# Identificando Indicadores de Browser Fingerprinting em Páginas Web

**Geandro Farias de Matos**[1], **Eduardo L. Feitosa**[1]

[1]Instituto de Computação (IComp) - Universidade Federal do Amazonas (UFAM)
Manaus – AM – Brazil

geandro@ufam.edu.br, efeitosa@icomp.ufam.edu.br

***Abstract.*** *Browser Fingerprinting techniques are those employed to identify (or re-identify) a user or device and are considered a potential threat to users' privacy. In this context, this article proposes a way to detect fingerprinting calls to JavaScript information provider objects on web pages and measure the level of severity of the threat to the user.*

***Resumo.*** *Técnicas de Browser Fingerprinting são aquelas empregadas para identificar (ou reidentificar) um usuário ou um dispositivo e são consideradas uma ameaça potencial à privacidade dos usuários. Neste contexto, este artigo propõe uma forma para detectar chamadas fingerprinting aos objetos fornecedores de informação do JavaScript em páginas Web e mensurar o nível de severidade à ameaça ao usuário.*

## 1. Introdução

Já é notório que empresas, sites e serviços coletam, rastreiam e monitoram informações de seus usuários através de técnicas de *Browser Fingerprinting*, representando um perigo a privacidade e a segurança. De acorodo com [Mayer 2009, Nikiforakis et al. 2013, Khademi et al. 2015, Saraiva 2016], essas técnicas coletam dados sistematicamente do navegador ou 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 única capaz de correlacionar as atividades dos usuários ao navegar na Internet .

As principais técnicas de *Browser Fingerprinting* utilizam JavaScript, linguagem adotada como padrão nos navegadores modernos, para obter acesso a objetos internos do HTML DOM (*Document Object Model*) [Khademi et al. 2015] e conseguir informações relacionadas ao sistema operacional, ao hardware e ao próprio navegador. Assim, tais informações podem ser combinadas em um identificador único que será empregado para rastrear e monitorar atividades do usuário.

Embora a literatura enumere diferentes formas de detectar *Browser Fingerprinting*, trabalhos sobre a detecção dos objetos JavaScript que fornecem informações do navegador ou hardware são escassos. Um dos poucos trabalhos propôs o uso de *regex* para identificar e quantificar termos suspeitos de ligação com *Browser Fingerprinting* [Saraiva 2016]. Entretanto, o uso do *regex* não permite identificar se uma chamada de objeto realmente aconteceu, o que pode gerar falsos positivos.

Neste cenário, esta pesquisa em andamento propõe a identificação de *Browser Fingerprinting* através da observação e identificação das chamadas aos objetos JavaScript

fornecedores de informações, empregando análise de Árvore Sintática Abstrata (AST), a fim de melhorar a capacidade de detecção e fornecer um arcabouço para futuras contramedidas. Especificamente, pretende-se: (i) utilizar e atualizar um dicionário de termos proposto por [Saraiva 2016], contendo objetos e propriedades da linguagem JavaScript, comumente utilizados em *Browser Fingerprinting*; e (ii) desenvolver um identificador sintático, baseado em AST, para identificar, no código fonte dos scripts das páginas coletadas, as chamadas aos objetos da linguagem JavaScript.

## 2. AST e Análise de código JavaScript

Árvore Sintática Abstrata (AST) é uma estrutura de dados, baseada em árvore, que representa o código fonte a ser inspecionado de forma programática. Os autores em [Damasceno et al. 2018] definem uma AST como uma representação intermediária do código fonte que favorece análises insensíveis ao fluxo, ignorando a ordem em que as instruções são executadas e possibilitando exploração de diversos caminhos a serem verificados.

A Figura 1 ilustra uma AST com chamdas a objetos ligados a *fingerprint*.

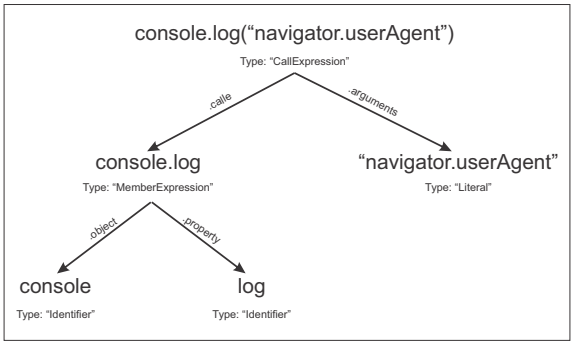


**Figura 1. Exemplo de AST avaliando um elemento de *fingerprinting***

Pode-se observar na Figura que **console.log("navigator.userAgent")** é o nó raiz da AST. Ele possui uma chamada de expressão com duas arestas que apontam para outros nós. A aresta $.calle$ aponta para o objeto **console.log**, um nó do tipo $MemberExpression$, que aponta para o objeto **console** e a propriedade **log**, ambos do tipo $Identifier$. A aresta $.arguments$ aponta para o argumento dessa chamada. Nota-se, com essa análise simplificada, o emprego de AST para detecção de *Browser Fingerprinting*, uma vez que **navigator.userAgent** - termo ligado a *fingerprint* - nessa condição, não pode ser classificado como um, pois a chamada **console.log** apenas imprimirá a palavra **navigator.userAgent**.

É importante destacar que ao utilizar uma AST, pode-se obter um relatório completo de análise contendo a estrutura do código de forma hierárquica e rótulos que identificam essa estrutura. É por este motivo que este trabalho utiliza AST para extrair do código das chamadas aos objetos fornecedores de informações.

## 3. WBF Analyze

Esta seção descreve a solução proposta, chamada WBF (*Web Browser Fingerprinting*) Analyse. A Figura 2 ilustra uma visão arquitetural da WBF Analyze.

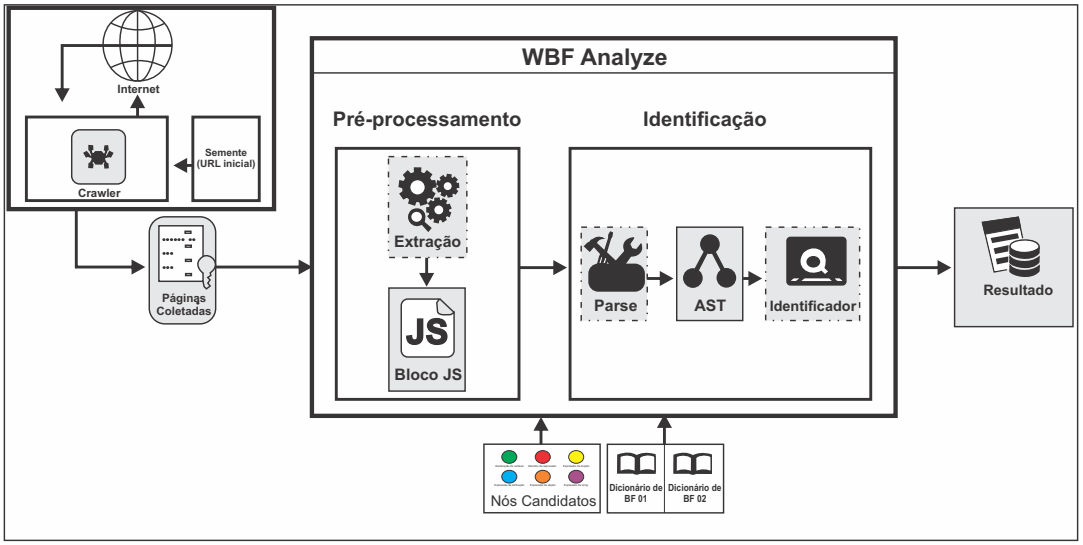


**Figura 2. Arquitetura da WBF Analyze**

Para funcionar, a WBF Analyze precisa dos códigos JavaScript das URLs que serão analisadas. Esse processo pode ser feito por um *Web Crawling*. Assume-se que todas as URLs são verificadas, quanto a sua validade (por exemplo, se estão mal-formadas, duplicadas ou com ausência de objetos JavaScript).

A primeira etapa, de fato, da WBF Analyze é o **pré-processamento** do código JavaScript, que basicamente consiste na limpeza do código, verificando e resolvendo casos de ofuscação do tipo *enconding* [Xu et al. 2012], muito empregada para atividades maliciosas e ataques. O resultado final é um bloco JavaScript formado apenas por códigos in-line e externos da URL analisada.

Na etapa seguinte, **identificação**, o bloco JavaScript é transformado, por um *parser*, em uma AST para análise. Qualquer *parser* que reconheça os tokens JavaScript pode ser usado. Após a conversão, o compoenente identificador recebe a AST como entrada e aplica três regras para detecção das chamadas de *Browser Fingerprinting*, conforme a Figura 3. As subseções seguintes explicam o funcionamento dessas regras.

### 3.1. Identificação - Regra 1: Extração

Na Figura 3, a *regra 1* recebe, além da AST, nós candidatos - expressões e operadores de JavaScript (por exemplo, *Assignment expression*, *Variable declaration*, entre outros) - como entrada e tentar extraí-los da AST. A AST será totalmente percorrida para extrair esses nós candidatos e essa varredura é feita de forma insensível ao fluxo para garantir que todos os caminhos possíveis da AST sejam examinados. Os resultados desta extração direcionada é um escopo reduzido, gerado para evitar que se façam manipulações que envolvam a edição da AST.

A Figura 4 ilustra o processo ocorrido na *regra 1*, onde foram encontrados os nós *VariableDeclarator* e *MemberExpression* na estrutura da AST e a extração dos termos *b*, *navigator* e *appName* foi realizada para o escopo reduzido.

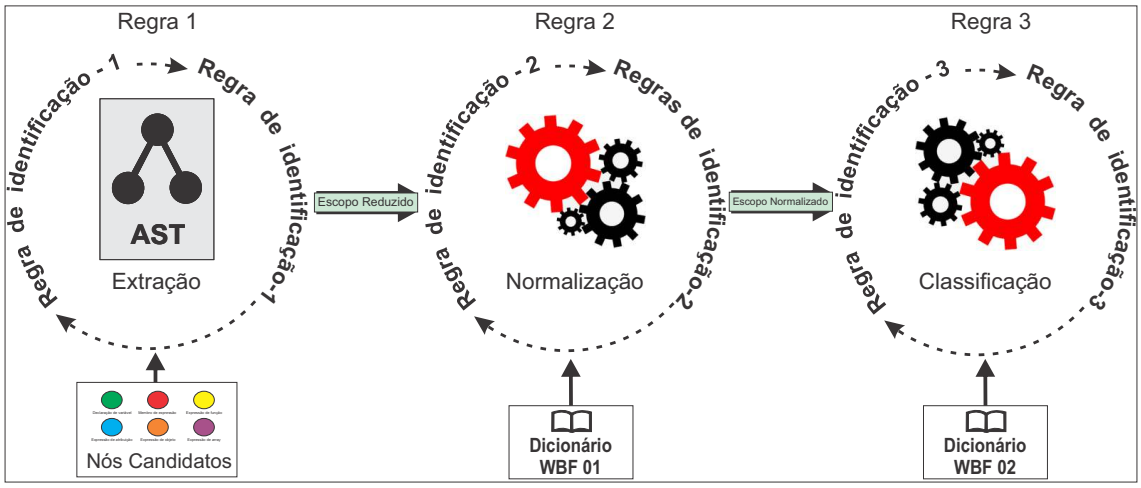


**Figura 3. Regras de Identificação**

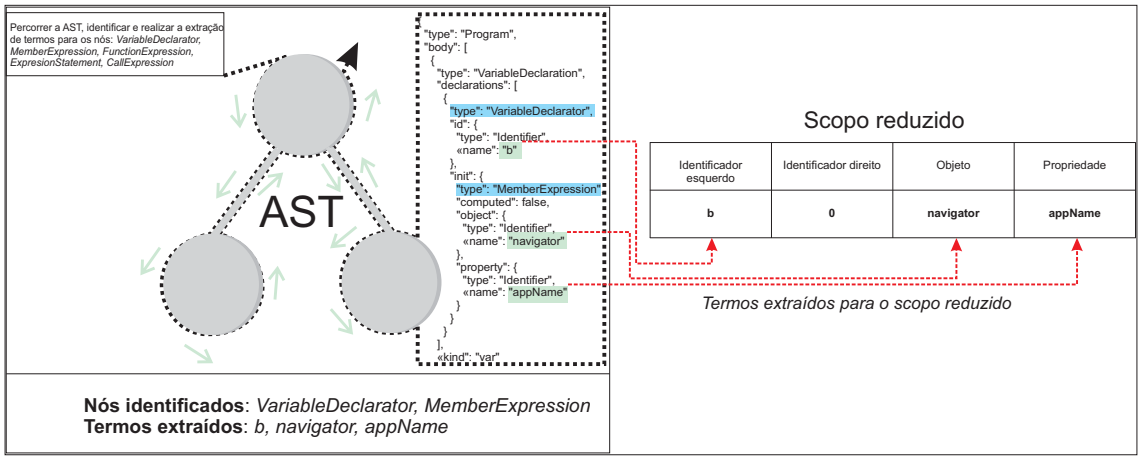


**Figura 4. Regra 1.**

### 3.2. Identificação - Regra 2: Normalização

A *regra 2* é responsável por normalizar a relação entre objetos e propriedades, a fim de produzir as chamadas que dão origem ao escopo normalizado. Essa regra recebe como entrada um escopo reduzido e aplica três processos (verificação, comparação e correspondência) com a finalidade de produzir um escopo normalizado, no formato das chamadas de objetos, como saída. Esses processos executam ações para transformar: (i) *objeto* + *propriedade*, em *objeto.propriedade*; (ii) *objeto* + *método*, em *objeto.método*; e (iii) casos especiais como *a.appName*, em *navigator.appName*.

O *processo de verificação* é responsável por: (i) verificar a existência de valores nos identificadores, nos objetos e propriedades do escopo reduzido; e (ii) executar ações com base nas verificações. Para tanto, faz uso de dois dicionários de *Browser Fingerprinting*, chamados BF 01 e BF 02. No primeiro, os objetos e propriedades ligados a *fingerprint* estão representados similarmento como no escopo reduzido. Já o segundo, as representações estão como chamadas completas. Ambos foram gerados de acordo com o trabalho de [Saraiva 2016]. A verificação também faz uso de: (i) um Escopo de Correspondência Temporária (ECT) para salvar os valores de objetos e propriedades que são

utilizados na verificação; (ii) um dicionário de identificadores, onde objetos que possuem valores são armazenados como par chave-valor; e (iii) um processo de produção, uma sub-rotina que produz uma chamada de objeto.

O *processo de comparação* é responsável por: (i) aplicar comparações baseada no dicionário BF 01; e (ii) executar ações com base nas comparações. Ele avalia se os valores apontados pelas chaves do escopo reduzido são iguais aos valores apontados pelas chaves do dicionário BF 01 ou do dicionário de identificadores. A comparação é um processo chamado pela verificação. A Figura 5 ilustra o processo de comparação.

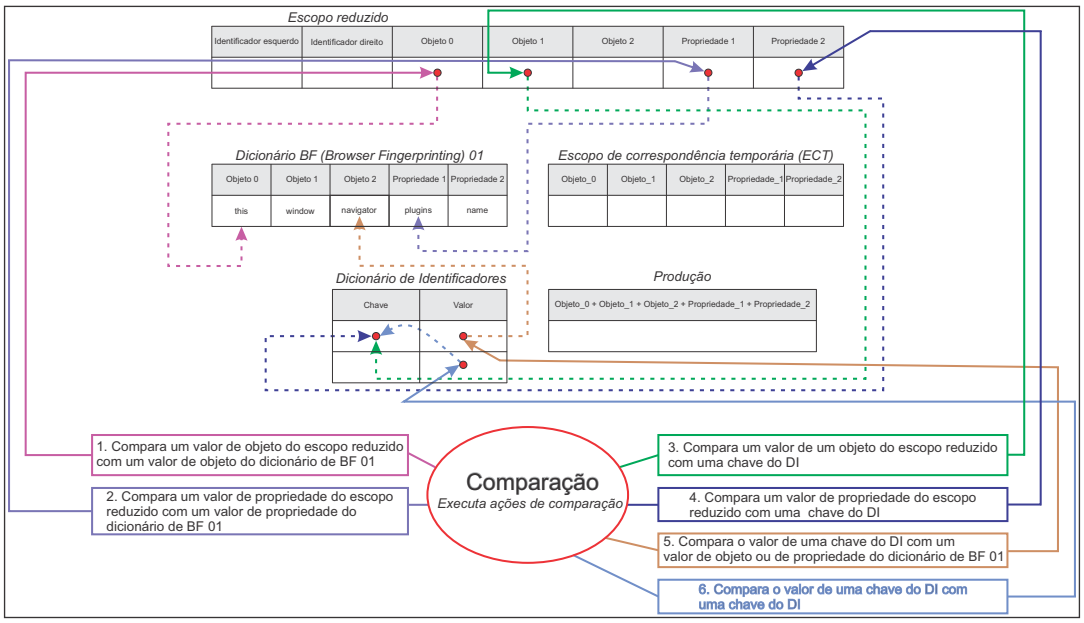


**Figura 5. processos da comparação.**

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

A comparação avalia casos do tipo objeto inteiro, como em *navigator.plugins.name*, orientando o processo de correspondência a salvar os valores que satisfaçam as comparações diretamente no *ECT*. Quando a comparação encontra um caso especial do tipo *b.appName*, onde tem-se uma ofuscação de string de palavra-chave, ela deve descobrir quem é **"b"** ou se esse **"b"** faz parte de uma relação estabelecida em uma cadeia de atribuições.

Por fim, o *processo de correspondência* é responsável por salvar no ECT os objetos e propriedades que foram aprovados pela comparação. Ele: (i) salva a correspondência de um valor de objeto do escopo reduzido com um valor de objeto dodicionário de BF 01 no ECT; (ii) salva a correspondência de um valor de propriedade do escopo reduzido com um valor de propriedade do dicionário BF 01 no ECT; e (iii) salva a correspondência

do valor de uma chave do dicionário de identificadores com um valor de objeto ou de propriedade do escopo reduzido.

### 3.3. Identificação - Regra 3: Classificação

A *regra 3* recebe como entrada o escopo normalizado e o dicionário BF 02 para realizar uma última avaliação comparativa antes de se ter certeza de que os falsos positivos foram eliminados e apenas chamadas a objetos em conformidade foram produzidas. Após essa avaliação, a classificação das chamadas é realizada segundo o método de [Saraiva 2016].

## 4. Atividades Realizadas, em Andamento e Futuras

Esta pesquisa já realizou: (i) a revisão sistemática completa, cujos resultados não poderam ser explicitados aqui, mas constam no texto de qualificação de mestrado do autor; (ii) a definição do conjunto de objetos fornecedores de informação do JavaScript (alguns mencionados na seção 3.1) relacionados a *Browser Fingerprinting*; (iii) implementação e teste do *parser* gerador de AST; (iv) atualização do dicionário de *Browser Fingerprinting* de [Saraiva 2016] para geração dos dicionários BF 01 e BF 02; e (v) implementação do pré-processamento e identificação e suas estruturas (dicionários, escopos e estruturas).

Atualmente, o trabalho está na fase de implementação das estruturas faltantes, como alguns elementos da *regra 2* e *regra 3*. Como atividades futuras, destacam-se: (i) a catalogação e montagem da base de dados para avaliar a solução; (ii) testes e avaliações da solução.

## 5. Agradecimentos

Esta pesquisa, conforme previsto no Art. 48 do decreto nº 6.008/2006, foi parcialmente financiada pela Samsung Electrônica da Amazônia Ltda, nos termos da Lei Federal nº 8.387/1991, através de convênio nº 003/2019, firmado com o ICOMP/UFAM.

## Referências

Damasceno, A., Rocha, T., and Souto, E. (2018). Taintjsec: Um método de análise estática de marcação em código javascript para detecção de vazamento de dados sensíveis. In *Anais Principais do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 196–209, Porto Alegre, RS, Brasil. SBC.

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

Mayer, J. R. (2009). Any person... a pamphleteer": Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, page 85.

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

Saraiva, A. R. e Feitosa, E. L. (2016). Determinando o risco de fingerprinting em páginas web. *Dissertação de Mestrado, Universidade Federal do Amazonas*, page 94.

Xu, W., Zhang, F., and Zhu, S. (2012). The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 9–16.