Universidade Federal do Amazonas
Instituto de Computação
Programa de Pós-Graduação em Informática

Geandro Farias de Matos

Identificando Indicadores de Browser Fingerprinting em Páginas Web

Manaus
Julho de 2021

Geandro Farias de Matos

Identificando Indicadores de Browser Fingerprinting em Páginas Web

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

Orientador: Prof. Dr. Eduardo Luzeiro Feitosa

Manaus
Julho de 2021

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Matos, Geandro Farias de
M433i Identificando indicadores de browser fingerprinting em páginas web / Geandro Farias de Matos . 2021
127 f.: il. color; 31 cm.

Orientador: Eduardo Luzeiro Feitosa
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Website Fingerprinting. 2. Rastreamento. 3. Análise Estática. 4. Javascript. I. Feitosa, Eduardo Luzeiro. II. Universidade Federal do Amazonas III. Título

FOLHA DE APROVAÇÃO

"Identificando Indicadores de Browser Fingerprinting em Páginas Web"

GEANDRO FARIAS DE MATOS

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

Prof. Eduardo Luzeiro Feitosa - PRESIDENTE

Prof. Rafael Roque Aschoff - MEMBRO EXTERNO

Prof. Carlo Marcelo Revoredo da Silva - MEMBRO EXTERNO

Manaus, 23 de Julho de 2021

Dedico esta dissertação a minha mãe Maria Auxiliadora Farias de Matos, um Jequitibá-rosa que nunca vergou diante das maiores tempestades que um dia ameaçaram seus galhos e raízes.

Agradecimentos

Primeiramente agradeço a Deus pelo dom da vida e por ter me concedido a oportunidade de concluir este trabalho diante de todas as tormentas e desafios que passei até chegar ao final dessa jornada.

Quero agradecer à minha família que sempre esteve ao meu lado e não me permitiu sucumbir diante dos desafios enfrentados.

À minha esposa e companheira amada Auxiliadora Fonseca Machado por segurar a minha mão quando minhas pernas não tinham forças para levantar e caminhar.

Ao meu filho amado, Murilo Fonseca de Matos, por compartilhar comigo as alegrias, as angústia e os risos de felicidade ao ver a WBF Analyzer dando os primeiros passos.

À minha irmã querida a Dr. Jorcemara Cardoso e ao meu cunhado Dr. Celso Ricardo por todos os aconselhamentos e por todas as orientações que me ajudaram a trilhar meu caminho com fé e perseverança.

Ao meu irmão Marcelo Cardoso por mostra-se sempre disponível a me ajudar nos momentos em que precisei.

Aos meus amigos poetas e compositores Rozinaldo Carneiro e Naferson Cruz por me estenderem a mão em momentos de dificuldades particulares.

Ao meu amigo poeta e compositor Demétrios Haidos por ser um companheiro de longas jornadas que sempre esteve ao meu lado com uma palavra de incentivo.

Às minhas queridas Diretora e Coordenadora Administrativa, Sandra Helena da Silva e Daniele Canto Hagra, por todo suporte em um momento de puro desespero e muita dor onde elas me ofertaram o seu bondoso coração.

Aos professores doutores José Luiz pereira da Fonseca e Ruiter Braga Caldas pela oportunidade de ingressar no mestrado e principalmente pela confiança de permanecer e fazer um excelente trabalho.

Ao Prof. Dr. Eduardo J. P. Souto pela aulas de metodologia científica que nortearam muitos aspectos da metodologia aplicada a este trabalho.

À minha cunhada Regina Carmem F. Machado por me prover abrigo quando estive longe do meu lar para realizar essa conquista.

Aos meus colegas do laboratório ETSS (*Emerging Tecnologies and Systems Security*) por todo companheirismo, compartilhamento de conhecimentos e orações em todos os momentos alegres e difíceis que passamos juntos.

Ao Dr. Francisco Marinho um amigo ímpar que esteve ao meu lado sempre me incentivando e foi de grande importância para a realização dessa conquista.

E para este final deixo um agradecimento muito especial e carinhoso ao meu orientador o Prof. Dr. Eduardo L. Feitosa um ser humano incrível de sorriso fácil que aprendi a amar e a admirar e que me ajudou com toda força e sabedoria a chegar ao primeiro lugar do pódio (Aprovação de minha defesa de mestrado).

“A coleta automatizada de dados é tão antiga quanto a própria internet.” .

(Ryan Mitchell).

Resumo

Técnicas de *fingerprinting* são aquelas empregadas para identificar (ou re-identificar) um usuário ou um dispositivo através de um conjunto de atributos (tamanho da tela do dispositivo, versões de softwares instalados, entre muitos outros) e outras características observáveis durante o processo de comunicação. Comumente conhecidas por *Browser Fingerprinting*, tais técnicas podem ser usadas como medida de segurança (na autenticação de usuários, por exemplo) e como mecanismo para vendas / marketing. Por outro lado, também podem ser consideradas uma ameaça potencial à privacidade Web dos usuários, uma vez que dados pessoais e sigilosos podem ser capturados e empregados para fins maliciosos nos mais variados tipos de ataque e fraudes. Neste contexto, esta dissertação propõe um método para detectar chamadas *fingerprinting* aos objetos fornecedores de informação do Javascript em páginas web e mensurar o nível de severidade à privacidade do usuário. O método foi avaliado em diferentes bases de páginas web (novas e outras utilizadas por outros trabalhos). Os resultados provam que trabalhos anteriores geravam muitos falsos positivos e eram incapazes de detectar um conjunto de objetos JavaScript formado por *new Date().getTime(), this.width, this.height navigator.userAgent, document.cookie* relacionados a *Browser Fingerprinting*.

Palavras-chave: Website Fingerprinting, rastreamento, análise estática, JavaScript.

Abstract

Fingerprinting techniques are those employed to identify (or re-identify) a user or device through a set of attributes (device screen size, installed software versions, and many others) and other observable characteristics during the communication process. Commonly known as website fingerprinting, such techniques can be used as a security measure (for user authentication, for example) and as a sales/marketing mechanism. On the other hand, they can also be considered a potential threat to users' web privacy since sensitive and personal data can be captured and used for malicious purposes in various attacks and fraud types. In this context, this dissertation proposes a method to detect fingerprinting calls to JavaScript information provider objects on web pages and to measure the level of disrespect to user privacy. We evaluated the method in different web page databases (new and others used by other works). The results prove that previous work generated too many false positives and were unable to detect a set of related JavaScript objects formed by *new Date().getTime(), this.width, this.height navigator.userAgent, document.cookie* to *Browser Fingerprinting.*

Keywords: Website fingerprinting, tracking, statistic analysis, Java Script.

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

listagem

Sumário

Capítulo 1

Introdução

É sabido que diferentes empresas, sites e serviços coletam, rastreiam e monitoram, sistematicamente, informações de seus usuários via web. Um estudo feito por Li et al. (2015) apontou que cerca de 46% dos sites listados no Alexa.com continham pelo, ao menos, um rastreador de terceiros. Esses rastreadores representam um perigo para a privacidade e a segurança dos usuários, pois coletam sigilosamente informações que são combinadas, em segundo plano, sem os usuários saberem de sua existência. Via de regra, como demonstrado em (Microsoft, 2013), essas informações sensíveis alimentam cartéis na Internet, direcionando, por exemplo, o usuário para publicidade indesejada e engendrando sua decisão de comprar ou não um produto. Esses rastreadores são chamados de *webtracker* [1], os quais possuem informações suficientes sobre o estado da condição financeira do usuário e podem classificar tais informações de acordo com seu poder aquisitivo. (Scism and Maremont, 2010).

Para alimentar os *webtrackers*, a primeira das tecnologias empregada foram os Cookies, que são pequenos arquivos de textos que podem armazenar senhas e guardar informações sobre os sites visitados (Nikiforakis et al., 2013). Contudo, por serem de fácil identificação e consequentemente apagados diretamente pelo navegador (a maioria dos navegadores permitem o gerenciamento), os Cookies perderam espaço para um novo método desenvolvido para continuar o rastreamento.

Conhecido como *Browser Fingerprinting* - também chamado de *website fingerprinting*, *Device intelligence*, *Machine fingerprinting*, *web fingerprinting* ou *Device fingerprinting*, esse método coleta dados sistematicamente do navegador ou do hardware do usuário por meio de um conjunto de configurações, atributos (tamanho da tela do dispositivo, versões de software instalado, entre muitos outros) e outras características observáveis durante comunicações para gerar uma identificação capaz de correlacionar suas atividades ao navegar na Internet (Saraiva, 2016). Essa coleta de dados sistêmica é possível porque os sites precisam saber informações para que possam ajustar o conteúdo disponibilizado de acordo com a tecnologia empregada pelo navegador do usuário.

[1] Web Tracking é a prática pela qual os operadores de sites coletam e compartilham informações sobre a atividade de um usuário específico na web.

Com essas informações disponíveis, os sites podem, por exemplo, ajustar o conteúdo ao tamanho da tela do usuário, bem como carregar recursos necessários para reprodução de áudio e vídeo com base nos plugins instalados.

Via de regra, a melhor forma de coletar dados é através de JavaScript[2] que, por padrão dos navegadores, permite acesso a objetos internos do HTML DOM (*Document Object Model*) como *Navigator*, *history* e *Screen* (Khademi et al., 2015) e, assim, obtém informações relacionadas ao sistema operacional, ao hardware do usuário e ao próprio navegador. Entretanto, recursos que foram desenvolvidos para fornecer funcionalidades aos sites também são capazes de expor informações e dados valiosos, através do *Browser Fingerprinting*, com a intenção de alimentar *webtrackers* e/ou atacantes.

Portanto, a ideia por trás do *Browser Fingerprinting* é a coleta de dados de natureza técnica do equipamento do usuário, a fim de combiná-los em um identificador único, para ser usado como meio de rastrear e monitorar suas atividades na Internet. Nesta dissertação, o alvo é o *Browser Fingerprinting*. Mais especificamente o *fingerprinting* que atua sobre os **"objetos JavaScript fornecedores de informações"** que formam o ecossistema do navegador.

1.1 Motivação

A prática de *Browser Fingerprinting* se tornou corriqueira na web, adotada por hackers até administradores de sites, por permitir o processamento dos dados no lado servidor e de ser de difícil detecção. Diferente dos Cookies, que dependem do armazenamento dos dados coletados no lado do cliente, a técnica de *Browser Fingerprinting* é invisível ao usuário comum da Internet, que não sabe se seus dados estão sendo coletados, onde sua identificação está sendo gerada e como será utilizada.

Existem trabalhos na literatura, como os de (Acar et al., 2013) e (Nikiforakis et al., 2015), que desenvolveram métodos de detecção de websites que geram a identificação única do usuário a partir dos dados coletados, os quais os autores chamaram de *fingerprinters* [3]. Também existem trabalhos ligados a detecção de objetos fornecedores de informações do navegador, como o de (Saraiva, 2016), que definiu uma classificação para avaliar o nível de risco oferecido por uma página web. O método de detecção proposto pela autora usa a identificação baseada em *regex*, o qual realiza a contagem de palavras suspeitas que referenciam os nomes dos objetos com base em um dicionário pré-estabelecido. Contudo, com *regex* não é possível identificar se uma chamada de objeto realmente aconteceu. Isso porque, encontrar no código uma chamada para o objeto *navigator* não representa por si só um *fingerprinting*, pois nem um valor de atributo será retornado. Ou seja, o trabalho só contou palavras, o que gera falsos positivos.

Considerando isso, a pesquisa nesta dissertação visa aplicar identificação baseada em AST (*Árvore de sintaxe abstrata*) de código JavaScript, extraído de páginas web,

[2] JavaScript é uma linguagem de programação que permite implementar funcionalidades mais complexas em páginas web.

[3] *fingerprinters* são sites que geram a identificação do usuário a partir de dados coletados pelos scripts de *fingerprinting*

para dar apoio ao processo de detecção de *Browser Fingerprinting*, diminuindo, assim, a coleta de dados abusiva e sem a anuência dos usuários

1.2 Objetivos

O objetivo geral desta dissertação é identificar *Browser Fingerprinting*, através da observação e identificação das chamadas aos objetos JavaScript fornecedores de informações, empregando análise da Árvore Abstrata Sintática (*AST*), a fim de aumentar a detecção deste tipo de rastreamento e fornecer um arcabouço para futuras contramedidas.

Como objetivos específicos, pretende-se:

- Elaborar um extrator para coletar o código JavaScript de páginas web.
- Criar um dicionário de objetos da linguagem JavaScript comumente utilizados em *fingerprinting*, dividido em níveis de profundidade que vai do primeiro objeto até a quinta propriedade.
- Desenvolver um identificador sintático, baseado em AST, para identificar no código fonte dos scripts das páginas coletadas as chamadas aos objetos da linguagem JavaScript.
- Adaptar e aplicar uma metodologia de classificação proposta por (Saraiva, 2016) para determinar o risco dos artefatos de *Browser Fingerprinting* encontrados em páginas web.

1.3 Contribuições Esperadas

Espera-se que, ao final do trabalho, as seguintes contribuições sejam alcançadas:

- Um método de detecção para identificar indicadores de *Browser Fingerprinting* em páginas *Web*
- Um dicionário de objetos de JavaScript relacionados a *Browser Fingerprinting*

1.4 Estrutura do Documento

O restante deste documento está organizado como segue: O Capítulo 2 apresenta os conceitos necessários para a compreensão desta proposta. O Capítulo 3 descreve os trabalhos relacionados sobre *Browser Fingerprinting* e as principais abordagens na área. O Capítulo 4 apresenta a visão geral e a metodologia. O Capítulo 5 apresenta a implementação da proposta e o protocolo experimental. Por fim, o Capítulo 6 expõe os resultados e atividades futuras.

Capítulo 2

Conceitos Básicos

Este Capítulo descreve os conceitos básicos necessários para o entendimento desta proposta de dissertação. Serão apresentadas definições de *Browser Fingerprinting*, sua classificação e suas formas de utilização para identificação de usuários unicamente na web. Também discute a aplicação das técnicas de ofuscação, especialmente em código JavaScript. Por fim, a aplicação de AST em código JavaScript é explicada.

2.1 Browser Fingerprinting

A RFC (*Request For Comments*) 6973 (Cooper et al., 2013) define *fingerprint* como "um conjunto de elementos de informação que caracteriza um dispositivo ou uma instância de uma aplicação", e *fingerprinting* como "o processo pelo qual um observador ou atacante identifica, de maneira única e com alta probabilidade, um dispositivo ou uma instância de um aplicativo com base em um conjunto de múltiplas informações".

Mayer (2009) foi o primeiro a aplicar esses conceitos em ambiente web e a demonstrar que era possível identificar usuários na web através de atributos do ecossistema de um navegador. Na Figura 2.1, retirada do trabalho de (Mayer, 2009), pode-se observar a resposta a uma requisição GET HTTP para o site da universidade de Princenton, que obteve como resposta uma série de cabeçalhos (*headers*) HTTP. Percebe-se, na linha **3**, que o *header User-Agent*[1] fornece uma string com diferentes informações sobre o navegador (versão - Mozilla/5.0; engine - Gecko/2008120121) e o dispositivo (Intel Mac OS X 10.5). Na prática, informações como a de **User-Agent** podem ser obtidas através de uma requisição **GET** (Figura 2.1) ou através de um script JavaScript, como observado na linha **"1"**, do código descrito na Figura 2.2, que obtém o valor da string por meio do comando **alert**, recebendo como parâmetro de entrada o objeto **window.navigator.userAgent**.

Uma vez que Mayer (2009) mostrou a possibilidade de obter atributos com informações que, combinadas, podem gerar identificadores únicos, outros pesquisadores

[1]User-Agent é um componente do cabeçalho do protocolo HTTP.

```
1 GET / HTTP/1.1
2 Host: www.princenton.edu
3 User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US
      ; rv : 1.9.0.5) Gecko/2008120121 Firefox /3.0.5
4 Accept: text/html, application /xhtml+xml, application /xml;q
      =0.9.*/*;q=0.8
5 Accept-Language: en-us , en;q=0.5
6 Accept-Encoding: gzip , deflate
7 Accept-Charset: ISO-8859-1,utf -8;q=0.7,*;q=0.7
8 Keep-Alive: 300
9 Connection: keep-alive
```

Figura 2.1: Resposta de uma requisição GET HTTP. Fonte: Adaptada de Mayer (2009).

```
1 alert(window.navigator.userAgent)
2 //alerta «Mozilla/5.0 (Windows; U; Wind98; en-US; rv:0.9.2) Gecko/20010725 Netscape6/.1
```

Figura 2.2: Obtendo informações de userAgent via JavaScript. Fonte: Autor.

passaram a atuar na área e a elaborar técnicas para identificação de usuários.

Eckersley (2010), com seu projeto Panopticlick (https://panopticlick.eff.org/), argumentou que as informações fornecidas pelo navegador, como, por exemplo, as dimensões da tela ou fontes instaladas, poderiam ser combinadas para criar um *fingerprint* do dispositivo e, consequentemente, identificar seu usuário. O autor observou que quanto mais comum o atributo for entre os diferentes usuários, menor é a entropia e, quanto mais o valor do atributo for estável, maior será o tempo de identificação. Nesse contexto, os experimentos de Eckersley demonstraram que 94,2% dos dispositivos possuíam um *fingerprint* exclusivo. Le et al. (2017) considera esta descoberta como o surgimento do *Browser Fingerprinting*.

Esta dissertação segue a nomenclatura especificada pela W3C[2] (*Word Wide web Consortium*), que definiu "*Browser Fingerprinting* como sendo a capacidade de um site para identificar ou re-identificar um usuário visitante, *userAgent* de navegador ou aplicação ou um dispositivo por meio de definições de configuração ou outras características observáveis". Portanto, *Browser Fingerprinting*, literalmente a impressão digital do navegador web, é a técnica que permite que um usuário tenha seu dispositivo identificado e possa ser rastreado por *web trackers*.

2.1.1 Classificação

Além da definição, o W3C também classificou as técnicas de *Browser Fingerprinting* em três tipos:

- **Passiva**: na qual o *fingerprinting* emprega características observáveis no conteúdo de solicitações web, sem utilizar qualquer código em execução no lado do cliente.

[2]https://www.w3.org/TR/fingerprinting-guidance/

Esse tipo de *fingerprinting* inclui o conjunto de cabeçalhos de solicitação *HTTP*, endereço IP e outras informações da rede.

- **Ativa**: considera técnicas nas quais scripts, principalmente via JavaScript, são executados no lado do cliente para observar características adicionais sobre o navegador. Técnicas ativas podem incluir o acesso ao tamanho da janela, enumerar fontes ou plugins, avaliação das características de desempenho ou os padrões de renderização de gráficos.

- **Cookie-like**: nessa categoria, usuários, *userAgents* e dispositivos podem ser reidentificados por um site que primeiro configura e depois recupera o estado armazenado de um navegador ou dispositivo. A reidentificação de um usuário ou inferências sobre ele, da mesma forma que os Cookies, permite o gerenciamento de estado para o protocolo *HTTP*. Também podem contornar as tentativas do usuário em limitar ou apagar os Cookies armazenados pelo *userAgent*.

Dos três tipos de *Browser Fingerprinting*, o ativo é o mais usual e o que obtém melhores resultados. Por isso, scripts de *fingerprinting* tendem a ser específicos do ambiente, ou seja, usam um conjunto de variáveis de ambiente para tornarem larga e rica em detalhes a avaliação que fazem das configurações do usuário. A Tabela A.1 ilustra algumas dessas variáveis que podem ser obtidas através de objetos de JavaScript, de acordo com (Hraška, 2018).

2.1.2 JavaScript e Browser Fingerprinting

O JavaScript é uma linguagem de programação interpretada que atua do lado do cliente e permite total interação com o usuário sem que exista a necessidade de intervenção do servidor web. Com JavaScript pode-se consultar, alterar ou excluir os elementos do DOM (*Document Object Model*) de uma página web. Além disso, o JavaScript é suportado por todos os principais navegadores do mundo. Esse contexto, faz da linguagem JavaScript uma ferramenta muito poderosa e versátil para ser usada por *Browser Fingerprinting*. O trabalho de Nikiforakis and Acar (2014) revelou que 400 milhões dos sites mais populares da Internet já usavam *Browser Fingerprinting* via JavaScript em 2015.

Nesse sentido, pode-se usar JavaScript para inspecionar o ecossistema do navegador e procurar, por exemplo, por objetos que forneçam características específicas do sistema onde o navegador foi instalado com a finalidade de criar identificações únicas. Essa obtenção de informação é realizada explorando-se o escopo global, que no caso do navegador é representado pelo objeto *Window*[3].

Ao contrário das linguagens estruturadas como **C** ou linguagens orientadas a objetos como Java e C++, o JavaScript é uma linguagem baseada em protótipos (Mozilla, 2020). Por esse motivo, a herança em JavaScript não se baseia em classes,

[3]O objeto *window* representa o objeto global do navegador, ou seja, a partir desse objeto outros podem ser chamados como, por exemplo, o objeto *navigator*

mas é construída com base em outros objetos, fazendo com que cada objeto tenha um link para outro objeto chamado *prototype* [4], criando uma cadeia interconectada de objetos que podem ser chamados um a partir do outro até que *"null"* seja encontrado. Em JavaScript, pois, a possibilidade de manipular objetos que alteram ou coletam informações da página web a partir do escopo global é um ponto fraco da linguagem a ser explorado em busca de vulnerabilidades que comprometam o sistema.

A Figura 2.3 exemplifica como criar uma cadeia de objetos em JavaScript.

Cadeia de objetos prototype

```
1 var objeto1  = {a: navigator};
2 //   objeto1 -----> Object.prototype -----> null
3
4 var objeto2 = Object.create(objeto1);
5 //   objeto2-----> objeto1
6 //              -----> Object.prototype -----> null
7 console.log(objeto2.a)
```

Figura 2.3: Ambientes de identificação: cadeia de objetos. Fonte: Autor.

Primeiramente cria-se um par chave e valor, como num dicionário, e o atribui-se para uma variável. No caso, na linha **"1"**, criou-se o *objeto*1, que recebeu como atribuição um objeto *a* e uma propriedade **navigator**. O *objeto*1 também aponta para *Object.prototype*, que aponta para *null*, encerrando a cadeia. Na linha **"4"**, criou-se o *objeto*2 chamando *Object.create* e passando como valor de entrada o *objeto*1. Nesse caso, o *objeto*2 aponta para o *objeto*1 e para *Object.prototype*, que aponta para *null*. Assim, pode-se chamar, a partir do *objeto*2, o *a* usando a notação de **"."** e, com isso, obter o valor de sua propriedade. Essa forma de chamar objetos e propriedades é utilizada pelos scripts de *Browser Fingerprinting*.

2.1.2.1 Scripts de Browser Fingerprinting

Os scripts de *Browser Fingerprinting* são códigos escritos em linguagem JavaScript, ou chamados via JavaScript, para obter as informações armazenadas nas variáveis de ambiente do navegador quando a página web é acessada. A Figura 2.4 ilustra, através da ferramenta de inspeção disponível nos navegadores, que códigos JavaScript são carregados junto à página principal do site.

De acordo com Figura a 2.4, na aba **rede**, percebe-se que o arquivo *script.js* foi carregado no momento em que o site foi acessado (**1**) e, como indicado em (**2**), após ser carregado, o script de *fingerprinting* coleta as informações sobre a configuração da máquina do usuário, executando funções que retornam dados fornecidos por objetos de JavaScript. Ao examinar o conteúdo do *script.js*, percebe-se que ele contém uma variável que recebe o retorno de uma função. Ao examinar o código da função

[4] O Prototype aplicado em JavaScript é um artifício para auxiliar os desenvolvedores a construir sites mais interativos e mais inteligentes.

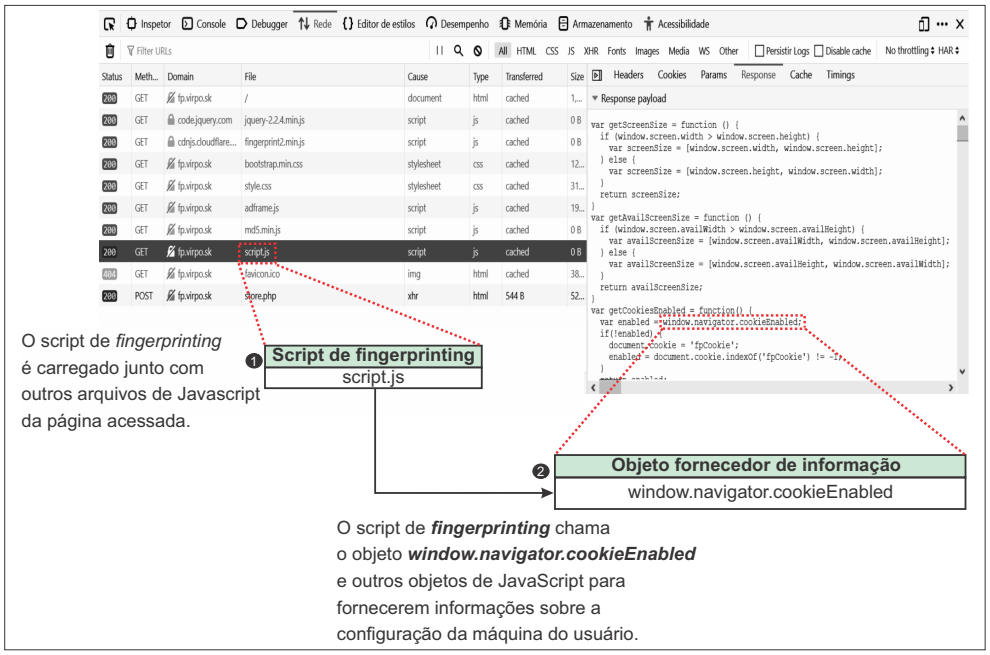


Figura 2.4: Script de fingerprinting para captura de dados. Fonte: Autor.

function(), percebe-se que a a variável declarada como *enabled* recebe o retorno do objeto *window.navigator.cookieEnabled*, o qual retorna um valor *Booleano* que indica quando Cookies estão habilitados ou não.

Já a Figura 2.5, seguindo mesmo cenário da Figura 2.4, mostra como os dados coletados são armazenados para serem enviados para os sites considerados *fingerprinters*.

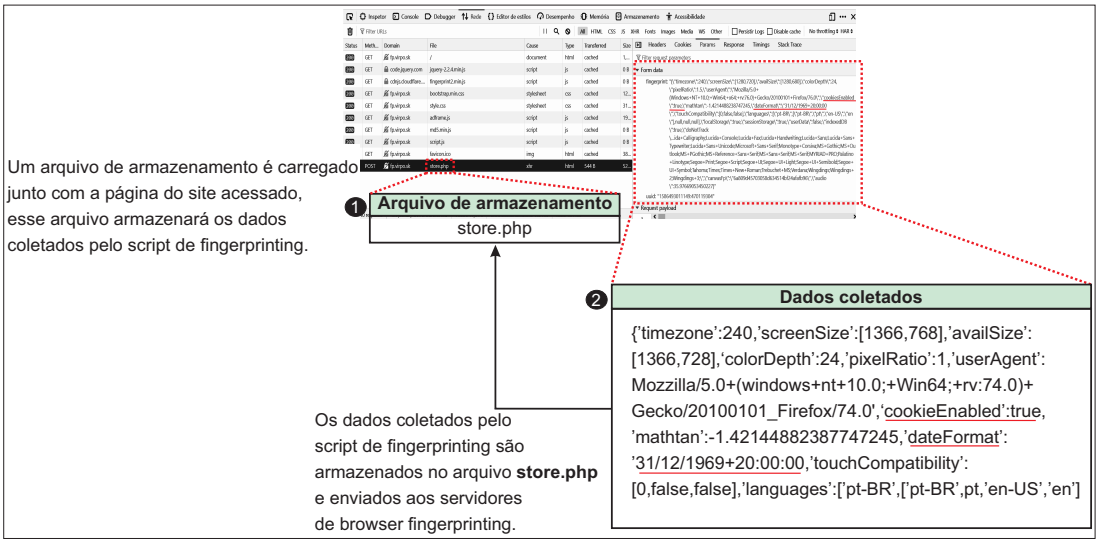


Figura 2.5: Script de fingerprinting para armazenamento de dados. Fonte: Autor.

Na Figura, observa-se que as informações coletadas pelo arquivo *script.js* são armazenadas no arquivo *store.php*, que também foi carregado junto à página principal do site acessado. Observa-se, em (**1**), que o arquivo *store.php* recebe as informações coletadas pelo arquivo *script.js* e, em (**2**), pode-se ver o conjunto de informações coletadas. Vale lembrar que, após a coleta, os dados são enviados aos servidores com a finalidade de gerar uma chave de identificação do usuário. Contudo, não é possível precisar o que é feito com as informações coletadas, em que local são armazenadas ou como as chaves de identificação do usuário são geradas. Em ambas as Figuras percebe-se a ação do *fingerprinting* em coletar os dados e em armazená-los para o envio aos servidores e, consequentemente, identificação do usuário. Ao analisar esse processo, percebe-se que muitos dados do dispositivo do usuário são coletados, deixando claro que o *Browser Fingerprinting* não é apenas uma ferramenta usada para gerar uma identificação e rastreamento do usuário, mas uma ferramenta poderosa para obtenção de informações pessoais, o que torna ações de detecção essencial para proteção da privacidade e segurança dos usuários.

Dessa forma, reconhecer quais objetos de JavaScript representam ações de *fingerprinting*, através da análise do código fonte da página web e, consequentemente, determinar o risco (Saraiva, 2016) que a página oferece ao usuário é fundamental.

2.2 Ofuscação de Código JavaScript

A ofuscação é a prática de tornar algo difícil de entender. Em programação, códigos são muitas vezes ofuscados para proteger a propriedade intelectual e impedir um atacante de utilizar engenharia reversa de um software proprietário. Com isso, a ofuscação pode envolver a criptografia de algumas partes ou de todo o código, removendo-se metadados que potencialmente poderiam revelar configurações internas do sistema em questão, renomeando classes e nomes de variáveis para rótulos sem sentido ou adição de código não utilizado para um binário do aplicativo (Beaucamps and Filiol, 2007).

Além disso, uma ferramenta capaz de ofuscar códigos pode ser utilizada para converter automaticamente o código-fonte diretamente em um programa, que continuará sendo executado da mesma maneira, porém, será mais difícil de decifrar. O exemplo da Figura 2.6 ilustra a ofuscação de código relacionada a *Browser Fingerprinting*, com o objetivo de ocultar a chamada dos objetos fornecedores de informações.

Na Figura 2.6, observa-se como ofuscar um código JavaScript e seu resultado. Ao acessar um site (a ofuscação foi realizada no site http://www.jsfuck.com/) que ofereça o serviço de ofuscação, passa-se para o site o código que se deseja ofuscar. No exemplo da Figura, foi passado como entrada a instrução $alert(navigator.appName)$ que produzirá a saída **"Netscape"** como referência ao nome do navegador. A palavra **"Netscape"**, impressa na tela, indica que o *fingerprinting* foi bem sucedido. É importante ressaltar que não foi gerada nenhuma identificação de usuário, apenas demonstrou-se como a requisição a um atributo pode ser ofuscada. Nesse sentido, a ofuscação é bem utilizada para embaralhar ou esconder códigos JavaScript benignos e maliciosos. Quem faz uso mal-intencionado das técnicas de ofuscação tem, muitas vezes, por objetivo, evitar a

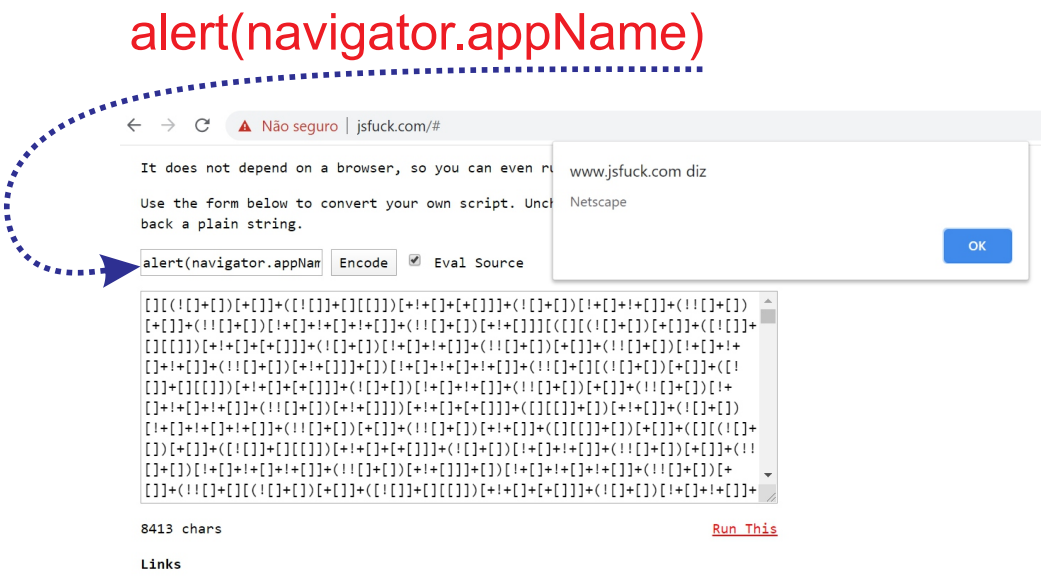


Figura 2.6: Primeiro exemplo de ofuscação de código JavaScript. Fonte: Autor.

detecção baseada em assinaturas (Lu and Debray, 2012).

Assim, existe um número que, apesar de pequeno, mostra que os scripts ofuscados geralmente ocorrem em categorias específicas de sites e, às vezes, escondem comportamentos relevantes à segurança e a questões de privacidade, como por exemplo, os ataques de *Browser Fingerprinting* (Skolka et al., 2019). Com isso, para se detectar o *fingerprinting* em códigos ofuscados, é necessário verificar os padrões aplicados em códigos JavaScript maliciosos. Para isso, é preciso entender as técnicas de ofuscação. A seguir são apresentadas algumas categorias de ofuscação de chamadas aos objetos JavaScript (Gorji and Abadi, 2014).

2.2.1 Randomização

A Figura 2.7 ilustra um exemplo detalhado com todos os tipos de randomização em código JavaScript.

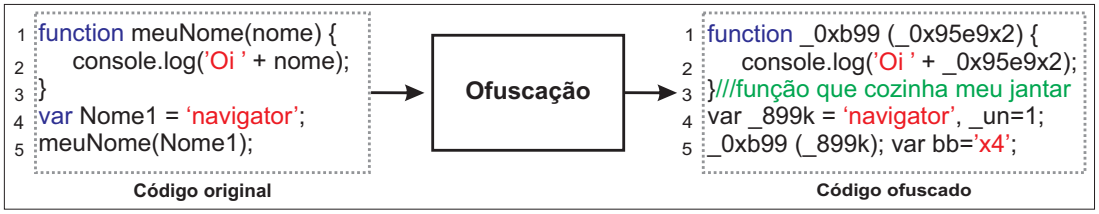


Figura 2.7: Segundo exemplo de ofuscação de código JavaScript. Fonte: Autor.

A randomização é uma técnica de ofuscação que permite inserir ou alterar aleatoriamente alguns elementos do código sem alterar a semântica. Em outras palavras,

adiciona "ruídos" no código e, com isso, gera um mascaramento que impossibilita a identificação do valor de um atributo confidencial. Essa técnica é provavelmente a técnica de ofuscação mais simples de implementar. Randomização de nomes de variáveis e funções, espaço em branco e randomização de comentários são técnicas comuns nessa categoria. A randomização de nomes de variáveis e funções substitui nomes de variáveis e nomes de funções por sequências aleatórias. A randomização de espaço em branco insere aleatoriamente caracteres de espaço em branco, incluindo espaço, tabulação, avanço de linha e retorno de carro. A lógica por trás disso é que o interpretador JavaScript ignora caracteres de espaço em branco.

Na Figura 2.7 percebe-se a aplicação de randomização na inserção de espaços em branco. Em um código JavaScript, o caractere *underline* (_) é considerado um espaço em branco pelo interpretador. Na linha **1** da Figura, o nome da função no código original ($function\ meuNome$) é ofuscado para $function$ $\_0xb99$ e o parâmetro de entrada ($nome$) é transformado em $\_0x95e9x2$, caracterizando uma randomização de nomes e variáveis. Na linha **3**, do código ofuscado, um comentário desnecessário, que não tem relacionamento com o código original, foi adicionado, Na linha **4**, a variável $Nome1$ foi transformada em $\_899k$. Observa-se na linha **5** que a variável $\_899k$ foi passada como parâmetro de entrada para a função ofuscada $\_0xb99$, e que a variável bb foi adicionada sem nunca ser utilizada.

2.2.2 Ofuscação de números

Outra técnica de ofuscação que pode ser aplicada em código JavaScript é a ofuscação de números. Ela permite criar uma representação de um número através de operações matemáticas, podendo, assim, escrever o mesmo número de maneiras diferentes.

A Figura 2.8 ilustra o uso desse tipo de randomização em códigos JavaScript.

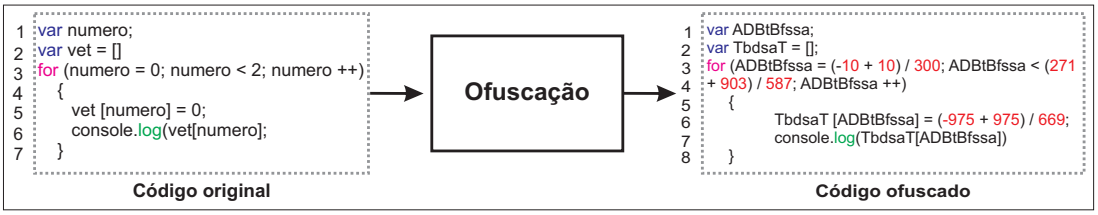


Figura 2.8: Ofuscação de números em código JavaScript. Fonte: Autor.

Na Figura 2.8, percebe-se que ofuscações por randomização de nome e variável foram realizadas no código (a variável $numero$ é transformada em $ADBtBfssa$, por exemplo). Falando diretamente sobre ofuscação de números, percebe-se que, na linha **3**, a operação aritmética $(-10 + 10)/300$ resultará no valor igual a 0. Na mesma linha, $(271 + 903)/587$ resultará no valor igual a 2 e a operação aritmética $(-975 + 975)/669$ resultará no valor 0.

O mesmo acontece na linha **5**. No código original tem-se $vet[numero] = 0$ e após a ofuscação ela passa a ser $TbdsaT[ADBtBfssa] = (-975 + 975)/669$, onde o

vetor vet (agora $TbdsaT$) receberá o valor 0, enquanto a operação do for for verdade.

2.2.3 Ofuscação de String

A ofuscação de string é uma técnica de ofuscação sofisticada que dificulta o entendimento da string para os seres humanos. Geralmente, ela envolve um decodificador personalizado, que vai desde uma simples função XOR a uma **cifra de César** mais complexa (Feinstein et al., 2007). Nesse sentido, a codificação e manipulação de strings são duas categorias principais utilizadas para ofuscar os dados. Com essa técnica pode-se dividir uma string em várias variáveis e posteriormente concatená-las em tempo de execução. Assim, as principais técnicas de ofuscação de string são:

- **Encoding**, na qual se altera a representação de uma string e a torna ilegível para seres humanos, mas não se altera o significado real da string.
- **Manipulação de strings**, na qual os scripts são divididos em pequenas strings. Randomização de código e encoding podem ser aplicados em conjunto para dificultar a detecção. Assim, é possível realizar a manipulação (i) concatenando strings, (ii) substituindo caracteres, e (iii) substituindo palavras-chave.

A Figura 2.9 ilustra uma ofuscação de string usando enconding, onde a linha $varnavi =' navigator.appName'$ é ofuscado para o formato Base64[5] no conjunto de caracteres de destino do formato UTF-8.

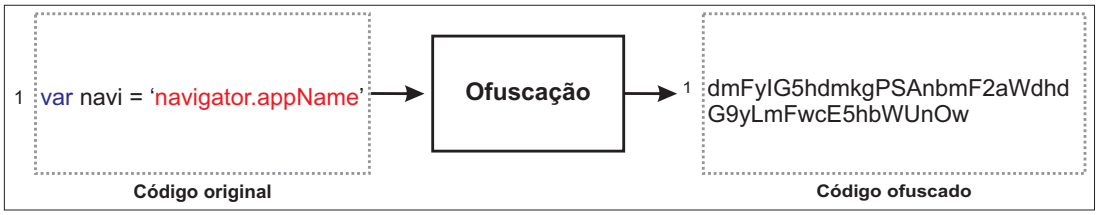


Figura 2.9: Ofuscação de string: Encoding. Fonte: Autor.

Já a Figura 2.10 ilustra um exemplo de como ofuscar um código usando concatenação de strings, momento em que se divide uma string em várias sub-strings, distribuídas em duas variáveis distintas e concatenadas pela função $eval()$.

Na Figura 2.10, percebe-se que, no código original, tem-se a função *alert* que recebe como parâmetro de entrada o método *navigator.appName*. Já no código ofuscado, na linha **1**, tem-se a variável $fing2$ que recebe como atribuição a string *"ale"+ "r"+ "t"+ "("+ "navi"+ "gator"* que é igual a *alert(navigator*. Em seguida, na linha **2**, a variável $fing1$ recebe como atribuição a string *"."+ "ap"+ "p"+ "Na"+ "me"+ ")"* que é igual a *.appName*. Finalizando, na linha **3**, tem-se a função $eval()$ que recebe como parâmetro

[5]Codificações **Base64** são usadas quando é preciso codificar dados binários que precisam ser armazenados e transferidos por mídia projetada para lidar com dados de texto. Fonte: https://www.base64encode.org/

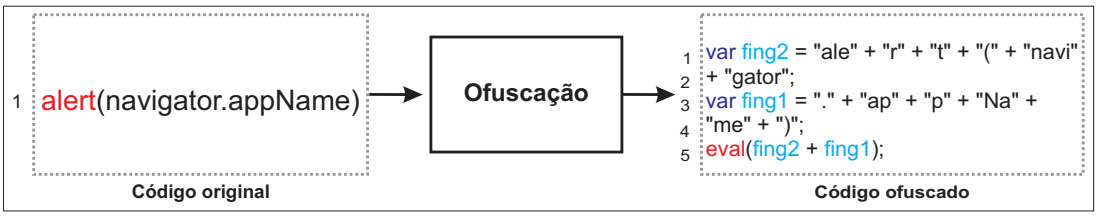


Figura 2.10: Ofuscação de string: Concatenação de strings. Fonte: Autor.

de entrada $fing2 + fing1$. A saída da função $eval()$ produzirá como resultado uma chamada ao objeto **navigator.appName**.

A Figura 2.11 exemplifica como é possível aplicar a substituição de strings.

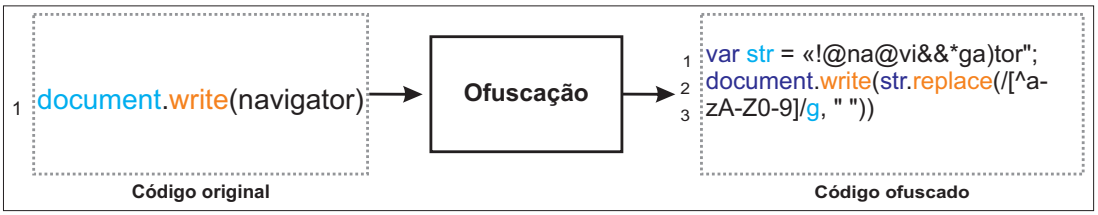


Figura 2.11: Ofuscação de string: Substituição de caracteres. Fonte: Autor.

Na Figura 2.11, essa substituição (ofuscação) pode ser realizada com a função *replace*() do JavaScript e a utilização de *regex* para substituir alguns caracteres na sequência especificada antes da execução do código. Assim, na linha **1**, a variável *str* recebe como atribuição a string *"navi&&ga)tor"*. Na linha **2**, o método *document.write* é chamado e recebe como entrada a função *str.replace*, que usa uma *regex* para negar qualquer caractere que não esteja no conjunto de caracteres alfanuméricos produzindo uma saída igual a *navigator*. Essa saída poderia, por exemplo, ser concatenada com *userAgent* para formar *navigator.userAgent* e assim ser obtida informação a respeito das configurações do navegador do usuário.

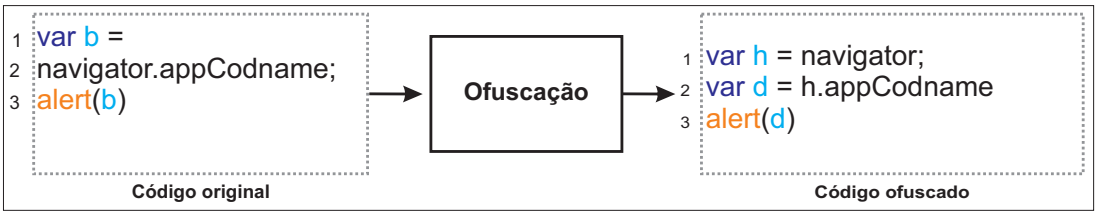


Figura 2.12: Ofuscação de String: Substituição de palavras-chave. Fonte: Autor.

Na Figura 2.12, o exemplo de um objeto em JavaScript dividido em palavras-chave é demonstrado. No código original observa-se que na linha **"1"** a variável **"b"** recebe como atribuição o objeto **navigator.appCodName**, após o processo de ofuscação, na linha **"1"** do código ofuscado que a variável **"h"** recebe como atribuição a

palavra **navigator** e na linha **"2"** a variável **"d"** recebe **h.appCodName**. Ao executar o script **h.appCodName** pode ser solicitado, porém ao buscar-se no código fonte pela cadeia **navigator.appCodName** não será encontrado, pois a chamada ao objeto foi dividida em várias variáveis.

Uma vez apresentados os principais tipos de ofuscação, é preciso lembrar que esta seção não tem a pretensão de esgotar o assunto sobre ofuscação de código JavaScript e nem as diversas formas existente de ofuscação, mas sim tentar demonstrar como as categorias podem ser aplicadas em *Browser Fingerprinting*.

Nesta dissertação, após identificar e realizar a desofuscação do código JavaScript, será preciso extrair as informações que possibilitarão a identificação do *fingeprinting* da página web. Uma abordagem existente para a extração de informações é a identificação baseada em AST de código JavaScript.

2.3 AST de código JavaScript

A Árvore de Sintaxe Abstrata (AST) é uma estrutura de dados baseada em árvore que representa o código fonte a ser inspecionado de forma programática. De acordo com Damasceno (2017), uma AST é uma representação sintática simplificada do código-fonte e, na maioria das vezes, é expressa em uma estrutura de dados, em formato de árvore, da linguagem de programação utilizada para a implementação. Uma AST também é chamada de representação intermediária do código fonte por favorecer as análises insensíveis ao fluxo (Nielson et al., 2010). Essas análises ignoram a ordem em que as instruções são executadas, possibilitando exploração de diversos caminhos a serem verificados.

A Figura 2.13 exemplifica um simples código em JavaScript e sua respectiva representação em formato de árvore sintática abstrata.

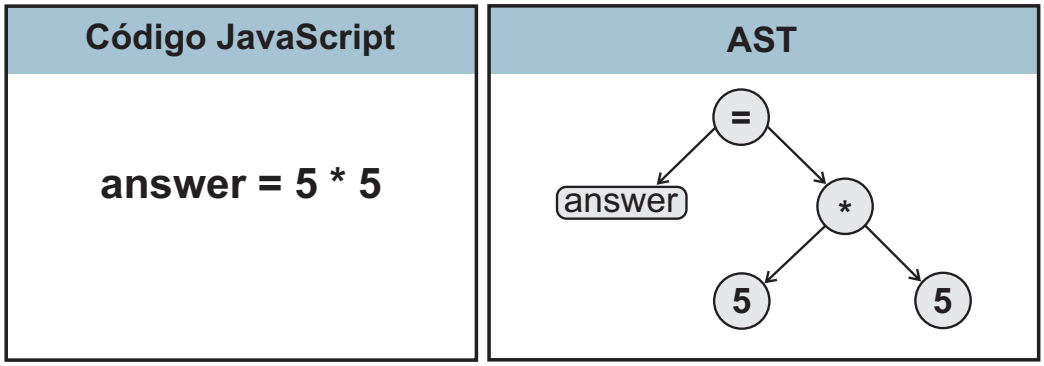


Figura 2.13: Exemplo de uma AST. Fonte: Adaptada de Damasceno (2017).

Via de regra, uma AST é utilizada em revisão de código fonte, pois examina o código sem executá-lo. Assim, pode-se verificar as características estáticas e como está organizada a estrutura do código JavaScript. Pode-se avaliar escopos, expressões ou declarações sem a preocupação com a redundância de detalhes sintáticos desnecessários

como espaços em branco, ponto e vírgula, chaves, cometários e outros. Além disso, não se tem a sobrecarga de execução de outras abordagens como a análise dinâmica, por exemplo (**?**).

A Figura 2.14 ilustra uma AST com chamadas a objetos ligados a *fingerprint*.

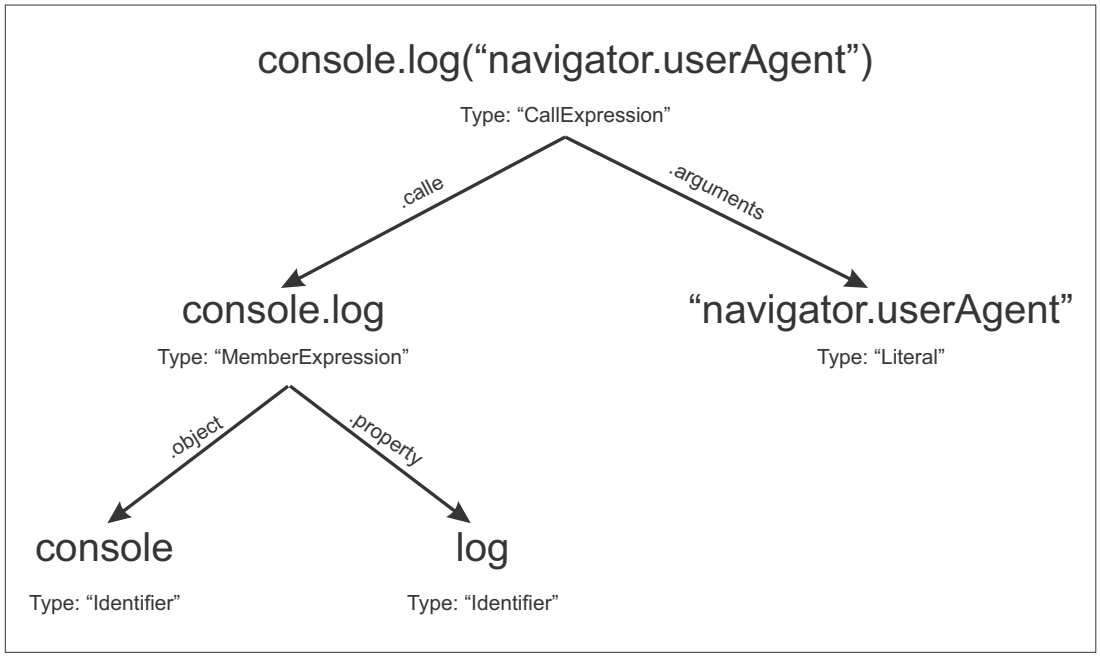


Figura 2.14: Avaliando um código com elementos de *fingerprinting* com AST. Fonte: Autor.

Pode-se observar que *type* representa o tipo do nó em questão. Para o código **console.log("navigator.userAgent")**, que é o nó raiz da AST, tem-se que o nó principal ou programa é do tipo $CallExpression$. Isto significa que se tem uma chamada de expressão. Essa chamada de expressão possui duas arestas que apontam para outros nós. A aresta *.calle* aponta para **console.log**, que é um nó do tipo $MemberExpression$, um membro de expressão, que por sua vez consiste em um objeto JavaScript, onde se tem o objeto **console** e a propriedade **log**. Também, pode-se ver que o nó **console** e o nó **log** são do tipo $Identifier$. Essa chamada de expressão possui uma aresta *.arguments* apontando para o argumento dessa chamada que é **"navigator.userAgent"**, que por sua vez é do tipo *literal*, ou seja, uma string. Percebe-se com essa pequena análise que **navigator.userAgent**, nessa condição, não pode ser classificado como um *fingerprinting*, pois a chamada de expressão **console.log**, recebendo como argumento de entrada a string **"navigator.userAgent"**, apenas imprimirá na tela a palavra **navigator.userAgent**.

É importante destacar que, ao utilizar uma AST, pode-se obter um relatório de análise bem completo, pois ela organiza a estrutura do código de forma hierárquica e cria rótulos que identificam essa estrutura, sendo uma excelente ferramenta para identificar e

extrair informações do código (He et al., 2018). Além disso, segundo Antal et al. (2018), é possível determinar os fluxos de controle sem a preocupação com a ordem de execução do fluxo de controle. Assim, pode-se realizar análises relacionais insensíveis ao fluxo que nos permitem rastrear vazamentos de informações sensíveis, rastrear chamadas de funções, chamadas a objetos, realizar análises de ponteiros, realizar comparações do código estático em relação ao código em execução, otimizações e comparações de objetos e propriedades com dicionários pré-estabelecidos.

Nesse sentido, vistoriando um código JavaScript, pensando na detecção de *Browser Fingerprinting*, pode-se utilizar a AST para extrair do código as chamadas aos objetos fornecedores de informações. O Capítulo 5 descreverá a aplicação de AST em código JavaScript para detecção de *Browser Fingerprinting*.

2.4 Considerações

Neste Capítulo, a fundamentação teórica foi conceituada e *Browser Fingerprintig* foi classificado. Além disso, foram realizadas explanações e discussões a respeito do processo de identificação de usuário. Foi estabelecido, também, a relação de *Browser Fingerprinting* com o código JavaScript e com a sua ofuscação. Também demonstrou-se como a aplicação da identificação baseada em AST de código JavaScript ajuda a identificar as chamadas aos objetos de JavaScript fornecedores de informações, que atuam como indicadores de *Browser Fingerprinting*.

De antemão, portanto, pode-se dizer que são muitos os desafios relacionados à identificação de indicadores de *Browser Fingerprinting*, pois esta é uma técnica de hábitos furtivos que se aproveita da dinamicidade da linguagem JavaScript para agir de forma transparente. Contudo, espera-se, no decorrer deste trabalho, preencher uma lacuna no processo de detecção *Browser Fingerprinting*

Capítulo 3

Trabalhos Relacionados

Este Capítulo abordará trabalhos relacionados em uma seção chamada de *Estudos em Browser Fingerprinting*. Essa seção apresentará e discutirá os trabalhos relacionados com a área de *Browser Fingerprinting* que avaliaram o código JavaScript, foco desta dissertação.

3.1 Estudos em Browser Fingerprinting

O estudo de Acar et al. (2013) (*"Fpdetective: dusting the web for fingerprinters"*) avaliou a presença de *fingerprinting* em código JavaScript. Nessa pesquisa, os autores avaliaram *fingerprinting* através de um *crawler* que visita os sites e coleta os dados sobre eventos que podem estar relacionados a *fingerprinting*. O objetivo é a classificação de um site que faz *fingerprinting* quando são carregadas mais de 30 fontes do sistema ou quando existe a enumeração de plugins ou mimeTypes. Essa técnica investiga o script que coleta os dados e que os envia para um servidor remoto fazer o *fingerprinting*. A pequisa encontrou *fingerprinting* no JavaScript de 404 sites de 1 milhão de sites avaliados. A taxa encontrada foi baixa, porém o JavaScript na época da pesquisa não dominava o mercado de fornecimento de interatividade da web. Os resultados foram satisfatórios no sentido de demonstrar que o uso da *fingerprinting* era uma prática em crescimento que merecia a atenção dos governantes e dos pesquisadores. Também foi importante no sentido de alertar como *website Fingerprinting* estabeleceria uma associação profunda com o JavaScript.

O trabalho de FaizKhademi et al. (2015) (*"Fpguard: Detection and prevention of browser fingerprinting"*) implementou um mecanismo que detecta, em tempo de execução, as ocorrências de *fingerprinting* no código JavaScript. De forma geral, ao atingir um valor limite especificado, o mecanismo classifica uma entrada (um site) como uma *fingerprinter*. Os valores limites possuem uma definição baseada em comportamento anormal estipulada pelos pesquisadores. O FPGuard também faz o uso da técnica de randomização, enviando valores anormais para as *fingerprinters* que atingem pontuação alta em sua classificação. Um dos problemas com essa abordagem é que os valores da

pontuação são vagos e não são bem definidos. Por esse motivo, fica difícil compreender o que é um comportamento anormal segundo os pesquisadores. Os valores limites, também, não são mencionados em nenhum lugar, além do FPGuard não está disponível para ser reproduzido e, por isso, não há como avaliá-lo para termos certeza de sua eficiência.

Já o estudo de Le et al. (2017) (*Towards accurate detection of obfuscated web tracking*) avaliou o código JavaScript para detectar *fingerprinting*. No método, os autores propuseram a detecção de rastreamento web em código ofuscado. Os autores montaram um *proxy* para coletar o trafego HTTP e analisar as chamadas as API do sistema. Segundo os autores, a maior vantagem da técnica é superar a ofuscação de código. No estudo em questão, foi observado que 9,13% dos sites analisados utilizavam Canvas *Fingerprinting*; 6,9% usavam código de texto sem formação e 2,2% usam rastreamento de tela ofuscado.

No geral, pelo menos 10,44% dos sites analisados utilizavam rastreamento baseado em tela, enquanto 2,25% usavam rastreamento ofuscado. A limitação da pesquisa está no fato de ter investigado apenas o Canvas *Fingerprinting* e não ter observado a existência de outros tipos. Porém, ela abre margem para pesquisas futuras no sentido de comprovar se atualmente apenas o método do Canvas é utilizado. Outro fato sobre o estudo é a naturalidade que a incidência de *fingerprinting* em códigos ofuscados é baixa, uma vez que a ofuscação de código pode prejudicar a execução dos scripts. Isso faz com que as assinaturas dos métodos não sejam ofuscadas para não impedir o seu funcionamento. Entretanto, objetos podem ser declarados dentro e fora de funções e, se forem declarados diretamente no escopo global do código, podem ser totalmente ofuscados. Acredita-se que os 2,5% sejam objetos que foram declarados diretamente no contexto global do código.

Laksono et al. (2015) (*JavaScript - based device fingerprinting mitigation using personal http proxy*) realizou um estudo que avaliou o código JavaScript a fim de identificar *fingerprinting*. A identificação envolveu a criação de um sistema baseado em confiança de valores, o que seria uma forma de classificação. Caso os valores detectados fossem confiáveis, os autores considerariam a página livre de *fingerprinting*. Nesse estudo, os autores usaram o $Seleniumweb - drive$, um *proxy* popularmente conhecido para renderização de conteúdo JavaScript. A técnica dos autores consiste em passar os dados renderizados para o *proxy* que analisará a existência de *fingerprinting*. Depois, uma filtragem, baseada em cabeçalho, é realizada quando a solicitação do usuário tiver sido passada para o servidor web. O *proxy* aguardará a resposta do usuário. A resposta recebida, então, é processada na forma de extração de endereço do servidor da web a partir do cabeçalho da resposta HTTP. Assim, baseados no valor de cada atributo da API JavaScript encontrado dentro da página web, os autores relataram em seu trabalho um valor de nível de confiança de 31,065. Foi definido um valor de 15,5796 como valor de limiar. Isso significa que, baseado no valor de confiança usado, poderiam indicar se um conteúdo JavaScript era ou não *fingerprinting*. Nesse caso, os autores constataram que apenas 50,1515% da API JavaScript que continham os atributos relacionados a *fingerprinting* representavam risco a privacidade do usuário.

Na pesquisa de Saraiva (2016) (*Determinando o risco de fingerprinting em páginas web*) foi avaliado o código JavaScript para identificar *fingerprinting* usando *regex* e foi criada uma metodologia de classificação. Para isso, a autora usou bases conhecidas para investigação do JavaScritp como Alexa.com e DMOZ. Observou, também, os aspectos léxicos do código JavaScript, onde foi aplicada uma *regex* para contar palavras suspeitas encontradas no código estático. Ademais, a autora criou um dicionário de termos comumente utilizados em *Browser Fingerprinting*, esses termos são referências aos objetos do JavaScript que fornecem informações como a versão do navegador ou lista de fontes instaladas. A classificação de Saraiva foi organizada em 3 níveis de risco sendo baixo, médio e alto. Além disso, organizou e distribuiu os termos de *fingerprinting* nos níveis do dicionário para então aplicar a classificação.

A classificação adotada é interessante, pois organiza os objetos JavaScript por níveis de risco e pode ser considerada sua maior contribuição. A classificação é atualizável e possui uma lista bastante extensa de objetos do JavaScript que são comumente utilizados para a prática de *Browser Fingerprinting*. No entanto, a técnica de detecção utilizando *regex* gera falsos positivos, uma vez que conta objetos como se fossem strings. Vale destacar que o trabalho de Saraiva (2016) observou a ocorrência de termos classificados como perigosos, que estavam no nível 3 de sua classificação, em 10.000 sites da base Alexa.com, com um percentual da classificação de 59% em sites considerados benignos.

No trabalho de Rausch et al. (2014) (*Searching for indicators of device fingerprinting in the JavaScript code of popular websites*), os autores avaliaram o código JavaScript para medir a frequência de certos scripts de *fingerprint* comumente usados para identificar os usuários. Essa pesquisa foi realizada investigando os 1000 principais sites dos Estados Unidos. Os autores examinaram e compararam as amostras somente com três scripts de *Browser Fingerprinting* conhecidos. Contudo, esta análise demonstrou-se frágil, pois tentou identificar apenas scripts conhecidos e, segundo a pesquisa, menos de 6% dos sites examinados utilizam um desses scripts. Portanto, torna-se difícil avaliar os resultados dessa pesquisa, pois outras técnicas que não foram mapeadas podem ser utilizadas e estar presentes nesses 1000 sites.

No estudo de van Zalingen and Haanen (2018), os autores objetivaram detectar a prática de *Browser Fingerprinting* através da análise estática do código JavaScript com classificação por aprendizado de máquina (utilizaram SVMs) e, para isso, a AST foi empregada como suporte à detecção de *fingerprinting*. Foram avaliados 32 domínios, dos quais 12 foram classificados como sites *fingerprinting*, e 20 domínios foram avaliados como domínios que não continham *fingerprinting*.

O método dos autores consistia em avaliar dois conjuntos predefinidos de scripts que passavam por uma fase de processamento para que os scripts pudessem ser melhor analisados. Com isso, as expressões de membro ocultas nas variáveis eram expandidas e, posteriormente, segundo os autores, era realizada a análise real dos scripts. De fato, para realizar a detecção, o código era avaliado em uma AST que permitia a representação do código em uma estrutura hierárquica que era percorrida para que todas as declarações de variáveis fossem armazenadas em um escopo atual. Se a inicialização ou atribuição da

variável fosse outra variável ou um membro de expressão, a inicialização era armazenada e, quando uma propriedade era solicitada em um membro de expressão, as variáveis armazenadas anteriormente eram então pesquisadas para tentar uma expansão dessa expressão de membro que retornasse uma chamada de objeto de JavaScript.

3.1.1 Discussão

Tabela 3.1: Estudos sobre *Browser Fingerprinting*

	Técnica			**Identificação**	
Trabalho Relacionado	**Léxica**	**AST**	**Proxy**	**Estática**	**Dinâmica**
Fpdetective: dusting the web for fingerprinters			x		x
Fpguard: Detection and prevention of browser fingerprinting			x		x
Towards accurate detection ofobfuscated web tracking			x		x
JavaScript - based device fingerprinting mitigation using personal http proxy			x		x
Determinando o risco de fingerprinting em páginas web	x			x	
Searching for indicators of device fingerprinting in the JavaScript code of popular websites	x			x	
Identificando indicadores de Browser Fingerprinting em Páginas web		x		x	
Detection of Browser Fingerprinting by Static JavaScript Code Classification		x		x	

Ao longo do tempo, diversas abordagens vem tentando detectar *fingerprinting* em páginas web e classificá-lo. Na Tabela 3.1 pode-se ver que as abordagens dinâmicas utilizando técnicas baseadas em *proxy* dominaram o cenário por um tempo. O problema com essas abordagens é que elas dependem da ocorrência de evento e esses eventos só ocorrem em certas condições como, por exemplo, se um determinado plugin estiver instalado na máquina do usuário. Abordagens desse tipo requerem muito poder computacional para avaliar grandes quantidades de linhas de código, pois os eventos são avaliados em tempo de execução. Outro problema, é que essas abordagens, principalmente as mais antigas, se preocuparam em provar a existência de sites *fingerprinters*, o

que foi coerente na época. Porém, isso acabou gerando um número relevante de falsos positivos e falsos negativos como o caso de abordagens como FPdetective e FPguard, pois eles procuravam identificar as *fingerprinters*. Com isso, se um site fosse identificado como fazendo *figerprinting*, teria-se um falso negativo e um falso positivo no caso inverso. Acontece que muitos sites fazem *fingeprinting* e não são *fingerprinters*, ou seja, não geram a identificação do usuário. Segundo essa classificação, todos os sites seriam *fingerprinters*

Além disso, outras abordagens que avaliaram a presença de *fingerprinting* em páginas web utilizaram identificação estática aplicando técnicas, que avaliavam o código JavaScript de forma léxica, aplicando *regex*, por exemplo, para fazer a identificação dos artefatos. Usar identificação estática garante rapidez na avaliação do código, pois nenhum evento precisa ser disparado para ser verificado e pode-se avaliar o código por completo. No entanto, avaliar do ponto vista léxico traz alguns problemas, pois não se tem como definir com precisão se um conjunto de dados léxicos avaliados é realmente um objeto que pode ser chamado ou se é apenas uma string declarada no código. As análise léxicas não nos possibilitam essas informações.

Esta dissertação propõe identificar as chamadas aos objetos de JavaScript que atuam como indicadores de *fingerprinting* em páginas web, através de uma identificação estática baseada em AST usando como ferramenta de suporte um dicionário contendo 78 objetos de JavaScript relacionados a *Browser Fingerprinting*. A abordagem de (Saraiva, 2016) foi a única que apresentou de maneira explícita o uso de um dicionário contendo 58 termos da linguagem JavaScript, porém o dicionário usado neste trabalho é diferente do dicionário de (Saraiva, 2016), pois o dicionário da WBF Analyzer (Método de detecção utilizado neste trabalho e explicado no capítulo 4) é exclusivamente utilizado para identificar chamadas de objetos de JavaScript, enquanto o dicionário de (Saraiva, 2016) pode realizar comparações com chamadas e termos ligados a *fingerprinting*. O problema com a possibilidade de realizar comparações com termos isolados é a quantificação de falsos positivos, ou seja, ao realizar uma comparação com um termo isolado, como *navigator*, esse termo seria contabilizado, mas o termo *navigator* empregado de forma isolada não é uma chamada de objeto. Além disso, o dicionário usado neste trabalho possui uma quantidade de objetos de JavaScript superior aos citados no dicionário de (Saraiva, 2016).

A abordagem que utilizou AST de código JavaScript para detectar *Browser Fingerprinting* mencionada no trabalho de (van Zalingen and Haanen, 2018) limitou-se a avaliar um conjunto de domínios muito restrito no que diz respeito a quantidade de sites avaliados. Outro fato sobre o trabalho é que os autores não deixam claro se avaliaram somente as páginas principais do domínio ou se avaliaram o conjunto formado pela página principal mais as páginas secundárias. Os autores também não descrevem com clareza se os códigos avaliados eram do tipo JavaScript in-line ou se eram do tipo JavaScript externo. É importante saber o tipo de JavaScript avaliado, pois se somente o JavaScript in-line foi avaliado, significa que o método em questão avaliou uma quantidade muito pequena de códigos, uma vez que os códigos in-line são pequenos fragmentos de código que são declarados diretamente no corpo do documento

HTML.

Outro problema é que os autores em (van Zalingen and Haanen, 2018) mencionam ter apenas avaliado as declarações de variáveis, o que significa que somente um nó alvo, no caso o *VariableDeclaration* da AST do código JavaScript, foi inspecionado, limitando o espectro da detecção. Além disso, o método em questão não é capaz de detectar o *Canvas fingerprinting* e outras chamadas de objetos ligados a *fingerprinting* como o método *Date().getTime()* devido a limitação do dicionário utilizado na pesquisa, que contém somente 35 chamadas de objetos de JavaScript. Outro problema com essa abordagem é que os autores mencionam que pesquisam os valores das variáveis declaradas anteriormente para tentar expandir os membros de expressões. Isso significa que foi utilizada a técnica de recursão diretamente na AST para tentar descobrir se, por exemplo, um *navigator* foi atribuído para uma variável *a*. O problema em usar recursão em JavaScript é a quantidade de vezes que uma chamada recursiva pode ser invocada, o que significa que não se pode avaliar uma quantidade grande de linhas de código, visto que as chamadas recursivas em JavaScript limitam-se a 20.000 possibilidades.

Tomando como base os trabalhos citados na Seção 3.1, a AST é uma técnica de identificação que permitir extrair informações do código com precisão. Ademais, é uma técnica que já foi avaliada em outras pesquisas e por este motivo fornece uma base sólida de análises, porém sua aplicação voltada para detecção de *Browser Fingerprinting* ainda é muito limitada restringindo-se, atualmente, a um estudo que realizou análises preliminares, avaliando uma pequena quantidade de dados e utilizando a AST como mecanismo de detecção. O método desenvolvido neste trabalho utiliza a AST como ferramenta de extração de dados e utiliza um algoritmo próprio para detectar, posteriormente ao processo de extração, as chamadas de objetos de JavaScript em um estrutura de dados chamada de escopo reduzido que reflete o problema a ser analisado e utiliza ferramentas de suporte como o dicionário de *Browser Fingerprinting* e o dicionário de identificadores, como será visto no capítulo 4. Assim, pode-se avaliar o fluxo do código rotulado com as características estruturais da linguagem onde se aplica um parser baseado em AST.

3.2 Considerações finais

Browser Fingerprinting é uma área desafiadora e surgem novos atributos, a medida que, a tecnologia dos navegadores evolui e não temos como classificá-los com 100% de exatidão, se são malignos ou benignos. Isso mostra que existe um caminho promissor para trabalhos futuros, pois algumas abordagens anteriores citadas falharam em sua identificação pela insistência de classificação baseada na demonstração de que os códigos de *fingerprinting*, para uso ilegitimo, são diferentes dos códigos de *fingerprinting*, para usos legítimos. As falhas foram devidas às insistências para detectar apenas uma classe de scripts de *fingerprinting*, focadas em atributos específicos. Além disso,as publicações citadas também não obtiveram êxito para definir claramente a linha entre sites *fingerprinters* e sites legítimos.

Capítulo 4

WBF Analyzer

Este Capítulo descreve a solução proposta nesta dissertação, chamada **WBF** (*web Browser Fingerprinting*) **Analyzer**. Para tanto, é apresentada a arquitetura para coleta, extração e avaliação das chamadas estáticas.

4.1 Visão Geral

O objetivo proposto nesta pesquisa é identificar *Browser Fingerprinting*, extraídos de páginas web, a fim de aumentar a detecção deste tipo de rastreamento e fornecer um arcabouço para futuras contramedidas. Uma arquitetura foi elaborada para alcançar o objetivo proposto e é ilustrada na Figura 4.1. Ela começa com a coleta de dados por meio de *Web Crawler* (subseção 4.1.1). Em seguida, esses dados são usados na fase de pré-processamento para serem limpos de ofuscação (subseção 4.1.2). O resultado é então usado na fase de identificação (subseção 4.1.3), que com a ajuda de um conjunto de dicionários produz a saída do sistema.

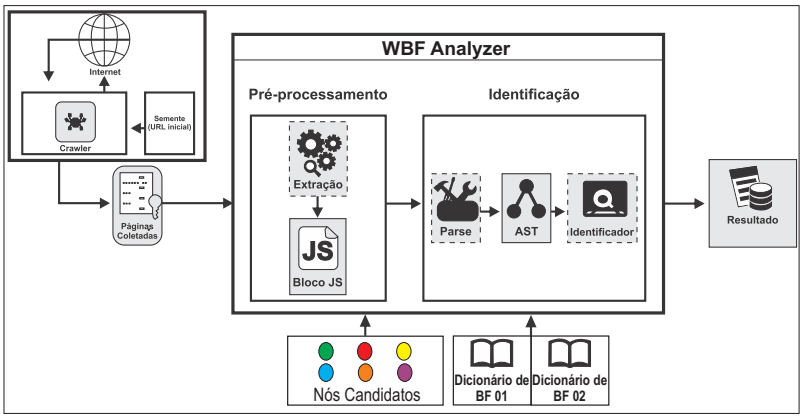


Figura 4.1: Arquitetura proposta. Fonte: Autor.

4.1.1 Coleta de dados

A **coleta de dados** é a processo da arquitetura que reúne códigos JavaScripts de URL que serão classificadas, de acordo com sua severidade ou não, em relação a aplicação de *Browser Fingerprinting*. Tal coleta é obtida através da busca e armazenamento do conteúdo de páginas web. Para tanto, um processo de *web crawling* é realizado, tendo como entrada diferentes URLs (*seeds*). No processo de coleta, um componente verifica se as URLs coletadas são válidas ou não. Esse procedimento permite eliminar URLs mal-formadas, bem como, entradas duplicadas e ausência futura de objetos.

É importante destacar que a coleta não precisa obrigatoriamente ser parte do processo de identificação de *Browser Fingerprinting*. Como alternativa ao processo de *Web Crawling*, uma base de códigos JavaScript pode ser passada como entrada para a WBF Analyzer.

4.1.2 Pré-Processamento

A primeira etapa da WBF Analyzer é o **pré-processamento** do código JavaScript, que basicamente contempla uma fase de extração que verifica e limpa o código do processo de ofuscação. A Figura 4.2 ilustra esse processo.

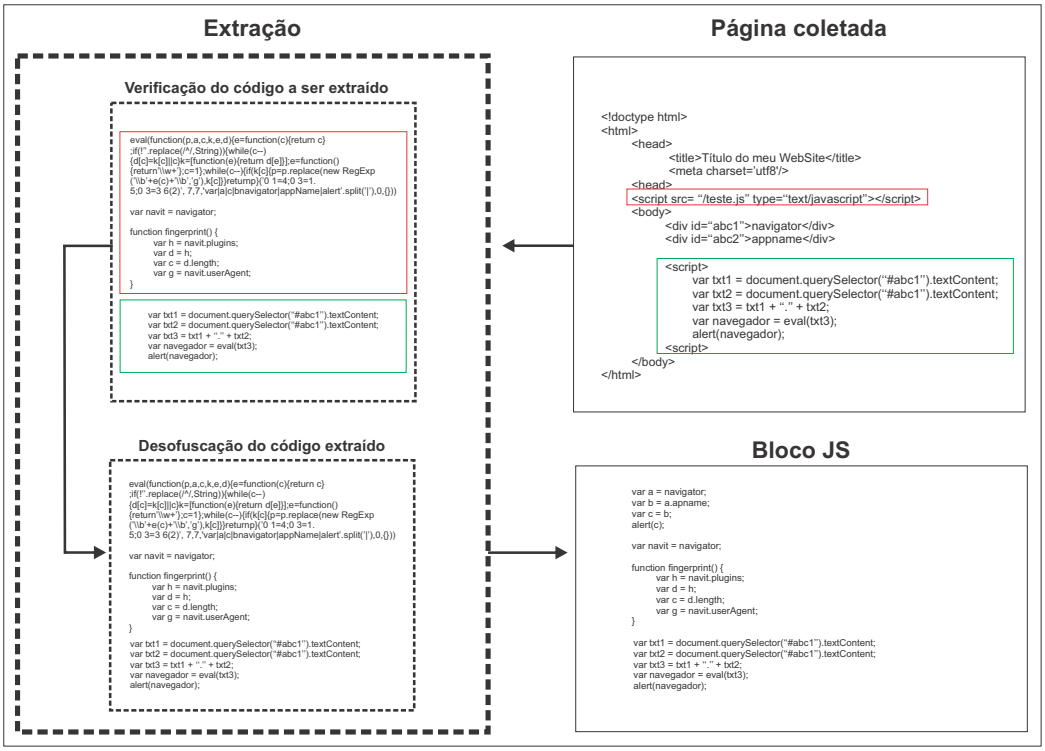


Figura 4.2: Exemplo da fase de extração. Fonte: Autor.

Em linhas gerais, seu funcionamento é o seguinte: após receber como entrada

a página coletada, é feita uma verificação dentro das tags $< html >< /html >$ para encontrar as subtags $< script >< /script >$, $< scriptsrc >< /script >$. A primeira tag $< script >< /script >$ fornece os códigos JavaScript que estão "in-line" na página, ou seja, foram declarados diretamente no corpo do documento *HTML*, e a segunda tag $< scriptsrc >< /script >$ fornece através do atributo src a localização dos scripts, que foram declarados externos ao corpo do documento *HTML*. Em seguida, ocorre a extração correta do código JavaScript. Além disso, tem-se nessa fase a limpeza da ofuscação, aplicada para que o código torne-se legível.

Ao final do tratamento, o código JavaScript é separado do HTML e limpo da ofuscação, conforme pode ser visto na Figura 4.2 (mais detalhes são explicados no Capítulo 5 em referência à utilização das bibliotecas BeatifulSoup, re, requests e Beautify), produzindo como saída um bloco JavaScript, formado apenas por códigos in-line e externos, que será enviado para a etapa de Identificação.

4.1.3 Identificação

Na etapa de identificação, o bloco JavaScript é recebido e um *parser* o transforma em uma AST, que é então repassada para o identificador. Nessa etapa, pode-se utilizar qualquer parser desde que ele reconheça os tokens da linguagem JavaScript. Após o identificador receber a AST como entrada, ele aplicará 3 regras necessárias para detecção das chamadas de *Browser Fingerprinting*.

A Figura 4.3 ilustra o identificador recebendo a AST, que é a representação abstrata do código a ser investigado como entrada, bem como os nós candidatos de onde serão extraídos os termos que irão gerar um escopo reduzido da AST. Também recebe como entrada o dicionário de *Browser Fingerprinting* que está dividido no dicionário de *Browser Fingerprinting* 01 e 02. O dicionário 01 será utilizado como ferramenta de suporte na normalização das chamadas e o dicionário 02 como ferramenta de suporte na classificação das chamadas.

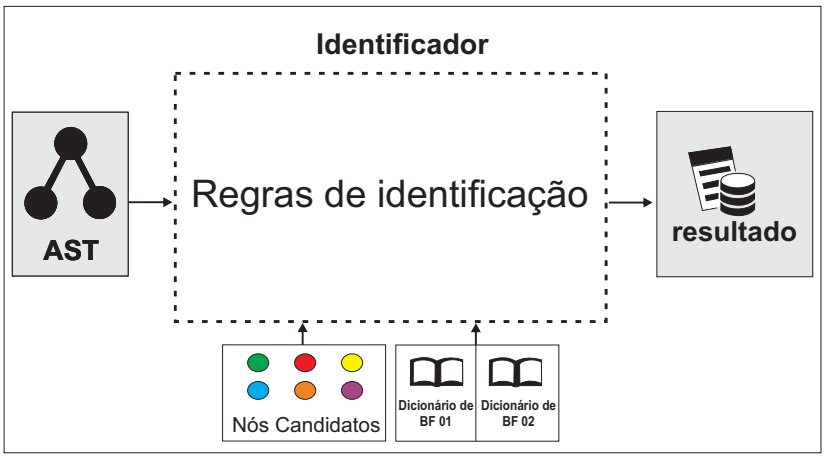


Figura 4.3: Identificador. Fonte: Autor.

O funcionamento do identificador é explicado através da execução de passos, chamados de regras de identificação, onde se tem a *regra de identificação 1 - extração*, a *regra de identificação 2 - normalização*, a *regra de identificação 3 - classificação*.

Na Figura 4.4, percebe-se que a *regra 1* (*extração*) recebe os nós candidatos como entrada. Esses são os nós da AST que serão visitados e seus conteúdos extraídos. O resultado dessa extração direcionada é um escopo reduzido que serve como entrada para a *regra 2* (*normalização*), que recebe também como entrada o dicionário de BF (*Browser Fingerprinting*) 01. Os termos extraídos, que possuem a relação *objeto + propriedade* e que forem compatíveis com o dicionário de BF 01, serão organizados no formato de uma chamada de objeto.

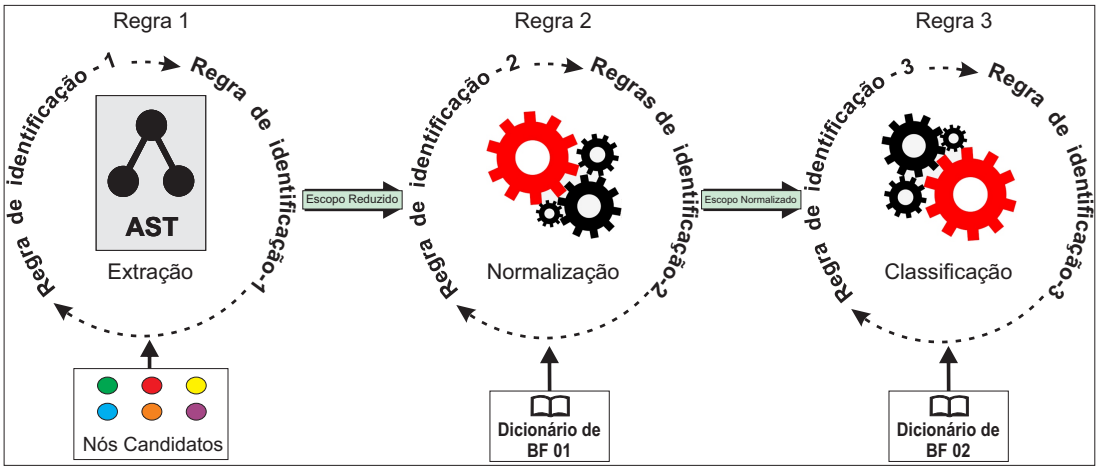


Figura 4.4: Regras de Identificação. Fonte: Autor.

Ainda na Figura 4.4, a regra 2 produz como saída um escopo normalizado que é enviado para a classificação, onde as chamadas serão contadas e avaliadas de acordo com o seu nível de risco. A classificação (*regra 3*), por sua vez, recebe como entrada o escopo normalizado e o dicionário de BF 02. O dicionário de BF 02 fará uma última avaliação comparativa antes da classificação para se ter a certeza de que os falsos positivos foram eliminados e produziram apenas chamadas a objetos, que estejam em conformidade com o dicionário de BF 02. Após essa avaliação, a classificação das chamadas é realizada segundo o método de Saraiva (2016)

4.1.3.1 Regra 1: Extração

Primeiramente, é necessário percorrer a(s) AST de entrada, assim, o conjunto de termos de interesse descritos na Tabela A.2 do apêndice A são percorridos e extraídos. A varredura é feita de forma insensível ao fluxo para garantir que todos os caminhos possíveis da AST sejam examinados.

Os resultados dessa verificação e extração são armazenados em um escopo reduzido, gerado para que não sejam feitas manipulações que envolvam a edição da AST. Com isso, pode-se armazenar os dados extraídos da AST em uma estrutura de

dados maleável como, por exemplo, uma lista encadeada. Evitando a perda de dados na AST e processos custosos como a inserção de registradores e de marcadores como feito em (Damasceno, 2017).

Na Figura 4.5, em (1), tem-se uma expressão de atribuição $a = navigator$. A princípio, não se pode fazer nada com essa linha de código. Porém, em tempo de execução, um script poderia facilmente transformá-la em $let\ a = navigator$ e ela deixaria de ser uma expressão de atribuição para ser uma declaração de variável e ***"a"*** poderia então compor uma outra expressão do tipo $let\ h = a.appName$. Assim, teria-se uma declaração de variável, que é membro de uma expressão, na qual se pode identificar que existe um objeto sendo atribuído ao operando ao lado esquerdo da expressão. Isso significa que o valor da propriedade desse objeto pode ser armazenado nessa variável.

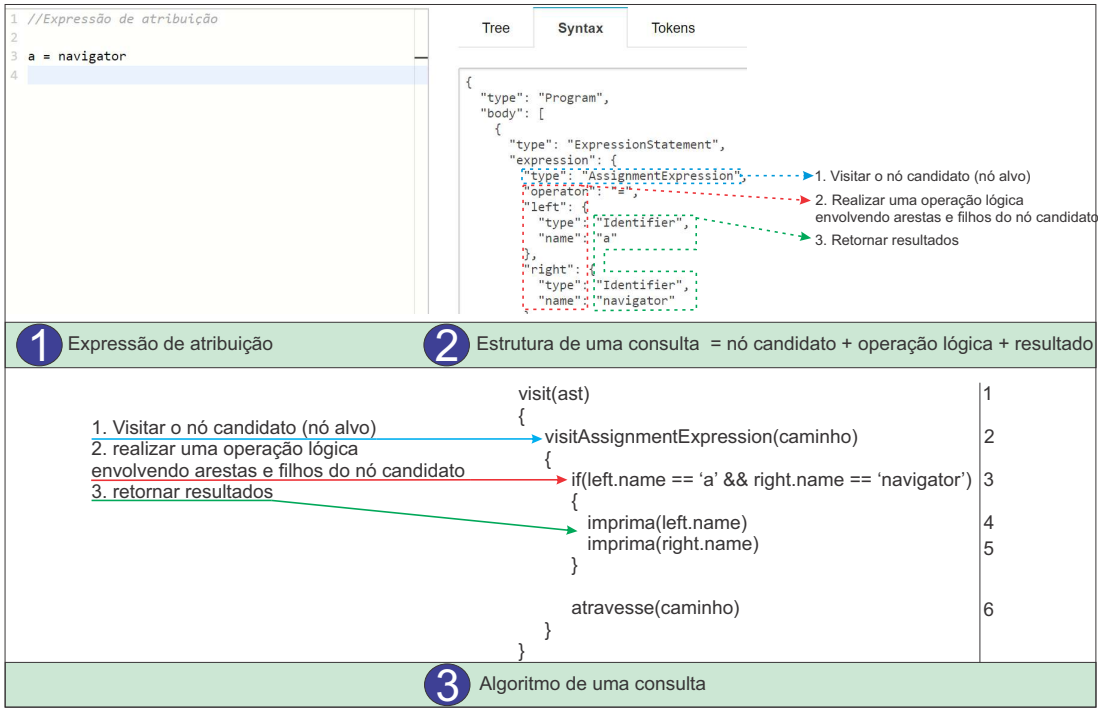


Figura 4.5: Algoritmo para extração do conteúdo de nós candidatos. Fonte: Autor.

Em (2), tem-se a estrutura de uma consulta que pode ser aplicada na AST. Essa estrutura é composta por um *nó candidato + uma operação lógica + um resultado*. Com esses três elementos, é possível gerar um algoritmo que possui a capacidade de consultar os nós da AST e extrair dados. Para se criar uma consulta em uma AST, primeiramente, tem-se que visitar o nó candidato, ou seja, o nó que previamente é o alvo. Depois, precisa-se realizar uma operação lógica que reflita o que se deseja extrair da AST. Após essa requisição, os resultados serão retornados e podem ser salvos em qualquer estrutura que se deseja armazenar os dados extraídos.

Em (3), tem-se o algoritmo de uma consulta. Na linha 1, a AST é passada

como entrada para a consulta (função *visit* recebe ast). Na linha 2 é necessário passar um caminho para chegar até o nó que se deseja visitar. Isso pode ser feito apenas invocando bibliotecas existentes associadas ao parser utilizado ou pode-se criar uma função de travessia baseada em algoritmos de busca em profundidade. Na linha 3, tem-se a operação lógica que permite a extração dos dados desejados. Nesse caso, diversas combinações lógicas podem ser aplicadas de acordo com a necessidade de extração. Nessa linha 3, é feito uma operação de decisão do tipo *IF* (estrutura de decisão) combinada com uma operação lógica do tipo *AND*, para extrair o nome do identificador esquerdo e o nome do identificador direito o qual será extraído somente se o valor do identificador esquerdo for igual a **"a"** e somente se o valor do identificador direito for igual a **"navigator"**. Nas linhas 4 e 5, pode-se imprimir na tela a informação extraída ou direcioná-la para uma estrutura de armazenamento de dados como uma lista, por exemplo. Na linha 6, tem-se uma função de travessia que recebe como entrada o caminho para se chegar até ao nó alvo, porém, como mencionado anteriormente, pode-se utilizar bibliotecas de travessia, associadas ao parser a ser utilizado para gerar a AST. Um conjunto de consultas reunidas com o objetivo de visitar diferentes nós da AST, ou até mesmo a AST inteira, transforma-se em um algoritmo de extração.

Para se extrair dados da AST é necessário ter um conjunto de nós candidatos que sofrerão a ação das consultas. Pode-se visitar a AST como um todo, como também pode-se visitar nós específicos da AST em que haja suspeita da ocorrência de atividades maliciosas. O problema de visitar a AST como um todo é que ao percorrer poucas linhas de código não se tem nenhum impacto sobre o tempo que se leva para realizar a extração dos dados. Porém, ao se visitar milhares de linhas de código tem-se uma explosão de nós que pode comprometer a performance de uma consulta. Por isso, serão apenas investigados nesta dissertação os nós candidatos que estão em conformidade com a Tabela A.2. Esses nós são potencialmente favoráveis ao *Browser Fingerprinting* em JavaScript, pois pode-se passar *array*, funções e objetos para uma variável. Como os dados coletados precisam ser armazenados na memória, mesmo que temporariamente, até serem salvos e enviados para as *Fingerprinters*, esses nós podem conter indicadores de *fingerprinting*.

A Figura 4.6 ilustra o processo ocorrido na Regra 1.

Na Figura 4.6, percebe-se que o identificador encontrou os nós *VariableDeclarator* e *MemberExpression* na estrutura da AST, realizou a extração dos termos *b, navigator e appName* e os realocou para um escopo reduzido. Um algoritmo de extração de termos pode ser empregado de acordo com os nós candidatos da Tabela A.2 ou nós específicos relacionados à consulta que se deseja realizar. Pode-se verificar as bibliotecas e documentações do parser, para se obter as definições dos nós candidatos, que for utilizado para gerar a AST. Pode-se observar na Figura 4.6, que existe, primeiramente, a identificação dos nós candidatos e, logo em seguida, é feita a extração dos termos. Vale destacar que nesse ponto não se aplica o dicionário de *Browser Fingerprinting 01* e nem o 02, pois o interesse é reduzir o universo de pesquisa aos possíveis contextos, em que podem ocorrer chamadas de objetos e criar um escopo reduzido, onde a normalização das chamadas pode ser aplicada.

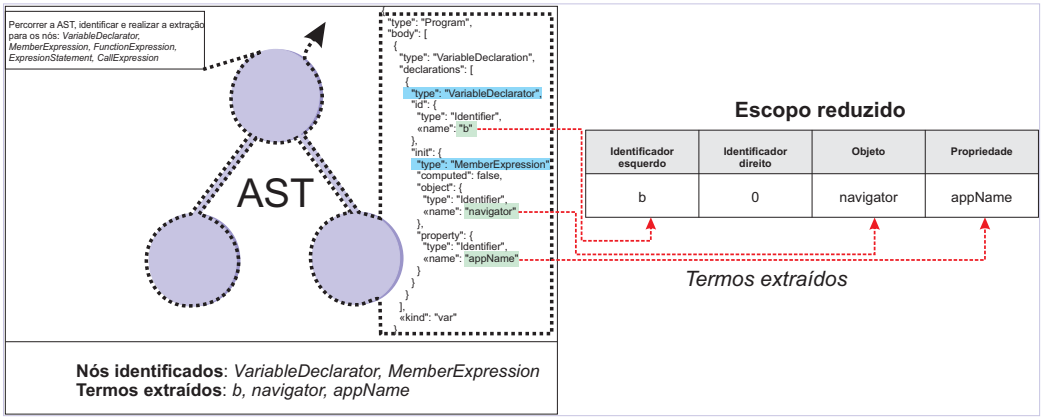


Figura 4.6: Aplicação da regra 1 para extração de elementos da AST. Fonte: Autor.

4.1.3.2 Regra 2: Normalização

A normalização é responsável por normalizar a relação entre objetos e propriedades para produzir as chamadas que dão origem ao escopo normalizado. Essa regra recebe como entrada um escopo reduzido, que é uma estrutura de dados resultante da fase de extração. Recebe o nome de escopo reduzido, pois houve uma redução dos dados que passaram pela *Regra 1 - extração*, sendo extraídos baseados em critérios chamados de nós candidatos. Essa estrutura é usada como base, onde serão aplicados processos de verificação, comparação e correspondência, com a finalidade de se produzir um escopo normalizado como saída para ser encaminhado para a fase de classificação.

Esses processos executam ações, conforme a Figura 4.7, para transformar:

- *objeto* + *propriedade*, em *objeto.propriedade*.
- *objeto* + *método*, em *objeto.método*.
- Casos especiais como *a.appName*, em *navigator.appName*.

Na Figura 4.8, tem-se um exemplo da estrutura que constitui o escopo reduzido. Ele é formado pelo identificador esquerdo, baseado no operando esquerdo de uma expressão de atribuição; identificador direito, baseado no operando direito de uma expressão de atribuição; objeto 0; objeto 1, objeto 2; propriedade 1 e propriedade 2, baseados na estrutura do dicionário de BF 01.

A normalização possui como saída um escopo normalizado, que é uma estrutura de dados produzida como saída da *Regra 2 - normalização* para ser enviada para a regra de classificação. Sua função é armazenar os dados que sofrerão a ação da contagem de frequência e da classificação das chamadas.

Na Figura 4.9, observa-se que a estrutura do escopo normalizado é composta por *Linha*, *Qtd* e *Chamada*. Linha armazena a origem da chamada encontrada no escopo reduzido. Nesse caso, os objetos e propriedades não estão concatenados. Essa informação

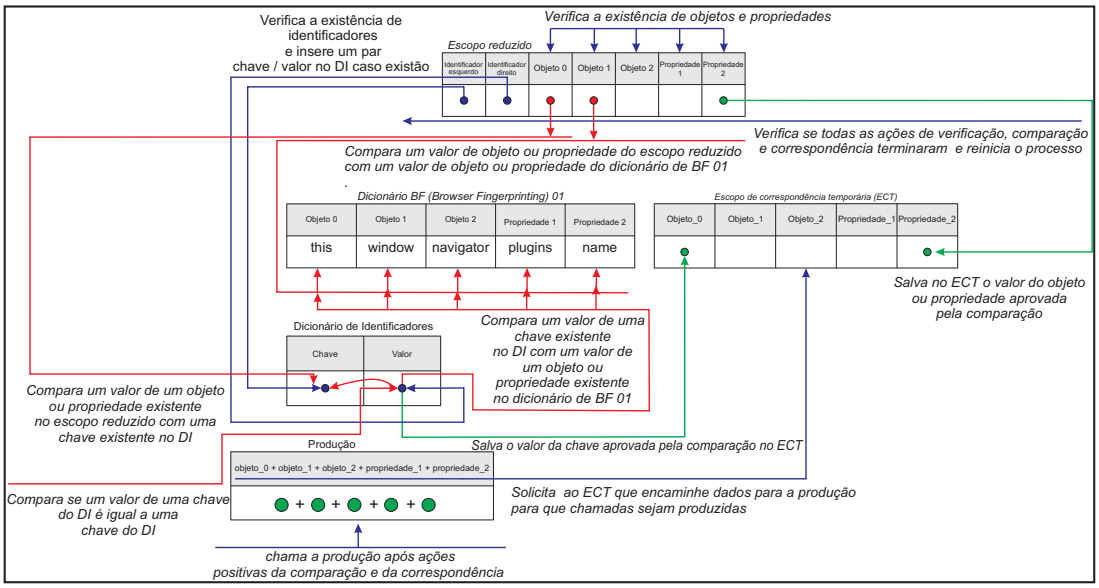


Figura 4.7: Ações da verificação, da comparação e da correspondência. Fonte: Autor.

Identificador esquerdo	Identificador direito	Objeto 0	Objeto 1	Objeto 2	Propriedade 1	Propriedade 2
h		d	plugins		name	

Figura 4.8: Escopo reduzido. Fonte: Autor.

é útil, pois se pode gerar um log do escopo reduzido para ser verificado de forma manual, posteriormente. Assim, pode-se avaliar, na implementação, se o algoritmo da produção das chamadas está funcionando corretamente. *"Qtd"* armazena um valor que representa a chamada encontrada. *"Chamada"* armazena o nome da chamada.

Linha	Qtd.	Chamada
1	1	navigator.plugins.name

Figura 4.9: Escopo normalizado. Fonte: Autor.

Na Figura 4.10, existem ferramentas de suporte para a verificação, para a comparação e para correspondência. Essas ferramentas foram divididas em ferramentas externas e internas.

Como ferramenta externa, tem-se o dicionário de *Browser Fingerprinting* 01, uma estrutura de dados em forma lista. Foi criado para armazenar os termos investigados que compõem o dicionário de BF 02. O dicionário de BF 01 possui uma estrutura, na

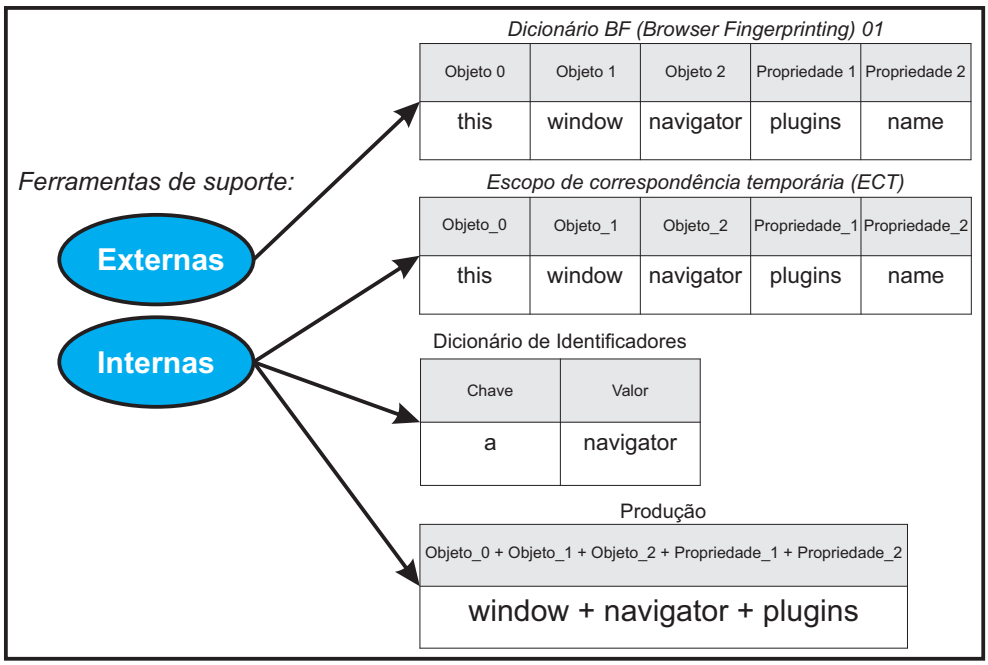


Figura 4.10: Ferramentas de suporte da normalização. Fonte: Autor.

qual os objetos e propriedades são representados como *tokens*. Cada *token* está num nível do objeto ou da propriedade referente a sua categoria. Dessa forma, pode-se armazenar um objeto 0, que pode ser *navigator*, e uma propriedade 1, que pode ser *plugins*. Como o dicionário possui 5 níveis de profundidade, pode armazenar objetos que sejam compostos por até três níveis de objetos e dois níveis de propriedades. O dicionário de BF 01 é usado pelo processo de comparação para que os objetos e as propriedades do escopo reduzido sejam identificados.

Já as ferramentas internas são compostas por:

- Escopo de correspondência temporária (ECT)
- Dicionário de identificadores
- Produção

O ECT (Escopo de Correspondência Temporária) pode ser qualquer estrutura que permita armazenamento de dados como, por exemplo, uma lista. O processo de correspondência utiliza o ECT para salvar os valores de objetos e propriedades que serão utilizados pela ferramenta de produção. Esse escopo temporário é limpo pelo processo de verificação.

O dicionário de identificadores é uma estrutura de dados baseada em objetos e possui sempre uma chave que aponta para um determinado valor. É usado pelo processo de verificação para armazenar referências a casos especiais, como *b.appName*, e pelo processo de comparação para avaliar o valor das chaves geradas pela verificação.

A produção é um processo, uma sub-rotina da verificação utilizada para produzir uma chamada de objeto após a avaliação dos processos de comparação e de correspondência. A produção age sobre os dados armazenados pelo ECT.

Processo de Verificação

O processo de Verificação é responsável por: (i) verificar a existência de valores nos identificadores, nos objetos e nas propriedades do escopo reduzido; e (ii) executar ações com base nas verificações.

Pode-se observar na Figura 4.11 que a verificação sempre inicia avaliando a existência do identificador direito (1). Caso ele exista, ela ignora objetos e propriedades e verifica se o identificador esquerdo existe (2). Caso o identificador esquerdo exista, a verificação, então, salva os valores do identificador esquerdo e do identificador direito no dicionário de identificadores (esse dicionário é um par chave e valor). A verificação salva o valor do identificador esquerdo como uma chave (4) e salva o valor do identificador direito como um valor correspondente a essa chave (5). Nesse caso, no dicionário de identificadores, o identificador esquerdo passa a ser uma chave que aponta para um valor, que é o valor encontrado pela verificação no identificador direito. Essas ações são necessárias, pois se esse cenário for identificado será necessário normalizar um caso especial do tipo *b.appName* ou *navigator.c*, onde tem-se que saber quem é **"b"** e quem é **"c"**. Deve-se saber quem são esses valores que estão estabelecendo relações do tipo de *objeto.propriedade*. Caso não exista valor no identificador direito, o identificador esquerdo é ignorado e a verificação avalia a existência de valores no *objeto 0*, em seguida no *objeto 1*, após, no *objeto 2*, depois na *propriedade 1* e, finalmente, na *propriedade 2* (3), exatamente nessa ordem.

A verificação também é a responsável por indicar que todos os identificadores, objetos e propriedades na lista de normalização atual foram verificados. A verificação somente vai para a próxima normalização após finalizar a primeira. Após a verificação detectar que todos os identificadores, objetos e propriedades foram avaliados e que objetos ou propriedades foram salvos pelo processo de correspondência, a verificação então chama a ferramenta de produção (6), uma sub-rotina da verificação, e solicita os valores armazenados no ECT (7) para, então, produzir uma chamada normalizada. Após a chamada ser produzida, a verificação limpa o ECT (8) e reinicia o processo de normalização (9).

Na Figura 4.12, em (1), *Verificando a existência do identificador direito*, a verificação está avaliando o item relativo à intersecção da coluna 2 com a linha 1, avaliando se o identificador direito existe. Essa avaliação é muito importante, pois se o identificador direito existir, isso significa que se pode ter um caso especial do tipo ofuscação de strings de substituição de palavra chave. Em (2), *Ignorando objetos e propriedades*, observa-se que a verificação sinaliza com uma descrição de *Identificador existente*, na intersecção da coluna 2 com a linha 1, informando que o identificador direito existe. Na sequência, na interseção da coluna 3 com a linha 1, observa-se que o objeto 0 foi ignorado, descrição *Objeto ignorado*. O mesmo acontece para o objeto 1 e para o objeto 2. Além disso, são ignoradas as propriedades 1 e 2, ambas com a

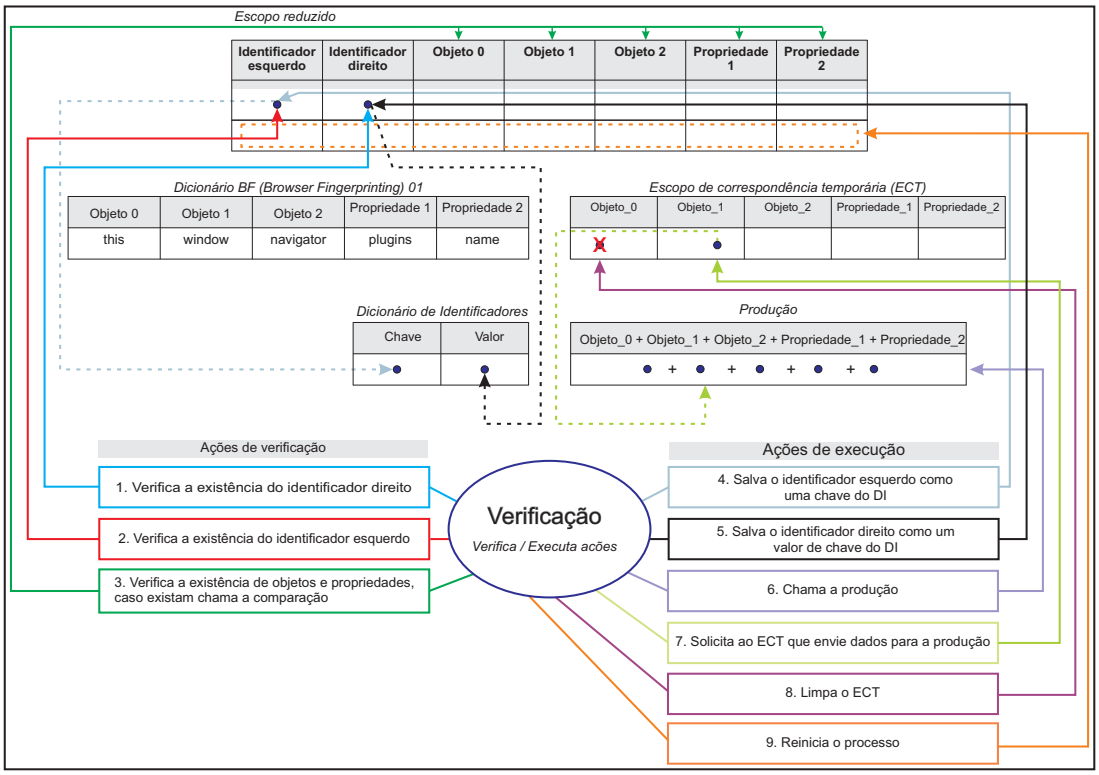


Figura 4.11: Processos da verificação. Fonte: Autor.

descrição de *Propriedade ignorada*. Portanto, nessa etapa, a verificação ignora, como visto, objetos e propriedades, pois trata-se de uma atribuição, na qual um valor do lado esquerdo foi atribuído para um operando do lado direito.

Em (3), *Verificando a existência do identificador esquerdo*, ainda na Figura 4.12, na intersecção da coluna 1 com a linha 1, a verificação está avaliando se o identificador esquerdo existe. Pode-se observar que o identificador direito já foi verificado e objetos e propriedades foram ignorados. Em (4), *Existência de identificadores esquerdo e direito confirmada*, o identificador esquerdo e o direito possuem a descrição de *Identificador existente*. Essa verificação é importante, pois para que sejam criados uma par chave e valor no dicionário de identificadores é necessário que o identificador esquerdo e o direito existam. Em (5), *Inserindo identificador esquerdo como uma chave no dicionário de identificadores*, o identificador esquerdo é inserido como uma chave no dicionário de identificadores. Essa chave apontará para o valor do identificador direito ao ser acessada. Em (6), *Inserindo identificador direito com um valor no dicionário de identificadores*, o identificador direito é inserido como o valor da chave que representa o identificador esquerdo. Dessa maneira, pode-se consultar o valor do identificador direito acessando a chave *"b"*.

Na Figura 4.13, a verificação avalia um caso em que o identificador direito não existe. Isso significa que uma expressão que consiste de um identificador, que recebe

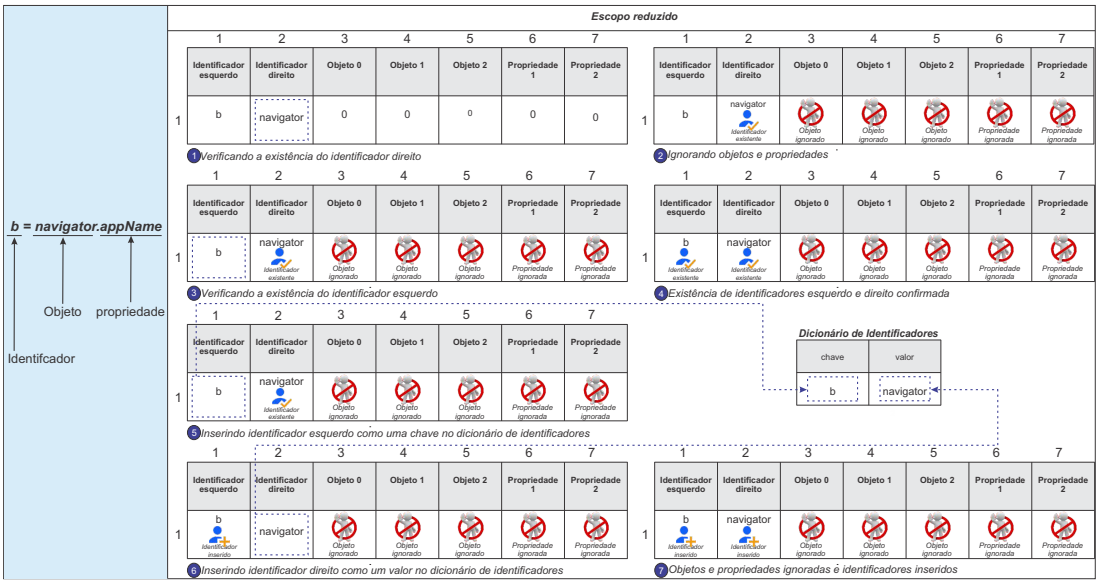


Figura 4.12: Passos da verificação caso o identificador direito exista. Fonte: Autor.

como atribuição a expressão $b = navigator.appName$, foi detectada. Essa expressão contém um identificador b que recebe um objeto (navigator) + uma propriedade (appName). Nesse caso, em (1) *Verificando a existência do identificador direito*, a verificação constata que o identificador direito não possui valores e, portanto, não existe. Em (2), *Ignorando o identificador esquerdo*, a verificação ignora o identificador esquerdo, pois o direito não existe, no caso o *"b"*, e passa a avaliar o *objeto 0*, (3) *verificando a existência do objeto 0*, o *objeto 1*, (4) *Verificando a existência do objeto 1*, o *objeto 2*, (5) *Verificando a existência do objeto 2*, a *propriedade 1*, (6) *Verificando a existência da propriedade 1*, a *propriedade 2*, (7) *Verificando a existência da propriedade 2* nessa ordem. Se a verificação avalia que existe um objeto ou uma propriedade, ela chama a *comparação*, senão ela avalia o próximo objeto. É importante ressaltar que a verificação somente avalia outra linha quando tem certeza que todos os objetos em ordem foram avaliados, levando sempre em consideração que primeiro deve-se avaliar a existência do identificador direito para que ações de decisão sejam tomadas.

Na Figura 4.14, percebe-se na linha 2 do escopo reduzido que em (1) a verificação constatou que não existem mais itens para serem avaliados, pois todos estão com uma descrição de *Identificador ignorado, Identificador inexistente, Salvo, Objeto inexistente, Objeto inexistente, Salvo, Propriedade inexistente*. Com isso, a verificação em (2) solicita ao ECT que envie os dados armazenados. Em (3), o ECT envia os dados armazenados para a produção. Em (4) a produção transforma os dados que recebeu do ECT em uma chamada de objeto. Nesse caso, foram enviados para a produção *navigator* e *appName* que foram transformados em *navigator.appName*. Nesse momento, o escopo normalizado é criado para ser a estrutura de dados de saída da regra 2 - normalização. Um campo *Linha* e um campo *Qtd* são adicionados ao escopo normalizado com as

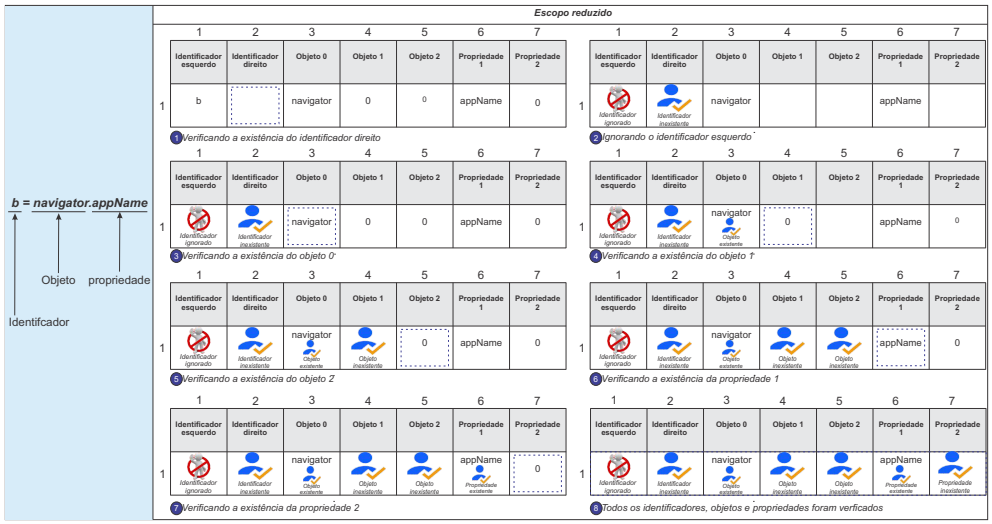


Figura 4.13: Passos da verificação caso o identificador direito não exista. Fonte: Autor.

finalidades já explicadas anteriormente. Em (5.1), a verificação detecta que existe uma chamada produzida no escopo normalizado. Em (5.2) ela adiciona a linha referente a chamada e em (5.3) ela adiciona um número que representa a quantidade da chamada. Em (6), após a chamada ser produzida, a verificação limpa o ECT. Em (7), após limpar o ECT, a verificação reinicia o processo de normalização e avalia uma nova linha.

Processo de Comparação

O processo comparação é responsável por: (i) aplicar comparação baseada em dicionário; e (ii) executar ações com base nas comparações.

A comparação é responsável por avaliar se os valores apontados pelas chaves do escopo reduzido são iguais aos valores apontados pelas chaves do dicionário de *Browser Fingerprinting* 01 ou do dicionário de identificadores. A comparação é um processo chamado pela verificação. Esse processo realiza primeiramente uma comparação com o dicionário de BF 01.

Na Figura 4.15, em (1) *Compara um valor de objeto do escopo reduzido com um valor de objeto do dicionário de BF 01*, em (2) *Compara um valor de propriedade do escopo reduzido com um valor de propriedade do dicionário de BF 01*, se a comparação não for satisfeita uma nova comparação é realizada em (3) *Compara um valor de um objeto do escopo reduzido com uma chave do DI* ou em (4) *Compara um valor de propriedade do escopo reduzido com uma chave do DI*. Se uma comparação de um valor de objeto ou propriedade do escopo reduzido com uma chave do DI for satisfeita, então uma comparação em (5) *do valor de uma chave do DI com um valor de objeto ou de propriedade do dicionário de BF 01* é realizada. Se houver a necessidade de normalizar um caso em que exista uma cadeia de relações de atribuições, uma comparação como em (6) *compara o valor de uma chave do DI com uma chave do DI* é realizada até que seja encontrado um valor de chave que satisfaça uma comparação com o dicionário de BF 01.

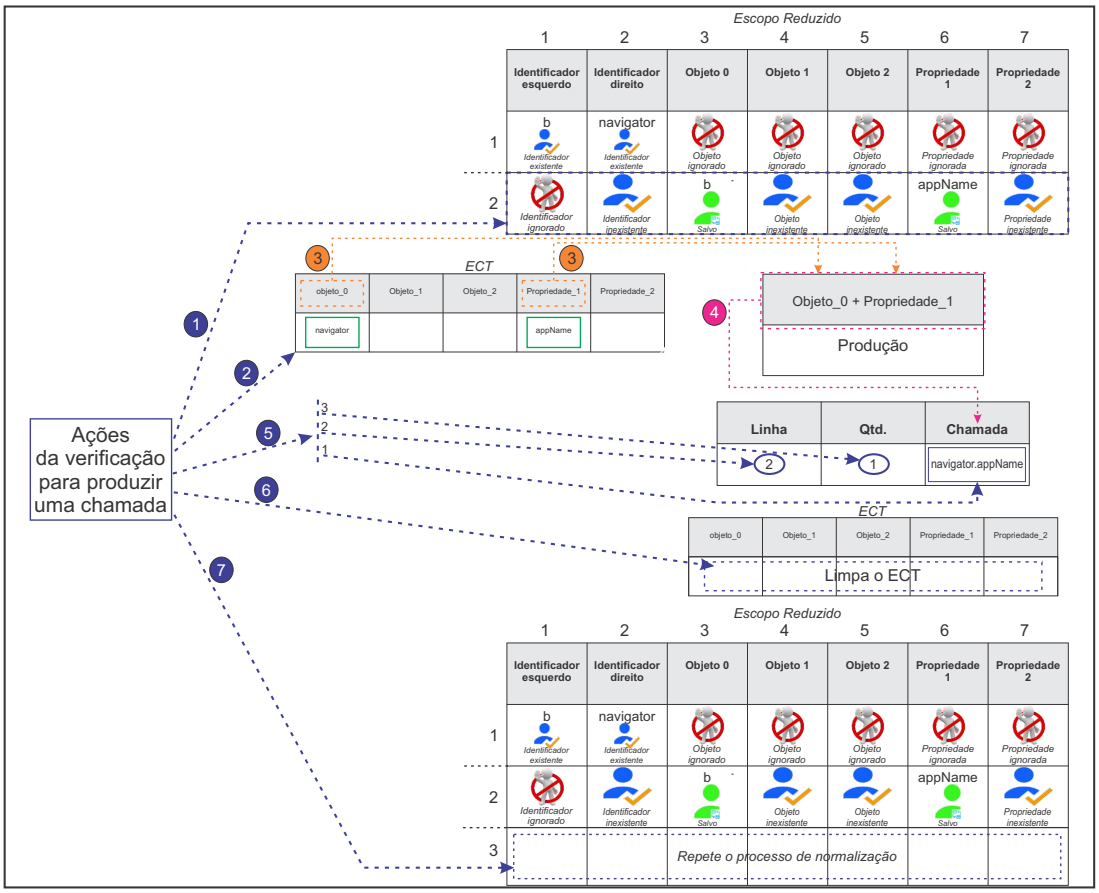


Figura 4.14: Passos da verificação para realizar a produção de uma chamada. Fonte: Autor.

A comparação avalia casos do tipo objeto inteiro como em *navigator.plugins.name* orientando o processo de correspondência a salvar os valores que satisfazem as comparações diretamente no *ECT*. Quando a comparação encontra um caso especial do tipo *b.appName*, no qual se tem uma ofuscação de string de palavra-chave, primeiramente ela deve descobrir quem é **"b"** ou se esse **"b"** faz parte de uma relação estabelecida em uma cadeia de atribuições. Quando a comparação descobre quem é o verdadeiro valor de *"b"*, ela aciona a correspondência e a orienta a salvar no ECT um valor encontrado no DI que satisfez uma comparação com o dicionário de BF 01. Nesse caso, a correspondência sabe que deve transferir valores que estão no dicionário de identificadores diretamente para o ECT.

Na Figura 4.16, a comparação está avaliando um item que está na intersecção da coluna 3 com a linha 2 do escopo reduzido. No caso o *objeto 0* está sendo avaliado. Em (1), a comparação avalia se o valor do *objeto 0* do escopo reduzido existe no dicionário de *Browser Fingerprinting*, caso ele não exista a comparação vai para o dicionário de identificadores. Em (2), a comparação avalia se o valor do *objeto 0* do escopo reduzido

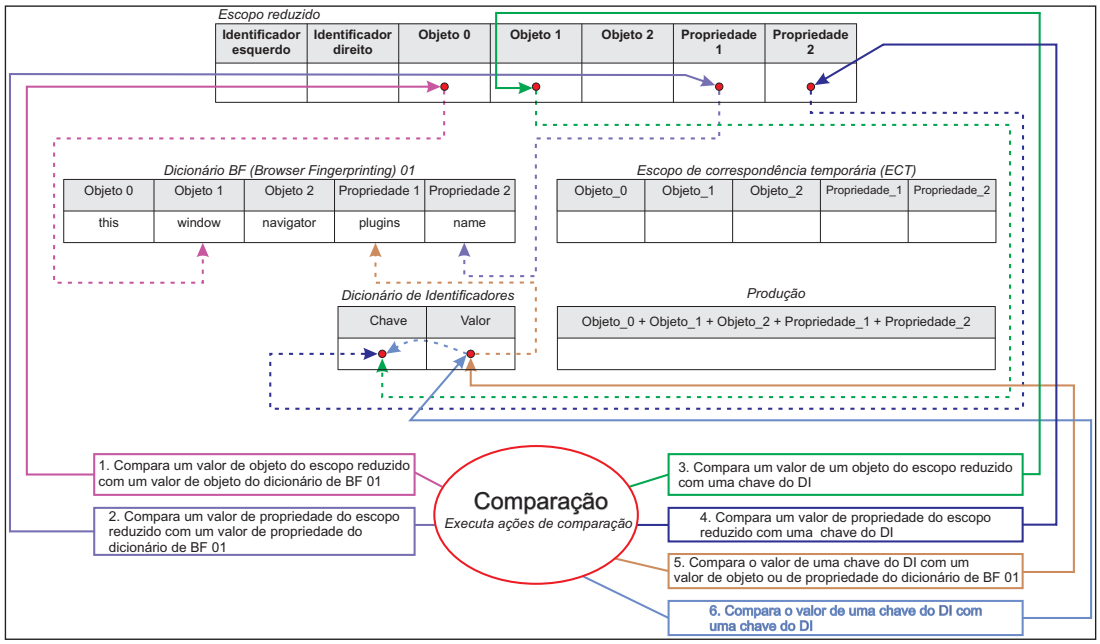


Figura 4.15: Processos da comparação. Fonte: Autor.

existe no dicionário de identificadores. Nesse caso, a comparação avalia se o valor do *objeto 0* é igual ao valor de uma chave encontrada no dicionário de identificadores. Se a chave do DI for igual ao objeto procurado a comparação, então, acessa o valor da chave e compara esse valor com o dicionário de BF 01. Em (3), se a comparação for verdadeira a correspondência, é chamada, senão a comparação continua executando, pois nesse caso tem-se que examinar todas a relações existente (*chave / valor*) no DI para que o objetivo de encontrar uma comparação válida com o dicionário de BF 01 seja alcançado. No caso da Figura 4.16, percebe-se que a comparação não encontrou o objeto *"b"* no dicionário de *Browser fingerprinting* 01 na primeira tentativa de comparação, então, nesse caso, ela avalia o dicionário de identificadores procurando por uma chave que seja igual ao valor do objeto que ela está avaliando. Ao encontrar um valor de chave que seja igual ao valor do objeto avaliado, ela acessa e avalia o valor da chave. Se o valor da chave existir no dicionário de BF 01, a comparação chama a correspondência, pois existe um valor de uma chave no dicionário de identificadores que corresponde a um valor de objeto existente no dicionário de BF 01, senão ela continua executando até que não hajam mais comparações a serem realizadas.

Na Figura 4.17, percebe-se na interseção da coluna 6 com a linha 2 do escopo reduzido que a comparação está avaliando a propriedade 1 com valor *appName*. Nesse caso, a comparação em (1) detectou que existe uma valor igual no dicionário de BF 01. Com isso ela chama a correspondência para salvar o valor da propriedade encontrada no ECT. Observa-se no exemplo, também, que na intersecção da coluna 3 com a linha 2 o objeto 0 com o valor de **"b"** possui uma descrição de salvo, indicando que a

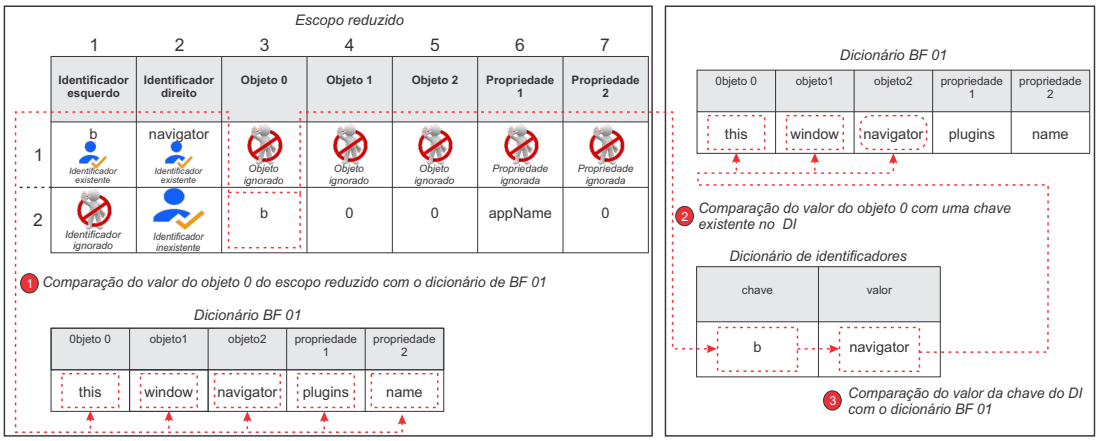


Figura 4.16: Processo de comparação aplicado em um objeto 0 ofuscado. Fonte: Autor.

correspondência já transferiu esse objeto para o ECT.

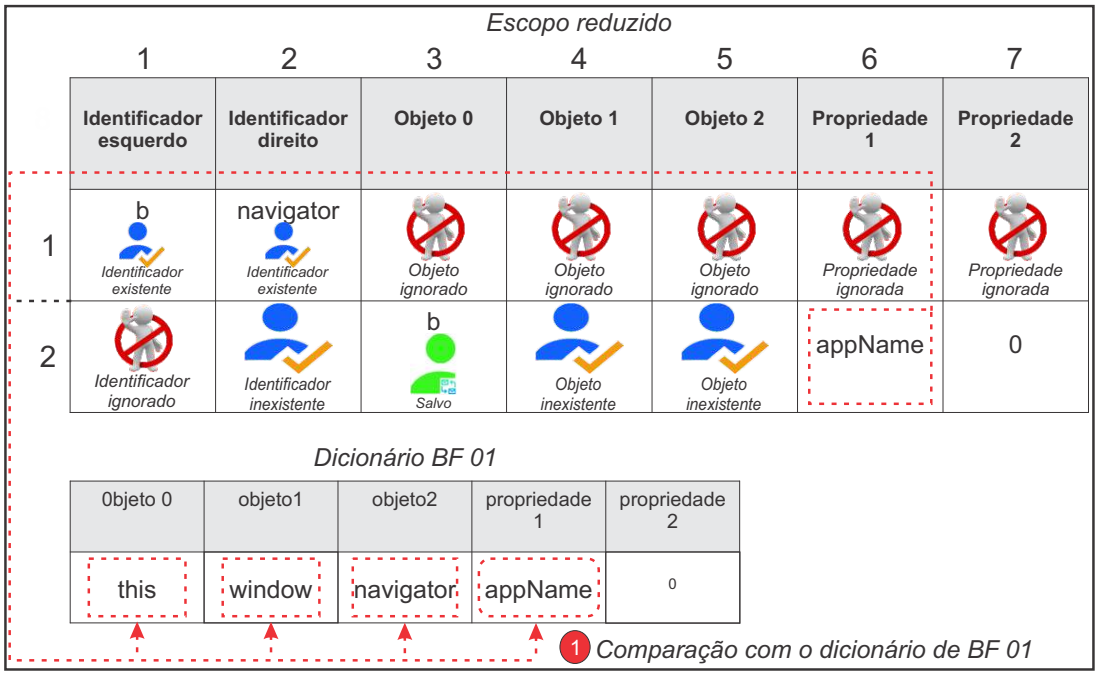


Figura 4.17: Processo de comparação aplicado em um objeto 0 não ofuscado. Fonte: Autor.

A Figura 4.18 demonstra como a comparação utiliza o dicionário de identificadores para resolver um problema de ofuscação de strings de substituição de palavra chave em uma cadeia de atribuições relacionadas. No exemplo da Figura, tem-se uma cadeia formada por um *"a"* que recebeu *"navigator"* como atribuição. Em seguida, o *"a"*

foi atribuído para *"b"*, que foi atribuído para *"c"*, que foi atribuído para *"d"* e finalmente *"h"* recebeu *"d.plugins"*. Para resolver esse problema, primeiramente, a comparação inicia uma avaliação *Top Down* no DI, onde em (1) o valor do objeto 0, no caso *"d"*, é comparado a primeira chave encontrada no DI. Então temos uma comparação entre *"d"* e *"a"* que resulta em negativo. Em (2), a comparação repete o processo comparando *"d"* com a chave *"b"*, depois com a chave *"c"* e finalmente com a chave *d*, encontrando uma comparação positiva. A comparação, então em (5), acessa o valor da chave *"d"*, que no caso é *"c"*, e em (6) compara o valor *"c"* com o dicionário de BF 01. Porém, o valor não é encontrado. Assim, uma comparação *Botton up* a partir desse ponto é iniciada em (7), onde o valor da chave *"d"*, ou seja, *"c"*, é comparado com a chave anterior que no caso é *"c"*. A comparação acessa o valor da chave *"c"* em (8) e em (9) compara *"b"* com o dicionário de BF 01. Como o resultado não foi alcançado, em (10) o valor da chave *"c"* é comparado com a chave *"b"*. Em (11), o valor da chave *"b"* é acessado e em (12) o valor *"a"* é comparado com o dicionário de BF 01 resultando em uma comparação negativa. Em (13), o valor da chave *"b"* é comparado com a chave anterior, então tem-se uma comparação onde o valor *"a"* é comparado com a chave *"a"* resultando em positivo. Em (14), a comparação acessa o valor da chave *"a"* e em (15) a comparação valor da chave *"a"* pergunta se *navigator* é igual a dicionário de BF 01?. Na primeira e na segunda comparação tem-se uma negação. Na terceira comparação tem-se um resultado positivo. Como um resultado positivo foi alcançado, a comparação chama a correspondência (16) para salvar o valor *"navigator"*, que foi encontrado no objeto 2 do dicionário de BF 01, no objeto_0 do ECT.

Esse processo pode ser aplicado até que uma condição seja satisfeita ou a comparação tenha percorrido todo o dicionário de identificadores. Se uma comparação de um *valor de chave* com uma *chave* não resultar em positivo, a comparação continua avaliando, apenas pulando as chaves que não resultam em positivo, isso para a avaliação feita em *Top Down* e para a avaliação feita em *Botton up*.

Processo de Correspondência

O processo de correspondência é responsável por salvar no ECT os objetos e propriedades que foram aprovados pela comparação.

Na Figura 4.19, a correspondência é um processo que em (1) *Salva a correspondência de um valor de objeto do escopo reduzido com um valor de objeto do dicionário de Browser Fingerprinting 01 no ECT*, em (2) *Salva a correspondência de um valor de propriedade do escopo reduzido com um valor de propriedade do dicionário de Browser Fingerprinting 01 no ECT* e em (3) *Salva a correspondência do valor de uma chave do DI com um valor de objeto ou de propriedade do escopo reduzido*. A correspondência atua sobre o escopo reduzido, sobre o ECT e sobre o dicionário de identificadores. Tem como principal ferramenta de suporte o ECT e sua missão é salvar objetos e propriedades aprovados pela comparação.

A Figura 4.20 apresenta um exemplo de um caso especial do tipo *b.appName*, onde o valor de *"b"* é igual a *navigator*. Em (1), a correspondência acessa uma chave no dicionário de identificadores que seja igual ao valor do objeto 0, ou seja, *"b"* e em

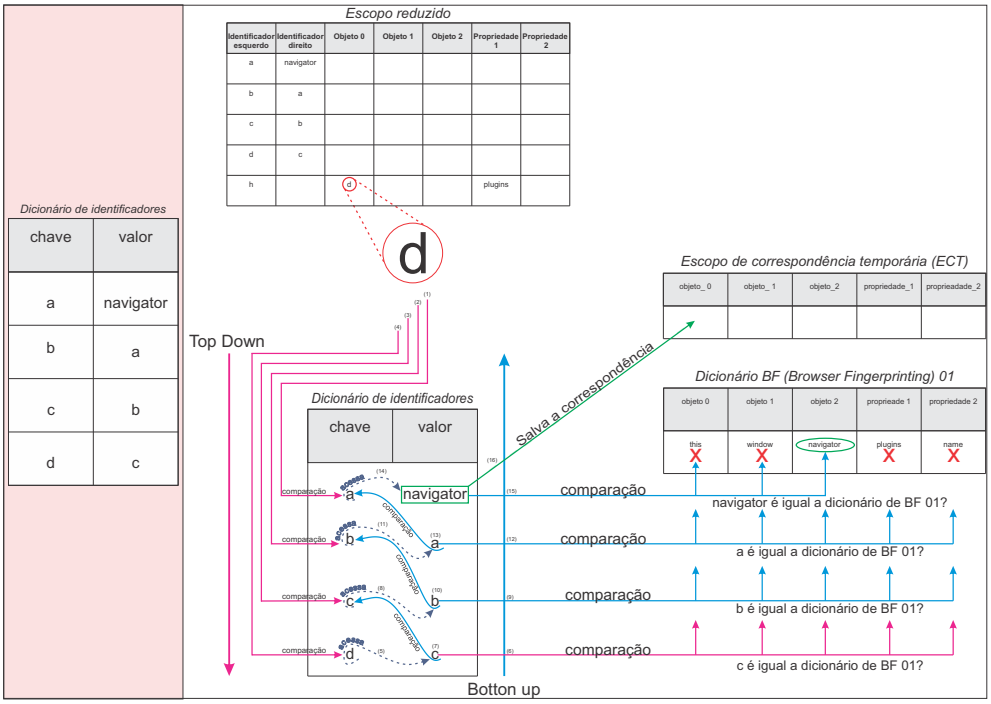


Figura 4.18: Aplicação do dicionário de identificadores para resolver o caso especial. Fonte: Autor.

(2) salva o valor dessa chave no ECT na posição relativa ao escopo reduzido.

Na Figura 4.21, em (1), a correspondência salva o valor da propriedade 1 do escopo reduzido no ECT. Como esse valor nativamente é uma propriedade de um objeto e foi aprovado pela comparação, não existe a necessidade da correspondência verificar o dicionário de identificadores.

4.1.3.3 Regra 3 - Classificação

A Regra 3 (**Classificação**) é responsável por aplicar a contagem de frequência, a classificação de risco da chamada e a classificação de risco da página web. A regra 3 recebe como entrada o escopo normalizado, produto da regra 2, e produz como saída o escopo classificado. Com a aplicação da regra 3, pode-se definir qual o risco de *Browser fingerprinting* que uma página web representa para o usuário.

Na Figura 4.22, o escopo classificado é uma estrutura de dados de saída, onde tem-se os campos *Frequência*, *Chamada*, *Risco da chamada* e o *Risco da página*. A Frequência armazena os números de vezes que uma chamada aparece no código fonte. A Chamada armazena o nome do objeto chamado. O Risco da chamada armazena a classificação de risco da chamada. O Risco da página armazena a classificação da página.

A classificação de risco da chamada e da página web são aplicadas segundo a

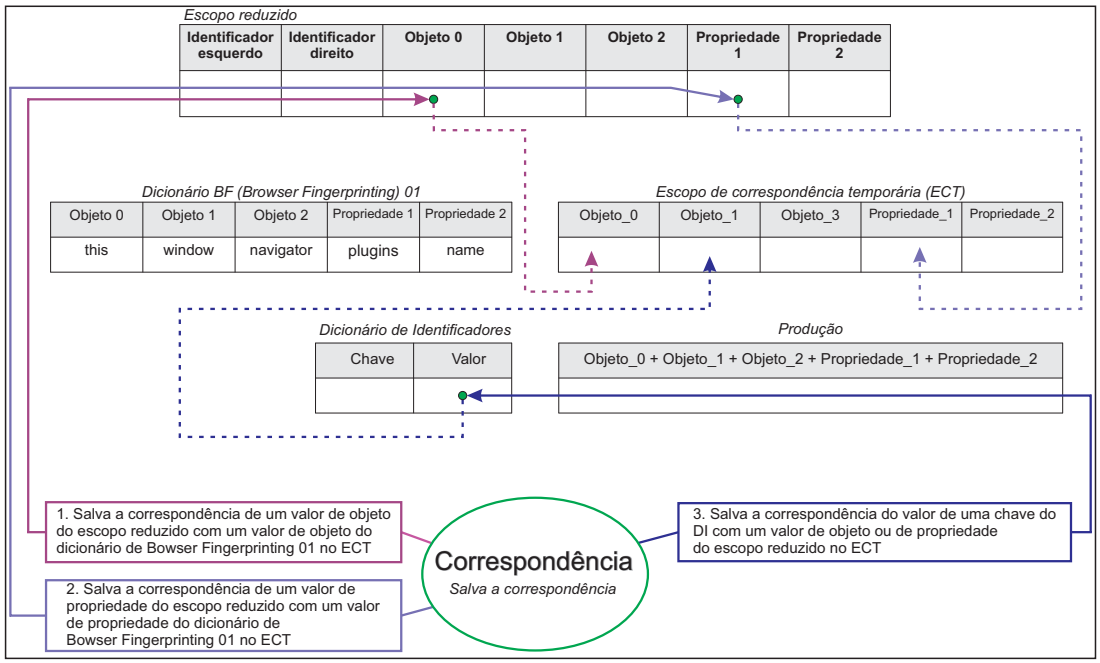


Figura 4.19: Processo de correspondência. Fonte: Autor.

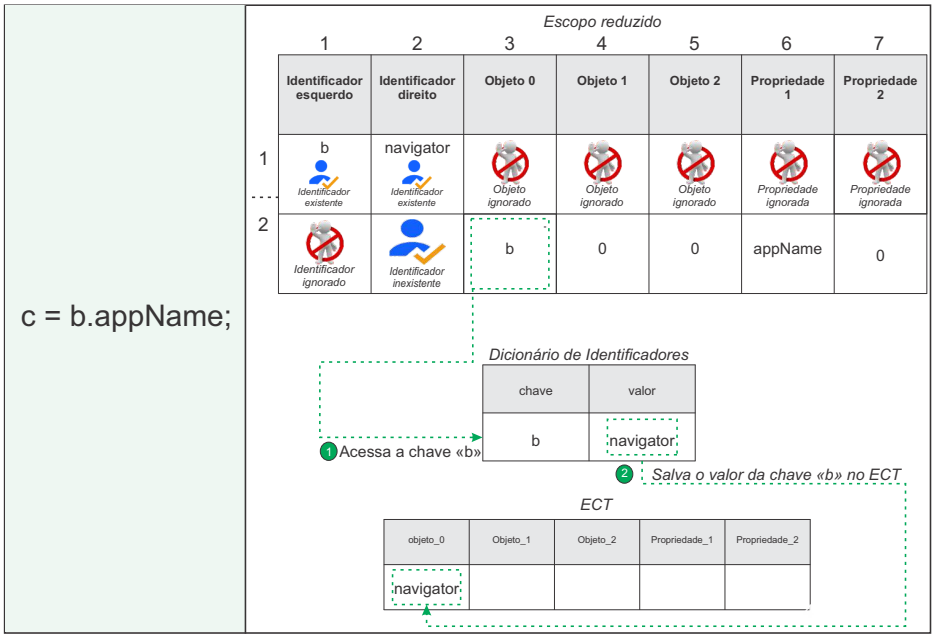


Figura 4.20: Correspondência sendo aplicada a chave b do objeto b do caso especial $b.appName$. Fonte: Autor.

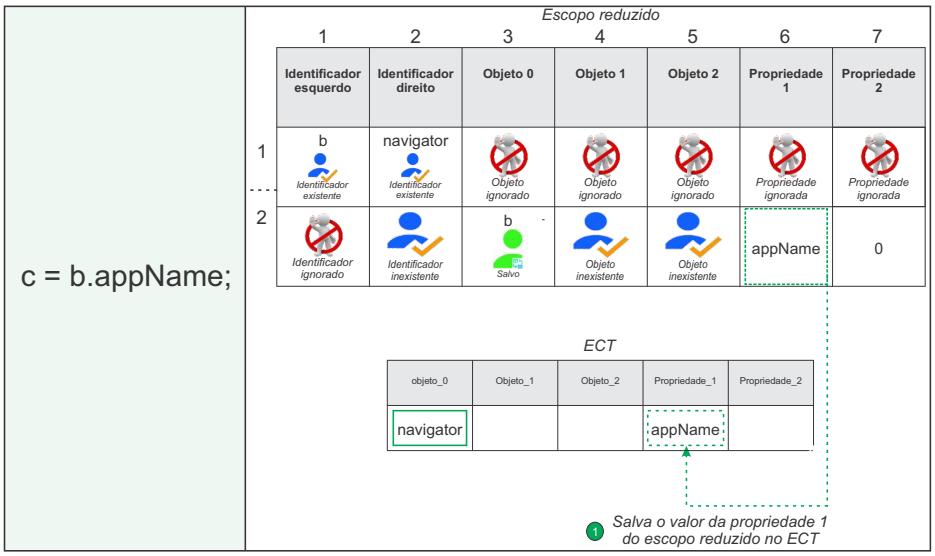


Figura 4.21: Correspondência sendo aplicada à propriedade 1 *appName* do caso especial *b.appName*. Fonte: Autor.

Escopo classificado

Frequência	Chamada	Risco da chamada	Risco da página
3	navigator.plugins.name	Médio	Alto
1	canvas.getImageData()	Alto	

Figura 4.22: Estrutura do escopo classificado. Fonte: Autor.

metodologia criada por (Saraiva, 2016), que gerou três níveis classificatórios:

- **Baixo**: nessa classificação encontram-se objetos que são costumeiramente usados pelos sites para identificar informações padrões a respeito do dispositivo do usuário, com a finalidade de adequação de conteúdo, fazendo com que o site torne-se responsivo adaptando às configurações da máquina do usuário. Segundo Saraiva (2016), não foram relatados ataques à segurança ou à privacidade que sejam ligados ao uso desses objetos. Objetos como *Window*, *Navigator*, *Screen*, *Document* e quase todas as suas propriedades foram as que se enquadraram em outras classificações como a classificação média e alta.

- **Médio**: esse nível classificatório é aplicado a objetos que podem retornar informações a respeito dos plugins instalados na máquina do usuário. Scripts que coletam essas informações agem como farejadores para malwares, pois, se as condições

de *fingerprinting* forem satisfatórias, o malware pode ser acionado em tempo de execução, para explorar vulnerabilidades nos plugins listados pelo script de *fingerprinting*. Saraiva (2016) relata que dados de plugins bancários ou alguma informação de algum software específico pode ser fornecida por essas verificações. A biblioteca *Modernizr* e a verificação de plugins e *MimeTypes* encontram-se nessa classificação.

- **Alto**: as chamadas a objetos como $canvas.toDataURL$, $canvas.getImageData$, $window.history$ e $navigator.getUserMedia$ estão nessa classificação. Segundo Saraiva (2016), vários ataques relatados usaram um desses objetos ou uma combinação deles para fazerem *fingerprinting* e explorarem vulnerabilidades da máquina do usuário.

Para se obter o risco da chamada e da página web, o método de (Saraiva, 2016) foi adaptado para ser aplicado e executado como segue:

1. Contagem de frequência
2. Classificação de risco da chamada
3. Classificação de risco da página

A regra 3, primeiramente, realiza a contagem de frequência que atua sobre o escopo normalizado e o escopo classificado. Essa contagem é de suma importância, pois ela indica a quantidade de vezes que uma chamada aparece num código fonte e será usada como uma variável no cálculo de classificação de risco da página web.

A *contagem de frequência* será demonstrada através das Figuras 4.23, 4.1.3.3, 4.25, 4.26. Também será adotada uma numeração que vai de (1) a (20) para explicar detalhes dessas figuras. O Ícone de um dicionário com uma lupa com um sinal $"+"$ na cor verde indica uma busca por uma chamada no dicionário de BF 02 que resultou em positivo. O ícone de uma lupa no escopo classificado indica uma busca de uma chamada no escopo classificado. Essa busca pode retornar um $"x"$ na cor vermelha, se a consulta não for positiva ou um ícone de verificação na cor verde se a consulta for positiva. Os caracteres $f++$ junto ao símbolo de uma engrenagem significam que uma frequência foi incrementada.

Segundo a Figura 4.23, em (1), uma comparação da chamada *navigator.plugins.name* é feita com o dicionário de BF 02. Em (2), como a comparação foi positiva a chamada *navigator.plugins.name* é adicionada ao campo *Chamada* do escopo classificado. Em (3), como a chamada foi aprovada, sua frequência é adicionada. Em (4), tem-se a primeira chamada adicionada e a sua respectiva frequência.

Após a primeira chamada ser adicionada, a próxima chamada é avaliada. Na Figura 4.1.3.3, em (5), compara-se a chamada com o dicionário de BF 02. Como a resposta é positiva, em (6), se uma chamada existente no escopo normalizado existir no escopo classificado, apenas sua frequência é adicionada (7). Nota-se que em (8), o ícone de verificação indica que a chamada existe e por este motivo não precisa ser adicionada

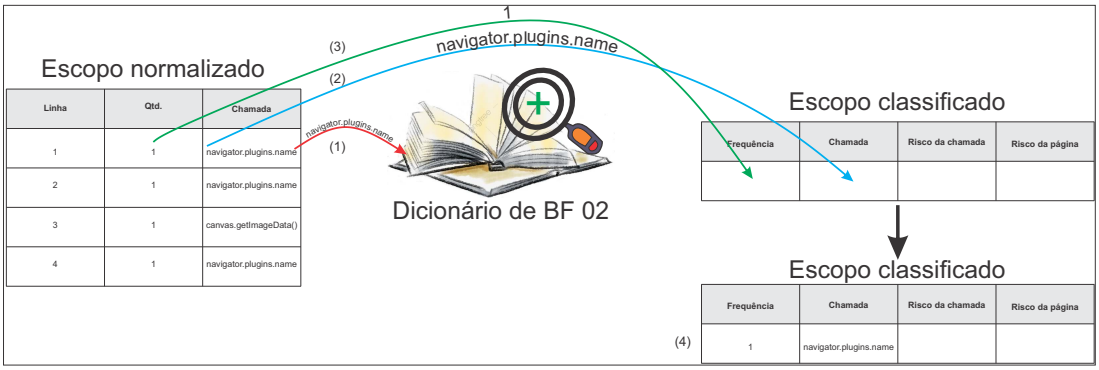
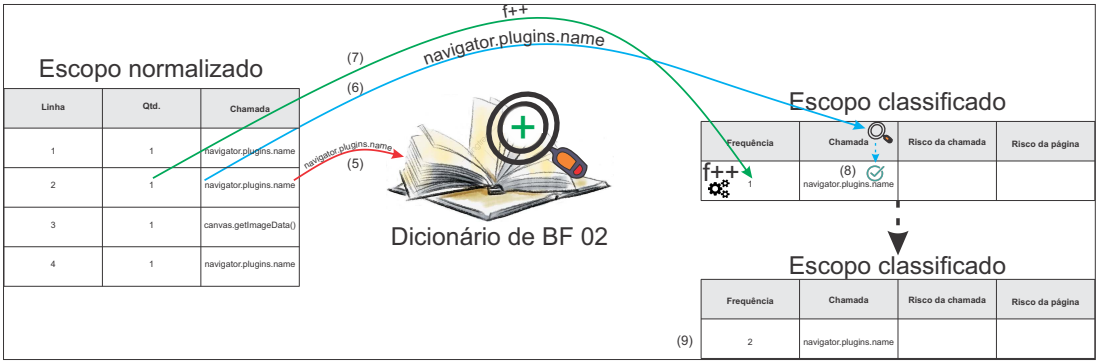


Figura 4.23: Contagem de frequência 01. Fonte: Autor.

novamente. Em (9), tem-se o escopo classificado indicando que existem duas chamadas de objeto do tipo *navigator.plugins.name*



Contagem de frequência 02. Fonte: Autor.

Na Figura 4.25, a classificação continua avaliando o escopo normalizado. A chamada *canvas.getImageData()* é comparada ao dicionário de BF 02 (10). Como a comparação é positiva, em (11) uma consulta ao escopo classificado é realizada para verificar se a chamada já foi adicionada ao escopo. Em (12), após a classificação realizar uma busca, ela detecta que o valor da chamada é diferente dos valores salvos. Com isso em (13), a chamada *canvas.getImageData()* é adicionada. Após a chamada ser adicionada, em (14), a contagem de frequência referente a chamada é adicionada. Finalmente, em (15) tem-se a chamada *canvas.getImageData()* e sua frequência fazendo parte do escopo classificado.

Na Figura 4.26, em (16), a classificação realiza uma nova comparação avaliando *navigator.plugins.name*. Como a comparação foi positiva, em (17) uma busca por *navigator.plugins.name* é realizada. Em (18), é constatado que essa chamada já existe no escopo classificado e então, em (19), sua frequência é adicionada. Em (20), tem-se o escopo classificado com todas as chamadas e frequências adicionadas. O escopo classificado está pronto para receber a aplicação da classificação de risco da chamada.

A classificação do risco da chamada indica, segundo os níveis classificatórios

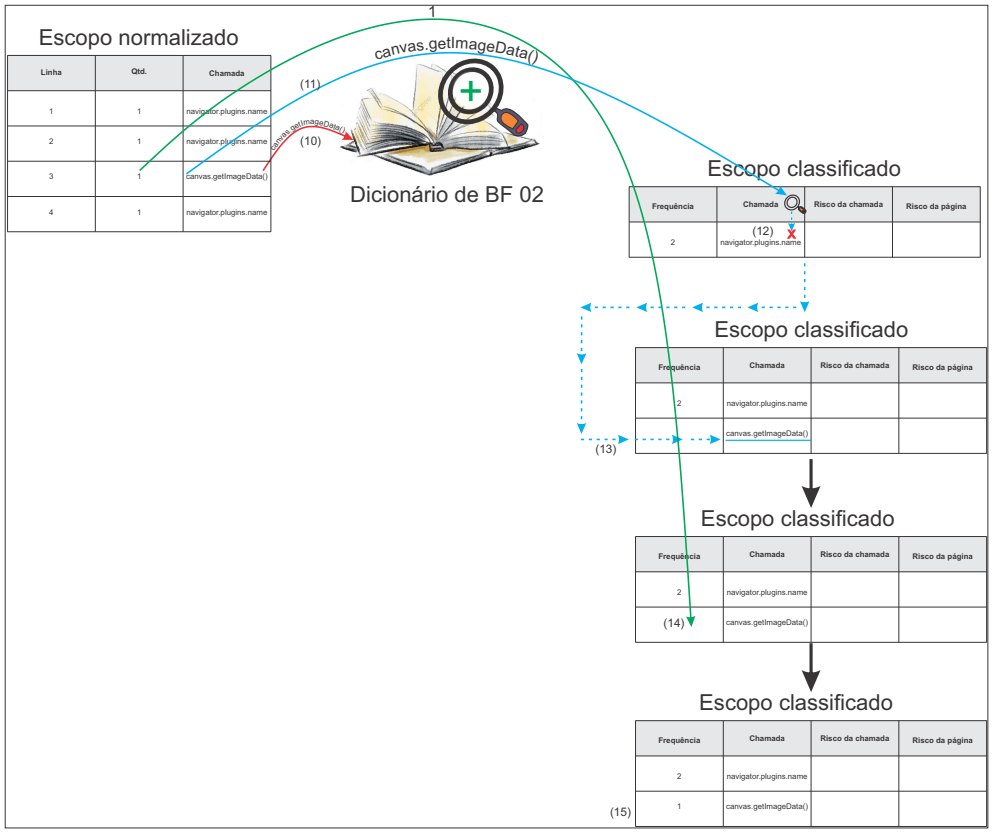


Figura 4.25: Contagem de frequência 03. Fonte: Autor.

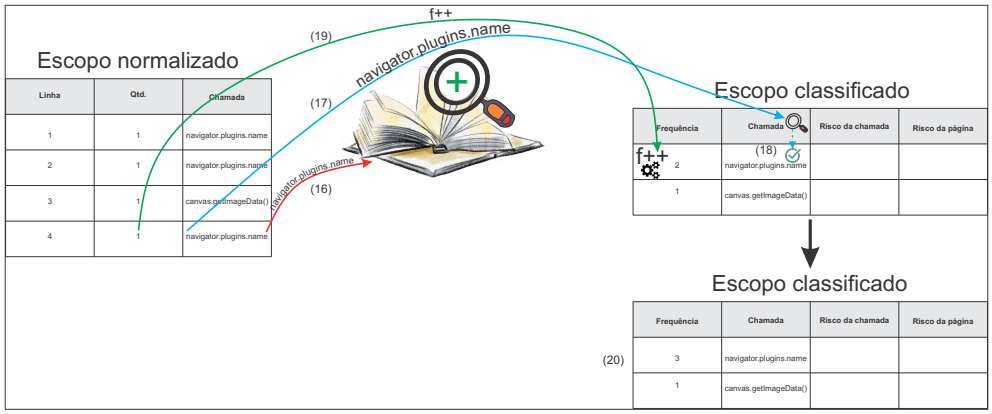


Figura 4.26: Contagem de frequência 04. Fonte: Autor.

de (Saraiva, 2016), o risco que uma chamada de objeto possui quando utilizada para fazer *fingerprinting*. O risco da chamada representa uma variável que será utilizada no cálculo de classificação de risco da página web.

Primeiramente, para aplicar-se à classificação de risco da chamada, é necessário converter o dicionário de BF 02 para os níveis classificatórios. Na Figura 4.27, o dicionário de BF 02 é passado como entrada para os níveis classificatórios e, então, tem-se um novo dicionário divido em três níveis classificatórios: *Nível 1 - Baixo*, *Nível 2 - Médio*, *Nível 3 - Alto*.

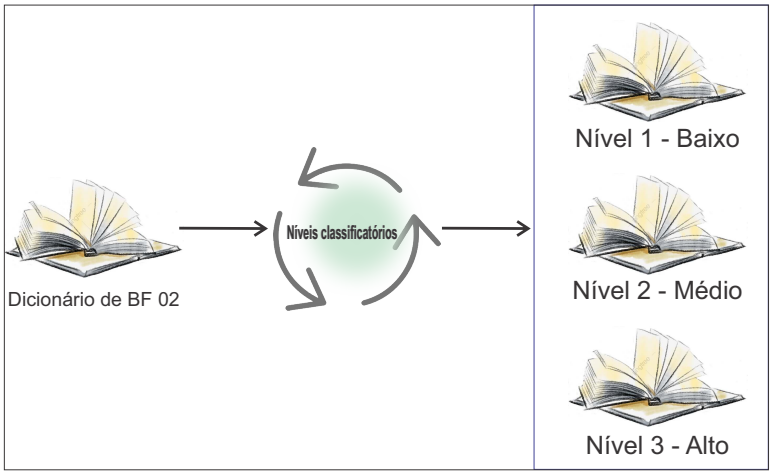


Figura 4.27: Conversão do dicionário de BF 02. Fonte: Autor.

É importante ressaltar que durante a contagem de frequência, o dicionário de BF 02 se mantém de acordo com o dicionário de BF 02 explicado na seção 4.1.4. A conversão foi necessária para ajustar o dicionário aos níveis classificatórios, dessa forma, as comparações são realizadas buscando as chamadas referentes ao seu nível de risco.

Na Figura A.1 localizada no apêndice A, pode-se verificar todas chamadas a objetos de *fingerprinting* alocadas em seus respectivos níveis de risco. Durante o processo de conversão, o dicionário de BF 02 mantém sua forma original que representa uma chamada de quatro formas distintas: (1) *this.window.navigator.userAgent*; (2) *window.navigator.userAgent*; (3) *this.navigator.userAgent* e (4) *navigator.userAgent*.

A classificação de risco da chamada consiste em aplicar uma comparação do valor existente no campo *Chamada* do escopo classificado com os níveis classificatórios do dicionário de BF 02 modificado. Na Figura 4.28, a chamada *navigator.plugins.name*, em (1), é comparada com as chamadas alocadas no nível 1 - Baixo. Porém, como essa chamada não existe nesse nível, ela não é encontrada, e um *"x"* vermelho indica a não existência da chamada. Em (2), o nível 2 - Médio é consultado. Um sinal de positivo na cor verde na lupa de busca indica que a chamada foi localizada. Em (3), uma seta na cor verde indica que a classificação média é aplicada. Em (4), tem-se o escopo classificado com o campo *Risco da chamada* preenchido com a classificação *Médio*. Como a chamada foi localizada no *Nível 2 - Médio*, a consulta ao *Nível 3 - Alto* foi desprezada.

Na Figura 4.29, tem-se uma aplicação da classificação de risco alto da chamada. Na Figura percebe-se que em (1), uma consulta foi realizada no *Nível 1 - Baixo*, porém não se obteve resultado positivo. Em (2), o *Nível 2 - Médio* foi consultado e

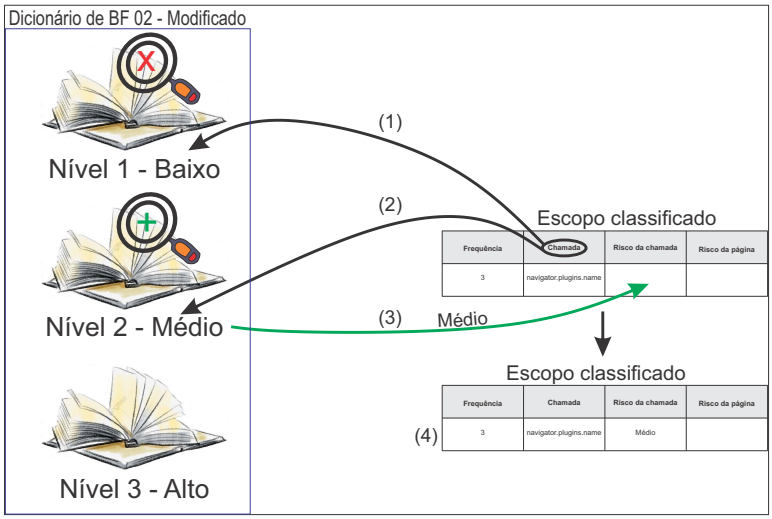


Figura 4.28: Classificação de risco da chamada - nível médio. Fonte: Autor.

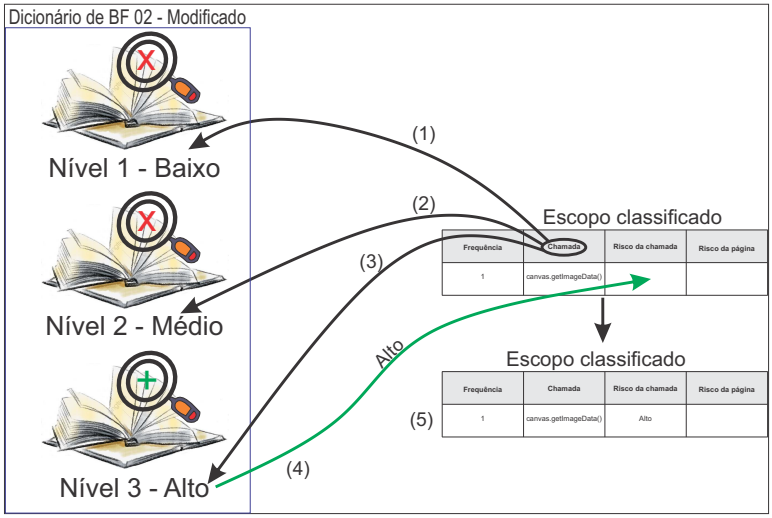


Figura 4.29: Classificação de risco da chamada - nível alto. Fonte: Autor.

novamente um resultado positivo não foi alcançado. Em (3), a chamada de objeto *canvas.getImageData()* foi localizada, pois um sinal de positivo foi indicado pela consulta aplicada. Em (4), a classificação de risco alto é aplicada. Em (5), tem-se o escopo classificado indicando a frequência da chamada *canvas.getImageData()* e seu nível de risco. Aplicar a contagem de frequência e a classificação de risco da chamada são importantes, pois irão representar variáveis no cálculo de média ponderada aplicada para que se possa determinar o risco da página web.

Na Figura 4.30, percebe-se que o escopo classificado está modificado. Agora se tem 3 *chamadas* de objeto do tipo *navigator.plugins.name* com classificação de

risco *Médio*. Tem-se também, 1 chamada de objeto do tipo *canvas.getImageData()* com classificação de risco *Alto*.

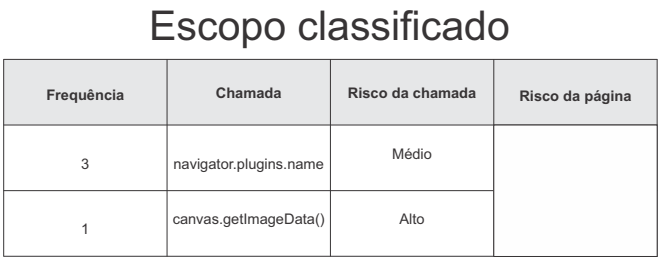


Figura 4.30: Contagem de frequência e classificação de risco aplicadas. Fonte: Autor.

Agora que se tem a frequência da chamada e do risco da chamada, pode-se aplicar a *Classificação de risco da página*. Para isso, precisa-se primeiramente atribuir pesos aos níveis classificatórios, conforme a Tabela 4.1.

Tabela 4.1: Peso do risco da chamada

Risco da chamada	Peso
Baixo	1/100
Médio	1/10
Alto	1/1

Dess forma, aplica-se o cálculo de média ponderada definido pela equação em (4.1):

$$\frac{\sum_{i=1}^{n} r_i f_i}{\sum_{i=1}^{n} r_i} \tag{4.1}$$

Na Figura 4.31, tem-se, como resultado final da regra de classificação, o escopo classificado contendo *3* chamadas de objeto do tipo *navigator.plugins.name* com o risco da chamada sendo *"Médio"*; *1* chamada de objeto do tipo canvas.getImageData() com o risco da chamada sendo *"Alto"*; e o risco da página sendo *"(alto")*.

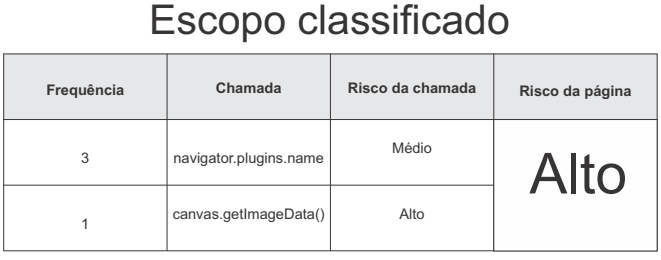


Figura 4.31: Escopo classificado: resultado final. Fonte: Autor.

4.1.4 Dicionário de Browser Fingerprinting

O dicionário de *Browser Fingerprinting* foi construído, após intensa pesquisa na literatura, por objetos contendo propriedades e métodos que possuem a capacidade de retornar os valores das variáveis de ambiente do navegador, através de uma chamada JavaScript organizada na forma ***objeto.propriedade*** ou ***objeto.método()***. Foram selecionados 11 objetos, contendo 43 propriedades e 14 métodos, totalizando 68 termos investigados, conforme a Tabela A.3 do apêndice A. A Seção 4.2 descreve todos esses objetos.

O Dicionário de *Browser Fingerprinting* foi dividido em dois dicionários que foram denominados de dicionário de BF 01 e dicionário de BF 02 (*onde BF significa Browser Fingerprinting*).

O dicionário de BF 01 será utilizado durante a execução da *regra 2 - normalização*, servindo como ferramenta de suporte externa, para que as comparações sejam realizadas durante a normalização das chamadas. Dessa forma, pode-se encontrar a relação objeto e propriedade como, por exemplo: *this.window.navigator.plugins.name*, bem como a relação objeto e método, como por exemplo, *this.window.navigator.getUserMedia()*. O dicionário de BF 01 está organizado em uma estrutura de 5 níveis de profundidade e os objetos e propriedades estão fatiados e alocados em níveis referentes a objetos ou propriedades. Dessa forma, pode realizar comparações com quatro níveis de objetos e um nível de propriedade; três níveis de objetos e um nível de propriedade; dois níveis de objetos e um nível de propriedade; um nível de objeto e um nível de propriedade. Vale destacar que, até o presente momento, nenhuma pesquisa avaliou tantos objetos e nessa profundidade de níveis.

A Tabela 4.2 detalha como o dicionário de BF 01 está estruturado em níveis de profundidade e como uma chamada de objeto está fatiada em *Objeto 0*, *Objeto 1*, *Objeto 2*, *Propriedade 1* e *Propriedade 2*. Essa estrutura é necessária, pois as chamadas estão dispostas de forma tokenizada na estrutura da AST. Por está razão, precisa-se aplicar a normalização da chamada, para que se possa estabelecer as relações entre objetos e propriedades.

Tabela 4.2: Estrutura do dicionário de BF 01

Objeto 0	**Objeto 1**	**Objeto 2**	**Propriedade 1**	**Propriedade 2**
this	*window*	*navigator*	*plugins*	*name*

O dicionário de BF 02 tem um formato mais simples e diferente do 01, não possuindo os objetos fatiados e divididos em níveis. Ao contrário, esse dicionário contém os termos em sua forma final, onde tem-se o objeto e suas propriedades sendo chamadas pela notação **"."**, que permite invocar a propriedade de um objeto ou seu método. A finalidade desse dicionário é ser utilizado durante a regra de classificação, onde ele será aplicado, permitindo que sejam apenas classificadas chamadas que estejam em conformidade com seus termos. Isso se faz necessário, pois apesar da regra de normalização ser bastante rigorosa em sua avaliação, podem vazar falsos positivos que

serão eliminados durante a aplicação da regra de classificação.

O dicionário de BF 02 possui uma estrutura interna que organiza uma chamada em três formas distintas como segue abaixo:

1. objeto.objeto.objeto.objeto.propriedade
2. objeto.objeto.objeto.propriedade
3. objeto.objeto.propriedade
4. objeto.propriedade

Dessa forma, pode-se obter, em (1), *this.window.navigator.plugins.name*. Em (2), *window.navigator.plugins.name* ou *this.navigator.plugins.name*. Em (3), pode-se obter *navigator.plugins.name*. Em (4), pode-se obter *navigator.plugins*. Assim, com essa estrutura interna, com a possibilidade de novas combinações, aumenta-se para 78 a quantidade de termos investigados.

A forma como o dicionário de BF 02 está estruturado pode ser consultada em `https://github.com/geandromatos/WBF_analyzer`. Verificar os comentários na seção de dicionario.

4.2 Objetos indicadores de *browser fingerprinting*

Segundo Khademi et al. (2015), os objetos ***navigator*** e ***screen***, com suas respectivas propriedades e métodos, são considerados eficazes para *Browser Fingerprinting*. Para Saraiva (2016), os objetos ***canvas***, ***document***, ***Moderniz***, ***history***, ***Storage***, ***devicePixelRatio***, ***window***, ***Date*** também são objetos considerados fornecedores de informações, desse modo as chamadas as esses objetos são fortes indicadores da presença de *browser fingerprinting*.

Por está razão, nesta dissertação, decidiu-se organizar os objetos, que vão ter suas chamadas observadas, em quatro categorias: Objetos especialistas do navegador, objetos especialistas da tela, objetos especialistas genéricos.

4.2.1 Objetos especialistas do navegador

1. *Navigator* é um objeto informativo que representa as propriedades do navegador, fornece o nome e a versão do navegador, os mimetypes, os plugins suportados, a plataforma, o idioma, o sistema operacional e outras propriedades. Foram encontradas referências para suas propriedades e métodos nos trabalhos de (Fiore et al., 2014), (Khademi et al., 2015), (Laksono et al., 2015) (Saraiva, 2016) e (Kobusińska et al., 2017).

4.2.2 Objetos especialistas da tela

1. *Screen* é um objeto informativo que contém informações da tela sobre configurações como resolução e profundidades de cor. Foram encontradas referências para suas propriedades e métodos nos trabalhos de (Khademi et al., 2015), (Laksono et al., 2015), (Saraiva, 2016), (Laperdrix et al., 2016), (Wu et al., 2017) e Saito and Koshiba (2019).

2. *Canvas* é um objeto que desenha conteúdo gráfico em tempo real usando *JavaScript*, pode ser utilizado via elemento <canvas> do html5 ou através da *API webGL*. O *canvas* pode ser usado para animação, gráficos de jogos, visualização de dados, manipulação de fotos e processamento de vídeo em tempo real. Foram encontradas referências para suas propriedades e métodos nos trabalhos de (Upathilake et al., 2015), (Saraiva, 2016), (Laperdrix et al., 2016), (Le et al., 2017), (Saito and Koshiba, 2019)

3. *DevicePixelRatio* é um objeto que retorna a proporção da resolução da tela em pixels indicando ao navegador quantos pixels reais da tela devem ser usados para desenhar um único pixel CSS [1]. Foram encontradas referências para suas propriedades e métodos no trabalho de (Saraiva, 2016)

4.2.3 Objetos especialistas genéricos

1. *Document* é um objeto que serve como entrada para o conteúdo da página web e provê funcionalidades globais ao documento.

2. *History* é um objeto usado para interagir com o histórico do navegador que é um registro que guarda as páginas que o usuário visitou.

3. *Storage* é um objeto que fornece acesso ao armazenamento de sessão ou para o armazenamento local e permite que sejam modificados ou excluídos os dados armazenados.

4. *Date* é um objeto que representa um único momento no tempo em que um formato independe de plataforma. O objeto Date contêm um número que representa milissegundos em uma contagem que vem sendo feita desde 1 de janeiro de 1970.

5. *Window* é um objeto global, isso quer dizer que uma propriedade de objeto window aponta para o próprio objeto window. Dessa forma, pode-se dizer que o objeto window, no caso do navegador, representa um objeto global ou a própria janela do navegador e, a partir dele, muitos objetos que atuam como suas propriedades podem ser invocados.

[1] CSS (*Cascading Style Sheets*) é uma linguagem que descreve como os elementos do html devsem ser estilizados, ou apresentados. fonte: https://www.w3schools.com/css/

6. *Moderniz* é um objeto que representa uma coleção de objetos que permitem aos desenvolvedores facilitarem a entrega de experiências em camadas, ou seja, que os melhores e mais recentes recursos dos navegadores sejam disponibilizados para o usuário de acordo com o suporte do seu navegador.

7. *MediaStream* é um objeto que cria e retorna um novo objeto que contém uma lista especificada de faixas. (especificada como uma matriz de objetos MediaStreamTrack).

   Para os objetos citados na seção ***4.1.2***, menos o objeto MediaStream, foram encontradas referências para suas propriedades e métodos no trabalho de (Saraiva, 2016) e foram feitas pesquisas de atualização e validação dos objetos, propriedades e métodos associados nos sites `https://modernizr.com/`, `https://developer.mozilla.org`, `https://www.w3schools.com/`

4.3 Considerações Finais

Neste Capítulo foi apresentado e detalhado como será realizada a identificação de indicadores de *Browser Fingerprinting*, baseada em AST, e a classificação do risco de uma página web. Diferente das soluções existente, a metologia avalia uma quantidade maior de objetos e em níveis mais profundos, somando e não competindo no universo da pesquisa, além de agregar a classificação de risco de (Saraiva, 2016), para aumentar as chances de obter resultados mais precisos ao identificar e classificar *Browser Fingerprinting*.

Capítulo 5

Implementações

Este Capítulo apresenta a implementação da **WBF Analyzer**, um método de análise estática de código JavaScript para identificar indicadores de *Browser Fingerprinting* em códigos estáticos de páginas web. Contudo, antes de iniciar o detalhamento de sua implementação, é necessário entender o funcionamento do parser Esprima, e a biblioteca utilizada para gerar a árvore abstrata.

5.1 Parser Esprima

Esprima é um parser que realiza análises léxicas e sintáticas de códigos JavaScript, transformando-os em uma representação intermediária. Com a biblioteca esprima pode-se gerar uma árvore abstrata sintática do código fonte. Em linhas gerais, ao se fornecer, como entrada, um arquivo com a extensão .js, a biblioteca é capaz de gerar uma estrutura hierárquica, conforme a Figura 5.1

Na etapa (1), o arquivo navigator.js, que contém o código fonte «var b = "navigator"», é tokenizado. A Esprima cria um nó do tipo *Program*, um nó raiz, que representa o programa em questão. Em seguida, começa efetivamente a criar a estrutura hierárquica da AST através de uma aresta *.body*, que aponta para o próximo nó. Nesse caso, contendo o conteúdo do arquivo.

Assim, na etapa 2, um nó do tipo *variableDeclaration* é criado. Ele possui um *.kind* indicando que é uma variável (palavra reservada chamada «var») e um *.range*, com valores entre 0 a 18, representando o espaço que a declaração de variável irá ocupar na árvore. O nó também tem uma aresta *.declarations* que aponta para um nó do tipo *variableDeclarator*. No caso do exemplo, este nó representa especificamente a declaração de uma variável. Os nós seguintes definem melhor a variável. Para isso, uma aresta *.declarations* aponta para o próximo nó.

Na etapa 3, um novo nó do tipo *variableDeclaration* é criado, agora apenas com *.range* entre 4 a 18. Isto significa que este nó está guardando uma informação e que para acessá-la é preciso visitar seus filhos navegando através das arestas *.id* e *.init*.

Ao seguir o caminho da aresta *.id* (etapa 4), chega-se ao nó do tipo $Identifier$,

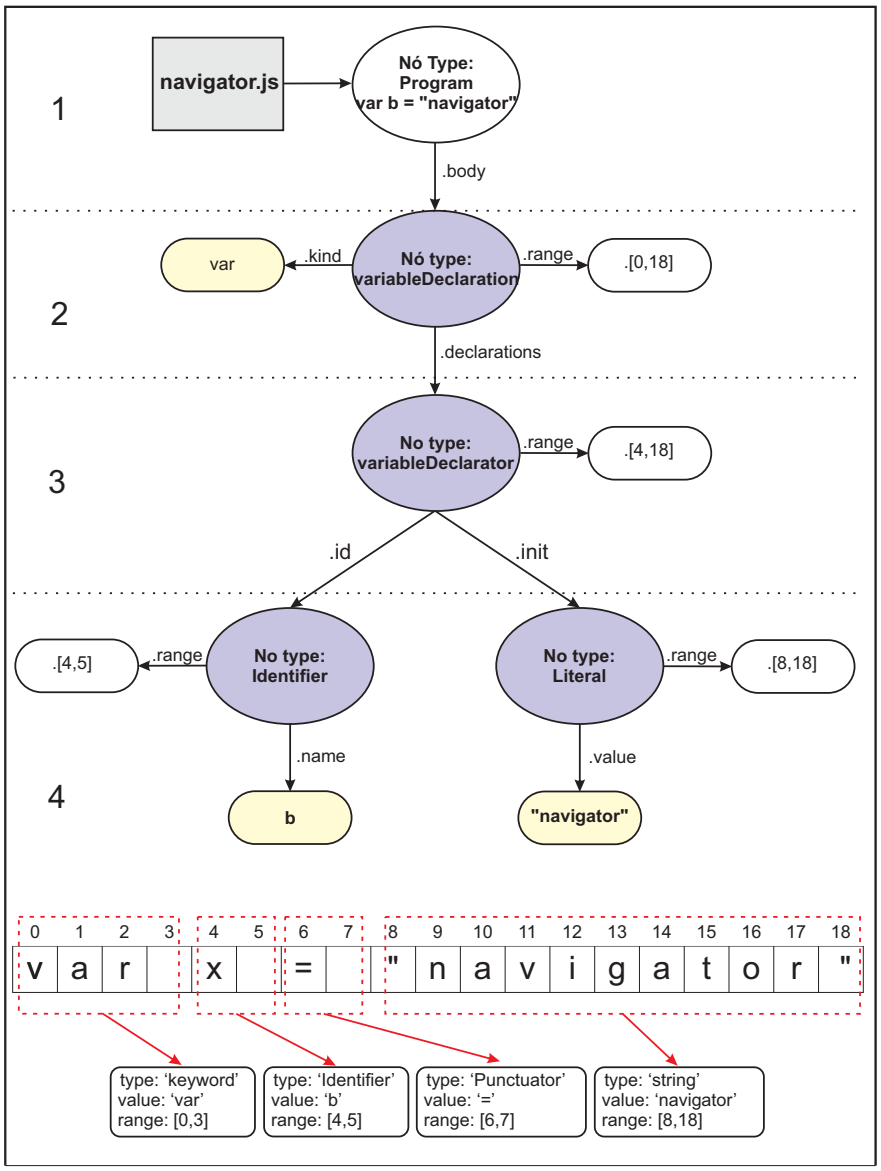


Figura 5.1: Exemplo de Árvore Abstrata da Esprima. Fonte: Autor.

que como o próprio nome diz, identifica a variável. Este nó possui apenas *.range*, entre 4 a 5, e uma aresta *.name* apontando para a letra *b*, o identificador da variável. É interessante notar que ela ocupa 2 espaços para permitir um espaço após a variável. Seguindo pelo caminho da aresta *.init*, chega-se a um nó do tipo *Literal*, com *.range*, entre 8 a 18, e uma aresta *.value*, que aponta para a palavra "navigator".

A Figura 5.2 ilustra a análise de uma AST na Esprima para extrair o objeto **window.navigator.cookieEnabled**. Vale destacar que o interesse é saber se existe um indicativo de um *fingerprinting*, ou seja, saber se esse mesmo objeto pode ser chamado por um script de *fingerprinting*.

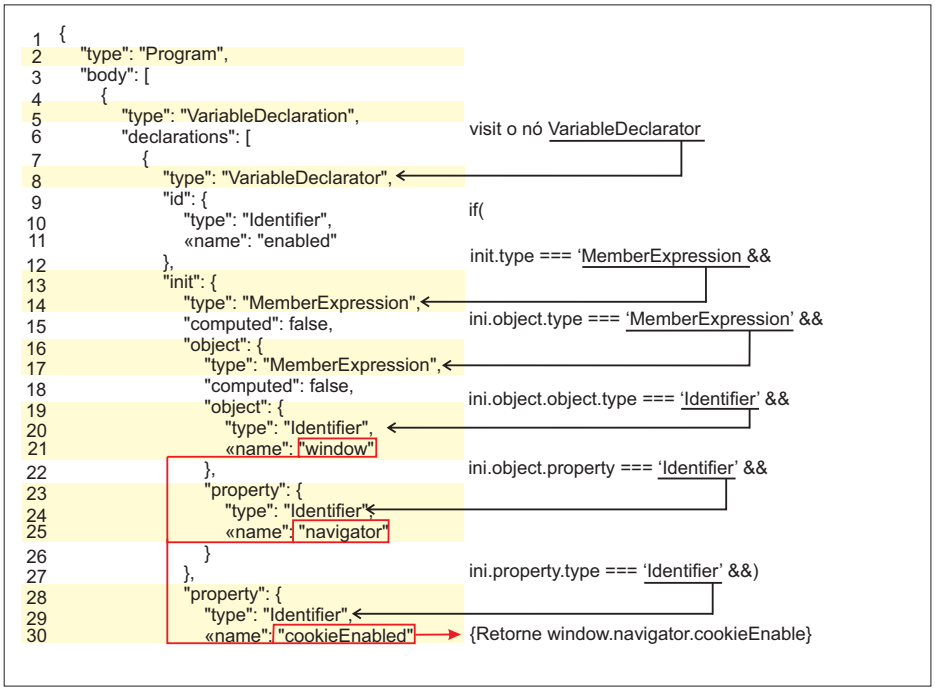


Figura 5.2: Parser: Extração de informação. Fonte: Autor.

Na linha **8**, vê-se o nó *VariableDeclarator*. Ao se chamar por *id.name*, obtém-se o nome da variável declarada no código (no caso é **enabled**). Ao continuar a análise no nó *VariableDeclarator*, também observa-se que ele foi iniciado com um objeto Javascipt. Sabe-se disso devido ao nó *init.type* ser igual a um **MemberExpression**, isso significa que se tem identificadores que juntos formam uma expressão. Ao se buscar pelas chaves *init.object.object.name*, chega-se no objeto *window*. Ao se continuar a busca, como em *init.object.property*, chega-se em **navigator** e, continuando a análise, em *init.property.name* com valor **cookieEnabled**.

Ao se concatenar os resultados (*window.navigator.cookieEnabled*), será retornado um valor booleano que indica quando os **cookies** estão habilitados ou não na máquina do usuário. É dessa forma que este trabalho, através da identificação baseada em AST, avalia se quando o código for executado ele executará algo relacionado a

fingerprinting.

É importante esclarecer que uma AST é normalmente uma estrutura grande. Por isso, é empregada uma AST reduzida, focada somente nas relações de atribuição que estão dispostas no código, visto que o foco desta dissertação é saber as chamadas a objetos e propriedades ligados a *fingerprinting* e não simplesmente suas aparições.

5.2 WBF Analyzer

A **WBF analyzer** foi implementada na plataforma *Node.js*, um ambiente de execução, multiplataforma, baseado no interpretador V8 do Google e que permite a interpretação de códigos JavaScript fora de um navegador web. A linguagem de programação Python foi usada para aproveitar os recursos voltados para ciência de dados. Para mais detalhes sobre a implementação, pode-se consultar https://github.com/geandromatos/WBF_analyzer, onde todo o código da aplicação está comentado e pode-se também consultar os arquivos leiame.txt para mais detalhes sobre cada parte da aplicação.

5.2.1 Coleta de Dados

A implementação da coleta de dados foi baseada na realização do processo de *web crawling*, com objetivo de extrair códigos JavaScript "in-line" e externos. Para tanto, as bibliotecas **urllib3** e **BeautifulSoup** foram utilizada. A primeira possibilita, através da classe *PoolManager*, extrair o conteúdo de múltiplas páginas HTML. A segunda permite a extração de trechos de códigos JavaScript através das ferramentas "find_all" e "find", que possibilitam a detecção dos dois tipos de *tags scripts* "in-line" e *src*.

Vale destacar que a **WBF Analyzer** também pode avaliar bases contendo códigos JavaScript, ao invés de usar um *web crawler*. Um algoritmo para leitura de bases com códigos JavaScript foi elaborado utilizando as bibliotecas **Beautify**[1] e **Walk**[2]. Esta última para percorrer diretórios.

5.2.2 Pré-Processamento

Uma vez que os códigos com a *tag script* foram coletados, entra em cena o pré-processamento para extrair os códigos JavaScript e remover todos os elementos não utilizados na elaboração da AST. O resultado final esperado é um bloco JS.

Para tanto, utilizam-se as bibliotecas **BeatifulSoup**, **re** e **requests**. Na **BeautifulSoup**, usa-se seu atributo ".string" para retornar apenas strings, enquanto a biblioteca **re** possui o método "sub", que usa uma regex para remover elementos específicos. Por fim, a biblioteca **requests** permite obter o conteúdo dos links para códigos JavaScript "in-line".

[1] O Beautify é uma biblioteca que auxilia em processos de desofuscação de códigos JavaScript. Mais informações podem ser obtidas em https://beautifier.io/

[2] Walk é uma biblioteca JavaScript para percorrer árvores de objetos. Mais informações podem ser obtidas em https://github.com/tckerr/walk

Para o pré-processamento de códigos vindos de bases externas, foram desenvolvidos algoritmos próprios da aplicação para realizar este procedimento. A listagem 5.1 ilustra o uso das bibliotecas para o pré-processamento de código JavaScript “in-line”.

```
1    for i in script_in_line:
2        i = str(i.string) + '\n'
3        i = re.sub(r"N"r"o"r"n"r"e", '', str(i))
4        i = re.sub('<!--', '', str(i))
5        i = re.sub('-->', '', str(i))
6        textJS += str(i)
```

listagem 5.1: Exemplo de uso das bibliotecas para JavaScript “in-line”

No laço da linha 1, percorre-se o Array que contém os códigos JavaScript “in-line”. Na linha 2, as tags existentes no código são removidas por meio da utilização do atributo “.string” da **BeautifulSoup**. Após isso, na linha 3, faz-se uso do método *sub* da biblioteca **re**, o qual recebe uma regex para rastrear a palavra *None* (que aparece no lugar das tags removidas) a fim de retirá-la do código. Posteriormente, nas linhas 4 e 5. usa-se ainda o método *sub* para rastrear e remover os marcadores de comentário HTML “$<! --$” e “$-- >$”. Por fim, na linha 6, faz-se uso da função *str*() do python para anexar os trechos de códigos “in-line” com o símbolo de nova linha $\backslash n$ para formar um único bloco de código.

A Listagem 5.2 ilustra o uso das bibliotecas para o pré-processamento de código JavaScript externo.

```
1    if requests.get(url_src2):
2       codigos_JavaScript.add(url_src)
3       if url_src2 in codigos_JavaScript:
4          url_src2 = requests.get(url_src2)
5          textoJs2 = url_src2.text
6          textoJs2 = tag(textoJs2)
7          textoJs2 = re.sub(r"N"r"o"r"n"r"e", '',
8          str(textoJs2))
```

listagem 5.2: Exemplo de uso das bibliotecas para JavaScript externo

Na linha 1 é feita uma verificação se é possível obter o conteúdo da página web para qual o link direciona. Se for possível, na linha 2, adiciona-se o link do código JavaScript externo para o Array “codigos_JavaScript”. Caso o link tenha sido acrescentado ao Array “codigos_JavaScript”, usa-se o método *get* da biblioteca **requests**, na linha 4, para capturar o conteúdo da página web. Na linha 5 é usado o atributo *.text* para converter o conteúdo para string. Depois de convertido, o código é passado para a função *tag*(), na linha 6, que retira as tags existentes nas extremidades do código. O pré-processamento para códigos JavaScript externo é finalizado na linha 7, aplicando o tratamento para remover a palavra *None*, que é o mesmo utilizado no pré-processamento de códigos JavaScript “in-line”.

5.2.2.1 Remoção de Tags

O algoritmo de remoção de *tags* retira elementos desconhecidos para a linguagem JavaScript, excluindo blocos de códigos que possuem características suspeitas. Isso é necessário para que a AST possa ser criada. No total, são seis casos removidos pelo algoritmo:

- Códigos que possuem a *tag script*, porém não tratam-se de código JavaScript. Ex: $< script\ type =' application/ld + json' >$;
- Códigos que possuem a *tag style*. Ex: $< style >$;
- Códigos que possuem a *tag !DOCTYPE*, *!doctype*, *BODY*, *body*, *HEAD*, *head*, *html* e *HTML*. Ex: $<!DOCTYPEHTMLPUBLIC >$;
- Códigos que possuem a *tag script* com o elemento "$defer >$ ". Ex: $< scripttype =$ "$text/JavaScript"defer >$;
- Elementos que estão entre "<!−−"e "−− >". Ex: $<!--MMMMMMMM-- >$;
- Elementos que possuem características de JSON. Ex: {"a"}

Vale destacar que qualquer elemento não reconhecido implica na não geração da AST pela Esprima.

5.2.2.2 Código Desorganizado

Além da remoção de *tags*, também é preciso ajustar códigos JavaScript desorganizados. Também foi desenvolvido um algoritmo que verifica, durante a extração, se existem *tags* com elementos ofuscados ou desorganizados, como é apresentado na Listagem 5.3, onde a segunda linha está mal formatada.

```
1        for(a = 0; a
2        < b; a++){}
```

listagem 5.3: Exemplo de código desorganizado

Para evitar este tipo de caso, foi utilizada a biblioteca **Beautify**, para desofuscar e embelezar o código. A biblioteca possui funções nativas que realizam a desofuscação de códigos.

Em linhas gerais, consegue-se organizar o código onde:

- Existem duas ou mais *tags* na mesma linha sem código entre elas. Ex: $< scriptsrc =$ "$http://www.0spam.com.js$" $>< /script >$;
- Existe uma *tag* no início da linha de código. Ex: $< script >$ $(adsbygoogle = window.adsbygoogle||[]).push()$;
- Existam *tags* quebradas no código. Ex: $<$ $/script >$;

- Exista uma *tag* no fim da linha de código. Ex: $for(i = 2; i <= 5; i++) < /script >$;

A remoção de *tags* que não sejam *script* funciona como uma continuação do tratamento de eliminação de elementos desconhecidos para a linguagem JavaScript, diminuindo, assim, as chances de aparecer algo que não permita que a AST seja gerada.

5.2.3 Formação do bloco JS

Após a coleta de dados ser realizada e o código ser pré-processado e desofuscado, o código é salvo na pasta blocoJs, um diretório no qual os arquivos, que estão prontos para serem analisados pela identificação, são salvos.

Os códigos são salvos por meio do uso da biblioteca **fs**, que significa *file system* e possui a função de salvar arquivos.

5.2.4 Identificação

Para o processo de Identificação foram utilizadas as bibliotecas **Esprima**, que cria a AST do código investigado, e **ast-types**, que percorrer a AST. Ambas são bibliotecas nativas do JavaScript.

As subseções seguintes descrevem o processo de implementação das regras de identificação.

5.2.4.1 Regra 1 - Extração

O processo de extração busca percorrer a AST para identificar e realizar a extração de termos para um escopo reduzido, implementado como um array. Cada elemento deste array de objetos possui 7 propriedades, como apresentado na Figura 4.8: identificador esquerdo, identificador direito, objeto 0, objeto 1, objeto 2, propriedade 1 e propriedade 2.

Na implementação da extração, a função *parseScript*, da biblioteca **Esprima**, cria e atribui a AST do código para a variável. Em seguida, essa variável é percorrida, utilizando-se a função *visit*, da biblioteca **ast-types**, como objetivo verificar os nós candidatos da AST a serem extraídos para o escopo reduzido.

Os nós candidatos que a implementação da extração busca são:

- *VariableDeclarator*. Ex: var h = navigator;
- *AssignmentExpression*. Ex: a = navigator;
- *MemberExpression*. Ex: oi = navigator.appName;
- *CallExpression*. Ex: a = canvas.toDataURL();
- *ArrayExpression*. Ex: a = [navigator.appName, navigator]; a[1].appName.

Vale destacar que a implementação permite adicionar um tipo de nó candidato, bastando apenas procurar pelo devido nó na biblioteca **Esprima** e realizar a adição no código implementado.

Para melhorar o entendimento sobre a implementação, será explicada a extração do nó *VariableDeclarator*, um dos mais relevantes .

5.2.4.2 Extração do nó VariableDeclarator

O nó candidato *VariableDeclarator* representa uma expressão de atribuição na árvore, no caso trata-se de uma atribuição realizada a partir da declaração de uma variável. Para melhor entender o processo de extração, é preciso entender que a representação da sintaxe do nó *VariableDeclarator* na **Esprima** permite capturar atribuições como: $varh = navigator$, $varh = this$ e $varh = newDate()$. Assim, além de uma atribuição normal, casos envolvendo palavras reservadas do JavaScript como "this" e "new" são percebidas.

O algoritmo 1 ilustra a extração do membro esquerdo da expressão (elemento que está ao lado direito do sinal =), Identificador Esquerdo (IDE), do ponto de vista da AST.

Algoritmo 1: Extração de tipos Identifier no membro esquerdo da expressão (nó VariableDeclarator)

Entrada : nó VariableDeclarator
Saída : Adição de termos no escopo reduzido

```
// node é o nó candidato que está sendo analisado
// IDE e IDD representam, respectivamente, o identificador
   esquerdo e o identificador direito do escopo reduzido
// new representa o tipo NewExpression, this representa o tipo
   ThisExpression e Ident representa o tipo Identifier
// node_i.c.n é a chamada de objeto node.init.callee.name
```

```
1  node ← nó VariabelDeclarator;
2  if node.id e node.init existirem then
3  |  if node.id.name existir then
4  |  |  IDE ← node.id.name;
5  |  end
6  |  if node.init.type = 'this' then
7  |  |  IDD ← "this";
8  |  end
9  |  else if node.init.type = 'new' then
10 |  |  IDD ← "new";
11 |  |  if node.init.calle.type = 'Ident' then
   |  |  |  // ex: new Date()
12 |  |  |  IDD ← new + node_i.c.n + ();
13 |  |  end
14 |  end
15 end
```

Observa-se na linha **1**, a variável node recebendo o caminho do nó *Variable-Declarator* o que permitirá a realização das operações envolvendo os elementos do nó.

Na linha **2**, verifica-se a existência das propriedades *id* e *init* do nó. Caso eles existam, é verificada, na linha **3**, a existência da propriedade *name*. Se a mesma existir, o identificador esquerdo do escopo reduzido a recebe. Após isso, nas linha **6** e **9** ocorrem operações que buscam verificar o tipo da propriedade *init* contida no nó. Caso o tipo seja um $ThisExpression$, o identificador direito do escopo reduzido recebe o token *this*. Caso o tipo seja igual a um $NewExpression$, o identificador direito do escopo reduzido recebe o token *new* e verifica também a existência da proriedade *callee*, pois se o tipo da propriedade *callee* for igual a $Identifier$, o identificador direito recebe o token *new* + o valor guardado pela propriedade *name* do nó $CallExpression$, com adição ainda de um parêntese, para indicar que este caso é um método.

Para extração do membro direito foi desenvolvido o algoritmo 2. Nele é ilustrado o caso onde a propriedade *init* possui a propriedade *name*. Caso a propriedade *name* for igual ao token *canvas*, o identificador esquerdo do escopo reduzido irá receber o valor de *name* da propriedade *id*+o token "_canvas", para tornar claro que a variável encontrada contém um objeto *canvas*. Caso não seja igual a canvas, então o identificador direito do escopo reduzido recebe o valor de *name* da propriedade id, sem ocorrer a marcação do identificador esquerdo pelo token "canvas".

Caso as propriedades *id* e *init* não possuam um tipo ou um nome, é verificado, na linha **10**, se existe uma propriedade "value"que seja igual a "string". Em caso positivo, a chamada de objeto é considerada um tipo string, como "navigator.appName". Nas linhas **14**, **17** e **20**, analisa-se a existência de fatores na string que a tornam inválida, como a presença de números, dos símbolos "<", ">"e de palavras reservadas como *var*, *let* e *const*, pois a intenção é somente extrair coisas que se encaixem em um padrão de uma operação de atribuição de palavras, como em "a = navigator". Na linha **23**, se a variável cont_string for igual a 0, isso significa que a string é válida. Na linha **24**, verifica-se a quantidade de elementos do array que armazena a string analisada. Se este só possuir um elemento, então o identificador direito do escopo reduzido recebe o elemento. Se o array possuir mais elementos, então os objetos 0, 1 e 2 e propriedades 1 e 2, do escopo reduzido, recebem os elementos. Logo, se existirem apenas dois elementos, o objeto 0 e 1 recebem os elementos, e desta forma os elementos são distribuídos no escopo reduzido de acordo com a quantidade de elementos do array.

5.2.5 Regra 2 - Normalização

A normalização utiliza a biblioteca **fs** para salvar o escopo normalizado. Seu processo é divido em funções, sendo a principal nomeada como verificação, pois verifica os dados vindos do escopo reduzido. As outras funções que compõem a normalização, comparação e correspondência, são chamadas de acordo com os dados. Também existem outras funções suportes como a de produção e a de limpeza.

Assim como o escopo reduzido, o escopo normalizado também foi desenvolvido como um array de objetos que possuem três propriedades: Linha, quantidade e chamada.

Algoritmo 2: Extração de tipos Identifier no membro direito da expressão (nó VariableDeclarator)

Entrada : nó VariableDeclarator
Saída : Adição de termos no escopo reduzido

```
   // IDE e IDD representam, respectivamente, o identificador
      esquerdo e o identificador direito do escopo reduzido
   // OBJ0 e OBJ1 representam, respectivamente, o objeto 0 e o objeto
      1 do escopo reduzido
   // Q_string é a quantidade de elementos no array string
 1 node ← caminho do nó VariabelDeclarator;
 2 if node.init.name existir then
 3 |   if node.init.name = 'canvas' then
   |   |   // ex: oi = canvas
 4 |   |   IDE ← node.id.name + "_canvas";
 5 |   |   IDD ← node.init.name;
 6 |   else
   |   |   // ex: oi = navigator
 7 |   |   IDD ← node.init.name;
 8 |   end
 9 end
10 else if node.init.value = 'string' then
11 |   string ← node.init.value;
12 |   string ← string.split(".");
13 |   cont_string ← 0;
14 |   if existir algum número em string then
15 |   |   cont_string + +;
16 |   end
17 |   if existir " < " em string ou existir " > " em string then
18 |   |   cont_string + +;
19 |   end
20 |   if existir "var"ou existir "let"ou existir "const"em string then
21 |   |   cont_string + +;
22 |   end
23 |   if cont_string = 0 then
24 |   |   if Q_string = 1 then
   |   |   |   // ex: ['navigator']
25 |   |   |   IDD ← string[0];
26 |   |   else
   |   |   |   // atribuição de acordo com a quantidade de elementos do
   |   |   |      array, ex: ['navigator','appName']
27 |   |   |   OBJ0 ← string[0];
28 |   |   |   OBJ1 ← string[1];
29 |   |   end
30 |   end
31 end
```

Além do escopo normalizado, a normalização produz o dicionário de identificadores, uma variável global que facilita a manipulação dos dados que esta estrutura guarda. Por fim, também é implementado o dicionário de *BrowserFingerprinting 01*, implementado como uma array, para ser usado pelas ferramentas de suporte a normalização. O dicionário de *BrowserFingerprinting 01* também foi declarado como variável global não apenas para facilitar a manipulação de dados, mas também devido o fato de que está localizado em um script diferente do script da normalização, como pode ser conferido em [3], sendo necessário chamá-lo como um código externo.

5.2.5.1 Verificação

O processo de normalização inicia quando a função verificação é invocada para percorrer todos os objetos do escopo reduzido e a cada objeto percorrido. O algoritmo 3 exemplifica a implementação da função verificação.

Algoritmo 3: Função da Verificação

Entrada : escopo reduzido
Saída : criação de par chave e valor do dicionário de identificadores

```
// DI é o dicionário de identificadores
// chave_valor é o par chave e valor formado a partir do IDE e do
   IDD
// ECT é o escopo de correspondência temporário
// produção é a função que concatena os valores do ECT e limpeza é
   a função que limpa o ECT
```

```
1  DI ← [ ];
2  Escopo_Normalizado ← [ ];
3  if IDD = vazio then
4  |  chame a comparação;
5  |  if ECT ≠ vazio then
6  |  |  produto ← produção;
7  |  |  if produto ≠ vazio then
8  |  |  |  produto é inserido em Escopo_Normalizado;
9  |  |  end
10 |  |  chame a função de limpeza;
11 |  end
12 else
13 |  if IDE ≠ vazio then
14 |  |  chave_valor ← {IDE : IDD};
15 |  |  DI.push(chave_valor);
16 |  end
17 end
```

Na linha **3** é verificado se o identificador direito é diferente de vazio. Se for, a função comparação é chamada. Essa, por sua vez, chama a correspondência e, juntas,

[3] *https : //github.com/geandromatos/WBF_analyser*

elas produzem o ECT, o escopo de correspondência temporário, um array simples que guarda a chamada produzida temporariamente. Na linha **5** é verificado se o ECT possui algum elemento. Se possuir, a função produção é chamada e o seu resultado é inserido no escopo normalizado.

Caso o identificador direito não exista, a verificação procura saber se o identificador esquerdo existe por meio de uma operação lógica realizada na linha **13**. Se o identificador esquerdo existir, então um objeto é criado contendo o identificador esquerdo como chave e o identificador direito como valor, e este objeto é inserido no dicionário de identificadores.

5.2.5.2 Comparação

Como mencionado anteriormente, a comparação é chamada caso o identificador direito do escopo reduzido não exista, logo, seu objetivo é verificar se os valores apontados pelas chaves do escopo reduzido, não incluindo o identificador esquerdo e direito, são iguais aos valores apontados pelas chaves do dicionário de *BrowserFingerprinting 01* ou do dicionário de identificadores.

É relevante ressaltar que, inicialmente, a comparação simplesmente busca os valores diretamente no dicionário de *BrowserFingerprinting 01*. Caso ela não encontre o valor procurado, ela realiza uma busca *top down* no dicionário de identificadores. Caso ela não encontre nada, ela ativa a comparação especial, que realiza uma busca *button up* no dicionário de identificadores.

O processo da função de comparação é ilustrado no algoritmo 4. Na linha **3** é iniciado um laço que percorre o dicionário de *Browser Fingerprinting 01*. A cada elemento percorrido é realizada uma operação lógica (linha **4**) que verifica se o objeto 0 do escopo reduzido é igual ao elemento corrente do dicionário de *Browser Fingerprinting*. Se forem iguais, a variável $\text{objeto}_{\text{ouprop ri}}$ recebe o objeto 0 do escopo reduzido e, em seguida, é enviada para regra da correspondência. Caso o objeto 0 do escopo reduzido não seja encontrado no dicionário de *Browser Fingerprinting 01*, a busca *top down* é invocada. Caso o valor da chave seja encontrado, a correspondência é chamada. Caso o valor não seja encontrado, a busca *button up*, também chamada de comparação especial, é chamada.

É relevante ressaltar que a busca *top down* não serve apenas para o objeto 0 do escopo reduzido, mas também para os seus outros componentes, sendo estes o objeto 1, objeto 2, propriedade 1 e propriedade 2.

5.2.5.3 Correspondência

O processo de correspondência consiste em salvar, no escopo de correspondência temporário (ECT), os objetos e as propriedades que foram aprovados pela comparação. Como já mencionado, o ECT foi desenvolvido como uma variável global para facilitar a manipulação dos dados que esta estrutura guarda. O processo da correspondência é ilustrado no algoritmo 5.

Algoritmo 4: Função da Comparação

Entrada : escopo reduzido
Saída : Adição de objetos e propriedades ao ECT

```
// Objeto_ouPropri recebe o elemento investigado que
   constituirá a chamada formada
// DB_01 é o dicionário de BrowserFingerprinting 01
// I_DB_01 é o índice dos elementos do dicionário de
   BrowserFingerprinting
// C_DB_01 é o comprimento do dicionário de
   BrowserFingerprinting
```

```
1  Objeto_ouPropri ← 0;
2  I_DB_01 ← 0;
3  while I_DB_01 <= C_DB_01 do
4  |  if OBJ0 = DB_01[I_DB_01] then
5  |  |  Objeto_ouPropri ← OBJ0;
6  |  |  chame a correspondência;
7  |  end
8  |  I_DB_01 + +;
9  end
10 if Objeto_ouPropri = 0 then
11 |  chame a busca top down;
12 end
```

Algoritmo 5: Função da Correspondência

Entrada : os objetos e propriedades que foram aprovados pela comparação
Saída : Adição de uma chamada ao escopo normalizado

```
// ECT e o escopo de correspondência temporário
// Objeto_ouPropri recebe o elemento investigado que constituirá a
   chamada formada
```

```
1  ECT.push(Objeto_ouPropri);
2  return ECT;
```

5.2.6 Regra 3 - Classificação

A classificação é responsável por contar a frequência de vezes em que uma chamada aparece, bem como denominar a classificação de risco da chamada e a classificação de risco da página web. A partir desses dados, ela forma o escopo classificado, construído como um *array* de objetos com 4 propriedades: Chamada, Frequência, Risco da chamada e Risco da página.

Como ferramenta de suporte, a classificação utiliza o dicionário de *Browser Fingerprinting 02*, aplicado para permitir que sejam apenas classificadas chamadas que estejam em conformidade com seus termos. O dicionário de *Browser Fingerprinting 02* é dividido em três níveis: alto, médio e baixo. O algoritmo 6 ilustra como a classificação compara as chamadas do escopo normalizado com o dicionário de *Browser Fingerpriting 02* completo.

Algoritmo 6: Comparando as chamadas do escopo normalizado com o dicionário de Browser Fingerpriting 02 completo

Entrada : escopo normalizado
Saída : Adição de uma chamada ao escopo classificado
// $cont_{EN}$ conta os elementos do escopo normalizado
// $chamada_{EN}$ é a chamada corrente do escopo normalizado
// C_{EN} é o comprimento do escopo normalzado
// DB_02 é o dicionário de BrowserFingerprinting 02
// C_{DB_02} é o comprimento do dicionário de BrowserFingerprinting 02
// I_{DB_02} é o índice dos elementos do dicionário de BrowserFingerprinting 02

```
1  cont_EN ← 0;
2  while cont_EN <= C_EN do
3  |  while I_DB_02 <= C_DB_02 do
4  |  |  if chamada_EN = DB_02[I_DB_02] then
5  |  |  |  chamada_EN é inserida no escopo classificado;
6  |  |  end
7  |  |  I_DB_02 + +;
8  |  end
9  |  cont_EN + +;
10 end
11 chame o algoritmo 7;
```

Na linha **2**, um laço é iniciado para percorrer o escopo normalizado. A cada chamada percorrida no escopo normalizado, um outro laço é iniciado para percorrer o dicionário de *Browser Fingerprinting 02* e, assim, comparar todos os seus elementos com a chamada corrente do escopo normalizado. Caso a chamada corrente do escopo normalizado exista no dicionário de *Browser Fingerprinting 02*, esta chamada é inserida no escopo classificado. Após comprovar que a chamada analisada existe no dicionário de *Browser Fingerpriting 02*, é contado a frequência de vezes que ela aparece no código, sendo este processo representado pelo algoritmo 7.

Algoritmo 7: Contagem de Frequência da Chamada

Entrada : escopo normalizado
Saída : Adição de uma chamada ao escopo classificado

```
// cont_EN conta os elementos do escopo normalizado
// C_EN é o comprimento do escopo normalzado
// chamada_EC é a chamada corrente do escopo classificado
// chamada_EN é a chamada corrente do escopo normalizado
// freq é a frequência de chamada_EC
1  if chamada_EC existir then
2  |  freq ← 0;
3  |  cont_EN ← 1;
4  |  while cont_EN <= C_EN do
5  |  |  if chamada_EC = chamada_EN then
6  |  |  |  freq++
7  |  |  end
8  |  |  cont_EN++;
9  |  end
10 end
11 freq é inserida no escopo classificado;
12 chame o algoritmo 8;
```

Na linha **1** é verificado se a chamada corrente do escopo normalizado foi classificada. Se ela existir no dicionário de **Browser Fingerprinting 02** completo, então um laço é iniciado, na linha **4**, para percorrer o escopo normalizado e verificar quantas vezes a chamada classificada se repete. Após a chamada ter sido classificada e a frequência da chamada ter sido registrada, a classificação busca saber o nível da chamada analisada, sendo o nível da chamada podendo ser alto, médio ou baixo. É relevante ressaltar que para classificar o nível da chamada, o dicionário de BF_02 completo foi dividido nos três níveis comentados anteriormente.

No algoritmo 8, é possível visualizar o processo de classificação para o nível da chamada. Na linha **4** é possível observar que um laço é iniciado para percorrer os elementos do dicionário de BF_02 alto, verificando, dessa forma, se a chamada investigada pertence ao grupo de nível alto. Caso a chamada investigada não pertencer ao grupo de nível alto, então a chamada é procurada no dicionário de BF_02 de nível médio. Caso ela não exista no grupo de nível médio, a chamada é buscada no dicionário de BF_02 nível baixo. Após ter classificado todas as chamadas encontradas no código, ter realizado a contagem de frequências e ter registrado o nível de cada chamada encontrada, a classificação executa sua última etapa, a qual consiste em determinar o nível da página web investigada a partir de todas as chamadas encontradas na própria página.

Para realizar este processo, é feita a contagem de chamadas de risco alto, médio e baixo e, após possuir a quantidade de chamadas de risco alto (100), médio (10) e baixo (1), o cálculo de média ponderada é feito. Primeiramente é realizada a multiplicação

Algoritmo 8: classificação do nível da chamada

Entrada : escopo normalizado
Saída : Adição de uma chamada ao escopo classificado

```
// DB_02 é o dicionário de BrowserFingerprinting 02
// chamada_EC é a chamada corrente do escopo classificado
// cont_DB_02_alto conta os elementos do dicionário de DB_02 alto
// C_DB_02_alto é o comprimento de DB_02 alto
// cont_DB_02_medio conta os elementos do dicionário de DB_02 médio
// C_DB_02_medio é o comprimento de DB_02 médio
// cont_DB_02_baixo conta os elementos do dicionário de DB_02 baixo
// C_DB_02_baixo é o comprimento de DB_02 baixo
// risco_chamada é o risco da chamada corrente do escopo classificado
1  cont_DB_02_alto ← 0;
2  cont_DB_02_medio ← 0;
3  cont_DB_02_baixo ← 0;
4  while cont_DB_02_alto < C_DB_02_alto do
5  |  if chamada_EC = DB_02[cont_DB_02_alto] then
6  |  |  risco_chamada ← 'alto';
7  |  |  risco_chamada é inserido no escopo classificado;
8  |  end
9  |  cont_DB_02_alto + +;
10 end
11 if risco_chamada ≠ 'alto' then
12 |  while cont_DB_02_medio < C_DB_02_medio do
13 |  |  if chamada_EC = DB_02[cont_DB_02_medio] then
14 |  |  |  risco_chamada ← 'medio';
15 |  |  |  risco_chamada é inserido no escopo classificado;
16 |  |  end
17 |  |  cont_DB_02_medio + +;
18 |  end
19 end
20 if risco_chamada ≠ 'medio' then
21 |  while cont_DB_02_baixo < C_DB_02_baixo do
22 |  |  if chamada_EC = DB_02[cont_DB_02_baixo] then
23 |  |  |  risco_chamada ← 'baixo';
24 |  |  |  risco_chamada é inserido no escopo classificado;
25 |  |  end
26 |  |  cont_DB_02_baixo + +;
27 |  end
28 end
```

entre o peso do nível e a quantidade de chamadas que possuem o nível em questão, no caso, cada nível possui um peso diferente, sendo o maior peso atribuído ao nível alto. Posteriormente, a soma dos valores do numerador é dividida pela soma dos pesos de cada nível, resultando no risco da página.

5.2.7 Considerações finais

Neste capítulo foi abordado a implementação da **WBF Analyzer**, na qual foram demonstrados algoritmos para o tratamento de código JavaScript, para a criação da AST, algoritmos das estruturas de pré-processamento, da identificação com a descrição da extração, da normalização e suas sub-rotinas como a verificação, a comparação e a correspondência. Também foi abordado como a regra 3 - Classificação foi implementada e como os dicionários de fingerprinting foram utilizados e como os cálculos de ponderação de risco foram aplicados.

Capítulo 6

Experimentos e Resultados

Este Capítulo descreve os experimentos e os resultados do processo de avaliação dos indicadores de *fingerprinting* e classificação dos sites quanto ao nível de severidade do *fingerprinting*. A primeira seção relata o protocolo experimental necessário (ambiente e base de dados), enquanto a segunda descreve o processo de aplicação da metodologia proposta e os resultados alcançados.

6.1 Protocolo Experimental

Para avaliar o método proposto, foram realizados vários experimentos a fim de verificar a capacidade de analisar códigos baixados através de um *web crawling* e encaminhados diretamente para a **WBF analyzer**.

Todos os experimentos foram executados em uma máquina Intel(R) Xeon(R) CPU E5-4617 @ 2.90GHZ (2 processadores), com 90 GB de memória RAM, 100 GB de disco, com sistema operacional Windows 10.

6.1.1 Base de Dados

Para realização dos experimentos, foram utilizadas cinco (05) bases de dados. A primeira base, chamada de Alexa (Ano 2016), contém 2002 sites (códigos) utilizados no trabalho de (Saraiva, 2016) e (Elleres, 2017). Esta base foi selecionada para verificar se a **WBF analyzer** pode identificar padrões ou quantidades de indicadores de *fingerprinting* não identificados ou quantificados por (Saraiva, 2016).

A segunda base, Top 50 Alexa (Ano 2021)[1], contém códigos de 50 sites coletados especificamente das páginas principais de cada um dos sites do Top 50 da Alexa.com, na categoria internacional. Os códigos foram coletados via *web crawling* pela **WBF analyzer**. Esta base foi montada para verificar se existe uma linha de crescimento de indicadores de *fingerprinting* e se os riscos, apresentados no trabalho de (Saraiva,

[1] http://www.alexa.com/

2016), permanecem atualmente ou se o *fingerprinting* baseado em objetos de JavaScript tornou-se obsoleto.

A terceira base, Top 50 Alexa - Dependências (Ano 2021), possui as mesmas características da base anterior (Top 50 Alexa do ano de 2021), mas contém os códigos tanto das páginas principais quanto das secundárias e, assim, formando um bloco de código único para análise. Esta base foi extraída para verificar se existe diferença entre uma avaliação feita pela **WBF analyzer** em códigos que compõe a página principal e uma avaliação contendo um conjunto de códigos de todas as páginas que compõe o domínio do site.

A quarta base, Canvas (Ano 2017), é oriunda do trabalho de (Elleres, 2017) e é composta por 8350 códigos de sites. Esta base foi selecionada especificamente para verificar se os indicadores de *fingerprinting* que predominam são de *Canvas Fingerprinting*, como mencionado em (Elleres, 2017).

A quinta e última base, *DMOZ* [2], também oriunda do trabalho de (Saraiva, 2016), contém 1564 sites. Esta base foi selecionada por ser considerada no trabalho de (Saraiva, 2016) como benigna.

Vale destacar que as comparações dos resultados nas bases investigadas, sempre que for possível, serão feitas com o trabalho de (Saraiva, 2016), pelas seguintes razões:

- O trabalho é focado em ocorrências e chamadas de objetos JavaScript relacionados com *Browser Fingerprinting*.
- O trabalho não limitou-se a um objeto de JavaScript como o Objeto Canvas estudado nos trabalhos de (Elleres, 2017; Le et al., 2017).
- É o trabalho seminal na investigação de termos de chamadas *JavaScript* (68 termos) relacionados a *Browser Fingerprinting*.
- O trabalho de (Saraiva, 2016) avalia o código JavaScript in-line e o código JavaScript externo, enquanto trabalhos como (Acar et al., 2013; EEF; Nikiforakis et al., 2013; Bujlow et al., 2017; Le et al., 2017) não deixam claro os tipos de código *JavaScript* que foram avaliados.
- No trabalho foi criada uma metodologia de classificação de risco, a qual foi adaptada para ser utilizada na **WBF analyzer**. Este trabalho não questiona a metodologia de risco estabelecida e sim o método utilizado para detectar as chamadas de *JavaScript* que são encaminhadas para o método de classificação.

6.1.2 Base Alexa (Ano 2016)

A base Alexa.com (Ano 2016) possui um total de 2002 sites. A Figura 6.1 ilustra os resultados da análise feita pela **WBF Analyzer**.

Do total, 83.8% (1678) dos sites foram considerados como contendo *fingerprinting*, isto é, a aplicação conseguiu detectar as chamadas de objetos JavaScript nesses

[2] http://DMOZ.org/

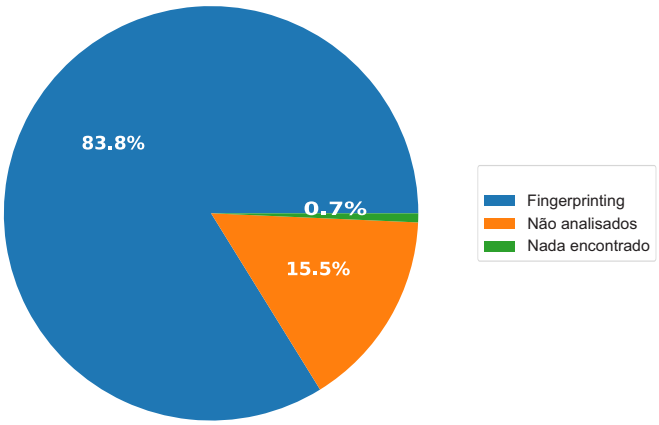


Figura 6.1: Análise da base Alexa (Ano 2016). Fonte: Autor.

sites. Já 15.5% (310) dos sites não foram analisados pela aplicação, devido ao fato da árvore abstrata do código não ser gerada devido a erros na sintaxe do código ou *tokens* não reconhecidos pela biblioteca Esprima. Por fim, em 0.7% (14) dos sites não foram encontradas chamadas de objeto JavaScript.

Em relação aos sites não analisados (15.5%), pode-se observar na Figura 6.2, na parte marcada em **2**, um erro no código (a espera por um ponto e vírgula).

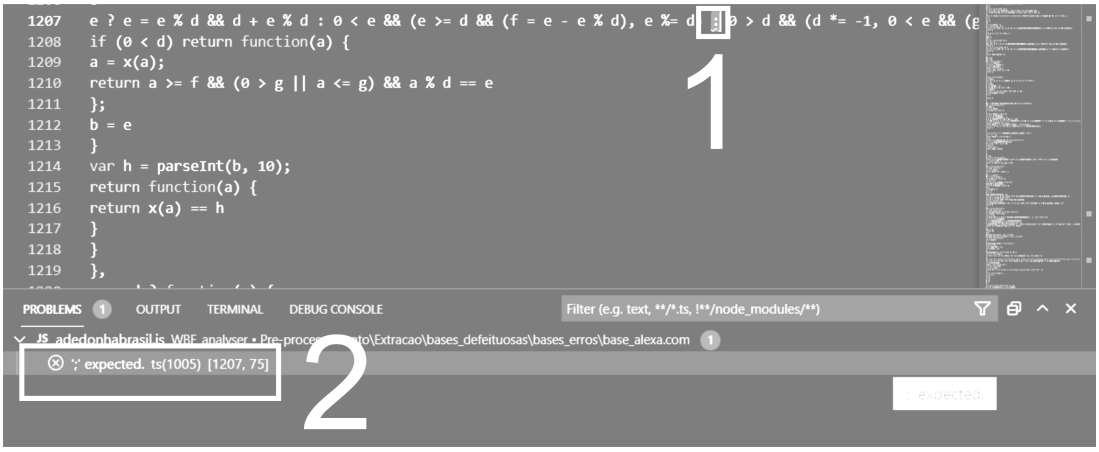


Figura 6.2: Exemplo de código com erro. Fonte: Autor.

Em **1**, ao analisar a linha 1207, o parser do JavaScript solicita que seja trocado ":" por um ";", pois entende que essa linha contém um erro. Nesse caso, a biblioteca não irá gerar a AST enquanto o erro permanecer. Esse tipo de erro não é de fácil correção, pois deve-se saber quando substituir dois pontos (:) por ponto e vírgula (;).

6.1.2.1 Classificação de Risco

A Figura 6.3 ilustra o resultado da classificação de risco na base Alexa.com ano 2016.

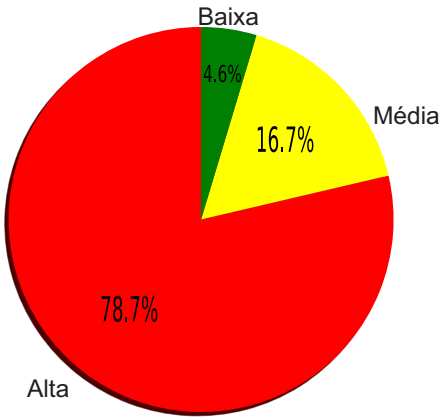


Figura 6.3: Classificação do risco base Alexa (Ano 2016). Fonte: Autor.

Observa-se na Figura que a **WBF Analyzer** conseguiu classificar os sites, sendo 78.7% (1320 sites) deles com nível de risco alto, 16.7% (280 sites) com nível de risco médio e 4.6% (78 sites) com nível de risco baixo. Os resultados obtidos são considerados satisfatórios, pois seguem a mesma tendência apresentada no trabalho de (Saraiva, 2016), embora, obviamente, os percentuais de classificação apresentem maiores porcentagens. A Tabela 6.1 ilustra essa análise comparativa.

Tabela 6.1: Análise comparativa da classificação da base Alexa (Ano 2016)

Autores	**Riscos**		
	Baixo	**Médio**	**Alto**
(Saraiva, 2016)	1%	40%	59%
WBF Analyzer	4.6%	16.7%	78.7%

Vale destacar que a **WBF Analyzer** consegue gerar a classificação através do escopo classificado, que é invidualizado para cada site. Em Saraiva (2016), o trabalho juntava todos os objetos da base e classificava os sites em relação aos níveis que cada objeto pertencia.

6.1.2.2 Chamadas e Invocações

No trabalho de (Saraiva, 2016), a autora menciona o termo *ocorrência*, mas não deixa claro se sua contagem se refere a ocorrência de uma chamada de objeto JavaScript (um conjunto *objeto.propriedade*; um objeto isolado, como *navigator*; ou uma propriedade isolada, como *userAgent*) ou a quantidade de invocações de uma chamada de objeto.

Já a **WBF Analyzer** identifica a chamada, no formato *objeto.propriedade*, e quantifica sua frequência (número de vezes que essa chamada de objeto aparece no código) e a organiza no escopo classificado da base ou do site analisado. Vale destacar que a **WBF Analyzer** descarta as aparições de nomes de objetos e propriedades que não sejam realmente uma chamada de objeto, pois seu algoritmo sabe identificar, por exemplo, que $navigator.userAgent = navigator.userAgent$ é uma chamada de objeto e não duas.

Assim, a Tabela 6.2 sumariza a quantidade de chamadas encontradas ao se analisar a base Alexa.com (ano 2016).

Tabela 6.2: Quantidade de chamadas da base Alexa (Ano 2016)

Nível Baixo	Nível Médio	Nível Alto	Total
95	43	19	157

Tabela 6.3: Quantidade de invocações de chamadas da base Alexa (Ano 2016)

Nível Baixo	Nível Médio	Nível Alto	Total de Invocações
131.859	19.295	6.376	157.530

A Tabela 6.3 fornece a quantidade de invocações para a base Alexa.com (ano 2016). Ao todo observou-se 157.530 invocações de chamadas de objetos JavaScript dos 1678 sites analisados. Desse total, 131.859 são chamadas para objetos JavaScript, classificados com risco baixo, 19.295 são chamadas para o objetos de risco médio e 6.376 são objetos de risco alto.

Por fim, a Tabela 6.4 ilustra um comparativo entre a quantidade de invocações de chamadas na base Alexa.com (ano 2016).

Tabela 6.4: Resultados comparativos para invocações de chamadas

Saraiva (2016)	WBF Analyzer
169.902	157.530

Embora o resultado da comparação apresente vantagem para o trabalho de (Saraiva, 2016), é preciso fazer uma contextualização. A **WBF Analyzer** fez sua análise sem aplicar a formação do bloco JS (função que unifica o código da página principal com o código de páginas que são dependentes do domínio principal, ou seja, junta código "in-line" com os externos da página principal). Isso foi feito por não se saber como foram feitas as análises de (Saraiva, 2016). Assim, a **WBF Analyzer** analisou todos os arquivos da base como arquivos independente, gerando escopos classificados para cada site da base. Os escopos classificados da base foram unidos posteriormente para quantificar as chamadas encontradas e suas invocações, e poder estabelecer uma classificação de risco para a base.

Além disso, o trabalho de (Saraiva, 2016) mencionou ter analisado de 10.000 sites, encontradas 10.025 ocorrências e 28.026 chamadas no código para classificação de nível alto; 27.074 ocorrências e 141.260 chamadas para o nível Médio; 261 ocorrências e 616 chamadas para o nível Baixo. No caso, como a autora utilizou uma *regex*, seu algoritmo não diferenciava uma relação do tipo *navigator.userAgent* = *navigator.userAgent*? (contando duas chamadas), coisa que a **WBF Analyzer** sabe fazer.

6.1.2.3 Escopo Classificado

Como forma de provar a importância e a eficácia do Escopo Classificado nas análise da **WBF Analyzer**, a Tabela 6.5 lista as 10 maiores chamadas (em frequência) encontradas na base Alexa.com. Vale lembrar que: (i) chamada é onde ficam armazenadas todas as chamadas de objeto no formato *objeto.propriedade* encontradas; (ii) frequência é onde fica registrado o número de vezes que uma chamada é invocada; e (iii) risco da chamada é onde fica registrado o risco da chamada, podendo ser baixo, médio ou alto.

Tabela 6.5: 10 Chamadas mais encontradas no Escopo Classificado

Chamada	Frequência	Risco da chamada
this.width	36651	Baixo
this.height	32204	Baixo
new Date().getTime()	20289	Baixo
navigator.userAgent	14170	Baixo
document.cookie	3556	Médio
window.innerWidth	3115	Baixo
this.video	2843	Médio
document.referrer	2487	Médio
window.innerHeight	2423	Baixo
canvas.getContext()	2355	Alto

6.1.2.4 Indicadores de Fingerprinting

Tomando como base a Figura 6.4, é possível observar chamadas de objeto pertencentes aos níveis baixo e médio da metodologia criada por (Saraiva, 2016). Nesse caso, tem-se a chamada *this.width* e *this.height*, que são respectivamente *screen.width* e *screen.height*.

```
33  1;screen;screen.height
34  1;screen;screen.width
```

Figura 6.4: Exemplo retirado da dissertação de (Saraiva, 2016).

Ao se analisar um recorte do dicionário de (Saraiva, 2016) para a detecção da chamada *screen.height* e *screen.width*, observam-se dois níveis de detecção, na qual, se existisse a palavra *screen* no código, ela seria contabilizada ou se existisse a chamada *screen.height* ou a chamada *screen.width*, ambas seriam detectadas. Porém, somete a palavra *screen* não é indicativo de uma chamada de objeto em JavaScript. É apenas uma palavra aleatória, declarada no código que poderia aparecer como uma atribuição do tipo $screen = screen.height$. No caso desse exemplo, pelo o que é visto no dicionário de (Saraiva, 2016), ela seria contabilizada (*screen*) como ocorrência e *screen.height*, como uma chamada.

No caso das chamadas *this.height* e *this.width*, elas até seriam extraídas pela *regex*, mas não passaram no crivo do dicionario de (Saraiva, 2016), pois não constam no dicionário. Isso significa que essas são chamadas inéditas descobertas pelo **WBF Analyzer** na análise da base Alexa.com, ano 2016. A prova é fornecida pela Tabela 6.5. Esta descoberta justifica a análise de bases antigas, pois ainda existem informações a serem descobertas que não foram mapeadas por outros pesquisadores.

Tomando como base a Tabela 6.6, que descreve os cinco (5) primeiros elementos mais chamados, é possível afirmar que os sites da base utilizam muitos indicadores de *fingerprinting* capazes de obter informações a respeito da largura e altura da tela, da data e hora do sistema, do tipo do navegador e dos dados de navegação armazenados.

Tabela 6.6: Indicadores de fingerprinting na base Alexa (Ano 2016)

Objeto	Função	Fingerprinting	Categoria
this.width	Retorna a largura da tela em pixels	Tela	Objetos da Tela
this.height	Retorna a altura total da tela	Tela	Objetos da Tela
new date().getTime()	Um número representando os milissegundos passados em 1 de janeiro de 1970 00:00:00 UTC e data atual	Data e Hora	Objetos Genéricos
navigator.userAgent	Retorna o cabeçalho de usuário do agente do navegador	Navegador	Objetos do Navegador
document.cookie	Retorna informações do usuário armazenadas em páginas web	Navegador	Objetos do Navegador

6.1.3 Base Top 50 Alexa (Ano 2021)

A base Top 50 Alexa (Ano 2021) é composta pelos 50 sites, listados na Alexa.com, no ano de 2021 e na categoria de sites internacionais. Nessa base, foram extraídos apenas os códigos JavaScript das páginas principais de cada site. A Figura 6.5 ilustra o resultado da avaliação feita pela **WBF Analyzer**, onde apenas um site (2% do total) não teve indicadores de *fingerprinting*.

Vale destacar que este único site, onde não foi detectado nenhum indicador de *fingerprinting* - domínio *Pandas.tv* - ocorreu porque o site estava em manutenção.

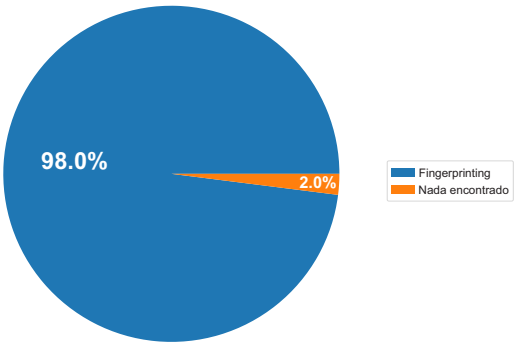


Figura 6.5: Análise da base Alexa.com ano 2021. Fonte: Autor.

6.1.3.1 Classificação de Risco

A Figura 6.6 ilustra o resultado da classificação de risco na base Top 50 Alexa (Ano 2021).

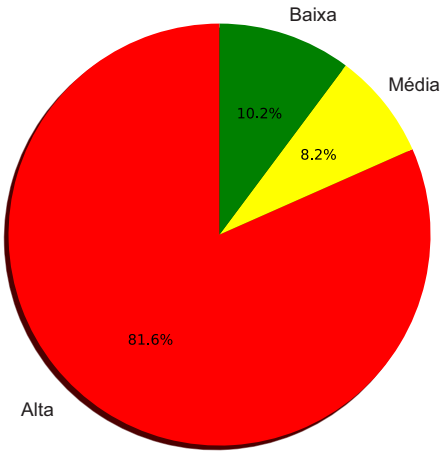


Figura 6.6: Classificação base Top 50 Alexa (Ano 2021). Fonte: Autor.

Observa-se na Figura que a **WBF Analyzer** conseguiu classificar 81.6% (40) dos sites como de risco alto, 8.2% (4) dos sites como de risco médio e 10.2% (5 sites) como de risco baixo. Diferente da base Alexa.com (Ano 2016), o resultado desta base não pode ser comparado com o trabalho de (Saraiva, 2016). Apenas como registro, a Tabela A.4 - Sites e risco da base Top 50 Alexa (Ano 2021), que pode ser consultada no apêndice A, lista os 50 sites avaliados e seu risco atribuído.

Novamente, é importante destacar que esta base contém apenas códigos “in-line” e externos da página principal dos sies citados na Tabela.

6.1.3.2 Chamadas e Invocações

Na base Top 50 Alexa (Ano 2021), foram detectadas um total de 125 chamadas de objetos e 5.592 invocações. Destas, 80 chamadas e 3.800 invocações foram classificadas como de risco baixo, 34 chamadas e 1.491 invocações como de risco médio e 11 chamadas e 301 invocações como risco Alto.

A Tabela 6.7 sumariza os resultados obtidos para a quantidade de chamadas encontradas.

Tabela 6.7: Quantidade de chamadas da base Top 50 Alexa (Ano 2021)

Nível Baixo	Nível Médio	Nível Alto	Total
80	34	11	125

Já a Tabela 6.8 sumariza os resultados obtidos para a quantidade de invocações de chamadas encontradas.

Tabela 6.8: Quantidade de Invocações de chamadas da base Top 50 Alexa (Ano 2021)

Nível Baixo	Nível Médio	Nível Alto	Total de Invocações
3.800	1.491	301	5.592

6.1.3.3 Escopo Classificado

A Tabela 6.9 contém todas as chamadas encontrada na análise da base Alexa.com ano 2021.

Tabela 6.9: Maiores Frequência do Escopo classificado da base Top 50 Alexa (Ano 2021)

Chamada	Freq.	Risco	Chamada	Freq.	Risco
new Date().getTime()	875	Baixo	screen.width	90	Baixo
screen.width	90	Baixo	screen.height	69	Baixo
document.referrer	284	Médio	this.domain	97	Baixo
document.cookie	430	Médio	new Date().setTime()	82	Baixo
window.navigator.userAgent	98	Baixo	window.history	67	Alto
navigator.userAgent	419	Baixo	this.width	488	Baixo
this.height	423	Baixo	window.innerWidth	116	Baixo
window.localStorage	105	Médio	localStorage.setItem()	55	Médio
window.innerHeight	100	Baixo	document.domain	68	Baixo
window.devicePixelRatio	50	Baixo	this.platform	13	Baixo
navigator.plugins	57	Medio	this.history	124	Alto
this.video	252	Medio	this.cookie	57	Medio

6.1.3.4 Indicadores de Fingerprinting

A análise fornecida pela **WBF Analyzer**, na Tabela 6.10, mostra os cinco (5) objetos que mais se destacaram na base.

Tabela 6.10: Objetos que mais se destacaram na base Top 50 Alexa (Ano 2021)

Chamada	Risco	Qtde. de Invocações
new Date()getTime()	Baixo	875
this.width	Baixo	488
document.cookie	Médio	430
this.height	Baixo	423
navigator.userAgent	Baixo	419

Nota-se que o objeto *date*(), invocando a propriedade *getTime*(), tem o maior percentual, o que nos fornece um indicador da tendência de *fingerprinting* para data e hora do computador. Percebe-se também a soma dos valores das chamadas *this.width* e *this.height* com 911 invocações, superando as 875 da chamada de objeto *Date*(). Este também é um indicativo de que nessa base existe uma tendência de *fingerprinting* para largura e altura da tela do usuário. Por fim, outro ponto importante é que se percebe que a predominância para os objetos classificados com risco baixo.

A Tabela 6.11 explica melhor a funcionalidade dos indicadores de *fingerprinting* da base Alexa 2021.

Tabela 6.11: Indicadores de *Fingerprinting* na base Top 50 Alexa (Ano 2021)}

Objeto	Função	Fingerprinting	Categoria
new date().getTime()	Um número representando os milissegundos passados em 1 de janeiro de 1970 00:00:00 UTC e data atual	Data e Hora	Objetos Genéricos
this.width	Retorna a largura da tela em pixels	Tela	Objetos da Tela
document.cookie	Retorna informações do usuário armazenadas em páginas web	Navegador	Objetos do navegador
this.height	Retorna a altura total da tela	Tela	Objetos da Tela
navigator.userAgent	Retorna o cabeçalho de usuário do agente do navegador	Navegador	Objetos do Navegador

6.1.3.5 Indicadores Canvas, History, MimeTypes e Plugins

Com intenção de verificar se a **WBF Analyzer** conseguiria avaliar os cinco sites mais potencialmente perigosos da base e se os sites, em 2021, ainda coletam os objetos que são descritos como potencialmente perigosos por (Saraiva, 2016), foram selecionados objetos que pertencem somente ao nível médio e alto.

É importante ressaltar que este é apenas um parêntese aberto nessa pesquisa, pois nossa intenção não é avaliar a periculosidade dos sites, assim como fez (Saraiva,

2016). Esta pesquisa visa desenvolver um método capaz de detectar indicadores de *fingerprinting* nos sites para que se possa avaliar a tendência de *fingerprinting* dos sites.

Para saber quais os cinco (5) sites que mais coletam os objetos *Canvas*, *history*, *MimeTypes* e *Plugins*, foi realizada uma pesquisa na base com um filtro estabelecido para esse grupo de objetos. Dessa forma, a **WBF Analyzer** pode futuramente fornecer, além dos indicadores de *fingerprinting* que predominam em um site, listas de sites potencialmente perigosos e sua classificação de risco.

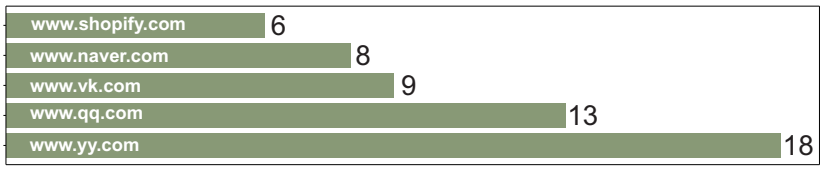


Figura 6.7: Sites que mais coletaram o *objeto Plugins*. Fonte: Autor.

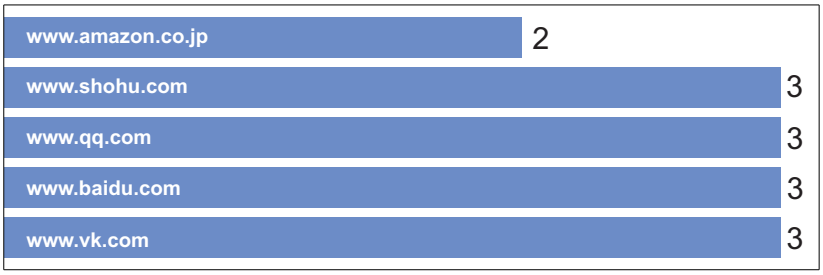


Figura 6.8: Sites que mais coletam o objeto MIME-types. Fonte: Autor.

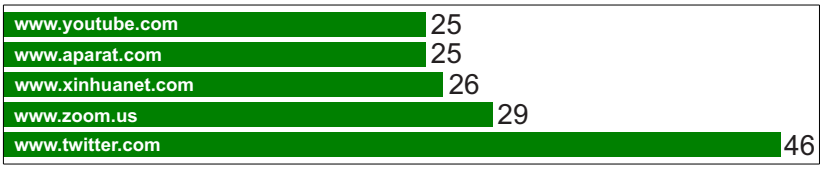


Figura 6.9: Sites que mais coletam o objeto History. Fonte: Autor.

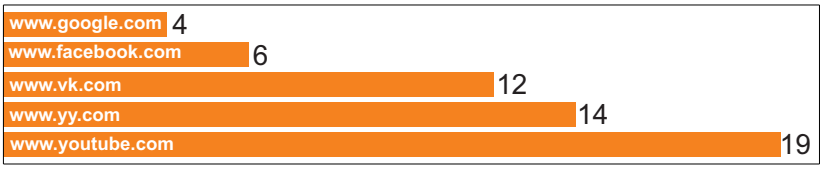


Figura 6.10: Sites que mais coletam o objeto Canvas. Fonte: Autor.

As Figuras 6.7, 6.8, 6.9, 6.10 exemplificam que, ao examinar apenas a página de um site, a análise não retorna um número expressivo de chamadas, mas consegue

identificar os sites que mais coletam os objetos que são passados como filtro para a aplicação.

6.2 Base Top 50 Alexa - Dependências (Ano 2021)

A base Top 50 Alexa - Dependências (Ano 2021) possui um total de 47 (quarenta e sete) sites. Deste total, 74.5% (35) dos sites foram considerados como contendo *fingerprinting*. 25.5% (12) dos sites não foram analisados. A Figura 6.11 ilustra os resultados da análise feita pela **WBF Analyzer**.

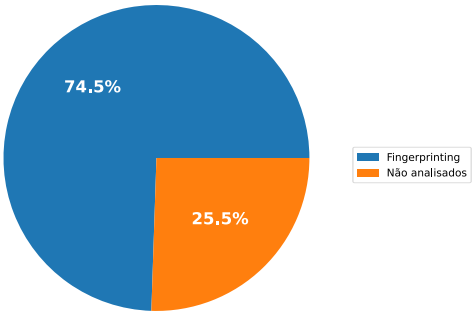


Figura 6.11: Análise da base Top 50 Alexa - Dependências (Ano 2021). Fonte: Autor.

Antes de explicar melhor os resultados, se faz necessária uma breve contextualização sobre a formação dessa base. O processo de *web crawling* foi realizado para 50 (*cinquenta*) sites rankeados na categoria internacional do site https://www.alexa.com/, que estabelece o rank com base no número estimado de visitas que o site recebe. Três desses sites não puderam ser obtidos. Os sites *aparat.com* e *pandas.tv* estavam fora do ar. Já o site *csdn.net* não permitiu baixar seu código, devido a defesas impostas pelo site.

A Figura 6.12 demonstra o esquema utilizado no alexa.com para realizar o processo de *web crawling* e formar um *BlocoJS*.

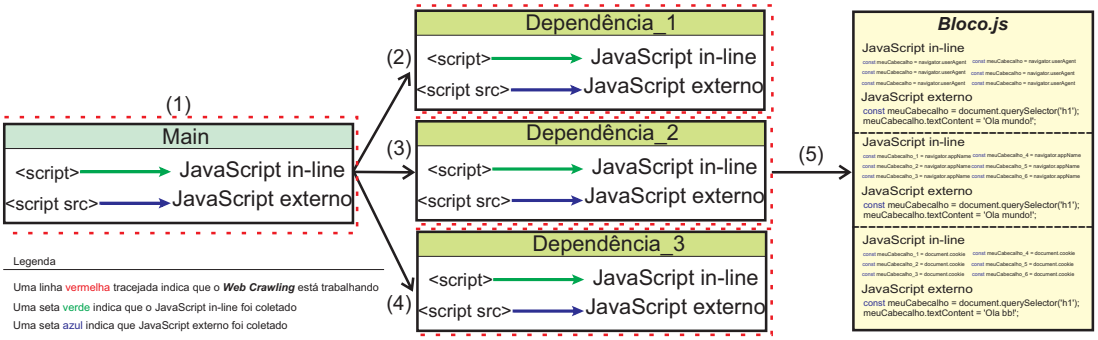


Figura 6.12: Esquema para *web crawler* formar um Bloco JS. Fonte: Autor.

Em (1), todos os scripts do tipo *JavaScript "in-line"*, que estão nas tags *<script>*,

e os *JavaScript externo*, que são apontados por *links* que estão entre as tags *<script src>* da página *main* do site, são coletados. Em seguida, em (2), o processo é repetido para a página de dependência_1. Em (3), o processo é repetido para a página de dependência_2. Em (4), o processo é repetido para a páginas de dependência_3 ou sub-página que compõe o domínio do site. Em (5), o código coletado é agrupado em um *Bloco.js*, que é formado na ordem da extração dos códigos e organizados de forma *TopDown*.

Na Tabela A.5, Dependências analisadas e não analisadas da base Top 50 Alexa - dependências (Ano 2021), que pode ser consultada no apêndice A, são apresentados os sites avaliados e suas respectivas quantidade de dependências analisadas e não analisadas. A Tabela pode ser lida como segue:

- O site `bongacams.com`, na primeira linha, é considerado pela **WBF Analyzer** como o *BlocoJS bongacams.com.js*, que foi formado a partir da união do código extraído de sua página *Main* e de suas páginas secundárias. Assim, o *BlocoJS* é formado por 201 arquivos, todos analisados.

- O site `login.microsoftonline.com`, na terceira linha, é formado a partir da união do código extraído de sua página *Main* e de suas páginas secundárias. Porém, ao executar o teste de abertura da AST, apenas 102 dos 107 permitiram análise.

Percebe-se, na Tabela mencionada acima, que um total de 7232 arquivos foram baixados, destes 4707 arquivos foram avaliados e 2525 não foram.

6.2.1 Classificação de Risco

A Figura 6.13 mostra que a base em análise foi classificada totalmente (100.0%) com risco **alto**.

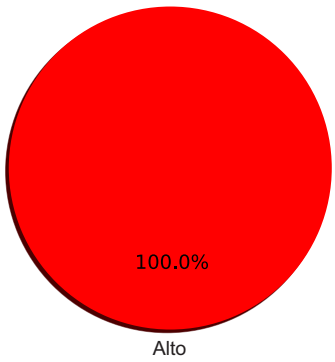


Figura 6.13: Classificação da base Top 50 Alexa - Dependências. Fonte: Autor.

6.2.1.1 Chamadas e Invocações

A Tabela 6.12 sumariza os resultados obtidos para a quantidade de chamadas encontradas.

Tabela 6.12: Quantidade de chamadas da base Top 50 Alexa - Dependências (Ano 2021)

Nível Baixo	Nível Médio	Nível Alto	Total
112	44	21	177

De acordo com a Tabela, foram encontradas 112 chamadas para o nível classificatório **Baixo**, 44 chamadas para o nível classificatório **Médio** e 21 chamadas para o nível classificatório **Alto**, totalizando 177 chamadas encontradas.

Na Tabela 6.13 percebe-se que a **WBF Analyzer** forneceu um total de 403.842 invocações de chamadas de objetos JavaScript para o nível **Baixo**, 117.632 para nível classificatório **médio** e 33.280 para o nível classificatório **alto**. Ao todo, observou-se um total de 554.754 invocações de chamadas de objetos JavaScript para a base avaliada, onde foram analisados 35 sites somando a análise de suas dependências como já explicado anteriormente.

Tabela 6.13: Quantidade de Invocações de chamadas da base Top 50 Alexa - Dependências (Ano 2021)

Nível Baixo	Nível Médio	Nível Alto	Total de invocações
403.842	117.632	33.280	554.754

A *Base Top 50 Alexa - Dependências* foi analisada com a aplicação da formação do *BlocoJS*, uma função criada para juntar todo o código da página principal com o código de páginas que são dependentes do domínio principal ou juntar apenas os código $in - line$ e os externos da página principal. Assim, nessa análise, todos os sites com dependências foram unidas e foi possível estabelecer as relações entre objetos e propriedades ofuscados por substituição de *String de palavra chave* ou não.

6.2.1.2 Escopo Classificado

Como forma de provar a importância e a eficácia do Escopo Classificado nas análise da WBF Analyzer, a Tabela 6.14 lista as 10 maiores chamadas (em frequência) encontra-das na *base Top 50 Alexa - Dependências*. Vale lembrar que Chamada é onde ficam armazenadas todas as chamadas de objeto no formato *objeto.propriedade* encontradas; Frequência é onde fica registrado o número de vezes que uma chamada é invocada; e Risco da chamada é onde fica registrado o risco da chamada, podendo ser baixo, médio ou alto.

Tabela 6.14: 10 Chamadas mais encontradas no Escopo Classificado

Chamada	Frequência	Risco da chamada
new Date().getTime()	87892	Baixo
this.width	64331	Baixo
this.height	57150	Baixo
navigator.userAgent	37851	Baixo
document.cookie	36018	Médio
document.referrer	23726	Médio
window.innerWidth	17318	Baixo
window.innerHeight	14163	Baixo
canvas.getContext()	13977	Alto
this.domain	12814	Baixo

6.2.1.3 Indicadores de Fingerprinting

A análise fornecida pela **WBF Analyzer**, na Tabela 6.15, mostra os cinco (5) objetos que mais se destacaram na base.

Tabela 6.15: Indicadores de Fingerprinting da base Top 50 Alexa - Dependências (Ano 2021)

Objeto	Função	Fingerprinting	Categoria
new date().getTime()	Um número representando os milissegundos passados em 1 de janeiro de 1970 00:00:00 UTC e data atual	Data e Hora	Objetos Genéricos
this.width	Retorna a largura da tela em pixels	Tela	Objetos da Tela
this.height	Retorna a altura total da tela	Tela	Objetos da Tela
navigator.userAgent	Retorna o cabeçalho de usuário do agente do navegador	Navegador	Objetos do Navegador
document.cookie	Retorna informações do usuário armazenadas em páginas web	Navegador	Objetos do Navegador

Nota-se que o objeto *date*(), invocando a propriedade *getTime*() tem o maior percentual, o que nos fornece um indicador da tendência de *fingerprinting* para data e hora do computador. Percebe-se também a soma dos valores das chamadas *this.width* e *this.height* ter 121.481 (*Ceto e vinte e um mil e quatrocentos e oitenta e um*) invocações, superando as 87892 (*Oitenta e sete mil e oitocentos e noventa e dois*) da chamada de objeto *Date*(). Esse também é um indicativo de que nessa base existe uma tendência de *fingerprinting* para largura e altura da tela do usuário. Por fim, outro ponto importante é que se percebe a predominância para os objetos classificados com risco baixo.

6.3 Base Canvas (Ano 2017)

A base *Canvas (Ano 2017)* possui um total de 8350 sites. A Figura 6.14 ilustra os resultados da análise feita pela **WBF Analyzer**.

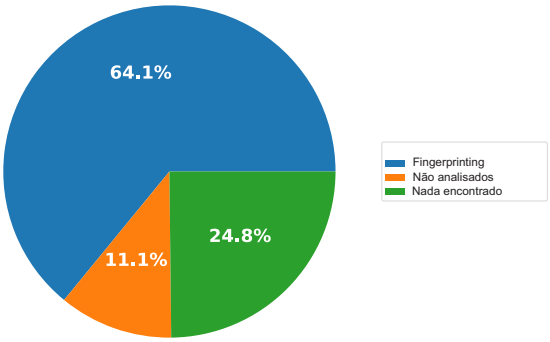


Figura 6.14: Análise da base Canvas (Ano 2017). Fonte: Autor.

Do total, 64.1% (5350) dos sites foram considerados como contendo $fingerprinting$. Já 11.1% (927) dos sites não foram analisados. Por fim, em 24.8.0% (2073) dos sites não foram encontradas chamadas de objeto *JavaScript*.

6.3.1 Classificação de Risco

A Figura 6.15 ilustra o resultado da classificação de risco da base $Canvas(Ano2017)$.

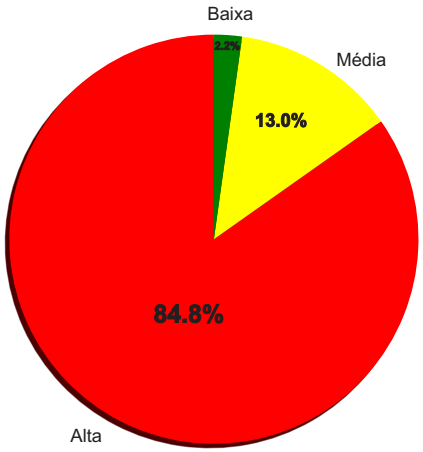


Figura 6.15: Classificação da base Canvas (Ano 2017). Fonte: Autor.

Observa-se na Figura que a **WBF Analyzer** conseguiu classificar os sites, sendo 84.8% (4523 sites) deles com nível de risco alto, 13.0% (696 sites) com nível de risco médio e 2.2% (118 sites) com nível de risco baixo.

6.3.2 Chamadas e Invocações

A Tabela 6.16 sumariza a quantidade de chamadas encontradas ao se analisar a base *Canvas (Ano 2017)*.

Tabela 6.16: Quantidade de chamadas da base Canvas (Ano 2017)

Nível Baixo	Nível Médio	Nível Alto	Total
110	47	18	175

Já a Tabela 6.17 fornece a quantidade de invocações para a base *Canvas* (ano 2017).

Tabela 6.17: Quantidade de Invocações de chamadas da base Canvas (Ano 2017)

Nível Baixo	Nível Médio	Nível Alto	Total de Invocações
557.531	124.175	48.095	729.801

A **WBF Analyzer** estabelece critérios para diferenciar chamadas de objetos de invocações de objeto.

- Se um objeto chama sua propriedade através da notação ponto (.), ela contabiliza como uma chamada encontrada.
- Se um objeto chama uma propriedade que já foi chamada, ela contabiliza essa repetição no código como a frequência de aparição da chamada.

De acordo com a Tabela 6.17, obteve-se um total 729.801 invocações de chamadas de objetos JavaScript. Desse total, 557.531 são chamadas para objetos JavaScript classificados com risco **Baixo**. 124.175 são chamadas para objetos de risco **Médio** e 48.095 são chamadas para objetos de risco **Alto**.

Fazendo uma comparação empírica, apenas de valores, sem qualquer avaliação realizada de fato, o trabalho de (Saraiva, 2016) encontrou 45.626 invocações nessa mesma base. Já a **WBF Analyzer** encontrou 729.801 invocações. Essa diferença é explicada, porque o método de (Saraiva, 2016) não resolve o problema da ofuscação de *string de substituição de palavra chave*.

6.3.2.1 Escopo Classificado

Como forma de provar a importância e a eficácia do Escopo Classificado nas análise da **WBF Analyzer**, a Tabela 6.18 lista as 18 maiores chamadas (em frequência) encontradas na base *Canvas*.

Foram encontradas 124.832 invocações para este objeto que pertence ao nível classificatório **Baixo**. É interessante notar que os objetos do nível **Baixo** é **Médio** são mais costumeiramente utilizados para coletar informações e produzir chaves identificadoras do usuário do que os objetos vinculados ao nível **Alto**.

Tabela 6.18: 18 maiores chamadas encontradas no Escopo Classificado

Chamada	Frequência	Risco da chamada
new Date().getTime()	124832	Baixo
this.width	99257	Baixo
this.height	82544	Baixo
navigator.userAgent	79271	Baixo
document.referrer	27487	Medio
document.cookie	22468	Medio
window.innerWidth	17387	Baixo
window.innerHeight	16137	Baixo
canvas.getImageData()	13673	Alto
window.localStorage	13398	Médio
screen.width	11189	Baixo
canvas.getContext()	10596	Alto
this.video	9720	Medio
screen.height	9583	Baixo
window.screen.width	9117	Baixo
window.screen.height	8616	Baixo
this.history	8421	Alto
canvas.toDataURL()	8215	Alto

A *WBF Analyzer* conseguiu identificar as chamadas de objetos principais desta base, segundo o trabalho de (Elleres, 2017) que também a analisou.

Elas estão destacas em amarelo, representando 13673 invocações para o objeto *canvas.getImageData*(), 10596 invocações para o objeto *canvas.getContext*(), 8215 invocações para o objeto *canvas.toDataURL*().

O objeto *canvas.toDataURL*() é o objeto mais potencialmente perigoso citado por (Saraiva, 2016). Porém, ele é o objeto que menos foi solicitado entre o grupo de objetos *Canvas* e o menos solicitado em relação aos 18 primeiros colocados na análise da **WBF Analyzer**.

6.3.3 Indicadores de Fingerprinting

A Tabela 6.19 descreve os cinco (5) primeiros elementos mais chamados. É possível afirmar que os sites da base utilizam muitos indicadores de *fingerprinting* capazes de obter informações a respeito da data e hora, do tipo do navegador, da largura e altura da tela e do último site que o usuário visitou.

6.4 Base DMOZ (Ano 2016)

A base *DMOZ (Ano 2016)* possui um total de 1564 sites. A Figura 6.16 ilustra os resultados da análise feita pela **WBF Analyzer**.

Do total, 83.0% (1298) dos sites foram considerados como contendo *fingerprinting*.

Tabela 6.19: Indicadores de *Fingerprinting* na base Canvas (Ano 2017)

Objeto	Função	Fingerprinting	Categoria
new date().getTime()	Um número representando os milissegundos passados em 1 de janeiro de 1970 00:00:00 UTC e data atual	Data e Hora	Objetos Genéricos
this.width	Retorna a largura da tela em pixels	Tela	Objetos da Tela
this.height	Retorna a altura total da tela	Tela	Objetos da Tela
navigator.userAgent	Retorna o cabeçalho de usuário do agente do navegador	Navegador	Objetos do Navegador
document.referrer	Retorna o URI da página que continha o link para a página atual	Navegador	Objetos do Navegador

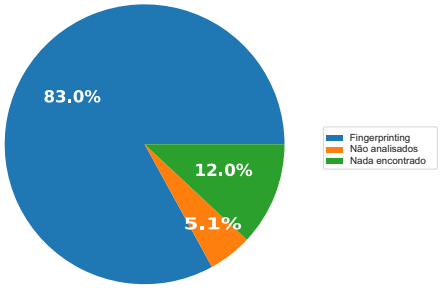


Figura 6.16: Análise da base DMOZ (Ano 2016). Fonte: Autor.

Já 5.1% (79) dos sites não foram analisados. Por fim, em 12.0% (187) dos sites não foram encontradas chamadas de objeto *JavaScript*.

6.4.1 Classificação de Risco

A Figura 6.17 ilustra o resultado da classificação de risco na base DMOZ (ano 2016).

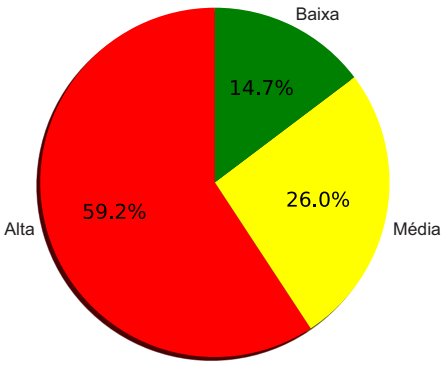


Figura 6.17: Classificação do risco da base DMOZ (Ano 2016). Fonte: Autor.

A **WBF Analyzer** conseguiu classificar os sites com a seguinte proporção:

59.2% (769) deles com nível de risco **alto**, 26.0% (338) com nível de risco **médio** e 14.7% (191) com nível de risco **baixo**. Os resultados obtidos são divergentes daqueles do trabalho de (Saraiva, 2016). A **WBF Analyzer** classificou mais sites com risco **alto**, enquanto o trabalho de (Saraiva, 2016) classificou mais com riscos **baixo** e **médio**. A Tabela 6.20 ilustra essa análise comparativa.

Tabela 6.20: Análise comparativa da classificação da base DMOZ (Ano 2016)

Autores	**Riscos**		
	Baixo	**Médio**	**Alto**
(Saraiva, 2016)	20.0%	47.0%	33.0%
WBF Analyzer	14.7%	26.0%	59.2%

6.4.1.1 Chamadas e Invocações

A Tabela 6.21 ilustra os resultados obtidos para a quantidade de chamadas encontradas na base DMOZ (ano 2016).

Tabela 6.21: Quantidade de chamadas da base DMOZ (Ano 2016)

Nível Baixo	**Nível Médio**	**Nível Alto**	**Total**
93	42	15	150

Observou-se nessa base, de acordo com a Tabela, foram encontradas 93 chamadas para o nível classificatório **Baixo** e 42 chamadas para o nível classificatório **Médio** e 15 chamadas para o nível classificatório **Alto**, totalizando 150 chamadas encontradas.

A Figura 6.18 exibe todas as 15 chamadas de risco **alto** identificadas.

this.history, window.history, canvas.getContext(), canvas.toDataURL(), canvas.getImageData(), this.go(), window.history.go(), widow.history.length window.history.back(), history.go(), history.back(), this.forward(), this.back(), this.history.go(), this.history.legth.

Figura 6.18: Chamadas de objeto do nível alto da base DMOZ (Ano 2016). Fonte: Autor.

Percebe-se na Figura que a *WBF Analyzer* consegue identificar a palavra reservada *this*, elemento não identificado no trabalho de (Saraiva, 2016). Também foi encontrado o objeto *history*, que em JavaScript pode aparecer de várias formas.

Já a Tabela 6.22 fornece a quantidade de invocações para a base DMOZ (ano 2016).

Tabela 6.22: Quantidade de Invocações de chamadas da base DMOZ (Ano 2016)

Nível Baixo	Nível Médio	Nível Alto	Total de Invocações
65.775	13.101	3.044	81.920

Ao todo, observou-se 81.920 invocações de chamadas de objetos JavaScript dos 1298 sites analisados. Desse total, 65.775 são chamadas para objetos JavaScript classificados com risco **baixo**. 13.101 são chamadas para o objetos de risco **médio** e 3.044 são objetos de risco **alto**.

Por fim, a Tabela 6.23 ilustra um comparativo entre a quantidade de invocações de chamadas na base DMOZ (Ano 2016).

Tabela 6.23: Resultados comparativos para invocações de chamadas

Saraiva (2016)	WBF Analyzer
16.621	81.920

Enquanto (Saraiva, 2016) identificou 16.621 chamadas ligadas a *fingerprinting* ao avaliar 1.584 sites, a **WBF Analyzer** conseguiu identificar 81.920 chamadas ligas a *fingerprinting* ao avaliar 1298 sites. Os resultados demonstram que o método proposto neste trabalho é mais eficiente para identificar chamadas de objetos do que o método proposto por (Saraiva, 2016). É preciso contextualizar que os valores apresentados na Tabela são apenas referente a comparação de contagem de chamadas, uma vez que (Saraiva, 2016) também contabilizou aparições isoladas dos objetos ou nomes de objetos usados com referência para a utilização de variáveis.

A Tabela 6.24 foi retirada do trabalho de (Saraiva, 2016) para realizar a comparação de sites avaliados. Já a Tabela 6.25 ilustra os resultados da **WBF Analyzer** para os mesmos sites.

Na Tabela 6.25, resultados na cor verde são aqueles nos quais a **WBF Analyzer** foi mais eficiente. Na cor preta estão os resultados em que houve empate e na cor vermelha os resultados em que a metodologia de detecção de (Saraiva, 2016) foi superior. Assim, nos resultados da comparação direta, no total houve 119 empates, 47 derrotas e 44 vitórias para a **WBF Analyzer**.

6.4.2 Escopo classificado

A Tabela 6.26 contém as 10 chamadas mais encontrada na análise da base DMOZ, enquanto a Figura 6.19 ilustra todas as chamadas de risco **alto** encontradas no escopo classificado.

De acordo com a Figura, percebe-se que os objetos de *fingerprinting* encontrados fazem parte do conjunto do objeto *History* e do conjunto do objeto *Canvas*. Esses objetos coletam informações do histórico do navegador como no caso do objeto *History*, que pode ser usado para saber quais sites um usuário visitou. Também, como no caso do objeto *Canvas*, podem guardar em seu estado atributos sob os quais as

Tabela 6.24: Ocorrência de termos encontrados nos sites da base DMOZ por (Saraiva, 2016)

websites	Nível 1 - Baixa					Nível 2 - Média					Nível 3 - Alta			
	innerWidth	innerheight	userAgent	screen.height	screen.width	referrer	cookie	mimeTypes	plugins	localStorage	getUserMedia	getImageData()	toDataURL()	History
robertkleingallery.com	0	0	0	2	2	0	0	0	0	0	0	0	0	0
pranapoweryoga.com	1	0	3	2	2	0	0	0	0	0	0	0	0	0
joegreenephoto.com	1	5	3	1	0	0	0	0	0	0	0	0	0	0
captel.com	0	0	0	1	1	0	0	0	0	0	0	0	0	0
dynamicpost.co.uk	5	5	1	0	0	0	0	0	0	0	0	0	0	0
oldogpaws.com	0	0	2	2	3	1	36	0	3	0	0	0	0	0
ilight-tech.com	9	4	8	1	1	1	2	4	8	0	0	0	0	0
bbc.com	10	**7**	11	32	32	52	118	6	9	7	0	0	0	0
gourmetsleuth.com	6	6	22	6	6	12	6	16	0	0	0	0	0	0
nordicademy.com.au	4	4	1	0	0	0	3	3	5	0	0	0	0	0
nikonusa.com	29	30	37	10	12	11	49	14	58	8	0	0	0	19
flairstrips.com	14	14	17	0	3	0	12	13	3	0	0	6	1	2
robotshop.com	8	8	13	3	3	7	13	4	6	4	0	3	1	1
fibergarden.com	13	17	9	2	2	5	20	4	5	4	0	3	1	4
newscooter4less.com	5	3	0	0	0	0	0	4	0	0	0	1	1	9

Tabela 6.25: Ocorrência de chamadas encontradas na base DMOZ pela WBF Analyzer

websites	Nível 1 - Baixa					Nível 2 - Média					Nível 3 - Alta			
	innerWidth	innerheight	userAgent	screen.height	screen.width	referrer	cookie	mimeTypes	plugins	localStorage	getUserMedia	getImageData()	toDataURL()	History
robertkleingallery.com	**0**	**0**	**0**	**2**	**2**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
pranapoweryoga.com	**1**	**0**	**3**	**3**	**3**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
joegreenephoto.com	**1**	**5**	**3**	**2**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
captel.com	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
dynamicpost.co.uk	**3**	**5**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
olddogpaws.com	**0**	**0**	**4**	**3**	**4**	**1**	**11**	**0**	**5**	**0**	**0**	**0**	**0**	**3**
ilight-tech.com	**8**	**8**	**14**	**28**	**29**	**1**	**1**	**1**	**5**	**0**	**0**	**0**	**0**	**0**
bbc.com	**9**	**6**	**11**	**11**	**13**	**13**	**62**	**9**	**12**	**1**	**0**	**0**	**0**	**0**
gourmetsleuth.com	**4**	**4**	**18**	**6**	**10**	**12**	**6**	**0**	**0**	**8**	**0**	**0**	**0**	**4**
nordcademy.com.au	**4**	**4**	**3**	**24**	**24**	**0**	**2**	**3**	**5**	**0**	**0**	**0**	**0**	**0**
nikonusa.com	**11**	**9**	**41**	**399**	**400**	**5**	**42**	**6**	**31**	**16**	**0**	**0**	**0**	**23**
flairstrips.com	**4**	**2**	**25**	**55**	**73**	**0**	**2**	**1**	**10**	**2**	**0**	**6**	**1**	**0**
robotshop.com	**4**	**6**	**24**	**34**	**31**	**5**	**10**	**0**	**7**	**8**	**0**	**3**	**1**	**1**
fibergarden.com	**5**	**8**	**15**	**25**	**25**	**3**	**9**	**0**	**1**	**9**	**0**	**3**	**1**	**0**
newscooters4less.com	**4**	**2**	**1**	**13**	**14**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

Tabela 6.26: Chamadas mais encontradas no Escopo Classificado

Chamada	Frequência	Risco da chamada
this.width	15238	Baixo
this.height	13081	Baixo
new Date().getTime()	12281	Baixo
navigator.userAgent	8125	Baixo
document.cookie	3126	Médio
document.referrer	2048	Médio
window.innerheight	1516	Baixo
this.video	1445	Médio
window.innerWidth	1403	Baixo
screen.width	1132	Baixo

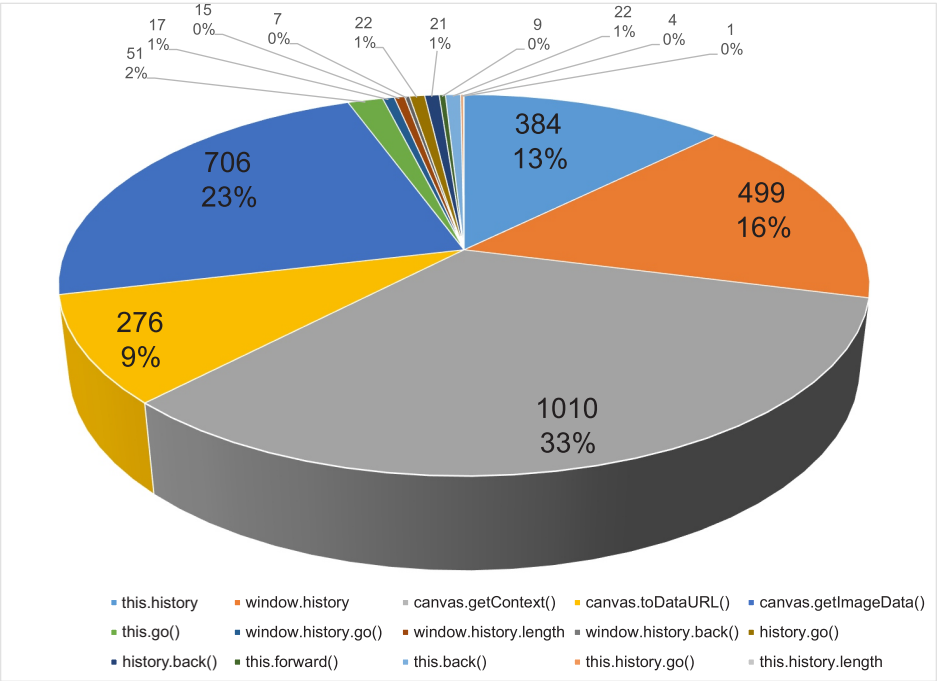


Figura 6.19: Chamadas de Risco Alto. Fonte: Autor.

primitivas gráficas operam e esses valores podem ser coletados e com eles produzidas chaves de identificação do usuário.

6.4.3 Indicadores de Fingerprinting

A Tabela 6.27 ilustra os indicadores de *fingerprinting* da base DMOZ.
Observou-se na base DMOZ que os sites tendem a fazer *fingerprinting* baseados

Tabela 6.27: Indicadores de *Fingerprinting* na base DMOZ (Ano 2016)

Objeto	Função	Fingerprinting	Categoria
this.width	Retorna a largura da tela em pixels	Tela	Objetos da Tela
this.height	Retorna a altura total da tela	Tela	Objetos da Tela
new date().getTime()	Um número representando os milissegundos passados em 1 de janeiro de 1970 00:00:00 UTC e data atual	Data e Hora	Objetos Genéricos
navigator.userAgent	Retorna o cabeçalho de usuário do agente do navegador	Navegador	Objetos do Navegador
document.cookie	Retorna informações do usuário armazenadas em páginas web	Documento HTML	Objetos Genéricos

em um conjunto de objetos que retornam informações a respeito largura da tela, da altura da tela, da data e hora, das informações do navegador e correlação de navegação do usuário. Percebe-se também que, dos cinco objetos mais invocados, quatro estão ligados ao nível classificatório **baixo** e apenas um objeto está ligado ao nível classificatório **médio** e não foram retornados entre os cinco objetos mais populares da base objetos ligados ao nível classificatório **alto**.

6.5 Discussão

Diferente do trabalho de (Saraiva, 2016), os resultados da **WBF Analyzer** demonstram que a ideia de formar de um *bloco.js* único contendo todos os códigos organizados de forma *top down* favorece a avaliação, pois casos como:

- *var t = navigator*, onde a variável *t* pode ser declarada no código $in-line$ de uma página e ser invocada através da notação . (*ponto*) no código $in-line$ de outra página, para chamar sua propriedade *plugins* e assim ter um $fingerprinting = t.plugins$ facilmente detectados.

Observa-se que, na comparação da Base Top 50 Alexa (2021) com a Base Top 50 Alexa - Dependências, os resultados diferem com uma margem de crescimento considerável no número de objetos encontrados em favor da Base Top 50 Alexa - Dependências.

O resultado demonstra que a **WBF Analyzer** obteve um desempenho considerável ao encontrar mais de 500.000 objetos fornecedores de informações, ao avaliar a base unindo todos os códigos $in-line$ e *externos* da página principal de cada site avaliado com os código $in-line$ e *externos* chamados por cada página secundária. Já na avaliação da Base Top 50 Alexa (2021), onde foram considerados apenas os códigos $in-line$ e *externos* da página principal dos sites avaliados, os resultados são inferiores.

Falando sobre a identificação de casos especiais como a ofuscação de string de substituição de palavras chave, a Figura 6.20 é um recorte dos logs do dicionário de identificadores e do escopo reduzido, que foram extraídos de um *bloco.js* da Base Top 50 Alexa - Dependências, provando que a **WBF Analyzer** foi capaz de detectar casos de ofuscação.

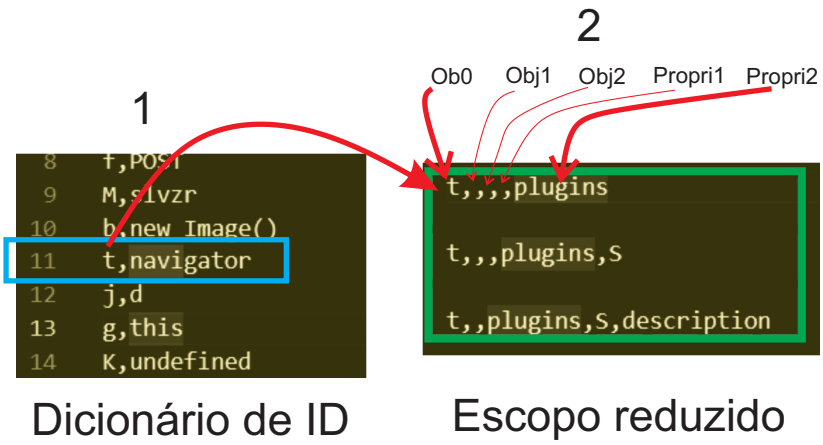


Figura 6.20: Exemplo de Detecção. Fonte: Autor.

Na Figura, percebe-se no Dicionário de ID que a chave t aponta para o valor *navigator*. Já no escopo reduzido, a chave t é alocado no $Ob0$ ($Objeto0$), os $Obj1$, $Obj2$, $Propri1$ estão vazios, e $Propri2$ aloca o elemento *plugins*. A partir dessas informações, os algoritmos de comparação e correspondência conseguem identificar a chamada *navigator.plugins*.

Quanto à classificação de risco das bases avaliadas, a Tabela 6.28 apresenta, de forma resumida, uma comparação entre os resultados obtidos nas 5 bases de dados, ilustrando os percentuais obtidos em cada uma.

Tabela 6.28: Percentuais de classificação das Bases de Dados

Base	**N Sites**	**Baixa**	**Média**	**Alta**
Base Alexa ($Ano2016$)	1678	4.6%	16.7%	78.7%
Base Top 50 Alexa ($Ano2021$)	49	10.2%	8.2%	81.6%
Base Top 50 Alexa - Dependências ($Ano2021$)	35	0.0%	0.0%	100.0%
Base Canvas ($Ano2017$)	5350	2.2%	13.0%	84.8%
Base DMOZ ($Ano2016$)	1298	14.7%	26.0%	59.2%

As bases que obtiveram as maiores classificações de sites com risco **alto** foram as bases Top 50 Alexa - Dependências (Ano 2021), Canvas (Ano 2017), Top 50 Alexa (Ano 2021), com 100%, 84,8% e 81,6%, respectivamente.

Os resultados da base Top 50 Alexa - Dependências provam que existe diferença de classificação dos sites ao ser avaliado o conjunto de páginas secundarias como se fosse um único bloco de código.

Os resultados da base Canvas (Ano 2017) demonstraram que essa base, apesar de ser considerada por (Elleres, 2017) como potencialmente voltada para o objeto *Canvas* e para o nível classificatório **alto**, tem predominância de chamadas com referência ao

nível classificatório **baixo**, sendo 110 chamadas desse nível contra apenas 18 chamas do nível **alto**. Também, nessa base, o número de invocações de chamadas do nível **baixo** é igual a 557.531 invocações, contra 48.095 invocações do nível **alto**. Por fim, verificou-se então que nessa base o indicar predominante de *fingerprinting* é a chamada de objeto JavaScript *new Date().getTime()* com 124.832 invocações, contra 10.596 invocações da chamada *canvas.getContext()*, 8215 invocações da chamada *canvas.getImageData()* e 13.673 invocações da chamada *canvas.toDataURL()*.

Os resultados da *base Top 50 Alexa (Ano 2021)* comprovaram que existe uma linha de crescimento na coleta de objetos JavaScript acentuada por sites que, em tese, não oferecem risco algum para os usuários. Na base, identificou-se que 81.6% dos sites apresentam risco **alto**, enquanto que na avaliação de (Saraiva, 2016) 59% dos sites foram avaliados como potencialmente perigosos.

Os resultados obtidos para a *base Alexa (Ano 2016)* comprovaram que a **WBF Analyzer** identificou um padrão de chamada que utiliza a palavra reservada *this* em JavaScript para acessar propriedades de objetos como em *this.userAgent*, *this.width*, *this.height* e outros. Esse padrão não foi documentado por (Saraiva, 2016; Acar et al., 2013; Beaucamps and Filiol, 2007; Bujlow et al., 2017) e outros pesquisadores.

Por fim, na *base DMOZ* observou-se que a **WBF Analyzer** conseguiu identificar 12,2% de risco **alto** a mais que a metodologia de identificação proposta por (Saraiva, 2016).

Discutindo um pouco mais sobre a classificação para os indicadores de *website Fingerprinting*, pode-se observar na Tabela 6.29, o conjunto de objetos dominante nas 5 (Cinco) bases analisadas. Quatro objetos são do nível classificatório **baixo** e dois são do **médio**.

Tabela 6.29: Indicadores de *Browser Fingerprinting*

	Nível Baixo				**Nível Médio**	
Bases	**new Date().getTime()**	**this.width**	**this.height**	**navigator.userAgent**	**document.cookie**	**document.referrer**
Alexa (Ano 2016)	20289	36651	32204	14170	3556	—-
Top 50 Alexa (Ano 2021)	875	478	423	419	430	—-
Top 50 Alexa - Dependências (Ano 2021)	87892	64331	57150	37851	36018	—-
Canvas (Ano 2017)	124832	99257	82544	79271	—-	27478
DMOZ (Ano 2016)	12281	15238	13081	8125	3126	

O objeto *this* chamando a propriedade *width* ocorreu 36651 vezes para a análise da base *Alexa (Ano 2016)*, enquanto que em (Saraiva, 2016) não foi mencionada

nenhuma ocorrência dessa chamada de objeto, significando que este objeto não foi detectado. O mesmo fato ocorre para o objeto *this*, chamando a propriedade *height*.

Na base *Top 50 Alexa (Ano 2021)*, observa-se que, apesar de terem sido avaliados, apenas 49 sites os resultados foram satisfatórios, pois foram identificados o conjunto de objetos que mais se destacaram como sendo o mesmo conjunto de objetos observados na análise da base *Alexa (Ano 2016)*. Além disso, a predominância de $fingerprinting$ de data e hora obteve o maior valor em relação aos outros indicadores avaliados.

Os resultados para a base *Top 50 Alexa - Dependências*, que avaliou em torno de 47 sites (códigos JavaScript) foram positivos, pois apresentam maior quantidade em relação ao quantitativo da base *Top 50 Alexa (Ano 2021)*, que possui o mesmo conjunto de sites avaliados. Esta avaliação demonstrou que ao avaliar os códigos das páginas secundárias, pode-se detectar muito mais objetos e assim saber qual a frequência de utilização desses objetos e suas propriedades. Nessa base, o objeto *new Date().getTime* manteve sua predominância como o objeto mais coletado pelos sites.

Quanto aos resultados da base *Canvas 2017*, apesar do objeto *Canvas* ter sido detectado pela **WBF Analyzer**, ele não se destacou o suficiente para compor o conjunto dos 5 (*Cinco*) objetos mais coletados da base. O objeto $document.referrer$ foi detectado nas outras bases avaliadas, mas se destacou com maior número de invocações na base *Canvas*. Essa base foi a única a apresentar um conjunto de objetos diferente das demais. Quanto ao indicador, o *fingerprinting* de data e hora foi coletado por 124832 vezes, também este objeto não foi relatado no trabalho de (Saraiva, 2016).

A base *DMOZ* foi a que apresentou menos coleta de informações em relação a comparação com as outras bases, tendo como indicador principal o objeto *this*, chamando a propriedade *width*, sendo assim sua tendência para *fingerprinting* de tela.

Para finalizar, é possível afirmar que o objetivo dessa dissertação (Um método para identificar um indicador ou um conjunto de indicadores de fingerprinting presente em páginas web) foi alcançado, uma vez que os resultados em base de sites reais comprovam que o método desenvolvido para detectar, quantificar e classificar chamadas de objetos Javascipt pode produzir resultados que favoreçam a análise da tendência de crescimento de coleta de dados feita pelos sites.

Capítulo 7

Conclusão

As técnicas de *fingerprinting* aplicadas a páginas web são comuns nos tempos atuais. Ainda assim, os usuários não são informados sobre a coleta de suas informações. Embora as formas, os mecanismos e suas consequências sejam conhecidos, temas como a privacidade dos usuários e métodos para identificar, bloquear e evitar a coleta de dados do usuário ainda são motivos de discussão.

Este trabalho propôs um método para detectar chamadas em códigos de páginas web que podem fornecer um indicativo do estado atual da coleta de dados via *fingerprinting* de objetos de JavaScript. Como forma de provar a eficiência do método, várias bases de páginas web, referente aos ano de 2016, 2017 e 2021, foram testadas e o resultado mostra que existe um conjunto de objetos de JavaScript formado por *new Date().getTime(), this.width, this.heigt navigator.userAgent, document.cookie* relacionados a *browser fingerprinting*, que persistiu nas 5 primeiras colocações em todas as bases avaliadas. Esses indicadores demonstram que bloquear a coleta de informações que possam gerar um identificador único do usuário nos tempos atuais é uma tarefa difícil, pois esses objetos são comumente utilizados em páginas web para fins benéficos como ajustar as preferências de navegação do usuário.

Portanto, a fim de impedir o bloqueio de páginas que coletam dados sem a anuência do usuário, os sites associam a identificação, o rastreamento e a coleta de informações a recursos que oferecem facilidade para a atividade de navegação na web.

7.1 Dificuldades encontradas

Durante a implementação deste tralho foram enfrentadas dificuldades em razão da:

- Extração de códigos de JavaScript dos sites modernos via *web crawling*.
- Leitura de bases extraídas por outros pesquisadores.
- Extração de dados da AST.

- Elaboração de um algoritmo para resolver a ofuscação de string de substituição de palavras chave.

Para burlar as defesas dos sites, foi preciso desenvolver um script de *web crawling* que permitisse o download dos códigos de JavaScritp a uma velocidade aceitável.

Para garantir a eficiência do pré-processamento da **WBF Analyzer** foi necessário criar scripts com capacidade de remover elementos que impediam a geração da AST. Foi um grade desafio mapear esses elementos, pois tratam-se de elementos textuais diversos como **%, : , ; < <> !** e outros.

Para implementar o script de extração de dados da AST foi importante compreender como funciona sua estrutura. A documentação da biblioteca Esprima é falha e não existem trabalhos acadêmicos relacionados que expliquem em detalhes o processo de caminhar, avaliar e extrair dados da AST.

Para implementar a normalização era necessário que o algoritmo da aplicação soubesse entender a diferença entre um objeto completo e um ofuscado por string de substituição de palavra chave. Inicialmente pensou-se em usar recursão, mas durante os testes verificou-se que em JavaScript o processo de recursão possui limitação a 20.000 chamadas recursivas, esse fato impossibilita a avaliação de código em torno de 2 milhões de linhas.

7.2 Contribuições Alcançadas

Esta dissertação contribui com a criação de um método para identificar *Browser Fingerprinting* empregando a análise da AST de código JavaScript.

Além disso, esta pesquisa demonstrou, através dos dados avaliados, que nunca houve uma queda na coleta de objetos JavaScript relacionados a *Browser Fingerprinting* no decorrer dos anos entre 2016 a 2021. Mostrou que, ao contrário do que se pensava, a coleta de informações através de objetos JavaScript está em uma linha constante de crescimento e, por esta razão, avalia-se que os sites estão utilizando *Browser Fingerprinting* para coletar mais informações e realizar mais rastreamento do usuário na web.

Este trabalho também contribuiu em provar que existe diferença na identificação de *Browser Fingerprinting* ao avaliar somente a página principal e avaliar a página principal juntamente com as páginas secundárias ou links que se relacionam diretamente com os sites. Demonstrou que a informação do usuário é coletada em cada ação que ele faz em uma página web, pois é grande a quantidade de invocações de chamadas que são encontradas ao se examinar os códigos JavaScript das páginas secundárias. Vale destacar que não foi encontrado referências que indiquem que algum trabalho científico avaliou todas as páginas secundárias de um site.

Este trabalho avançou um passo na fronteira de pesquisa na área de *Browser Fingerprinting*, uma vez que deu continuidade ao trabalho de (Saraiva, 2016), implementando seu método de classificação na **WBF Analyzer** e resolvendo um dos problemas de ofuscação de termos *fingerprinting* citado em sua seção de trabalhos futuros.

7.3 Trabalhos Futuros

As pesquisas sobre *fingerprinting* estão sendo atualizadas constantemente, pois novos experimentos surgem, trazendo para a comunidade científica inovações que aprimoram processos focados nessa área. Os trabalhos futuros estão elencados a seguir:

1. Desenvolver uma interface web para a **WBF Analyzer**;
2. Aprimorar o algoritmo da normalização da **WBF Analyzer** para identificar casos complexos de ofuscação de string de substituição de palavra chave. A Figura 7.1 ilustra um caso de ofuscação de string com dois níveis de objeto sendo atribuído para uma variável. Neste trabalho foi tratado o caso de atribuição de um nível de objeto ofuscado.

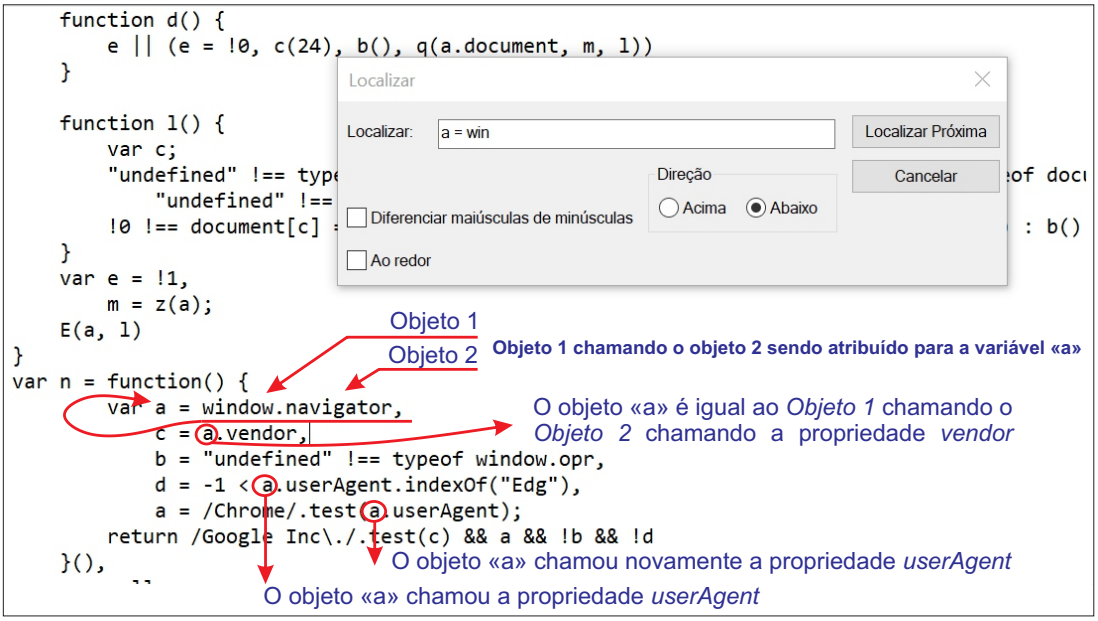


Figura 7.1: Caso complexo de ofuscação de *string*. Fonte: Autor.

3. Aprimorar o *web crawling* utilizando automações de ações em Selenium[1], a fim de imitar um comportamento humano ou utilizando aprendizado de máquina para este fim.
4. Documentar a seção de resultados da **WBF Analyzer**
5. Aprimorar **WBF Analyzer** para identificar chamadas de objetos JavaScript relacionados à *web phishing*

[1]Selenium é um framework portátil para testar aplicativos web, nele podem ser criados comandos que executam ações de forma dinâmica e automatizada em páginas web como clicar em links por exemplo.

Referências Bibliográficas

G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. Fpdetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1129–1140. ACM, 2013.

G. Antal, P. Hegedus, Z. Tóth, R. Ferenc, and T. Gyimóthy. Static javascript call graphs: A comparative study. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 177–186. IEEE, 2018.

P. Beaucamps and É. Filiol. On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3 (1):3–21, 2007.

T. Bujlow, V. Carela-Español, J. Sole-Pareta, and P. Barlet-Ros. A survey on web tracking: Mechanisms, implications, and defenses. *Proceedings of the IEEE*, 105(8): 1476–1510, 2017.

A. Cooper, H. Tschofenig, D. B. D. Aboba, J. Peterson, J. Morris, M. Hansen, and R. Smith. Privacy Considerations for Internet Protocols. RFC 6973, July 2013. URL https://rfc-editor.org/rfc/rfc6973.txt.

A. B. Damasceno. TaintJSec: um método de análise estática de marcação em código Javascript para detecção de vazamento de dados sensíveis. Master's thesis, Programa de Pós-Graduação em Informática, Instituto de Computação, Universidade Federal do Amazonas, 2017.

P. Eckersley. How unique is your web browser? In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 1–18. Springer, 2010.

EEF. Panopticlick is your browser safe against tracking? https://panopticlick.eff.org/. Acessado em : 11-12-2019.

P. A. d. P. Elleres. Detecção de Canvas Fingerprinting em páginas Web baseada em Modelo Vetorial. Master's thesis, Programa de Pós-Graduação em Informática, Instituto de Computação, Universidade Federal do Amazonas, 2017.

A. FaizKhademi, M. Zulkernine, and K. Weldemariam. Fpguard: Detection and prevention of browser fingerprinting. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 293–308. Springer, 2015.

B. Feinstein, D. Peck, and I. SecureWorks. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. *Black Hat USA*, 2007, 2007.

U. Fiore, A. Castiglione, A. De Santis, and F. Palmieri. Countering browser fingerprinting techniques: Constructing a fake profile with google chrome. In *2014 17th International Conference on Network-Based Information Systems*, pages 355–360. IEEE, 2014.

A. Gorji and M. Abadi. Detecting obfuscated javascript malware using sequences of internal function calls. In *Proceedings of the 2014 ACM Southeast Regional Conference*, page 64. ACM, 2014.

X. He, L. XU, and C. CHA. Malicious javascript code detection based on hybrid analysis. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 365–374. IEEE, 2018.

P. Hraška. Browser fingerprinting. Master's thesis, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, 2018.

A. F. Khademi, M. Zulkernine, and K. Weldemariam. An empirical evaluation of web-based fingerprinting. *IEEE Software*, 32(4):46–52, 2015.

A. Kobusińska, J. Brzeziński, and K. Pawulczuk. Device fingerprinting: Analysis of chosen fingerprinting methods. In *International Conference on Internet of Things, Big Data and Security*, volume 2, pages 167–177. SCITEPRESS, 2017.

T. D. Laksono, Y. Rosmansyah, B. Dabarsyah, and J. U. Choi. Javascript-based device fingerprinting mitigation using personal http proxy. In *2015 International Conference on Information Technology Systems and Innovation (ICITSI)*, pages 1–6. IEEE, 2015.

P. Laperdrix, W. Rudametkin, and B. Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 878–894. IEEE, 2016.

H. Le, F. Fallace, and P. Barlet-Ros. Towards accurate detection of obfuscated web tracking. In *2017 IEEE International Workshop on Measurement and Networking (M&N)*, pages 1–6. IEEE, 2017.

T.-C. Li, H. Hang, M. Faloutsos, and P. Efstathopoulos. Trackadvisor: Taking back browsing privacy from third-party trackers. In *International Conference on Passive and Active Network Measurement*, pages 277–289. Springer, 2015.

G. Lu and S. Debray. Automatic simplification of obfuscated javascript code: A semantics-based approach. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 31–40. IEEE, 2012.

J. R. Mayer. *Any person... a pamphleteer": Internet Anonymity in the Age of Web 2.0*. PhD thesis, Faculty of the Woodrow Wilson School of Public and International Affairs, Princeton University, 2009.

Microsoft. Don't get scroogled by gmail. https://news.microsoft.com/2013/02/07/dont-get-scroogled-by-gmail/, 2013. Acessado em : 11-12-2019.

Mozilla. Herança e cadeia de protótipos (prototype chain). https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain, 2020. Acessado em : 04-05-2020.

F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2010.

N. Nikiforakis and G. Acar. Browse at your own risk. *IEEE Spectrum*, 51(8):30–35, 2014.

N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy*, pages 541–555. IEEE, 2013.

N. Nikiforakis, W. Joosen, and B. Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, pages 820–830, 2015.

M. Rausch, N. Good, and C. J. Hoofnagle. Searching for indicators of device fingerprinting in the javascript code of popular websites. In *Proceedings, Midewest Instruction and Computing Symposium*, 2014.

T. Saito and R. Koshiba. Examination and comparison of countermeasures against web tracking technologies. In *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 477–489. Springer, 2019.

A. R. Saraiva. Determinando o risco de Fingerprinting em páginas Web. Master's thesis, Programa de Pós-Graduação em Informática, Instituto de Computação, Universidade Federal do Amazonas, 2016.

L. Scism and M. Maremont. Insurers test data profiles to identify risky clients. *The Wall Street Journal*, 19:2010, 2010.

P. Skolka, C.-A. Staicu, and M. Pradel. Anything to hide? studying minified and obfuscated code in the web. In *The World Wide Web Conference*, pages 1735–1746. ACM, 2019.

R. Upathilake, Y. Li, and A. Matrawy. A classification of web browser fingerprinting techniques. In *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2015.

T. van Zalingen and S. Haanen. Detection of browser fingerprinting by static javascript code classification. 2018.

Y. Wu, D. Meng, and H. Chen. Evaluating private modes in desktop and mobile browsers and their resistance to fingerprinting. In *2017 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2017.

Apêndice A

Outras Informações

Tabela A.1: Variáveis de ambiente

Variável	Valor	Descrição
Screen size	1440x900	Fornece a resolução da tela
Available size	1440x827	Fornece a resolução da tela
Color depth	24	Fornece a profundidade de bits da tela
Pixel ratio	2	Fornece a proporção da resolução em pixels
AdBlock	true	Informa se o adBlock está instalado
Cookies enabled	true	Informa se Cookies estão habilitados
Do not Track (DNT)	false	Informa se o rastreamento está habilitado
Plugins	Chrome PDF Plugin,...	Informa os plugins instalados
IE plugins	empy	Informa os plugins instalados
Indexed database	true	Informa se existem dados armazenados
Local storage	true	Informa se existem dados armazenados
Session storage	true	Informa se existem dados armazenados
User-agent	StartFragmentMozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AplewebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 ...	Fornece dados sobre o navegador

Tabela A.2: Nós candidatos

Nó candidato	Descrição	Código
Assignment expression	Valor atribuído a um operando do lado esquerdo	a = navigator
Assignment expression with object expression	Objeto atribuído a um operando do lado esquerdo	a = {navigator: appName} b = {navigator, appName}
Assignment expression with array expression	Array atribuído a um operando do lado esquerdo	a = [navigator, appName]

Assignment expression with function expression	Uma expressão de função atribuída ao operando do lado esquerdo	g = navigator
		i = function(){
		return g.appName
		
		}

Assignment expression with member expression	Membro de expressão atribuído ao operando do lado esquerdo	d = navigator.appName
Variable declaration with object expression	Declaração de variável inicializada com expressão de objeto	var m = {navigator: appName} var l = {navigator, appName}
Variable declaration with array expression	Declaração de variável inicializada com expressão de array	var j = [navigator.appName, navigator, appName]
Variable declaration	Declaração de variável com objeto sendo atribuído ao operando do lado esquerdo	var h = navigator
Variable declaration with member expression	Declaração de variável com membro de expressão sendo atribuído ao operando do lado esquerdo	var x = navigator.appName var s = a.appName

Variable declaration with function expression	Declaração de variável com expressão de função sendo atribuída ao operando do lado esquerdo	g = navigator
		i = function(){
		return g.appName
		
		}

Níveis	*Objetos*
Baixo	this.window.navigator.userAgent; window.navigator.userAgent; this.navigator.userAgent; navigator.userAgent; this.window.navigator.hardwareConcurrency; window.navigator.hardwareConcurrency; this.navigator.hardwareConcurrency; navigator.hardwareConcurrency; this.window.navigator.bluetooth; window.navigator.bluetooth; this.navigator.bluetooth; navigator.bluetooth; this.window.navigator.appCodename; window.navigator.appCodename; this.navigator.appCodename; navigator.appCodename; this.window.navigator.appCodename; window.navigator.appCodename; this.navigator.appCodename; navigator.appCodename; this.window.product; this.product; window.product; this.window.productSub; window.productSub; this.productSub; this.window.navigator.vendor; window.navigator.vendor; this.navigator.vendor; navigator.vendor; this.window.navigator.appMinorversion; window.navigator.appMinorversion; this.navigator.appMinorversion;navigator.appMinorversion; this.window.navigator.maxTouchPoints; window.navigator.maxTouchPoints; this.navigator.maxTouchPoints; navigator.maxTouchPoints; this.window.navigator.getInstalledRelatedApps(); window.navigator.getInstalledRelatedApps(); this.navigator.getInstalledRelatedApps(); navigator.getInstalledRelatedApps(); this.window.navigator.language; window.navigator.language; this.navigator.language; navigator.language; this.window.navigator.browserLanguage; window.navigator.browserLanguage; this.navigator.browserLanguage; navigator.browserLanguage; this.window.navigator.userLanguage; window.navigator.userLanguage; this.navigator.userLanguage; navigator.userLanguage; this.window.navigator.languages; window.navigator.languages; this.navigator.languages; navigator.languages; this.window.navigator.cpuClass; window.navigator.cpuClass; this.navigator.cpuClass; navigator.cpuClass; this.window.navigator.oscpu; window.navigator.oscpu; this.navigator.oscpu; navigator.oscpu; this.window.navigator.plataform; window.navigator.plataform;this.navigator.plataform; this.navigator.plataform; this.window.navigator.geolocation; window.navigator.geolocation; this.navigator.geolocation; navigator.geolocation; this.window.navigator.vendorSub; window.navigator.vendorSub; this.navigator.vendorSub; navigator.vendorSub; this.window.navigator.online; window.navigator.online; this.navigator.online; naigator.online; this.window.navigator.appVersion; window.navigator.appVersion; this.navigator.appVersion; navigator.appversion; this.window.navigator.appName; window.navigator.appName; this.navigator.appName; navigator.appName; this.window.navigator.doNotTrack; window.navigator.doNotTrack; this.navigator.doNotTrack; navigator.doNotTrack; this.window.screen.availHeight; window.screen.availHeight; this.screen.availHeight; screen.availHeight; this.window.screen.availWidth; window.screen.availWidth; this.screen.availWidth; screen.availWidth; this.window.screen.colorDepth; this.window.screen.colorDepth; screen.colorDepth; this.window.screen.height; window.screen.height; this.screen.height; screen.height; this.window.screen.pixelDepth; window.screen.pixelDepth; this.screen.pixelDepth; screen.pixelDepth; this.window.screen.with; window.screen.with; this.screen.with; screen.with;this.window.matchMedia(); window.matchMedia(); this.matchMedia(); this.window.devicePixelRatio; window.devicePixelRatio; this.devicePixelRatio; devicePixelRatio; this.window.inner.height; window.inner.height; this.inner.height; inner.height; this.window.outer.width; window.outer.width; this.outer.width; outer.width; this.window.outer.height; window.outer.height; this.outer.height; outer.height; Date.getTimezoneOffset(); new Date();
Médio	this.window.document.referrer; window.document.referrer; this.document.referrer; document.referrer; this.window.document.cookie; window.document.cookie; this.document.cookie; document.cookie; this.window.document.domain; window.document.domain; this.document.domain; document.domain; this.window.document.createElement(); window.document.createElement(); this.document.createElement(); document.createElement(); Modernizr.geolocation; Modernizr.video; this.window.navigator.cookieEnabled; window.navigator.cookieEnabled; this.navigator.cookieEnabled; navigator.cookieEnabled; this.window.navigator.javaEnabled(); window.navigator.javaEnabled(); this.navigator.javaEnabled(); navigator.javaEnabled(); this.window.navigator.mimeTypes; window.navigator.mimeTypes; this.navigator.mimeTypes; navigator.mimeTypes; this.window.navigator.plugins.filename; window.navigator.plugins.filename; this.navigator.plugins.filename; navigator.plugins.filename; this.window.navigator.plugins.name; window.navigator.plugins.name; this.navigator.plugins.name; navigator.plugins.name; this.window.navigator.plugins; window.navigator.plugins; this.navigator.plugins; navigator.plugins; this.window.navigator.plugins.description; window.navigator.plugins.description; this.navigator.plugins.description; navigator.plugins.description; this.window.navigator.plugins.length; window.navigator.plugins.length; this.navigator.plugins.length; navigator.plugins.lenght; this.window.localStorage; window.localStorage; this.localStorage; this.window.sessionStorage; window.sessionStorage; this.sessionStorage
Alto	this.window.navigator.getUserMedia(); window.navigator.getUserMedia(); this.navigator.getUserMedia(); navigator.getUserMedia(); canvas.getContext(); canvas.toDataURL(); canvas.getImageData(); this.window.history; window.history; this.history; this.window.history.length; window.history.length; this.history.length; this.window.history.back(); window.history.back(); this.history.back(); history.back(); this.window.history.forward(); window.history.forward(); this.history.forward(); history.forward(); this.window.history.go(); window.history.go(); this.history.go(); history.go(); MediaStream.getVideoTracks(); MediaStream.getAudioTracks(); navigator.mediaDevices.getUserMedia(); this.window.navigator.mediaDevices.getUserMedia(); window.navigator.mediaDevices.getUserMedia(); this.navigator.mediaDevices.getUserMedia(); navigator.mediaDevices.getUserMedia()

Figura A.1: Dicionário de BF 02 convertido. Fonte: Autor.

Tabela A.3: Dicionário de BF 02

Termos Investigados
this.window.navigator.userAgent; window.navigator.userAgent; this.navigator.userAgent; navigator.userAgent;
this.window.navigator.hardwareConcurrency; window.navigator.hardwareConcurrency; this.navigator.hardwareConcurrency;
navigator.hardwareConcurrency; this.window.navigator.bluetooth; window.navigator.bluetooth; this.navigator.bluetooth;
navigator.bluetooth; this.window.navigator.appCodename; window.navigator.appCodename; this.navigator.appCodename;
navigator.appCodename; this.window.navigator.appCodename; window.navigator.appCodename; this.navigator.appCodename;
navigator.appCodename; this.window.product; this.product; window.product; this.window.productSub; window.productSub; this.productSub;
this.window.navigator.vendor; window.navigator.vendor; this.navigator.vendor; navigator.vendor; this.window.navigator.appMinorversion;
window.navigator.appMinorversion; this.navigator.appMinorversion; navigator.appMinorversion; this.window.navigator.maxTouchPoints;
window.navigator.maxTouchPoints; this.navigator.maxTouchPoints; navigator.maxTouchPoints;
this.window.navigator.getInstalledRelatedApps(); window.navigator.getInstalledRelatedApps(); this.navigator.getInstalledRelatedApps();
navigator.getInstalledRelatedApps(); this.window.navigator.language; window.navigator.language; this.navigator.language;
navigator.language; this.window.navigator.browserLanguage; window.navigator.browserLanguage; this.navigator.browserLanguage;
navigator.browserLanguage; this.window.navigator.userLanguage; window.navigator.userLanguage; this.navigator.userLanguage;
navigator.userLanguage; this.window.navigator.languages; window.navigator.languages; this.navigator.languages;
navigator.languages; this.window.navigator.cpuClass; window.navigator.cpuClass; this.navigator.cpuClass;
navigator.cpuClass; this.window.navigator.oscpu; window.navigator.oscpu; this.navigator.oscpu;
navigator.oscpu; this.window.navigator.plataform; window.navigator.plataform; this.navigator.plataform;
this.navigator.plataform; this.window.navigator.geolocation; window.navigator.geolocation; this.navigator.geolocation;
navigator.geolocation; this.window.navigator.vendorSub; window.navigator.vendorSub; this.navigator.vendorSub;
navigator.vendorSub; this.navigator.online; naigator.online; this.window.navigator.appVersion;
window.navigator.appVersion; this.navigator.appVersion; navigator.appVersion; window.navigator.online;
this.window.navigator.appName; window.navigator.appName; this.navigator.appName; navigator.appName;
this.window.navigator.doNotTrack; window.navigator.doNotTrack; this.window.navigator.online; this.navigator.doNotTrack;
navigator.doNotTrack; this.window.screen.availHeight; window.screen.availHeight; this.screen.availHeight;
screen.availHeight; this.window.screen.availWidth; window.screen.availWidth; this.screen.availWidth;
screen.availWidth; this.window.screen.colorDepth; window.screen.colorDepth; screen.colorDepth;
this.window.screen.height; window.screen.height; this.screen.height; screen.height;
this.window.screen.pixelDepth; window.screen.pixelDepth; this.screen.pixelDepth; screen.pixelDepth;
this.window.screen.width; window.screen.width; this.screen.width; screen.width; this.window.matchMedia();
window.matchMedia(); this.matchMedia(); this.window.devicePixelRatio; window.devicePixelRatio;
this.devicePixelRatio; devicePixelRatio; this.window.inner.height; window.inner.height;
this.inner.height; inner.height; this.window.outer.width; window.outer.width;
this.outer.width; outer.width; this.window.outer.height; window.outer.height;
this.outer.height; outer.height; Date.getTimezoneOffset(); new Date();
this.window.document.referrer; window.document.referrer; this.document.referrer; document.referrer;
this.window.document.cookie; window.document.cookie; this.document.cookie; document.cookie;
this.window.document.domain; window.document.domain; this.document.domain; document.domain;
this.window.document.createElement(); window.document.createElement(); this.document.createElement(); document.createElement();
Modernizr.geolocation; Modernizr.video; this.window.navigator.cookieEnabled; window.navigator.cookieEnabled;
this.navigator.cookieEnabled; navigator.cookieEnabled; this.window.navigator.javaEnabled();
window.navigator.javaEnabled(); this.navigator.javaEnabled(); navigator.javaEnabled();
this.window.navigator.mimeTypes; window.navigator.mimeTypes; this.navigator.mimeTypes; navigator.mimeTypes;
this.window.navigator.plugins.filename; window.navigator.plugins.filename; this.navigator.plugins.filename; navigator.plugins.filename;
this.window.navigator.plugins.name; window.navigator.plugins.name; this.navigator.plugins.name; navigator.plugins.name;
this.window.navigator.plugins; window.navigator.plugins; this.navigator.plugins, navigator.plugins;
this.window.navigator.plugins.description; window.navigator.mediaDevices.getUserMedia(); this.navigator.mediaDevices.getUserMedia();
navigator.mediaDevices.getUserMedia() window.navigator.plugins.description; this.navigator.plugins.description;
navigator.plugins.description; this.window.navigator.plugins.length; window.navigator.plugins.length;
this.navigator.plugins.length; navigator.plugins.length; this.window.localStorage; window.localStorage;
this.localStorage; this.window.sessionStorage; window.sessionStorage; this.sessionStorage;
this.window.navigator.getUserMedia(); window.navigator.getUserMedia(); this.navigator.getUserMedia() navigator.getUserMedia();
canvas.getContext(); canvas.toDataURL(); canvas.getImageData(); this.window.history;
window.history; this.history; this.window.history.length; window.history.length;
this.history.length; this.window.history.back(); window.history.back(); this.history.back();
history.back(); this.window.history.forward(); window.history.forward(); this.history.forward();
history.forward(); this.window.history.go(); window.history.go(); this.history.go();
history.go(); MediaStream.getVideoTracks(); MediaStream.getAudioTracks(); this.window.navigator.mediaDevices.getUserMedia();

Tabela A.4: Sites e risco da base Top 50 Alexa (Ano 2021)

Ordem	Site	Risco	Ordem	Site	Risco
1	pandas.Tv	Nada encontrado	2	bongacams.com	Alto
3	intl.alipay.com	Médio	4	login.microsoftonline.com	Baixo
5	facebook.com	Alto	6	outlook.live.com	Alto
7	stackoverflow.com	Alto	8	weibo.com	Alto
9	vk.com	Alto	10	www.17ok.com	Baixo
11	twitter.com	Alto	12	www.360.cn	Alto
13	www.aliexpress.com	Alto	14	www.amazon.co.jp	Alto
15	www.adobe.com	Alto	16	www.amazon.com	Alto
17	www.amazon.in	Alto	18	www.baidu.com	Alto
19	www.aparat.com	Alto	20	www.bing.com	Baixo
21	www.csdn.net	Alto	22	www.google.com.hk	Alto
23	www.ebay.com	Alto	24	www.huanqiu.com	Alto
25	www.instagram.com	Baixo	26	www.google.com	Alto
27	www.jd.com	Alto	28	www.linkedin.com	Alto
29	www.naver.com	Alto	30	www.microsoft.com	Alto
31	www.netflix.com	Médio	32	www.office.com	Alto
33	www.okezone.com	Alto	34	www.qq.com	Alto
35	www.reddit.com	Baixo	36	www.shopify.com	Alto
37	www.sina.com.cn	Alto	38	www.sohu.com	Alto
39	www.tianya.cn	Alto	40	www.taobao.com	Alto
41	www.twitch.tv	Alto	42	www.wikipedia.org	Médio
43	www.tmall.com	Alto	44	www.yahoo.co.jp	Alto
45	www.yahoo.com	Alto	46	www.yy.com	Alto
47	www.zhanqi.tv	Médio	48	xinhuanet.com	Alto
49	zoom.us	Alto	50	www.youtube.com	Alto

Tabela A.5: Dependências analisadas e não analisadas da base Top 50 Alexa - dependências (Ano 2021)

Sites	Dependências	Analisadas	Não Analisadas
bongacams.com	201	201	0
intl.alipay.com	110	110	0
login.microsoftonline.com	107	102	5
outlook.live.com	122	118	4
stackoverflow.com	201	195	6
twitter.com	108	108	0
vk.com	116	113	3
weibo.com	193	193	0
www.17ok.com	203	203	0
www.360.cn	78	76	2
www.adobe.com	188	188	0
www.aliexpress.com	144	144	0
www.alipay.com	86	85	1
www.amazon.co.jp	203	201	2
www.amazon.com	201	0	201
www.amazon.in	201	0	201
www.baidu.com	102	97	5
www.bing.com	115	104	11
www.ebay.com	201	201	0
www.facebook.com	147	119	28
www.google.com	135	135	0
www.google.com.hk	117	111	6
www.huanqiu.com	196	193	3
www.instagram.com	118	87	31
www.linkedin.com	197	0	197
www.microsoft.com	200	0	200
www.naver.com	200	0	200
www.netflix.com	120	114	6
www.office.com	202	182	20
www.okezone.com	316	190	126
www.qq.com	201	0	201
www.reddit.com	199	0	199
www.shopify.com	165	129	36
www.sina.com	99	90	9
www.sohu.com	62	62	0
www.taobao.com	191	0	191
www.tianya.cn	131	0	131
www.tmall.com	60	55	5
www.twitch.tv	163	163	0
www.wikipedia.org	201	201	0
www.xinhuanet.com	19	195	2
www.yahoo.co.jp	176	151	25
www.yahoo.com	111	62	49
www.youtube.com	186	0	186
www.yy.com	31	29	2
www.zhanqi.tv	44	0	44
zoom.us	187	0	187
Total	**7232**	**4707**	**2525**