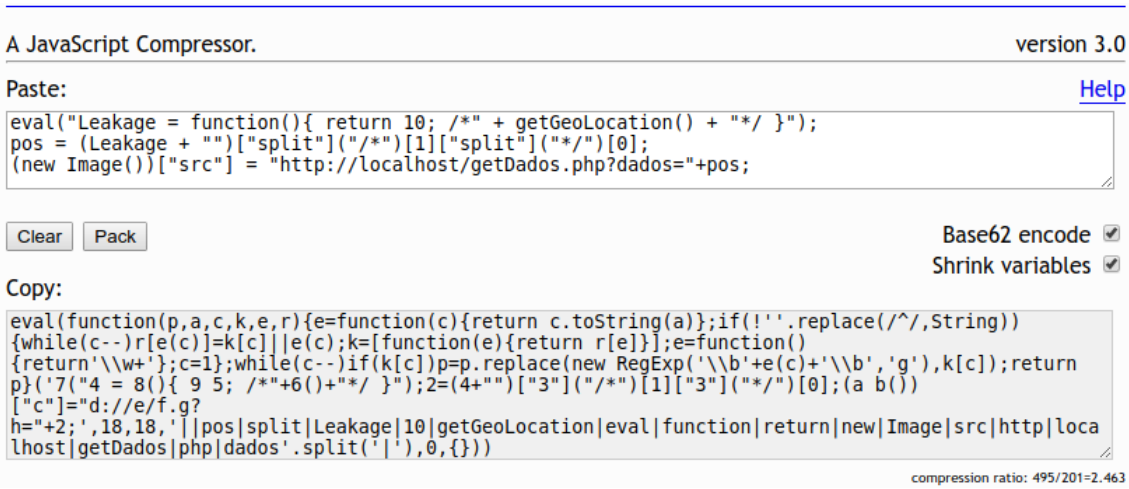


Figura 5.11. Código gerado pelo *Packer*.

Resultado da propagação do *taint tag* no *Packer*

```
1  ----------------------
2  || Nível: 0
3  || Nome: raiz
4  || Registradores Locais:
5  ||   temp_0_5953198160603002 = Variavel {
6    valor: '||pos|split|Leakage|10|getGeoLocation|eval|function|return|new|Image|
         src|http|localhost|getDados|php|dados',
7    tainted: 0}
8  ||   temp_0_6198358132507966 = Variavel {
9    valor: 'function (){ return 10; /*[-3.0880814,-59.9659753]*/ }',
10   tainted: 1}
11 ||   temp_0_451124834576798 = Variavel { valor: '[-3.0880814,-59.9659753]*/ }',
       tainted: 1}
12 ----------------------
13 ====================================
14 ----------------------
15 || Nome: Variaveis Globais
16 || Registradores Globais:
17 ||   temp_0_367289211359239 = Variavel {
18   valor:
19    { estrutura:
20       { range: [Object],
21         type: 'FunctionExpression',
22         id: null,
23         params: [Object],
24         defaults: [],
```

```
25          body: [Object],
26          generator: false,
27          expression: false } },
28    tainted: 0}
29  ||    Leakage = Variavel {
30    valor:
31     { estrutura:
32        { range: [Object],
33          type: 'FunctionExpression',
34          id: [Object],
35          params: [],
36          defaults: [],
37          body: [Object],
38          generator: false,
39          expression: false,
40          tainted_eval: 1 } },
41    tainted: 1}
42  ||    pos = Variavel { valor: '[-3.0880814,-59.9659753]', tainted: 1}
43  ||    temp_0_7206422871497422 = Variavel { valor: HTMLImageElement {}, tainted:
       1}
44  ----------------------
```

É possível observar nas linhas 42 e 43 do resultado de propagação do código ofuscado pelo *Packer* que o *taint tag* propagou normalmente para os demais objetos. Na linha 42 é possivel observar o registrador da variável ***pos*** marcado com *taint tag* positivo. O mesmo ocorre na linha 43 com o objeto *HTMLImageElement*.

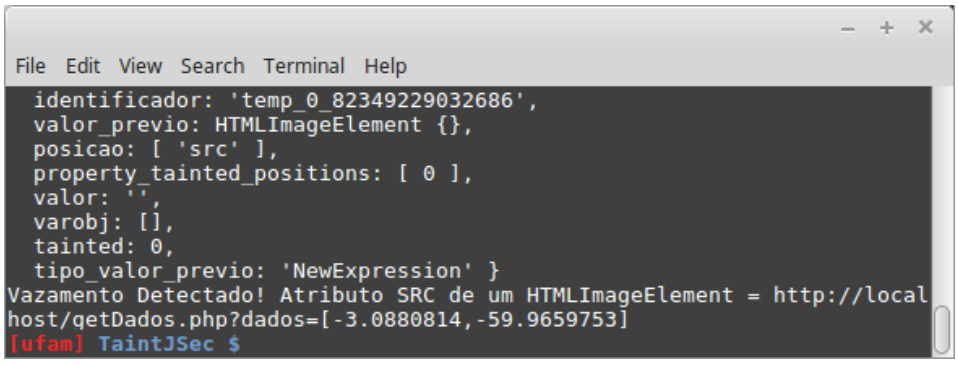


Figura 5.12. Detecção do vazamento de informação no código gerado pelo *Packer*.

Logo, a propagação da *tag* ocorreu mesmo com o código ofuscado e foi possível detectar o vazamento de informação, conforme apresenta a Figura 5.12.

Capítulo 6

Conclusão

Este capítulo apresenta as conclusões deste trabalho de pesquisa, as limitações e as contribuições para a comunidade científica. Além de possíveis trabalhos futuros, que visam sanar limitações da análise.

6.1 Considerações Finais

Este trabalho apresentou o *TaintJSec*, uma abordagem de análise estática, para a detecção de vazamento de informação sensível em código *JavaScript*. Diferente das abordagens existentes, o *TaintJSec* é capaz de verificar código *JavaScript* explícitos e implícitos. Mais ainda, o *TaintJSec* consegue analisar a propagação de um dado sensível (*taint data*) na execução da função *eval*, a qual é bastante utilizada em técnicas de ofuscação de código. Além de conseguir identificar o vazamento de informação sensível em códigos ofuscados por ferramentas criadas especificamente para tal finalidade.

A avaliação do *TaintJSec* utilizando 14 conjuntos de testes mostraram que a acurácia da abordagem proposta é superior às ferramentas *JPrime*, *ScanJS*, *JSpwn*, *KSLint*, *Jalangi* e *jsTaint*.

Entretanto, apesar da abordagem *TaintJSec* ter obtido bons resultados, há muitas questões a serem melhoradas, como por exemplo:

- não verifica todos os caminhos possíves do fluxo de execução. De tal modo que uma operação condicionante *IfStatement* checa somente o escopo verdadeiro se a condição for verdadeira, ou checa o escopo falso se a condição for falsa;
- não analisa o comportamento dos *Event Listeners*. Logo, funções executadas por meio de eventos são ignoradas;
- não tem suporte a *callbacks* como o objeto *new Proxy*;

- não tem suporte a função geradora;
- não tem suporte ao modo estrito (diretiva *"use strict"*).
- não tem suporte a códigos *JavaScript* com comportamento assíncrono.

6.2 Lições Aprendidas

Durante a implementação do *TaintJSec* surgiram alguns problemas, os quais foram sanados após nova fase de planejamento e pequisa. Esses problemas apresentavam-se como limitações da plataforma *Node.js*, limitações nos módulos escolhidos para as fases da abordagem ou alguma característica da linguagem *JavaScript* que não havia sido prevista.

Nesta seção são descritos os problemas enfrentados durante este trabalho de pesquisa e as medidas utilizadas para contorná-los.

Limitações do *Node.js*

O microprocessador possui grande parte das funções nativas do *JavaScript* e, por isso dispensa o trabalho de criação das mesmas. Porém, não fornece suporte às funções e métodos nativos presentes nos navegadores.

No mundo real, a possibilidade de encontrar um código *JavaScript* que não utilize alguma função ou método que só existem nos navegadores é muito baixa. São exemplo de objetos que o *Node.js* não oferece suporte são Audio e Video.

Para contornar essas limitações foram utilizados os objetos *document* e *window* fornecidos pelo módulo jsDom. Desse modo, foi possível identificar a existência de métodos exclusivos do HTML e não da linguagem *JavaScript*. Para o objeto Audio foi utilizado o objeto *window.Audio*.

Contudo, os objetos que não eram fornecidos ou emulados pelos módulos do *Node.js* ainda causavam problemas durante a análise, como o objeto *navigator.userAgent*, que não é fornecido pela versão mais recente do *Node.js* e nem pelo módulo jsDom. Para resolver esse problema, tais objetos podem ser emulados por meio da criação de um objeto *navigator* com o atributo *userAgent* e um valor literal fixo.

Cheerio x jsDom

Cheerio[1] é um módulo do *Node.js*, que fornece um subconjunto de operações da biblioteca *jQuery* possibilitando o acesso ao DOM de códigos HTML mesmo em *server-side*.

Foi desenvolvido como uma alternativa ao módulo jsDom e, segundo seus criadores, é até oito vezes mais rápido que o jsDom. Porém, objetos criados dinamicamente a partir

[1] https://github.com/cheeriojs/cheerio

de operações como *document.createElement* não são adicionados ao DOM gerado pelo Cheerio e, devido a isso, ocorreram erros durante a análise de código. Essa limitação tornou inviável a utilização desse módulo durante a implementação do *TaintJSec*. Em substituição, foi utilizado o jsDom.

Contexto de Escopo em *JavaScript*

A linguagem *JavaScript* executa um contexto de escopo em dois tempos. No primeiro momento são verificadas todas as possíveis declarações de variáveis e funções. Para cada operação de declaração de variável é atribuído o valor *undefined* à variável, e toda declaração de função criada pelo usuário é interpretada, mas não executada.

A Figura 6.1 apresenta um exemplo de código *JavaScript* e os dois momentos da leitura do código. No primeiro momento, é interpretada somente a operação de declaração de variável na linha 3, mas sem a atribuição. O valor adicionado para a variável b é *undefined*.

No segundo momento, as demais operações são interpretadas. Então, a operação de atribuição é executada e o valor de b é atribuído à variável a.

Um programador desatento poderia imaginar que o código apresentaria erro devido a atribuição da linha 1, onde a variável b não existe. Contudo, a linha 1 só é executada no segundo momento da interpretação das operações do escopo, quando a variável b passaria a existir após o primeiro momento da interpretação das operações.

	Os momentos da execução de um escopo	
Código *JavaScript*	1º Momento	2º Momento
1 `<script>`		
2 `a = b;`	linha 2 é ignorada	a = undefined; (valor de b)
3 `var b = 2;`	var b = undefined;	var b = 2;
4 `</script>`		

Figura 6.1. Exemplo de código *JavaScript* e os dois momentos de execução do escopo.

Para implementar essa característica da linguagem *JavaScript*, o framework SAFE realiza uma tarefa de organização da estrutura do código de entrada, colocando as operação de declaração de função e declaração de variável antes de qualquer outra operação do código. Somente quando o código está reestruturado é que o analisador do SAFE é executado. A Figura 6.2 apresenta um exemplo de como o SAFE reescreve o código *JavaScript*, movendo todas as operações de declaração de função para o início do código, e abaixo delas insere as operações de declaração de variável sem a atribuição de valores.

Código Original	Código Reestruturado
1 x; 2 var x = f(); 3 function f() { return 42; }	1 function f() { return 42; } 2 var x; 3 x; 4 x = f();

Figura 6.2. Exemplo de um código *JavaScript* e a sua versão reestruturada pelo SAFE.

A coluna Código Original da Figura 6.2 apresenta na linha 2 e linha 3 operações de declaração de variável e função, respectivamente. Essas operações são movidas para o início do código *JavaScript* formando um código reestruturado. Primeiro a declaração de função é inserida na linha 1 do Código Reestruturado e logo em seguida, na linha 2 é inserida a operação de declaração de variável, sem o valor de atribuição. Com o código reestruturado, o SAFE pode analisar o fluxo de execução do código original da mesma maneira como um interpretador de *JavaScript* do mundo real o faria.

O *TaintJSec* utiliza uma abordagem diferente do SAFE para implementar essa execução de escopo em dois momentos. Em vez de reestruturar o código original, o *TaintJSec* analisa a AST duas vezes. Na primeira análise da AST, somente as declarações de função e variável são inseridas nos registradores. Os registradores armazenam as variáveis declaradas e atribuem o valor *undefined* a elas. Todas as demais operações da AST não são inseridas nos registradores, mas são analisadas para que novas operações de declaração de variável possam ser encontrada dentro do escopo de operações como *IF*, *Try-Catch* e laços de repetição.

Quando a primeira análise da AST chega ao fim, os registradores de objetos contém todas as funções declaradas no escopo e todas as variáveis declaradas estão com o valor *undefined*. Então é iniciada a segunda análise da AST, contabilizando e inserindo os valores obtidos nas demais operações.

Assim, a característica da linguagem *JavaScript* de executar o contexto duas vezes é respeitada e todas as computações ocorrem corretamente.

6.3 Trabalhos Futuros

Neste trabalho foram identificadas as seguintes possibilidades de trabalhos futuros:

1. A inclusão completa do DOM na análise realizada, podendo reconhecer métodos próprios dos navegadores, os quais não existem por padrão no *Node.js* e nem em módulos adicionais, tais como *navigator.userAgent*;

2. Estender a *taint propagation* pelos objetos do HTML para permitir a marcação de *tags* HTML que possuam dados sensíveis no valor de pelo menos um de seus atributos ou em seu conteúdo *innerHTML*;

3. A criação de um controle de *taint tags* para os objetos do tipo *array*, pois atualmente existe somente um *taint tag* para o objeto inteiro, de tal maneira que esse controle poderia observar a propagação do *taint tag* por posição do objeto, diminuindo o número de falsos positivos.

Referências Bibliográficas

[1] J. Ye, C. Zhang, L. Ma, H. Yu, and J. Zhao. Efficient and Precise Dynamic Slicing for Client-Side JavaScript Programs. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 449–459, March 2016.

[2] A. M. Fard and A. Mesbah. JavaScript: The (Un)Covered Parts. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 230–240, March 2017.

[3] Stack Overflow. Developer Survey. https://insights.stackoverflow.com/survey/2017, 2016.

[4] Sora Bae. Concolic Testing with Static Analysis for JavaScript Applications. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 7–8, New York, NY, USA, 2014. ACM.

[5] RedMonk. The Developer-Docused Industry Analyst Firm. http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/, 2017.

[6] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An Empirical Study of Client-Side JavaScript Bugs. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 55–64, Oct 2013.

[7] Mozilla Foundation. WebExtensions | MDN. https://developer.mozilla.org/pt-BR/Add-ons/WebExtensions. Acessado: 07/12/2017.

[8] Google LLC. What Are Extensions? https://developer.chrome.com/extensions. Acessado: 07/12/2017.

[9] Microsoft Corporation. WinJS - A Windows Library for JavaScript. http://www.buildwinjs.com/, 2014.

[10] Cheng Zhao. Electron - Build Cross Platform Desktop Apps with JavaScript, HTML and CSS. https://electronjs.org. Acessado: 07/12/2017.

[11] Node.js Foundation. Node.js. https://nodejs.org. Acessado: 14/05/2016.

[12] Linux Foundation. Tizen - An Open sSource, Standards-based Software Platform for Multiple Device Categories. https://www.tizen.org/, 2012. Acessado: 07/12/2017.

[13] S. Peneti and B. P. Rani. Data leakage Prevention System with Time Stamp. In *2016 International Conference on Information Communication and Embedded Systems (ICICES)*, pages 1–4, Feb 2016.

[14] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] B. S. Pinto, F. C. B. Boeira, P. Minatel, P. C. Pires, I. Souza, A. Silva, and J. Shin. Mobile Data Leakage Prevention using Packet Inspection Approach.

[16] H. Kuzuno and S. Tonami. Signature Generation for Sensitive Information Leakage in Android Applications. In *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, pages 112–119, April 2013.

[17] Dapeng Wang, Guang Jin, Jiaming He, Xianliang Jiang, and Zhijun Xie. A Grey List-Based Privacy Protection for Android. *JSW*, 9:1525–1531, 2014.

[18] Shuen Wen Hsiao, Shih-Hao Hung, Roger Chien, and Chih Wei Yeh. PasDroid: Real-Time Security Enhancement for Android. *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 229–235, 2014.

[19] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132, New York, NY, USA, 2014. ACM.

[20] A. M. Fard and A. Mesbah. JSNOSE: Detecting JavaScript Code Smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, Sept 2013.

[21] Jacques A. Pienaar and Robert Hundt. JSWhiz: Static Analysis for JavaScript Memory Leaks. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[22] Y. Ko, H. Lee, J. Dolby, and S. Ryu. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 541–551, Nov 2015.

[23] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type Refinement for Static Analysis of JavaScript. *SIGPLAN Not.*, 49(2):17–26, October 2013.

[24] D. Liu, H. Wang, and A. Stavrou. Detecting Malicious JavaScript in PDF through Document Instrumentation. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 100–111, June 2014.

[25] Z. Zhao and F. C. Colon Osono. TrustDroid: Preventing the Use of SmartPhones for Information Leaking in Corporate Networks Through the Used of Static Analysis Taint Tracking. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 135–143, Oct 2012.

[26] Z. Yang and M. Yang. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *2012 Third World Congress on Software Engineering*, pages 101–104, Nov 2012.

[27] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[28] Per Christensson. "JavaScript Definition", TechTerms. Sharpened Productions. https://techterms.com/definition/javascript, 2014.

[29] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* ECMA (European Association for Standardizing Information and Communication Systems), 5.1 edition, June 2011.

[30] David Flanagan and Paula Ferguson. *JavaScript: O Guia Definitivo.* O Reilly, 4 edition, 2004.

[31] B. Carter. HTML Architecture, a Novel Development System (HANDS): An Approach for Web Development. In *2014 Annual Global Online Conference on Information and Computer Technology*, pages 90–95, Dec 2014.

[32] W3C. JavaScript Specification. http://www.w3.org/standards/webdesign/script, 2016.

[33] Pedro Teixeira. *Professional Node.Js: Building JavaScript Based Scalable Software*. Wrox Press Ltd., Birmingham, UK, UK, 1st edition, 2012.

[34] Ismail Adel AL-Taharwa, Hahn-Ming Lee, Albert B. Jeng, Kuo-Ping Wu, Cheng-Seen Ho, and Shyi-Ming Chen. JSOD: JavaScript Obfuscation Detector. *Security and Communication Networks*, 8(6):1092–1107, 2015.

[35] M. Jodavi, M. Abadi, and E. Parhizkar. Jsobfusdetector: A binary pso-based one-class classifier ensemble to detect obfuscated javascript code. In *2015 The International Symposium on Artificial Intelligence and Signal Processing (AISP)*, pages 322–327, March 2015.

[36] Jornal Nacional. Imprensa americana diz que ataque à Sony partiu do governo norte-coreano. http://g1.globo.com/jornal-nacional/noticia/2014/12/imprensa-americana-diz-que-ataque-sony-partiu-do-governo-norte-coreano.html. Acessado: 09/12/2017.

[37] P. Papadimitriou and H. Garcia-Molina. Data Leakage Detection. *IEEE Transactions on Knowledge and Data Engineering*, 23(1):51–63, Jan 2011.

[38] UOL Notícias. Dados de 57 milhões de usuários do Uber foram expostos. https://tecnologia.uol.com.br/noticias/redacao/2017/11/21/uber-ocultou-roubo-de-dados-de-57-milhoes-de-usuarios-por-hackers-diz-site.htm. Acessado: 09/12/2017.

[39] Marco A. Maia. BYOD: Produtividade X Segurança. http://segurancadainformacao.modulo.com.br/o-que-e-byod. Acessado: 09/12/2017.

[40] S. Morales-Ortega, P. J. Escamilla-Ambrosio, A. Rodriguez-Mota, and L. D. Coronado-De-Alba. Native malware detection in smartphones with android os using static analysis, feature selection and ensemble classifiers. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–8, Oct 2016.

[41] Tejas Saoji. Implementing Dynamic Coarse & Fine Grained Taint Analysis for Rhino JavaScript. http://scholarworks.sjsu.edu/etd_projects/519, 2017.

[42] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, May 2012.

[43] Suan Hsi Yong and Susan Horwitz. Using Static Analysis to Reduce Dynamic Analysis Overhead. *Formal Methods in System Design*, 27(3):313–334, Nov 2005.

[44] A. S. Novikov, A. N. Ivutin, A. G. Troshina, and S. N. Vasiliev. The Approach to Finding Errors in Program Code Based on Static Analysis Methodology. In *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, June 2017.

[45] António Cardoso Soares. Asund: Solução de classificação estática em Node.js para aplicações JavaScript. Master's thesis, Faculdade de Engenharia da Universidade do Porto, Porto, Portugal, 2017.

[46] M. Alawairdhi. Static Analysis Based Business Logic Modelling from Legacy System Code: Business Process Model Notation (BPMN) Extraction Using Abstract Syntax Tree (AST). In *2015 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6, May 2015.

[47] Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. Code Comparison System based on Abstract Syntax Tree. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, pages 668–673, Oct 2010.

[48] H. Kikuchi, T. Goto, M. Wakatsuki, and T. Nishino. A source code plagiarism detecting method using alignment with abstract syntax tree elements. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6, June 2014.

[49] F. M. Lazar and O. Banias. Clone detection algorithm based on the Abstract Syntax Tree approach. In *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 73–78, May 2014.

[50] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[51] G. Blanc, D. Miyamoto, M. Akiyama, and Y. Kadobayashi. Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts. In *2012 26th International Conference on Advanced Information Networking and Applications Workshops*, pages 344–351, March 2012.

[52] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZZLE: Fast and Precise In-browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[53] Ariya Hidayat. ECMAScript Parsing Infrastructure for Multipurpose Analysis. http://esprima.org. Acessado: 14/05/2016.

[54] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. *CoRR*, abs/1404.7431, 2014.

[55] Mozilla Foundation. SpiderMonkey JavasScript Engine. https://developer.mozilla.org/pt-BR/docs/Mozilla/Projects/SpiderMonkey. Acessado: 18/09/2017.

[56] Martin Odersky. The Scala Programming Language. http://www.scala-lang.org. Acessado: 18/09/2017.

[57] Mozilla Foundation. Rhino. https://developer.mozilla.org/pt-BR/docs/Mozilla/Projects/Rhino. Acessado: 18/09/2017.

[58] PLRG at KAIST. Scalable Analysis Framework for ECMAScript (SAFE) Version 2.0. http://plrg.kaist.ac.kr/. Acessado: 18/09/2017.

[59] Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. Analysis of JavaScript Web Applications Using SAFE 2.0. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 59–62, Piscataway, NJ, USA, 2017. IEEE Press.

[60] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript, 2012.

[61] Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/Main_Page. Acessado: 21/09/2017.

[62] Sunspider JavaScript Benchmark. https://webkit.org/perf/sunspider/sunspider.html. Acessado: 21/09/2017.

[63] Octane JavaScript Benchmark. https://developers.google.com/octane. Acessado: 21/09/2017.

[64] LINQ for JavaScript. https://jslinq.codeplex.com. Acessado: 21/09/2017.

[65] Emscripten LLVM. http://kripken.github.io/emscripten-site. Acessado: 21/09/2017.

[66] Nishant Das Patnaik and Sarathi Sabyasachi Sahoo. JSPrime - A JavaScript Static Security Analysis Tool. https://github.com/dpnishant/jsprime.

[67] Paul Theriault. ScanJS - Static Analysis Tool for JavaScript Code. https://github.com/pauljt/scanjs.

[68] Duarte Monteiro, Nishant Das Patnaik, and Paul Theriault. JSpwn - JavaScript Static Code Analysis. https://github.com/dvolvox/JSpwn.

[69] Domenic Denicola. JSDOM - A JavaScript Implementation of the DOM and HTML Standards. http://www.npmjs.com/package/jsdom. Acessado: 11/05/2016.

[70] Bear Bibeault and Yehuda Katz. *jQuery in Action, Third Edition*. Manning Publications Co., Greenwich, CT, USA, 3rd edition, 2015.

[71] D. R. Baquero. VM (Executing JavaScript) | Node.js Documentation. https://nodejs.org/api/vm.html#vm_vm_runinnewcontext_code_sandbox_options. Acessado: 14/05/2016.

[72] Simon Lydell. Eslump - CLI Tool for Fuzz Testing JavaScript Parsers and Suchlike Programs. http://www.npmjs.com/package/eslump. Acessado: 20/02/2017.

[73] Alireza Gorji and Mahdi Abadi. Detecting Obfuscated JavaScript Malware Using Sequences of Internal Function Calls. In *Proceedings of the 2014 ACM Southeast Regional Conference*, ACM SE '14, pages 64:1–64:6, New York, NY, USA, 2014. ACM.

[74] Prakasam Kannan. Taint and Information Flow Analysis Using Sweet. js Macros, 2016.

[75] CircleCell. JSCompress - The JavaScript Compression Tool. https://jscompress.com. Acessado: 11/02/2017.

[76] Yosuke Hasegawa. Aaencode - Encode Any JavaScript Program to Japanese Style Emoticons. http://utf-8.jp/public/aaencode.html. Acessado: 01/06/2016.

[77] M. Kleppe. JSFuck - Write any JavaScript with 6 Characters: []()!+. http://www.jsfuck.com. Acessado: 01/02/2016.

[78] Antônio P. F. F. dos Santos, Rui M. S. Ribeiro, and Filipe M. G. Silva. Jscrambler - Make Your JavaScript Application Protect Itself. https://jscrambler.com. Acessado: 01/06/2017.

[79] Dean Edwards. Packer - A JavaScript Compressor 3.0. http://dean.edwards.name/packer. Acessado: 15/09/2017.