Faculdade de Engenharia da Universidade do Porto

# You can hide but you can't run: browser extensions fingerprinting

Lucas Vieira Casalderrey Vilard Stein

Final Version

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática e Computação

Supervisor: Nuno Flores

Second Supervisor: Susana Lima

October 31, 2022

# You can hide but you can’t run: browser extensions fingerprinting

**Lucas Vieira Casalderrey Vilard Stein**

Mestrado em Engenharia Informática e Computação

October 31, 2022

# Abstract

The most well-known and popular web browsers, such as Google Chrome, allow users to improve their browsing experience with third-party extensions. Extensions are small software modules that run in the browser and enable the modification of the user interface, ad blocking, cookie management, among other uses. These extensions often run with high privileges, which if misused can compromise the application's integrity. Furthermore, there are several extensions that can harm a website, either by blocking advertisements - such as *AdBlock* - or suggesting competitor applications that offer better deals - such as *Shoptimate*. Therefore, it is increasingly relevant for a website to know what extensions the user has installed, either to detect and mitigate possible threats or to prevent *Customer Journey Hijacking* attempts. Extension detection can also enable new ways to profile the user base. The detection of certain extensions can reveal specific information about a user, such as religion or political affiliation. For these reasons, it is also in the users' interest that these vulnerabilities are measured and that it is studied how extensions can be detected and fingerprinted to prevent any malicious behavior.

In this work we aim to study the most common and better methods of extension detection and fingerprinting, and their effectiveness in the number of extensions they can identify. These methodologies vary and take advantage of different extension properties to identify them. We consider two types of extensions: non-stealthy and stealthy. The first ones are extensions that manipulate DOM elements, and therefore, can be easily detected. The latter are extensions that do not manipulate the DOM and require more complex detection methods, for example through their Web Accessible Resources or based on the type of CSS they inject into the websites. To implement on-site extension detection, it is not only important to have a high detection rate, but also to implement detection in a way that it doesn't harm the user experience. This work also conducts an analysis on the performance cost of each implemented method to understand the viability of each one. A detection script, that combines three methods and can be injected into any website, is developed based of a dataset of 1,000 of the most popular extensions from the Google Chrome web store. Detection using Web Accessible Resources proves to be the better method in both detection rate and performance, while detection through CSS injection demonstrates to be impracticable without harming the website's performance.

**Keywords**: Extensions Fingerprinting, Browser Security, Extensions Security, User Privacy

# Resumo

Alguns dos browsers mais populares, como por exemplo o Google Chrome, permitem aos utilizadores melhorar a sua experiência de navegação através do uso de extensões. Extensões são pequenos módulos de software que correm no browser e permitem a modificação da user interface, bloquear anúncios, gestão de cookies, entre outras aplicações. Estas extensões geralmente são executadas com privilégios que, se usados incorretamente, podem comprometer a integridade do website. Além disso, existem várias extensões que podem prejudicar um website, seja através do bloqueio de anúncios - como o *AdBlock* - ou através da sugestão de outros websites concorrentes que oferecem melhores ofertas - como o *Shoptimate*. Por esta razão é cada vez mais relevante para um website saber quais as extensões o utilizador tem instaladas, seja para detectar e mitigar possíveis ameaças ou para prevenir tentativas de *Customer Journey Hijacking*. A detecção de extensões pode também permitir novas formas de extrair informação do perfil dos utilizadores. A detecção de certas extensões pode revelar informações específicas sobre um utilizador, como religião ou afiliação política. Desta forma, é também do interesse dos utilizadores que estas vulnerabilidades sejam medidas e que seja estudado como detetar e identificar extensões para prevenir qualquer comportamento malicioso.

Neste trabalho, pretende-se estudar os melhores e mais comuns métodos de detecção e *fingerprinting* de extensões, e a sua eficácia no número de extensões que consegue identificar. Essas metodologias variam e aproveitam-se de diferentes propriedades das extensões para identificá-las. Consideramos dois tipos de extensões: *non-stealthy* e *stealthy*. As primeiras são extensões que manipulam elementos DOM e, portanto, podem ser facilmente detectadas. Estas últimas são extensões que não manipulam o DOM e requerem métodos de detecção mais complexos, como por exemplo, através dos *Web Accessible Resources* ou do tipo de CSS que injetam nos websites. Para implementar detecção de extensões é importante não só ter uma taxa de detecção elevada, como também fazer a detecção de forma que não prejudique a experiência do utilizador. Neste trabalho é realizado também uma análise ao custo de desempenho de cada método implementado, para se tirar conclusões sobre a viabilidade de cada um. Um *script* de detecção, que combina três métodos e pode ser injetado em qualquer website, é desenvolvido com base num *dataset* de 1,000 extensões populares do Google Chrome. A detecção usando *Web Accessible Resources* mostra ser o melhor método tanto na taxa de detecção como no desempenho, enquanto que a detecção por *CSS injection* demonstra ser impraticável sem prejudicar o desempenho do site.

**Keywords**: Fingerprinting de Extensões, Segurança dos Browsers, Segurança das Extensões, Privacidade do Utilizador

# Acknowledgements

I want to give a special thanks to everyone who took part not only in this process of writing my dissertation but also throughout all my life.

I thank my supervisors, Nuno Flores and Susana Lima, for all the support given over the months, for always guiding me in the right direction and for always demanding the best from me. To the company JScrambler for always providing me with all their support and resources, and assuring I was given the best possible conditions to succeed.

I am incredibly grateful to my mother who sculpted me into the man I am today, and for being my inspiration every single day. I would also like to thank all my friends for their support, especially Guilherme Antunes for the good company during harder times, Tiago Teixeira for his moral support and positivity, and Inês Fernandes for keeping me focused on the goal, always with the right mindset and giving me the right tools to be able to overcome the challenges that have arisen. Without them, it certainly would not have been possible.

Lucas Stein

*“ If I have seen further*
*it is by standing on the shoulders of giants. ”*

Isaac Newton

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| AST | Abstract Syntax Tree |
| CSS | Cascading Style Sheets |
| DOM | Document Object Model |
| HTML | Hyper Text Markup Language |
| UI | User Interface |
| URI | Uniform Resource Identifier |
| WAR | Web Acessible Resource |

# Chapter 1

# Introduction

This chapter introduces the problem under study and the objectives and contributions of this dissertation. Section 1.1 makes a contextualization of the domain on which this work is carried out and briefly mentions some of the concerns. Then, in Section 1.2, an explanation is given on why there is a need for research to be done in this field and the relevance of the work. In Section 1.3 the general and specific objectives of the dissertation are described, and finally, in Section 1.4, the structure of this document is presented.

## 1.1 Context

It is estimated that around 63.2% of the world population, that is 4.93 billion people, have access and use the internet frequently [31]. In North America and Europe, nearly 90% of the population accesses the internet daily [31], and it's undeniable that these numbers will tend to grow as society moves to a digital world with each passing year. A large percentage of these accesses are made through Web Browsers such as Google Chrome or Mozilla Firefox [31]. Browsers nowadays are more than simple vehicles for accessing the internet. They are complex platforms with a rich set of features and constitute an important role in the entire internet ecosystem. Throughout the years, some security concerns have been raised on how browsers and how websites collect and handle user information [1] [2] [12] [22]. This information enables third parties to make a digital profile of the user [17], allowing them to target the user base with ads custom-made specifically for them. Most users are uncomfortable with this and claim this constitutes a breach of their privacy [18] [30].

Furthermore, one way browsers have allowed users to improve and personalize their browsing experience is through the use of extensions. Browser extensions are small programs the user can install and run in the browser. Research has shown that extensions can be taken advantage of to track users online and create their digital profiles [17]. Even though browsers do not provide a JavaScript call that allows web pages to know which extensions the user has installed, they can be detected through various indirect methods as described on Chapter 3. This introduces a new security threat because it enables the collection of even more data from users. However, detecting what extensions can also be a way to prevent malicious or unwanted behavior. There are extensions, such as *AdBlock*, that block any advertisement made on the web page, which can directly harm a website. For this reason, there has been a lot of research done in this field, in an attempt to understand how secure browsers and extensions are, and how they can be used to fingerprint users.

## 1.2 Motivation

There has been a growing interest in protecting users' privacy. It has been considered an essential topic in the software development field [18] [22] [30]. As web extensions have been more and more present in everyday user internet activities, it's important to measure the threat of using them and how they leave users exposed to potential malicious attacks.

Ideally, from the user's perspective, these extensions should be invisible to the websites, impossible to be detected so that they can't be used to profile the user base. For this reason, fingerprinting through extensions should be explored to understand extensions and browser vulnerabilities. Detecting these vulnerabilities may lead to future work proposing new countermeasures that improve browser security and protection of users' privacy. From the websites' perspective, there is great value in knowing what extensions the user has installed for several reasons. By the end of the 2020 year, it was estimated that over 40% of internet users used some kind of ad blocking [38]. *AdBlock* and *AdBlock Plus* are also the most downloaded extensions from the Google Chrome Store ever with over 500 million downloads. This can present a real threat to many websites' form of income. There are also popular extensions like *Shoptimate* that suggest better offers and prices to users trying to buy something from a website. Websites that are able to detect the presence of such extensions could implement measures to mitigate the harm these types of extensions may bring. This creates a scenario where there is a clash of security goals, and as it was previously stated by another researcher [34] this work does not intend to assume the interest of one side over the other. The primary motivation is to analyze how extensions can be detected on the most popular web browser.

Throughout the years, there have been presented many methods to identify extensions as described on Chapter 3. Most of them were tested and validated on Google Chrome and Mozilla Firefox. Extensions can vary significantly in implementation. Some manipulate DOM elements and can be more easily detected - non-stealthy extensions - others only inject CSS to the page or make no changes at all - stealthy extensions. The latter need more complex methods to be detected, for example, it has been shown that it is possible to take advantage of their WARs to detect their presence [34]. The problem is that each method targets a specific characteristic and if a particular extension is not implemented using that characteristic, it cannot be detected by that method. Thus, there needs to be a mixed approach that implements various methods, targeting various characteristics in such a way where the number of extensions able to be detected is maximized.

Furthermore, the solution needs to work efficiently to be run on a real website with other common purposes. If one website wants to implement extension detection for whatever reason, the implementation cannot harm the main purposes of the website, in terms of performance. There are hundreds of thousands of extensions available for download. In an ideal world, where the implementation of massive on-site detection across all these extensions is applied, it should be possible to try to detect them all without performance issues. Nonetheless, the solutions that exist now are very taxing performance wise, and can't be implemented on a normal website without harming the overall user experience. Because of this, it's important not only to continue research on how to maximize the detection rate of the methods, but also find a way to be efficient enough to fulfill the previously mentioned prerequisites.

## 1.3 Objectives

This dissertation's main objectives are the following:

1. Develop and document the implementation of different website extension detection methods;
2. Implement a script that combines multiple website extension detection methods;
3. Analyze the detection effectiveness and performance cost for each implemented method.

The main goal is to develop a script that can be injected into a website and be capable of identifying the extensions a user has installed during run time. The detection mechanisms will be implemented using a combined and mixed approach based on several previously proposed approaches. As mentioned previously, some extensions are more stealthy than others, as some change the content of the web page and others do not, meaning it is necessary to implement different methods to detect different extensions. This work aims to accurately identify multiple extensions, stealthy or not, for the Google Chrome browser. The decision to validate using this browser is explained in Chapter 3. In Chapter 6 every single implementation is detailed and every step of development is described in a way where every implementation can be easily reproducible. This was made in an effort to streamline each implementation in hopes that they can be improved in the future.

Furthermore, this work aims to make an analysis on the performance cost for each individual implementation, so that conclusions can be taken on the viability of these methods to implement massive on-site extension detection.

## 1.4 Document Structure

This document is structured in the following chapters: Chapter 1 gives an introduction to the problem under study and the main objectives and motivations for this dissertation work, Chapter 2 introduces the concept of extensions. It describes what they are, where they are applied, how they work, where they fit into the browsers' architecture, and the differences between extensions from different browsers. Chapter 3 presents the current state-of-the-art techniques to detect extensions. It enumerates some browser and extensions vulnerabilities and how they can be used to profile users. It details all the major methods to detect extensions and provides one implementation example for each one of them. Chapter 4 provides the formal declaration of the problem this work is trying to solve, the solution this work proposes and the methodology to validate it. Chapter 5 describes the development process of the work done. It documents every step taken during the implementation process for each one of the methods. Chapter 6 makes an analysis on the results of the implementations. It analyses the detection rate and performance viability of each method. Finally, Chapter 8 concludes the document by enumerating some points that can be taken away from this work and what can be done in the future to further develop it.

# Chapter 2

# Web Browser Extensions

This chapter introduces the most important concepts and some complementary themes to the main topic of extensions. Section 2.1 explains what browser extensions are. In Section 2.2 the architecture of an extension is described and where they are inserted in the browser. Then, in Section 2.3, the environment where the extensions are used (browsers) is explained, including a mention of the main browser that will be studied in this dissertation, Google Chrome. Finally, in Section 2.4 a summary of what is discussed in the chapter is made.

## 2.1 What are extensions?

Browser extensions are small software modules installed into the user's browser that add new features and functionality, "extending" and enhancing the browser's capabilities. Extensions can also be used to remove unwanted features such as advertisements or pop-ups. They are essentially a set of HTML, CSS and JavaScript files that implement the extension functionalities. Although there are some cases in which the browser companies develop their own extensions, for the most part, extensions are developed and maintained by third-parties. They are usually completely free but each browser contains its own set of available extensions for download. Microsoft's Internet Explorer, or Edge [23], calls them add-ons and separates them into five different categories such as toolbars, search providers, accelerators, tracking protection and extensions. Google Chrome [10] simply calls them extensions and separates them into themes and apps. Mozilla Firefox [24] makes no categorization and calls them extensions or plug-ins.

Web browsers process a ton of information. The simple task of loading a web page requires an exchange of information. Extensions are basically specialized agents in the middle, running in the browser, that have access to the flow of information processed by the browser, and can manipulate or use this information to create or deny some features and functionality for the user. This makes extensions far more powerful than some users realize [18]. Browsers can see and track everything a user does on the internet such as back account information or social media passwords. Extensions can potentially access this information. Even if the extension does not have any malicious intent, just by being able to access potentially sensitive information means it can be considered a security hazard for the user.

Some examples of the most downloaded extensions from the Google Chrome store are *Ad-Block*, used to eliminate advertisements from web pages including pop-ups; *LastPass* is a password manager; *Evernote Web Clipper* lets users save web pages, articles, PDFs and other things they find interesting to their Evernote account.

## 2.2 Extension Architecture

On the Google Chrome browser, an extension architecture is divided into three parts: UI, background page and content scripts. The user interacts with the extension through the UI, such as the settings page, where he can activate or deactivate it. The background page is the main component of an extension and is a implicit page that contains the background scripts. Finally, content scripts are the scripts injected by the extension that run alongside web applications. These scripts are the ones that can manipulate the content of the web page.

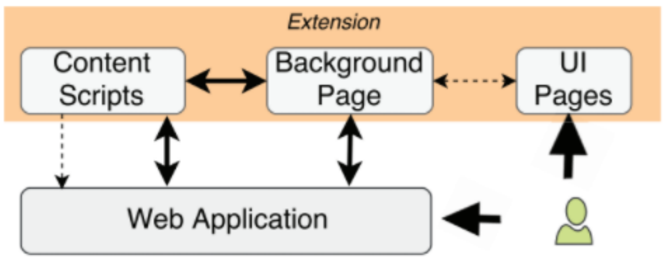


Figure 2.1: Extension architecture
*Note:* Reprinted from [35]

## 2.3 Application

Throughout software development there are a lot of programs that are classified as extensions. This work focuses on browser extensions, meaning it is important to understand the system on which this type of extensions exist and how it works.

### 2.3.1 Google Chrome and other web browsers

As mentioned on Section 2.1, browser extensions are small programs that run in the browser. A browser is a software application used to access content on the *World Wide Web*. It fetches information from the web, which is transferred using the *Hypertext Transfer Protocol*. After retrieving this information, the rendering engine starts processing this data and reads the code (written in *HTML*) to create the text and images users see on the screen. Examples of popular web browsers are Google Chrome, Mozilla Firefox, Microsoft Edge and Safari by Apple. Chrome is by far the most used web browser and contains the largest browser extension market [8].

Google Chrome is a web browser released by Google in 2008. It was built using free software components from Apple's WebKit, a rendering web engine, and Mozilla Firefox. Chrome was developed using a multi-process architecture where a single browser kernel process runs in privileged mode to access system resources, on the behalf of multiple rendering processes. Each rendering process is a web page, that is, a browser tab. A rendering process cannot directly access the system or network, it can only make requests to the main browser process. Regarding extensions specifically, the Chrome system was designed on the assumption that extensions are not malicious but can be buggy (benign-but-buggy assumption) [14]. Therefore, the extensions were integrated into Chrome with an architecture that aims to protect the extensions from possible attacks from websites that take advantage of the extensions' bugs. Since the extension code runs in the rendering process that communicates with the main browser process, the browser has to be protected from extensions attacks to not cause potential damage to the system. Browser and extension security is described more in detail on Section 2.3.2.

### 2.3.2 Browser and Extension Security

When talking about browsers and the security of using extensions, it's important to consider two different perspectives:

1. First is whether a website can potentially use the extensions a user has installed to carry out an attack;
2. Second is whether malicious extensions can be implemented, that take advantage of the browser to attack its users.

The result of this dissertation work will mainly focus on the first case, however both are important. In order to prevent extensions from being a vehicle for possible attacks, browsers do not offer any direct way to check which extensions a user has installed. However, as it was already mentioned and will be explained in Chapter 3, there are multiple indirect ways that have been proposed over the years that allow this to be achieved. This presents a huge security and privacy risk for the user [1] [2] [12] [22].

It has already been shown that a high percentage of internet users can be uniquely identified by the extensions they use [13]. In addition, studies have also proven that attackers can take advantage of some bugs that extensions may have to extract users' private information [12] [14] [26]. However, browsers like Google Chrome implement extension architectures that have the exact purpose of preventing attacks of this type. Chrome's extension architecture follows three security principles: least privilege, privilege separation and isolation [22]. Chrome separates privileges for different components of an extension, with the most privileged component not being able to interact directly with the webpage (thus being isolated). On the other hand, the component that can interact with the page, in this case the content script, has to declare in the *manifest.json* file all the permissions it needs to execute properly. This way, the component that is potentially vulnerable to malicious websites is the one with the least permissions granted.

Beyond this protection, Chrome's extension system does not offer users any protection against malicious extensions. Studies have already been carried out showing that there is a section of Chrome extensions with the purpose of attacking users [27] [29] [32]. In an analysis carried out on the Opera and Chrome browsers between 2016 and 2018 [18], researchers found that more than 3000 extensions were leaking sensitive information from their users. The top 10 extensions on this list were used by more than 60 million users. With this, it can be concluded that the use of extensions is a huge security risk and it is necessary to have a greater focus on this topic, both on the part of users and on the part of those who implement browser systems.

## 2.4 Summary

A browser extension is a small software module with the purpose of extending the browser features and personalizing the user's experience. All the most popular web browsers allow the use of extensions, which are in most cases, developed by third parties. Although the architecture of browsers like Google Chrome was developed to protect extensions from malicious websites, almost no security is guaranteed to users against the extensions themselves. Over the years, several studies have been carried out that prove that extensions pose a security risk and infringe on user's privacy. However, most users are unaware of the risk associated with the use of extensions.

# Chapter 3

# Extension Detection

This chapter describes the current state of the art, mainly focusing on the most popular extension detection techniques. Section 3.1 introduces the concept of extension fingerprinting, the two categories into which fingerprinting techniques are divided, what is the difference between stealthy extensions and non-stealthy extensions and why there is a need to use different techniques to detect extensions. In sections 3.2, 3.3 and 3.4, the three most effective extension fingerprinting techniques are described. One example of implementation for each is given and the results obtained during the validation process. Some limitations and what can be done to improve each technique is also mentioned. Then, in Section 3.5, other detection techniques are mentioned and finally, in Section 3.6, a summary of the conclusions that can be drawn from the state of the art research is made.

## 3.1 Extension Fingerprinting

Extension fingerprinting is the process of detecting and identifying the extensions that a user has installed on their browser. This detection is not a simple task, because browsers do not offer a direct way for the website to check which extensions are installed. However, over the years, several indirect methods have been proposed that are able to identify extensions [16] [17] [21] [34] [36] [40]. There are no flawless methods, that is, methods that can detect 100% of the available extensions. This is due to the fact that each technique takes advantage of different browser or extension characteristics, whether behavioral or static, to make the detection. As extensions vary in terms of behavior and implementation, some using certain features that others don't, detection methods never had a perfect success rate. Figure 3.1 shows how many extensions can four different techniques detect, and the overlap between them.

Detection techniques are usually divided into two different categories: behavioral detection and non-behavioral detection [34]. The first type of detection involves techniques that identify extensions through their behavior. For example, the first technique that's described in Section 3.2

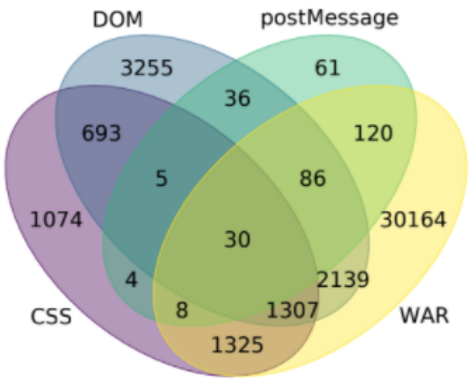


Figure 3.1: Number of extensions four different fingerprinting techniques can detect
*Note:* Reprinted from [21]

is a representative example of this. Through the changes that an extension can make to a web page, it is possible to detect it [37]. This type of technique requires extensions to make detectable changes in order to be identified. A disadvantage of this category is that these methods are prone to returning false positives because there could be multiple extensions with similar behaviors. The second category, non-behavioral detection, is related to static detection of extensions, with no need for them to have some kind of behavior to be identified [34]. An example of this type of detection is detection through WARs which is discussed in Section 3.3.

As far as extension fingerprinting is concerned, in this dissertation extensions are divided into two different categories: stealthy and non-stealthy extensions. Non-stealthy extensions are extensions that are easily detected as they do not make any effort to hide their behavior. Usually these are extensions that make changes to the UI of a website, which can be detected through behavioral fingerprinting techniques. Stealthy extensions are extensions that usually run in the background, often having no behavior directly perceptible to the user or the website. For this reason, these types of extensions need more complex detection methods and as they often cannot be identified through their behavior, requiring the use of non-behavioral techniques.

## 3.2 Detection through DOM manipulation

Detection through DOM manipulation is a type of behavioral detection. This technique takes advantage of the fact that some extensions exhibit a behavior that is observable on web pages. If the behavior of an extension towards a page is unique enough and separable from the behavior of other extensions (such as the elements inserted by the extension being unique), then the extension can be easily identified. The problem is that there is a lot of overlap between different extensions. This makes the behavioral analysis more complex and hard to implement, because it has to take into account several aspects. The main idea of an implementation of this method, and perhaps the most important for the success of the implementation, is to create behavioral signatures for each behavior. The more different behaviors can be differentiated and have unique signatures, the greater the number of extensions that can be identified. This process is also important to countermeasure potential false positives. One type of parameterization that can be included in signatures, for example, is the type of changes that a certain behavior has made to the DOM. These changes can be adding new elements to the tree, removing elements that already exist in the tree or editing the parameters of an element in the tree. A base implementation of this method creates test pages where each extension individually is ran to verify the behaviors it has. Behavioral signatures are then created for each extension. With this information, on the page where the fingerprinting is implemented, the behavior of the extensions that are present is monitored, also creating the behavioral signatures with exactly the same process. By matching these signatures with those that had already been created, we are able to identify which extensions the user has installed.

An example of an implementation that uses this fingerprinting technique is the *Carnus* system [17]. *Carnus* uses several detection approaches, including detection through DOM manipulation, detection through WARs, which is explained in Section 3.3, detection through inter-communication and detection through intra-communication. Focusing only on the first type of detection, the first step in implementing this system was to create test pages, which they called *honeypages*, where all extensions were ran individually and exhibited detectable behavior. Since there are several factors that can lead an extension to trigger page alterations, their approach was to create, in the *honeypages*, all the possible elements that can be created using HTML, that is, all tags and types with various combinations of associated attributes. In addition to this, they also included certain specific keywords on the page which led to some extensions being triggered. The choice of these keywords was made through an analysis of the extensions available on the Chrome Web Store. They made the decision to run each extension three times on the *honeypage* to ensure that the behaviors did not change. In case there were different behaviors for the same extention, all variations were considered. For each extension they created the behavioral signature and stored this information into a database. The behavior signature they used is basically a series of additions and removals of elements, where editing an element can be thought of as one removal followed by one addition. For instance, the behavior signature for adding the image *<img src='image.png'>* would be the sets *[{"<img", "src='image.png'>"}, {}]*. In this example, the second set is empty because nothing is removed. Finally, on the page where the extensions were to be identified, the behavioral signature of the extensions present was calculated *on-the-fly*, which were then compared to the signatures created previously. In cases where the added content is dynamic (for example, the element attribute is based on the date or time of day), there will never be a perfect match of signatures and therefore in *Carnus* they added a small percentage of missmatch acception. If the signatures were sufficiently similar, they considered it to be a match. Obviously this can cause some problems and lead to false positives. They countered this by double checking, using the other detection methods that *Carnus* implements. If the other techniques are also unable to identify the extension, then they consider the extension to be non-identifiable by the system. Figure 3.2 ilustrates an overview of the full Carnus implementation.

In the validation phase of the *Carnus* system, a dataset of 102,482 extensions was used. In

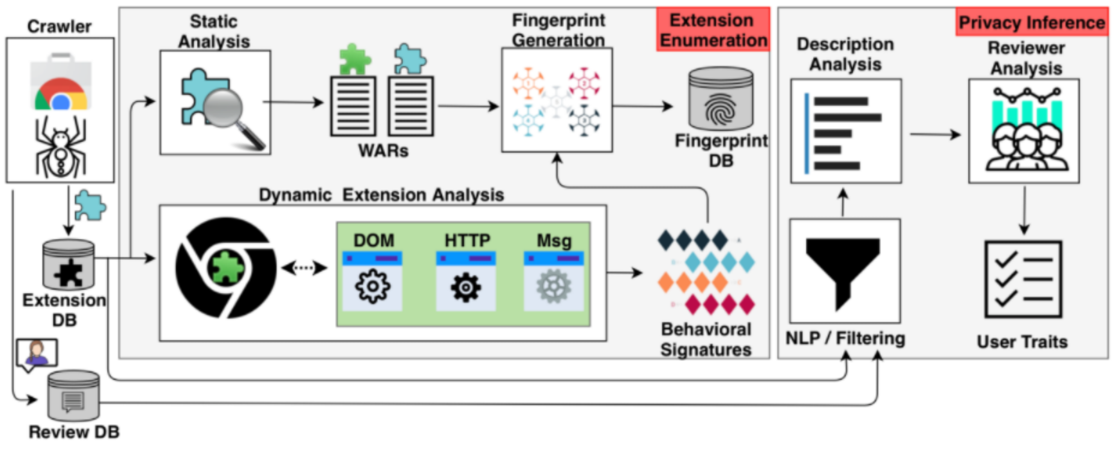


Figure 3.2: Overview of the *Carnus* system architecture
*Note:* Reprinted from [17]

total, 5,793 extensions were detected, that is, a percentage of 5.7%, through the detection method by DOM manipulation. However, only 2,998 extensions (2.9%) were uniquely detected using this method. It is important to mention that these results were obtained in test pages where each extension was tested individually, and when the validation was performed in environments with multiple extensions installed, the success rate slightly decreases. The success of this detection technique heavily depends on the behavior signature creation process and the way in which different behaviors are parameterized. If this process is improved, the number of identifiable extensions could potentially increase.

## 3.3 Detection through WARs

Detection through WARs is a technique that was proposed in 2017 [34] and is considered non-behavioral detection. Web Accessible Resources or simply WARs are files inside the extension, such as images, that are needed during runtime for the extension to run properly. These files need to be mentioned in the metadata file of the extension. The browser can make a request for a specific WAR and check if it is present in the current session. This way it is possible to know which WARs are present. Added to this, if we know which extensions use which WARs, then the presence of a particular WAR will indicate the presence of a particular extension. It was from this simple idea that this detection technique was idealized, which ended up resulting in an implementation with one of the highest detection success rates.

In [34] this technique was put into practice through a two-step implementation. As mentioned before, we need to know which WARs are associated with which extensions and therefore, the first step of the implementation was to create a database that stores this information. For example, in Chrome and Firefox browsers, this can be achieved with a script that iterates through all available extensions and collects the metadata file associated with each one. Then with a parse of this file it is possible to know which WARs are declared. In the Chrome browser for example, the metadata file is called '*manifest.json*' and the declaration of WARs is always done in the '*web_accessible_resources*' field.

After the database was created, they moved on to the second step of implementation. On the test page, they iterated over the extensions that they discovered to have WARs associated with them, and for each one, made a request for the WAR. All WARs are present in the URL '*chrome-extension://<extensionid>/<pathToFile>*' in case of Google Chrome. <*extensionid*> is the extension's unique identifier and <*pathToFile*> is the same relative URL as in the root package. The request can be done using *XMLHttpRequest* for example. Then, through the positive or negative return of the request, they knew if the WAR was present in the session, which in turn indicated if the extension was installed in the user's browser. It is important to note that, unlike behavioral techniques like the one described in Section 3.2, due to the unique identifiers of the extensions, this technique does not return false positives.

After testing this implementation, in the validation process, researchers found that they were able to uniquely identify 28% of the total extensions and, in the case of Google Chrome, more than 50% of the top 1,000 extensions in terms of number of users. This detection success rate is currently the highest rate among the most commonly used techniques due to the high number of extensions that use WARs. However, due to the architecture of Mozilla Firefox, there is not much need for the use of WARs and therefore the detection results in this browser are considerably lower.

Despite the high success rate, [34] mentions some limitations that, if resolved, can further increase the pool of extensions detectable through WARs. One of the limitations presented is the fact that it is possible to use WARs without mentioning them in the metadata file, making detection through this method impossible. Extensions can hide the use of WARs by, for example, representing these resources with strings using URIs, or by storing these files on external servers and requesting them during runtime. It is uncertain how many extensions hide WARs through these methods, but if techniques that contradict these practices are implemented, the detection success rate has the potential to increase.

## 3.4 Detection through CSS injection

One of the most recent but most effective ways to fingerprint extensions was presented in [21]. Researchers realized that, just like regular web pages, extensions rely on CSS to implement their user interfaces. There are multiple occasions in which an extension may need to use CSS, not only to make the UI that is invisible to the website (such as the UI that appears when the user clicks the extension icon) but also to inject some new UI element into the web page (such as a new button that executes a specific feature). Modern browsers allow scripts to request the styling of DOM elements. On Google Chrome, for example, it is possible to make a request to the API - *getComputedStyle* - that returns all the styles applied to that specific DOM element, including inline CSS and CSS applied by JavaScript execution. Through this, it is possible to gather which IDs and class names a specific extension uses in its style sheets and then create trigger DOM elements with those IDs and classes assigned to them. Then, whenever the styles applied to these trigger DOM elements are different from the default ones, the extension is present. The author calls this fabricated DOM elements, *tripwires*.

One possible implementation of this technique is presented in [21]. This implementation is divided into four different stages and the processing pipeline can be viewed in figure 3.3.

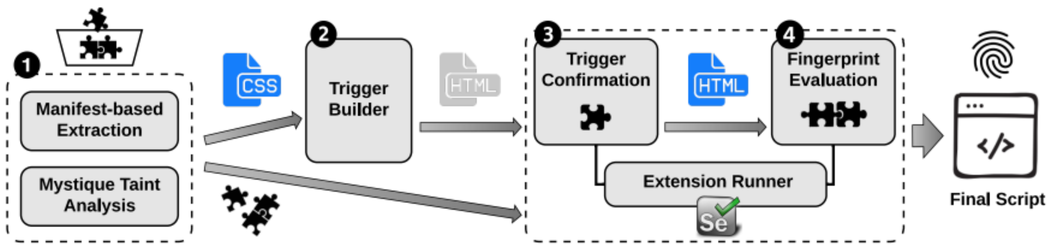


Figure 3.3: Example of one implementation using detection through CSS injection
*Note:* Reprinted from [21]

**Stage 1.** The first stage is the stage where the injected style sheets are gathered from the extensions. First there needs to be a dataset of extensions chosen to detect. In [21] 116,485 extensions were collected from the Google Chrome store, removing some irrelevant theme based extensions. There are two types of style injection that need to be collected: declarative injection and programmatic injection. In case of Chrome extensions, each one has a unique *manifest.json* file. This file has a *content_scripts* field where developers declare the CSS style sheets that are applied to the DOM elements by the extension. After parsing this file they were able to easily extract the information they needed. To detect programmatic injection the process was more complicated. A tool named *Mystique* [4] was developed precisely to detect this type of CSS injection. In [21] they used Mystique's web interface to capture CSS injections.

**Stage 2.** After collecting all the styles that are injected by the extensions they wanted to detect, the next stage was to create the trigger DOM elements, or *tripwires* as they call them. They created various DOM elements with IDs and class names assigned to them. These IDs and classes are based on the information extracted from the previous stage. They limited the number of triggers to 50 per each extension. Adding more triggers would be redundant as they only needed a few to uniquely identify extensions.

**Stage 3.** After stage 2 they had 50 triggers per extension created, but they needed to verify that these triggers actually work and would activate whenever its respective extension was present. There may be many reasons why the triggers don't work: classes can change during runtime or there may be other CSS rules that overlap and take precedence. Hence, stage 3 focuses on making this verification. The verification was done in two different ways: first the test page was executed three times to verify that the behavior was always the same and unchanged; and secondly, another element that mimics the trigger was made but with one difference, it didn't have the associated classes and IDs. The goal was to have two elements with the exact same DOM hierarchy and verify if CSS rules would apply to one but not the other.

**Stage 4.** In the final stage they needed to make sure that there was no trigger overlap between extensions, meaning two different extensions could not have the same exact trigger. This could potentially lead to false positives. Many extensions use generic or repeated rules. Let's imagine for example that two different extensions contain one DOM element with the same CSS rule and one user has only one of these two extensions installed on the browser. The trigger will fire for both extensions and will inform that both are present, when in reality only one is. The way they mitigated this was to test all triggers with every extension individually and check if the trigger would activated on the presence of wrong extensions.

After testing and running this implementation, they found that a total of 4,446 extensions from the initial dataset of 116,485 were identified. Even though 17,712 extensions inject at least one CSS file, not all of them could be detected. Some of these extensions only act in specific domains and did not trigger the *tripwires* on the test page. Others, they were not able to create reliable triggers to detect them. The rest of them had overlapping triggers (detected on stage 4) that would make impossible to distinguish one extension from the other. Analyzing these results, it can be concluded that there are 17,712 potentially fingerprintable extensions using this method, thus representing the ceiling of the method for that dataset. These limitations are caused by the way this technique was implemented and shows there is potential for improvement.

## 3.5 Other Techniques

The three techniques mentioned in sections 3.2, 3.3 and 3.4 are the methods with the highest detection rate and therefore the most effective and the ones that have received the most attention from the scientific community. However, over the years, other fingerprinting techniques have been proposed. Some of these techniques are described in this section of the document.

One of the techniques that hasn't received much attention from the research community is detection through the *postMessage()* method [17] [21] [35]. Using the *postMessage()* method and with an extension's unique identifier, a website can send a message to the extension with that ID. The extension will have to have a listener implemented, which waits for this message and in turn, responds to the request. The way extensions can be identified with this method is basically to make a request that the extension is expecting, and check if there was a response or not. If there is a response then it means the extension is present. Although this method is quite effective in identifying extensions that have listeners implemented, the problem is that the vast majority of extensions do not use this browser feature and therefore the success rate of detection is very low. In [21], using their dataset (total of 116,485 extensions), they found that this method would only be able to detect 350 extensions. For this reason, not many implementations and studies have been carried out to further validate the technique.

Another less common fingerprinting technique is detection through HTTP requests, called Inter-communication based fingerprinting [17]. Through a specific API and the *getEntriesByType("resource")* method, it is possible to monitor requests made by the browser page. This way it is possible to then run the extensions individually on a test page and check what requests each extension makes. Then, on the page where the detection is made, the requests are monitored and if they are the same as the ones tested before, then the matching extensions are present. It is a relatively simple method to implement and does not lead to many false detections. However, in the *Carnus* system, this method was implemented and it was found that, of the 102,482 extensions they tested, only 105 (0.1%) were uniquely detected with this technique. These results show that this technique is not very effective in detecting extensions.

In addition, there are several methods that focus on classifying installed extensions into various categories through their perceived behavior, rather than identifying them [32]. This categorization is important because it is often in the interest of the website or the user, to know if the extensions are malicious or not instead of identifying which specific extensions are present [30] [32]. For this reason, these methods are considered to exceed this dissertation's scope, and therefore, they will not be further detailed nor implemented.

## 3.6 Summary

After an extensive analysis of the state of the art, of the different techniques that have been proposed over the years, it can be concluded that identifying extensions (extension fingerprinting) is not a simple task due to the fact that there is no ideal solution. Each technique targets a specific feature that some extensions use but others don't. For this reason, in order to obtain a high detection success rate, it is necessary to combine multiple approaches, preferably approaches that do not have much detection overlap between them. These are the parameters to be taken into account when choosing techniques to implement into the solution. The details of the solution are described in Chapter 4.

Table 3.1: Detection statistics for each fingerprinting technique

| **Technique** | **Dataset** | **Number of extensions detected** | **Number of extensions uniquely detected** | **Percentage of detected extensions** | **Percentage of uniquely detected extensions** |
|---|---|---|---|---|---|
| DOM | 102,482 | 5,793 | 2,998 | 5.7% | 2.9% |
| WARs | 43,429 | 12,154 | - | 28% | 22%* |
| CSS | 116,485 | 4,446 | 1,074 | 3.8% | 1% |

Table 3.1 compiles data on detection rates of each of the mentioned fingerprinting techniques. It is important to mention that in the implementation of detection by WARs [34] no data was mentioned regarding the amount of extensions that the technique was able to uniquely detect. Since several techniques were not implemented in [34], there was no method of comparing. However, in [17] it was found that (according to the dataset used) detection through WARs was able to uniquely identify 22% of the extensions.

Additionally, some gaps or aspects that can be improved in the methods were also reported. In the detection by DOM manipulation, the detection rate was low (on the *Carnus* implementation) due to the process of creating behavior signatures. If this process is improved, the rate could potentially increase. In detection by WARs, the detection rate is already high. However, this method fails to detect some extensions because they hide the use of WARs through different methods. If these ways of hiding are detected, it would be possible to reach an even higher percentage of detection. Finally, in the detection by CSS injection, extensions that only act in specific domains were not considered, thus leaving out a large number of extensions that can potentially be detected using this technique.

Furthermore, in all these implementations, no data was mentioned about the performance cost and viability. The main objective of this dissertation is not only to combine the different methods and trying to improve them, but also to study their performance cost and implement them in a way where a injected script can carry out massive on-site extension detection during run time, without harming the user experience and other common functionalities of the website.

# Chapter 4

# Problem and Proposed Solution

This section serves to present the problem and the solution that this dissertation proposes. In Section 4.1 the problem is outlined, followed by the description of the proposed solution in Section 4.2. Finally, in Section 4.3, we talk about the validation process.

## 4.1 Problem

As it was already established throughout this document, web security and protection of users' privacy is an extremely important topic in the field of computer science today. Since extensions are part of this ecosystem and are widely used by users of the most famous browsers, it is important to know the associated risks. Therefore, the questions that this dissertation aims to answer are the following:

- How can we detect and identify which extensions the user has installed on their browser?
- Which methods are viable to implement on-site extension detection during run time, without harming the user experience?
- How can extension fingerprinting methods be combined in order to maximize detection rate whilst still maintaining performance viability?

There already exist several fingerprinting methods to detect extensions, as established in Chapter 3. Previous work focuses on the detection aspect of the methods without analysing the performance cost of each one. In this dissertation, the objective is to not only study the detection efficiency of several methods, but also to describe how each one of them was implemented. Then, make an analysis of the performance cost of each one, propose a way to combine them and develop a script capable of detecting extensions in a way that's viable performance wise.

## 4.2 Solution Approach

To answer the questions raised in Section 4.1, a script was developed, that implements three different extension fingerprinting methods. The three methods were selected taking into account the detection success rates of previous implementations and also because each one of them targets different features of the browser and extensions. Therefore there is not much detection overlap (different methods detect the same extensions). These three methods are: detection by DOM manipulation, detection by WARs and detection by CSS injection. The methods were detailed in sections 3.3, 3.4 and 3.5 respectively and their implementation in this work is described in sections 6.2, 6.1 and 6.3 respectively. In case an extension can be detected by two implemented methods, a decision needs to be made on which of the methods is used to detect that extension. This decision is based on the performance analysis of Section 6.2. The final script implementation is detailed in Chapter 7.

## 4.3 Validation

To validate the developed script and each individual method, a dataset of extensions was selected to test the detection rate. This dataset creation process is detailed in Section 5.1. The script and individual methods were tested on Google Chrome using a ***'selenium'*** and ***'Node.js'*** environment. A Chrome process is executed with one dataset extension installed and the detection is tested. The metric used to compare different methods is the percentage of extensions that the script was able to identify. The performance impact that each implementation has is also measured. The performance impact is measured using website loading time and script execution time. To test the performance viability, the script is also injected into real complex websites via proxy, and the performance cost is measured.

# Chapter 5

# Extension Detection Methods Implementation

This chapter serves to describe every step of the development process for the individual extension detection methods. First, we start by detailing the dataset creation process and analysing the dataset in Section 5.1. Then, the various methods' implementations are explained in sections 5.2, 5.3 and 5.4. Section 5.5 concludes the chapter by summarizing the development process and enumerating some conclusions that can be made.

## 5.1 Dataset

This section explains why there is a need for a dataset, how it was created and how the extension data was collected from the browser web store without an accessible API. Section 5.1.1 explains the creation process, and Section 5.1.2 describes all the extensions' details.

Generally, to implement a extension detection method, there are two phases:

1. First the extensions' files and data need to be iterated so that we can extract the necessary information to target them. This information will vary depending on the method we are implementing;
2. Then, the script that will be run on the website where the detection is to be implemented needs to be developed. The script accesses the information collected in the first stage of implementation and uses it to check for presence of the extension.

So, the first step in the project's development is establishing a dataset of browser extensions with which the necessary information can be collected and the developed techniques can be tested and validated. Ideally, the dataset would consist of all the extensions currently available for download from the Google Chrome web store. At the time this dissertation was developed, there are a total of 137,345 extensions. However, using all available extensions was not possible due to time constraints, processing power, and storage memory. Therefore, choosing a smaller sample of extensions was necessary to validate the detection rate of the implemented techniques. One of the components of this work is precisely the study of the performance of the mentioned techniques. For this purpose, the extensions in the dataset can simply be duplicated as many times as necessary because the detection of one particular extension does not influence the website's performance.

### 5.1.1 Creation

As mentioned previously, choosing a small sample of extensions is necessary. However, the Google Chrome web store does not have an accessible API where the extensions files and meta-data are available for access. Therefore, it would be necessary to develop a web scraper to collect the needed information or to resort to an external solution that could provide us with this data. Since developing the web scraper would take up valuable time and would not be entirely helpful in the context of the project, the best decision is to opt for the second option. The selected API is provided by the *Node* package named ***'chrome-webstore'*** [33].

Now, having access to an API that provides information about the extensions we want to study, it is still necessary to choose what extensions are a part of the dataset. Considering the context and Use Cases of this work, the best option was to have a mix of extensions with a high number of downloads as well as extensions that are more niche and do not have that many downloads. In 2020, a website called ***DebugBear*** studied the impact of the individual installation of 1,000 extensions on overall browser performance [39]. A year later, in 2021, ***DebugBear*** updated the data and performed the same study with the same list of extensions. It is important to mention that this study only focused on the impact that the extensions themselves had on performance, not their detection, which is what we intended to focus on in this work. The decision was made to use the same list of extensions that were used in the mentioned study for our dataset. Table 5.1 displays some of the most popular extensions from the mentioned list, most of them with more than 10 million downloads.

Table 5.1: Top 20 dataset extensions by number of downloads.

| Extension Name | Category | Number of Installs |
|---|---|---|
| Google Translate | Productivity | 10M+ |
| Google Hangouts | Social & Comunication | 10M+ |
| Netflix Party | Fun | 10M+ |
| Tempermonkey | Productivity | 10M+ |
| Adblock - best ad blocker | Productivity | 10M+ |
| Adblock Plus - free ad blocker | Productivity | 10M+ |
| uBlock Origin | Productivity | 10M+ |
| Adobe Acrobat | Productivity | 10M+ |
| Chrome Remote Desktop | Productivity | 10M+ |
| Pinterest Save button | Productivity | 10M+ |
| LastPass: Free Password Manager | Productivity | 10M+ |
| Avast Safeprice \| Comparison, deals, coupons | Shopping | 10M+ |
| Avast Online Security | Social & Comunication | 10M+ |
| Skype | Social & Comunication | 10M+ |
| Cisco Webex Extension | Social & Comunication | 10M+ |
| Grammarly for Chrome | Productivity | 10M+ |
| Honey | Shopping | 10M+ |
| AVG SafePrice \| Comparison, deals, coupons | Shopping | 9M+ |
| Screencastify - Screen Video Recorder | Productivity | 8M+ |
| Hola free VPN, unblock any site! | Productivity | 8M+ |

In the time frame between the extension list being created and this work starting development, some extensions were permanently removed from the browser web store. This resulted in a slight decrease in the dataset to a total of 947 extensions instead of 1,000 extensions.

### 5.1.2 Analysis

Extensions fall into various categories like *'Productivity'*, *'Shopping'*, *'Developer Tools'*, among others. The distribution of dataset extensions by categories is displayed in figure 5.1. More than half of the extensions (55.2%) are categorized as *'Productivity'* extensions.

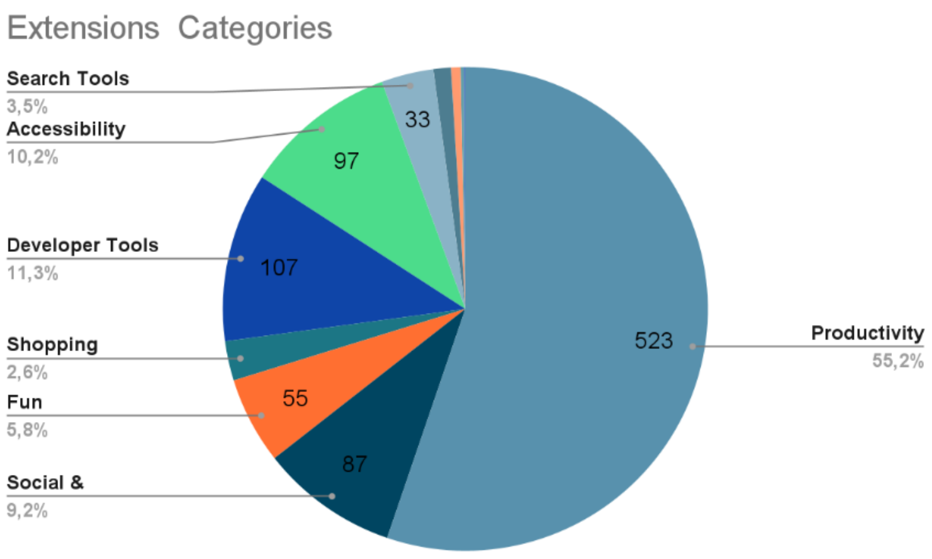
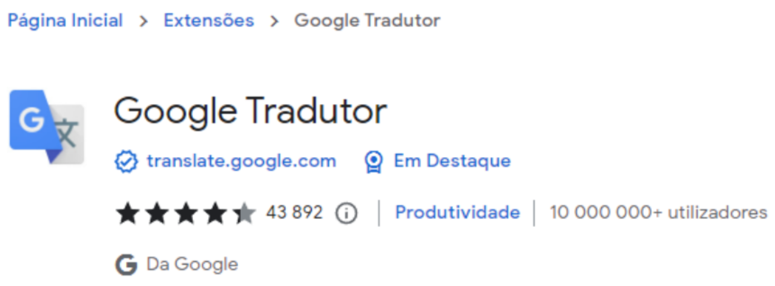


Figure 5.1: Extension distribution by web store categories.

As for the number of downloads, the Google Chrome web store does not show the exact number of downloads per extension. Figure 5.2 presents an example of the type of information displayed on an extension page.


Página Inicial > Extensões > Google Tradutor

Google Tradutor
translate.google.com Em Destaque
43 892 | Produtividade | 10 000 000+ utilizadores
Da Google

Figure 5.2: Example of an extension page on the Chrome Web Store.

All extensions from this dataset have between 100,000+ and 1,000,000+ downloads. It should be noted that because of the way the extension recommendation system works in the Chrome web store (the number of downloads is highly regarded as a quality metric), the entities behind the extensions are indirectly encouraged to artificially increase this number, which can lead to false values [39]. Google combats this by regularly doing checks to validate whether the downloads are legitimate or not. If the number of downloads is considered not legitimate, the extension is permanently removed from the store [39].

Regarding the size in memory that the files of each dataset extension occupy, the distribution is displayed in figure 5.3. In total, the entire dataset occupies **4.52 GB**. Suppose we wanted to process all extensions available for download on the store. Assuming that the distribution of sizes would be the same and that there are 137,345 extensions, the dataset could potentially occupy **655.54 GB**.

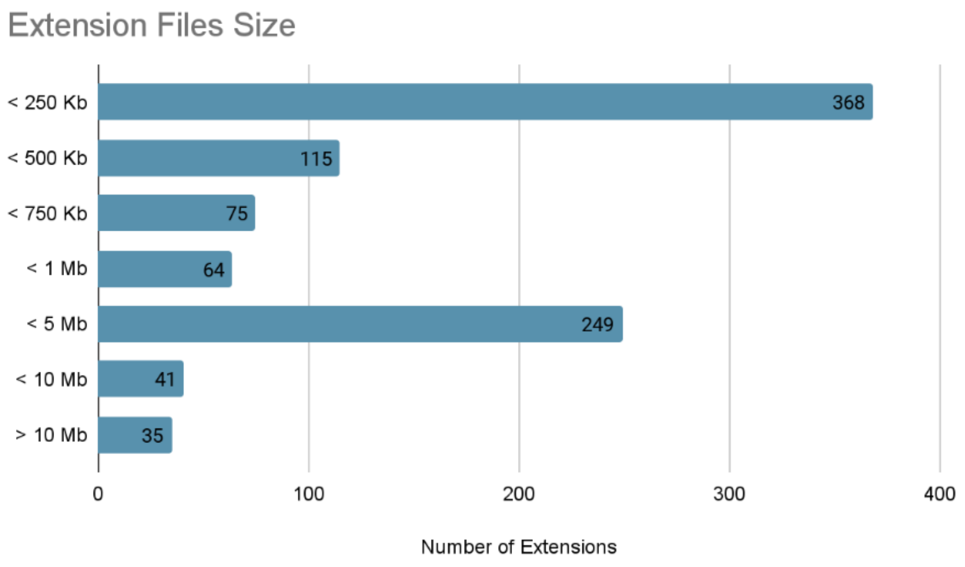


Figure 5.3: Distribution of dataset extensions file size.

Figure 5.4 shows the average distribution of file sizes by extension category. Data shows that the average memory occupied by each extension category does not vary much, however certain categories take up considerably less space, such as the case of the *'Search Tools'* category, and the detection of these types of extensions may be lighter in terms of data processing performance.

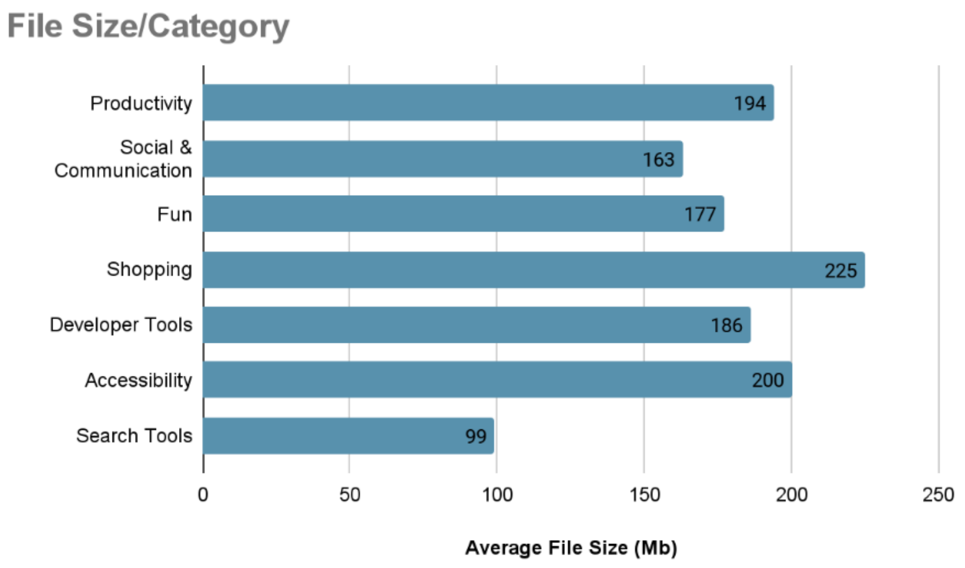


Figure 5.4: File size of dataset extensions by store category.

Figure 5.5 shows the rating distribution (from zero to five stars) that the dataset extensions have on the store. More than 60% of extensions are rated at or above four stars. Finally, figure 5.6 shows the distribution of the number of users who rated the extension on the web store.

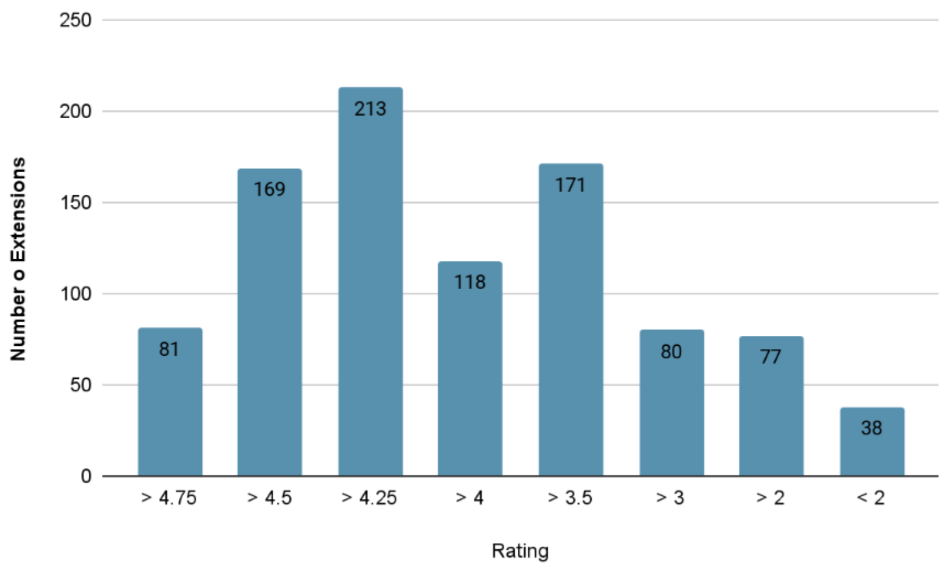


Figure 5.5: Distribution of dataset extension ratings.

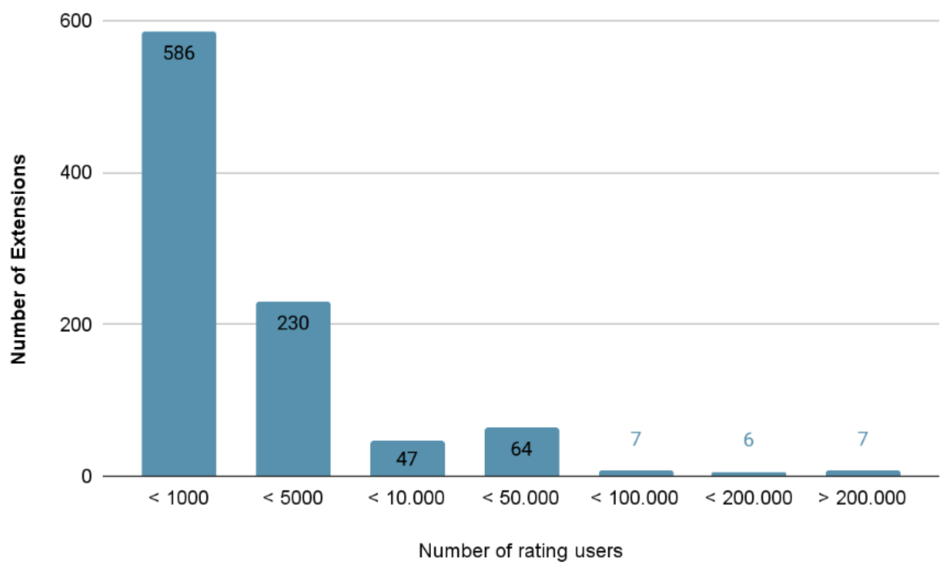


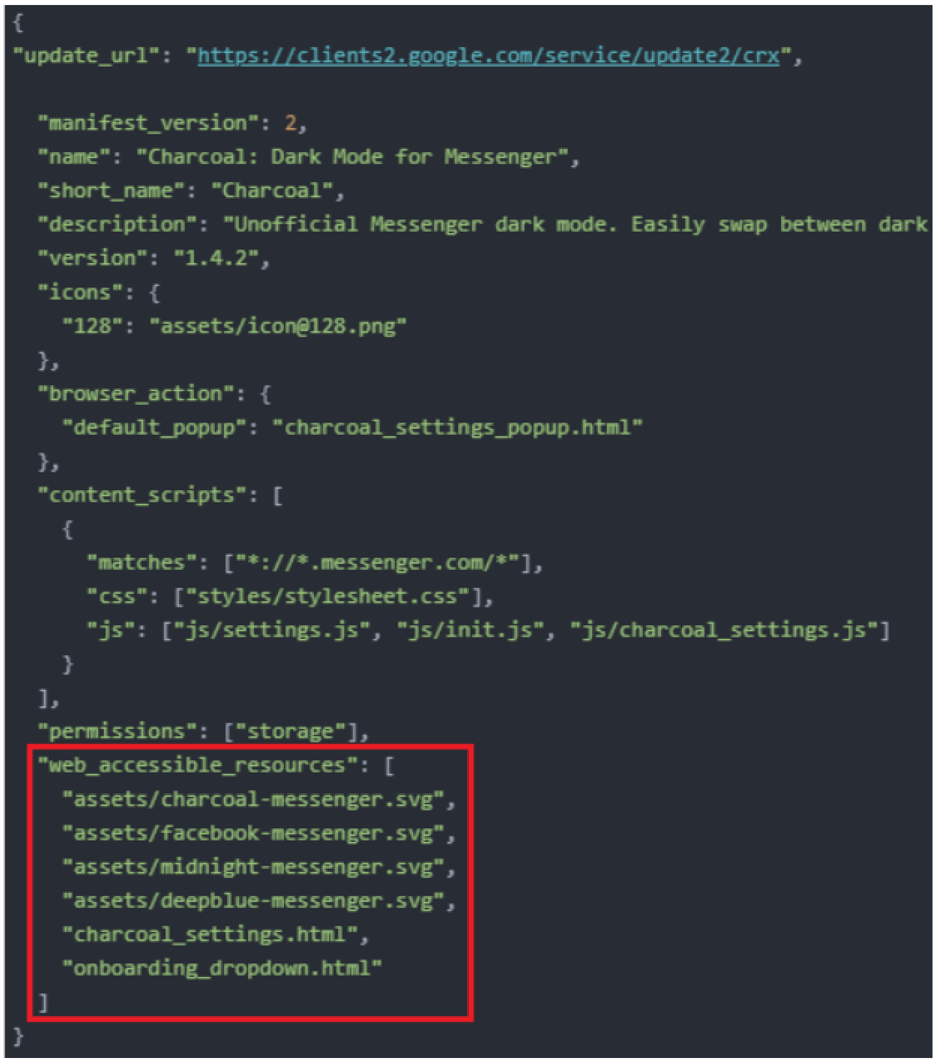
Figure 5.6: Distribution of number of rating users by dataset extensions.

As we can see from the number of downloads and reviews, some dataset extensions are pretty popular and are constantly at the top of the list of most used extensions throughout the years. The following chapters show that many of them are highly detectable and vulnerable to the implemented detection methods. The dataset is quite varied and is a realistic sample of the variety of extensions available for download from the Google Chrome web store.

## 5.2 Detection through WARs

Due to the high detection rate results from previous implementations, the first implemented methodology was the detection through the extensions' *Web Accessible Resources* (*WARs*). As described in Section 3.3, this method is the technique that previously presented the best results in terms of detection rate. Therefore, it has the highest priority among the three implemented methods. This means that in the final script if an extension can be detected using *WARs*, it should be the method used to detect it. The performance analysis is also performed in Section 6.2.2, where the results support this decision.

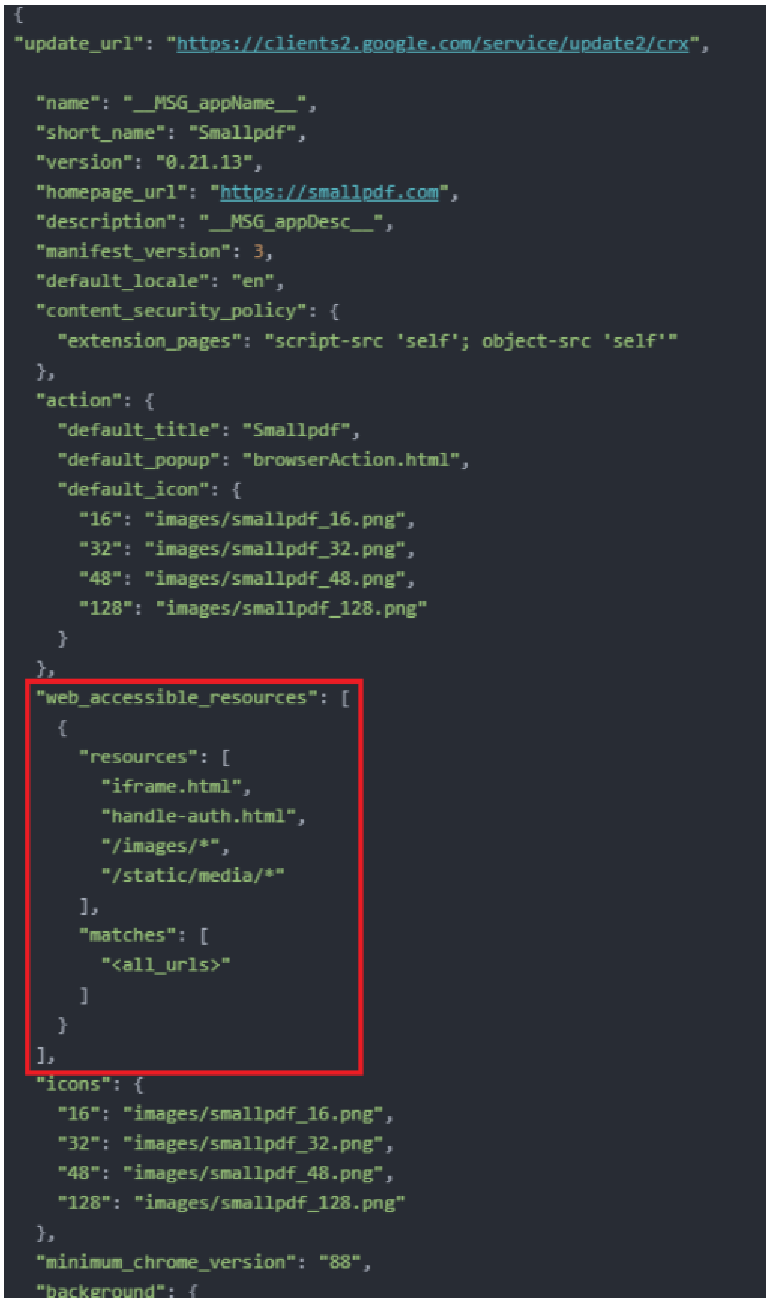
However, in terms of implementation, the method is pretty straightforward, and there is not much freedom to make changes in order to try to potentially improve performance. The first step of the implementation is to fetch the manifest file of each extension present in the dataset. Then, the manifest files are parsed in order to find and associate the *WARs* to each extension. At the time of this work, there are two different formats that the manifest file of the extensions can have: *manifest v2* and *manifest v3*. These differences are displayed in figures 5.7 and 5.8 respectively.



```
{
"update_url": "https://clients2.google.com/service/update2/crx",

  "manifest_version": 2,
  "name": "Charcoal: Dark Mode for Messenger",
  "short_name": "Charcoal",
  "description": "Unofficial Messenger dark mode. Easily swap between dark
  "version": "1.4.2",
  "icons": {
    "128": "assets/icon@128.png"
  },
  "browser_action": {
    "default_popup": "charcoal_settings_popup.html"
  },
  "content_scripts": [
    {
      "matches": ["*://*.messenger.com/*"],
      "css": ["styles/stylesheet.css"],
      "js": ["js/settings.js", "js/init.js", "js/charcoal_settings.js"]
    }
  ],
  "permissions": ["storage"],
  "web_accessible_resources": [
    "assets/charcoal-messenger.svg",
    "assets/facebook-messenger.svg",
    "assets/midnight-messenger.svg",
    "assets/deepblue-messenger.svg",
    "charcoal_settings.html",
    "onboarding_dropdown.html"
  ]
}
```

Figure 5.7: Extension *manifest v2* format.



```
{
"update_url": "https://clients2.google.com/service/update2/crx",

  "name": "__MSG_appName__",
  "short_name": "Smallpdf",
  "version": "0.21.13",
  "homepage_url": "https://smallpdf.com",
  "description": "__MSG_appDesc__",
  "manifest_version": 3,
  "default_locale": "en",
  "content_security_policy": {
    "extension_pages": "script-src 'self'; object-src 'self'"
  },
  "action": {
    "default_title": "Smallpdf",
    "default_popup": "browserAction.html",
    "default_icon": {
      "16": "images/smallpdf_16.png",
      "32": "images/smallpdf_32.png",
      "48": "images/smallpdf_48.png",
      "128": "images/smallpdf_128.png"
    }
  },
  "web_accessible_resources": [
    {
      "resources": [
        "iframe.html",
        "handle-auth.html",
        "/images/*",
        "/static/media/*"
      ],
      "matches": [
        "<all_urls>"
      ]
    }
  ],
  "icons": {
    "16": "images/smallpdf_16.png",
    "32": "images/smallpdf_32.png",
    "48": "images/smallpdf_48.png",
    "128": "images/smallpdf_128.png"
  },
  "minimum_chrome_version": "88",
  "background": {
```

Figure 5.8: Extension *manifest v3* format.

The difference between the two formats is that in the *manifest v2* format, the extensions can only declare which *WARs* are allowed to be accessed on any web page (this version of the manifest was discontinued as of January of 2022, and it is no longer allowed to publish extensions with this format). In *manifest v3* format, as we can see by the ***'matches' tag*** in figure 5.8, the extensions can declare on which *URLs* the *WARs* are accessible [9] [11]. This change to the manifest format was made in January 2022. Eventually, all manifests will have this format, which will affect the detection rate of this method. If we fetch a WAR from a web page that was not declared in the manifest, the file will not be accessible, and therefore the detection fails.

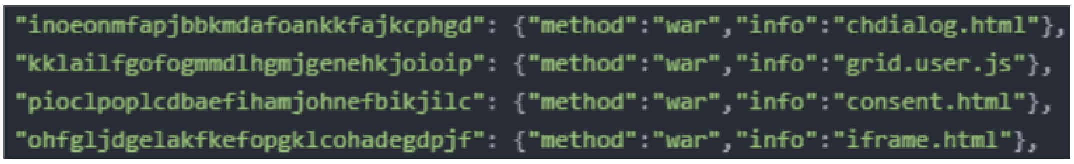
It is important to note that only one *WAR* is needed to detect a given extension. Many extensions declare several *WARs* in the manifest, but we only need to store one. The choice of which *WAR* to use to perform the detection is also irrelevant since it will only be necessary to verify the presence of the file in the current session without any type of additional operation on that same file.

Often extensions do not declare specific files but groups of files using '*' or entire directories. In these cases, it is explicitly necessary to have access to the extension files in order to find a file that matches the description. Google Chrome uses a packaging format named ***'crx'***. All extension files are packaged in a ***'crx'*** file. Because of this, in order to have access to the files of an extension, we must first obtain its ***'crx'*** and then unpack it. An extension's ***'crx'*** file can be obtained directly using the URL 'https://clients2.google.com/service/update2/crx?response=redirect&prodversion=**(PRODVERSION)**&acceptformat=crx2,crx3&x=id%3D**(EXTENSIONID)**%26uc' where:

- **(PRODVERSION)** is the browser version and;
- **(EXTENSIONID)** is the extension ID of which we intend to get the ***'crx'*** package.

After downloading all the ***'crx'*** files for all the dataset extensions, it is now necessary to unpack them. For this, an external library called ***'unzip-crx'*** can be used [28]. This library receives a ***'crx'*** file as input and unpacks it. After unzipping all the ***'crx'*** files obtained in the previous step, we now have access to all the extension's files.

Having now the extension's files, we can select a *WAR* from each of them to do the verification. In the final *JSON* containing the information needed to detect extesnions, the data regarding the extensions that are detectable with *WARs* is given in the format exemplified in figure 5.9.



```
"inoeonmfapjbbkmdafoankkfajkcphgd": {"method":"war","info":"chdialog.html"},
"kklailfgofogmmdlhgmjgenehkjoioip": {"method":"war","info":"grid.user.js"},
"pioclpoplcdbaefihamjohnefbikjilc": {"method":"war","info":"consent.html"},
"ohfgljdgelakfkefopgklcohadegdpjf": {"method":"war","info":"iframe.html"},
```

Figure 5.9: Format of the information about extension *WARs* in the final *JSON* file.

Finally, in the second phase of implementation, in the script where the test is performed, the *URL* for the fetch is constructed in the same way as explained in Section 3.3. A request can be made for a *WAR* using the *URL* 'chrome-extension://**(EXTENSIONID)**/**(EXTENSIONWAR)**' where:

- **(EXTENSIONID)** is extension ID and;
- **(EXTENSIONWAR)** is the extension *WAR* directory.

A fetch request is performed for each extension that needs to be tested. The result will be returned within a ***'Promise'*** object. In the case of a positive answer, the extension is installed.

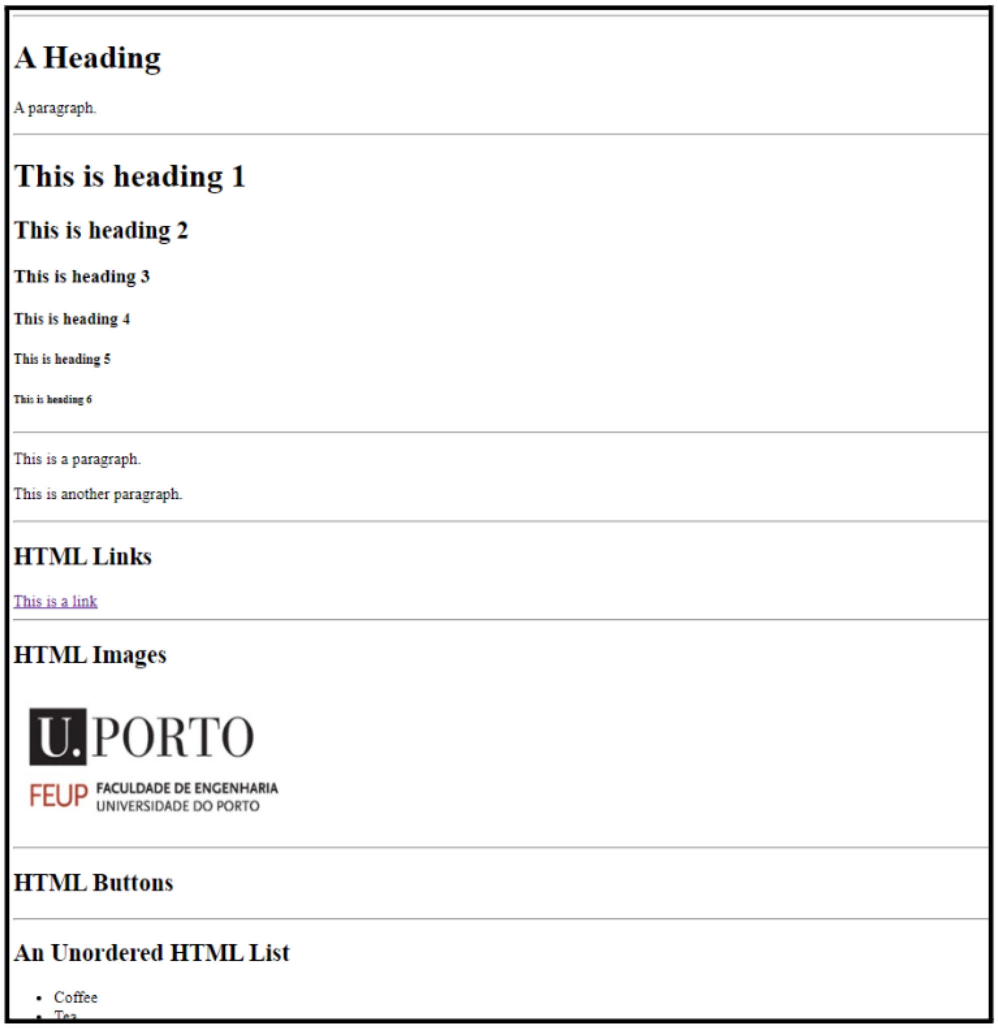
Otherwise, the extension is not installed. The code snippet for this part of the implementation is the following:

```
fetch("chrome-extension://" + id + "/" + ext.war)
        .then(() => ( resolve(true) ))
        .catch(() => ( resolve(false) ))
```

By this point, this method's development is complete. The results for this method's implementation are presented and analyzed in Section 6.1.1. Section 6.2.2 contains the performance analysis for this method.

## 5.3 Detection through DOM manipulation

To detect extensions through DOM manipulation, it is also necessary to do some processing beforehand to discover what changes each individual extension makes to a webpage. With this in mind, a *selenium* environment can be used to run a Chrome browser with only one dataset extension installed. The objective is to detect what changes that specific extension makes to the web page's DOM. The name *"honey page"* was given to refer to the page that was used. This page is composed of a series of diversified *HTML* elements having the objective of covering several structures and all the existing elements. A snippet of the *"honey page"* is shown in figure 5.10. Extensions that only make changes to certain elements with a particular class or ID are extensions that only target specific domains. Therefore, it is impossible to detect using this method in other domains, so there is no need to try to detect the changes that an extension of this type does to the DOM of a random page.



A Heading

A paragraph.

This is heading 1

This is heading 2

This is heading 3

This is heading 4

This is heading 5

This is heading 6

This is a paragraph.

This is another paragraph.

HTML Links

This is a link

HTML Images

HTML Buttons

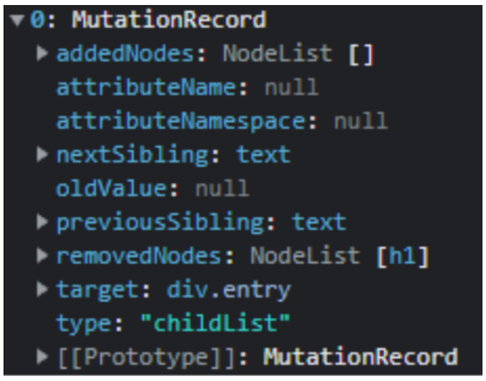
An Unordered HTML List

- Coffee
- Tea

Figure 5.10: Snippet of the honey page used to detect extension behavior.

The detection of extension behavior is done through the ***’MutationObserver’*** object. The ***’MutationObserver’*** object detects all changes made to a page’s DOM. Through the ***’observe()’*** method, we can set the ***’MutationObserver’*** to observe a particular element. Similarly to a Javascript event, a function is called when changes are made to the element’s node or child nodes.

When one of the elements is changed, the function receives an array of ***’MutationRecord’*** objects as an argument with the information that was changed. An example of one ***’MutationRecord’*** object is shown in figure 5.11.



```
▼0: MutationRecord
  ▶addedNodes: NodeList []
   attributeName: null
   attributeNamespace: null
  ▶nextSibling: text
   oldValue: null
  ▶previousSibling: text
  ▶removedNodes: NodeList [h1]
  ▶target: div.entry
   type: "childList"
  ▶[[Prototype]]: MutationRecord
```

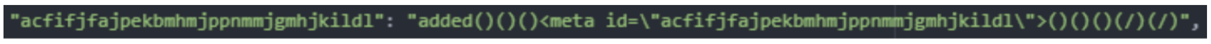
Figure 5.11: Example of one ***'MutationRecord'*** object.

The example of figure 5.11 displays a removal of an element ***'h1'***. Elements that are added can be accessed in the array in the ***'addedNodes'*** attribute and elements that are removed can be accessed in the ***'removedNodes'*** attribute. A change to an element, such as editing an attribute or ***'innerHTML'***, is represented by a removal followed by an addition, that is, the ***'MutationRecord'*** will have the element before editing in the ***'removedNodes'*** attribute and the element after editing in the ***'addedNodes'*** attribute.

This way, it is possible to detect and save the changes that each extension does to the page. Each change is stored in a string with the following format "**(TYPE)**()()()**(ELEMENT)** ()()(/)(/)" where:

- **(TYPE)** can be *'added'* or *'removed'* depending on the type of change that was made and;
- **(ELEMENT)** is the *HTML* element that has changed.

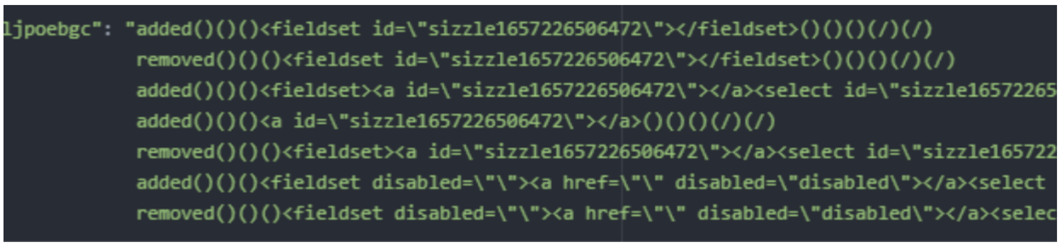
The parentheses serve to separate the information. For example, an addition of the *HTML* element “<meta id="acfifjfajpekbmhmjppnmmjgmhjkildl">” by the extension with ID *“acfifjfajpekbmhmjppnmmjgmhjkildl”* is represented in *JSON* as shown in figure 5.12.



```
"acfifjfajpekbmhmjppnmmjgmhjkildl": "added()()()<meta id=\"acfifjfajpekbmhmjppnmmjgmhjkildl\">()()()(/)(/)",
```

Figure 5.12: Example of one *DOM* behavior stored in the final *JSON* file.

A more complex example of behavior on the part of the extension with ID *“aopfgjfeiimeioiajeknfidlljpoebgc”* can be seen in figure 5.13.



```
ljpoebgc": "added()()()<fieldset id=\"sizzle1657226506472\"></fieldset>()()()(/)(/)
            removed()()()<fieldset id=\"sizzle1657226506472\"></fieldset>()()()(/)(/)
            added()()()<fieldset><a id=\"sizzle1657226506472\"></a><select id=\"sizzle16572265
            added()()()<a id=\"sizzle1657226506472\"></a>()()()(/)(/)
            removed()()()<fieldset><a id=\"sizzle1657226506472\"></a><select id=\"sizzle165722
            added()()()<fieldset disabled=\"\"><a href=\"\" disabled=\"disabled\"></a><select
            removed()()()<fieldset disabled=\"\"><a href=\"\" disabled=\"disabled\"></a><selec
```

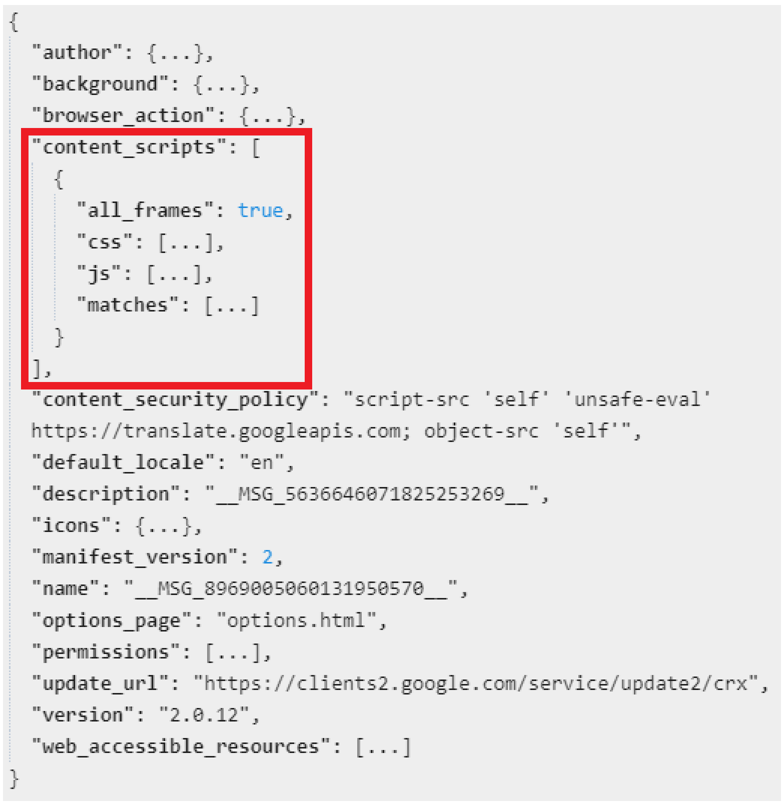
Figure 5.13: Example of a more complex *DOM* behavior stored in the final *JSON* file.

For the second phase of implementation, we need to collect the changes that are made to the current website where the script is running and compare them to the changes gathered in the previous step. During run time, the same process is made, one ***'MutationObserver'*** object is placed in the root of the page. Then, this pattern of alteration is compared to each individual extension behavior. In cases where the patterns exactly match, we consider the extension to be present.

This method's detection results are presented and analyzed in Section 6.1.2. The performance analysis is done in 6.2.3

## 5.4 Detection through CSS injection

The first step to implement detection through *CSS* injection is analysing the style sheets that every dataset extension potentially injects. In the manifest file of each extension, similarly to *WARs*, all the style sheets that are meant to be injected are enumerated. These files can be seen on the manifest, under the ***"content_scripts"*** attribute. Figure 5.14 shows one example of a manifest file containing the ***"content_scripts"*** attribute.



```
{
  "author": {...},
  "background": {...},
  "browser_action": {...},
  "content_scripts": [
    {
      "all_frames": true,
      "css": [...],
      "js": [...],
      "matches": [...]
    }
  ],
  "content_security_policy": "script-src 'self' 'unsafe-eval'
  https://translate.googleapis.com; object-src 'self'",
  "default_locale": "en",
  "description": "__MSG_5636646071825253269__",
  "icons": {...},
  "manifest_version": 2,
  "name": "__MSG_8969005060131950570__",
  "options_page": "options.html",
  "permissions": [...],
  "update_url": "https://clients2.google.com/service/update2/crx",
  "version": "2.0.12",
  "web_accessible_resources": [...]
}
```

Figure 5.14: Example of one manifest file containing ***"content_scripts"***.

As we can see, the *"content_scripts"* attribute contains another attribute named ***"css"***. This contains an array with all the *css* files the extension intends to use. Notice that *"content_scripts"* also contains a *"matches"* attribute, meaning that some extensions only inject in certain domains which certainly harms the detection rate results.

After having access to this information, the next step is to parse each individual dataset extension's *css* files. The final objective of this step is to parse all the *css* rules into iterable JavaScript objects so that later they can be easily iterated and we can create a trigger *HTML* element for each rule. This can be accomplished by using an external package named ***"cssjson"*** [19]. This package parses a *css* file and converts it to JSON. For example, the following *css* code:

```
#nav-below {
    border-bottom: 1px solid #ddd;
    margin-bottom: 1.625em;
    background-image: url(http://www.example.com/images/im.jpg);
}
```

Is converted into:

```
"#nav-below": {
      "children": {},
      "attributes": {
        "border-bottom": "1px solid #ddd",
        "margin-bottom": "1.625em",
        "background-image": "url(http://www.example.com/images/im.jpg)"
      }
    }
```

The next step of implementation is to parse the selector itself. To build a trigger that works, the *HTML* element needs to have exactly the same structure as defined by the selector. To parse this part of the rule, the browser *css* parser could be used but since this work is developed using *Node*, that is not an option. Another external package can be used named ***"css-selector-parser"*** [6]. This package parses the selector part of a *css* rule and converts it into an iterable *AST* object. For example, the following selector:

```
a[href^=/], .container:has(nav) > a[href]:lt($var)
```

Is converted into:

```
{type: 'selectors',
 selectors:
  [{type: 'ruleSet',
    rule:
    {tagName: 'a',
     attrs: [{name: 'href',
          operator: '^=',
          valueType: 'string',
          value: '/'}],
 type: 'rule'}},
{type: 'ruleSet',
 rule:
 {classNames: ['container'],
  pseudos:
  [{name: 'has',
    valueType: 'selector',
    value: {type: 'ruleSet',
            rule: {tagName: 'nav', type: 'rule'}}}],
    type: 'rule',
    rule:
    {tagName: 'a',
     attrs: [ { name: 'href' } ],
     pseudos: [ { name: 'lt', valueType: 'substitute', value: 'var' } ],
     nestingOperator: '>',
     type: 'rule'}}}]}
```

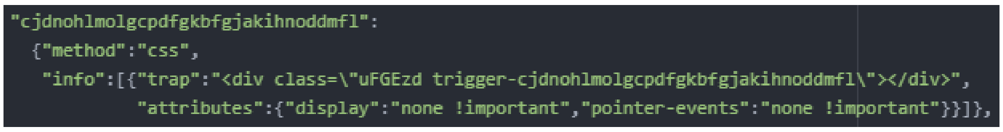
By this point, all the information needed to create workable triggers is gathered. Every single *css* rule from all the dataset extensions is iterated and a respective *HTML* element created. There were some details that needed to be taken into consideration:

- For this work, the default element tag used is the *'div'* tag. What does this mean? In cases where the *HTML* element tag is not mentioned in the selector, the trigger will be created using a *'div'* tag. The reason for this decision is that this tag is one of the most general, generic and common tags. Therefore, there probably won't exist that many websites injecting general style sheets specifically for *'div'* elements. This means that in most cases the triggers won't be manipulated by other entities besides the extension itself, which is the best case scenario.

- The trigger creation implementation is certainly not perfect, meaning that that are some types of *css* rules that can't be turned into triggers. All the main *css* selectors are covered, but not all. Examples of selectors that are not covered are the attribute targeting types (*'[attribute]'*, *'[attribute=value]'*, *'[attribute*=value]'*, among other variations) and the state selectors (*':hover'*, *':focus'*, among others). This can potentially be problematic as some extension may only inject this type of *css*. However, these are not the most commonly used selectors and even if we could create triggers for the state selectors, the triggering of the element would depend on the actions of the user to make the trigger have the correct state precisely when the detection is tested. Furthermore, not all created triggers need to be used. For these reasons, we think these issues don't have a meaningful impact on detection.

- When outputting the triggers to the final usable *JSON* the attributes values need to be included for the detection to work.
- Every trigger was created with a unique class identifier using the extension ID, so that when the detection check needs to be made, it is easier to target each extension's triggers.

One example for the output of this processing is displayed in figure 5.15. As it was explained in Section 3.4, for the second phase of implementation, every trigger on the *"trap"* attribute is invisibly injected into the detection page and then a check is made to see if this element has the correct attribute values. For this work, we consider an extension present when all the triggers are activated. For a trigger to be activated, it needs to have all the correct attribute values, this is, the attribute values that were gathered for that specific trigger in the previous phase of implementation. This is important because if not all attributes are considered, there can be cases of false positives. An element can randomly have the correct value for a certain attribute and the extension is incorrectly identified. Of course this can still happen even with multiple triggers and multiple considered attributes, but the greater the number of these elements, the lower the chance of false positives occurring. It was for this reason that in previous implementations of this method, 50 triggers for each extension was used to make the detection. However, as it is described in Section 6.2, this is not viable for massive extension detection in terms of performance.



```
"cjdnohlmolgcpdfgkbfgjakihnoddmfl":
  {"method":"css",
   "info":[{"trap":"<div class=\"uFGEzd trigger-cjdnohlmolgcpdfgkbfgjakihnoddmfl\"></div>",
            "attributes":{"display":"none !important","pointer-events":"none !important"}}]},
```

Figure 5.15: Example of the final *JSON* format for the information needed to detect through *CSS* injection.

This method's implementation detection results are presented in Section 6.1.3 and the performance analysis is done in Section 6.2.4.

## 5.5 Summary

In this stage of development the three detection methods were individually implemented. This chapter serves as documentation for every decision and step taken during implementation.

In previous work where extension detection is made there is not a clear description on how the methods were implemented. There are also no tools available where extension detection can be applied. This created the need to implement each method from scratch. In the end, all implementation were successful as each method is capable of detecting extensions. The results are further analyzed in chapter 6. These implementation can serve as baseline and starting point for future work in this field and there are multiple ways to improve on detection rate and performance. Future work possibilities are mentioned in Section 8.2.

Overall, detection by *WARs* proves to be the simplest and quickest method to implement, and it also shows to have the best results. Detection through CSS injection is the most complex of the three. This implementation requires multiple steps of iteration through the data and the interpretation of CSS rules. For this work, external tools are used to process the information but this is not the ideal scenario.

# Chapter 6

# Detection Rate and Performance Analysis

In this chapter we validate and analyse the results obtained from the implementation process. This dissertation work aims to focus not only on the detection effectiveness, but also on the performance cost and viability of each method. In cases where a website need to implement extension detection, both need to be taken into consideration so that the user experience is not harmed. Section 6.1 presents and analyses the detection rate for each method. Section 6.2 evaluates each implementation on the performance cost and concludes which methods are viable in this context. Finally, Section 6.3 summarizes this chapter's information and presents the conclusions that can be made about the results.

## 6.1 Detection Rate Results

This section details the detection effectiveness for each individual method implementation. The detection effectiveness is measured through the number of extensions from the dataset the method is able to correctly identify. For each method it is also described a way to predict the detection rate. This prediction value represents the ceiling for each implementation and in the best case scenario it represents the detection rate the method is actually able to achieve.

Each method is tested individually. A ***'selenium'*** environment is used where a single Google Chrome process is launched containing one extension installed. Then the detection script is executed. The script uses the information from the *JSON* file and checks for all the dataset extensions. The expected result is that every extension comes back negative, except for the one that is actually installed in the browser. If this check is positive then the method is able to correctly detect this specific extension. In the end, the detection rate is the number of extensions correctly detected divided by the total number of dataset extensions.

### 6.1.1 Detection through WARs

The ***'selenium'*** environment previously described is used to verify the correct execution of the method, thus validating the implementation. Depending on whether the extension contained *WARs* declared in its manifest file, the expected response is either positive or negative. The detection script makes a request for the wanted WAR and verifies whether it is present or not in the current session. As Section 3.3 mentioned, this detection method cannot originate **false positives**. However, for each extension that is tested, all the others are also tested to verify that all returned negative results. In an actual situation, the normal execution process will be testing all dataset extensions, thus it makes sense to test this way. All validation tests are performed five times to ensure that the results would always be the same. The results are displayed as one since all the tests returned the same values. The final results are shown in table 6.1.

Table 6.1: Detection statistics for *WARs* method

| | Dataset | Expected Detection | Detection |
|---|---|---|---|
| Number of Extensions | 947 | 465 | 376 |
| Percentage (%) | - | 49.1% | 39.7% |

The number of extensions expected to be detected are calculated using the number of extensions that contained *WARs* declared in the manifest. The difference of 9.4% (89 extensions) is due mainly to two reasons:

- First, as already mentioned, due to the new *manifest v3* format, extensions can make *WARs* only available in specific domains, thus becoming invisible to detection by *WARs* in any other domain.
- Second, due to browser updates, it happens regularly that specific extensions stop working and are removed from the browser web store, especially in less popular extensions. Every instance of ***'selenium'*** used to run the tests is launched with the files that had been fetched and saved locally for precisely that reason. However, in cases where the extension becomes broken and stops working, the test could not be performed and resulted in a negative detection.

However, it was possible to convert successfully 80.8% of what was expected. Furthermore, since all other cases are explained by the two reasons described above, the correct implementation of the method is validated and presents a high detection rate for the considered dataset.

### 6.1.2 Detection through DOM manipulation

The validation of this method is done similarly to what is described previously. We consider an extension to be present when precisely the same pattern of behaviors is detected. It is important to note that in the regular operation of a website, changes can also be made to the *DOM* of the page.

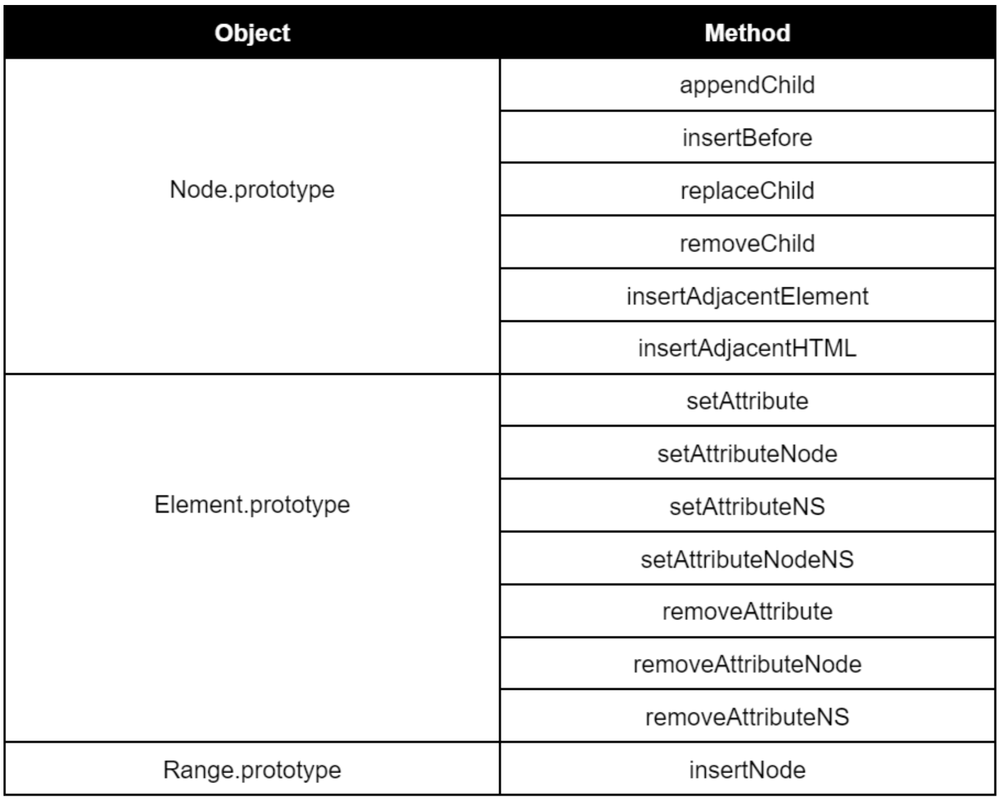
Therefore, the changes made by a particular extension are mixed with behaviors made by other entities. For this reason, an extension cannot only be considered present when all its behaviors are checked in the same order and directly followed by one another. This work also considers an extension present when all its behaviors are verified, even if the order is changed. The detection results are shown in table 6.2.

Table 6.2: Detection statistics for *DOM* manipulation method

| | **Dataset** | **Expected Detection** | **Detection** |
|---|---|---|---|
| Number of Extensions | 947 | 632 | 150 |
| Percentage (%) | - | 66.7% | 15.8% |

For the considered dataset, the technique presents a detection rate of 15.8%, being able to detect 150 of the 947 total dataset extensions. Compared to state of the art, it is a considerably high percentage which can be mainly due to two reasons: the dataset used, we can conclude that the technique is more effective within the group of the most popular extensions, and secondly; the *'honey page'* that is used to capture behavior beforehand. The detection success of this technique depends on how the extension behaviors are collected. In cases where the webpage where the detection will be made is known, the best possible scenario is to use that same page to collect the behaviors, thus having fewer variables that can make the behaviors vary.

In this case, the expected detention value was calculated by iterating over all files of all dataset extensions and, using *REGEX* to find code instances that contain operations that change the pages' *DOM*. All operations considered are shown in figure 6.1.



| Object | Method |
|---|---|
| Node.prototype | appendChild |
| | insertBefore |
| | replaceChild |
| | removeChild |
| | insertAdjacentElement |
| | insertAdjacentHTML |
| Element.prototype | setAttribute |
| | setAttributeNode |
| | setAttributeNS |
| | setAttributeNodeNS |
| | removeAttribute |
| | removeAttributeNode |
| | removeAttributeNS |
| Range.prototype | insertNode |

Figure 6.1: *DOM* altering operations that are considered to predict detection.

All extensions where at least one of these operations is found in their files are considered to make changes to the *DOM* and, therefore, could potentially be detected using this method. However, from this prediction, only **23.7%** of the detections were confirmed successfully. This is mainly due to the fact that many of these changes target specific domains and few extensions make general changes to general *HTML* elements. For this reason, the success of this method is highly dependent on how the behaviors are gathered beforehand. This is why it is very advisable to detect behaviors on the actual website where the detection will take place.

### 6.1.3 Detection through CSS injection

For this validation process, exactly the same environment is used as in the other methods. A script injects every trigger into the page and then, after a few seconds, a check is made to see if the trigger has the right attribute values assigned.

As it was previously explained in Section 5.4, an extension is considered present when all the used triggers are activated. Even though a large number of triggers are created for some extensions, only five triggers per extension are used in this test. The reason is for performance issues, which will be further discussed in Section 6.2. The way by which the five triggers are chosen is not by random selection. Priority is given to triggers that originate from more specific selectors, like IDs or classes. The more specific a selector is, the better, because it lowers the chance of causing other entities besides the extension itself to target the trigger element (causing false positives in the process).

The results for this implementation are presented in table 6.3.

Table 6.3: Detection statistics for *CSS* injection method

| | **Dataset** | **Expected Detection** | **Detection** |
|---|---|---|---|
| Number of Extensions | 947 | 188 | 62 |
| Percentage (%) | - | 19.9% | 6.5% |

The way by which the expected detection is calculated is by counting how many dataset extensions declare *CSS* files in the ***"content_scripts"*** attribute of the manifest. The difference between the expected result and the actual detection rate is mainly because of domain issues. The majority of extensions that inject some type of *CSS*, only do it in certain domains. This makes them impossible to detect using this method. Other failed detections that do not fall into this case scenario, are due to limitations in the trigger creation process. Some triggers are incorrectly created, thus making them unable to be activated by the respective extension.

Overall, this method is by far the least effective in detecting the dataset extensions. This is mainly because of the low number of extensions that actually inject any type of *CSS*. Even comparing the expected results to the other methods, it would still be the weakest approach.

## 6.2 Performance Analysis

This section presents the performance results for each implemented method. A performance analysis is carried out and the viability to apply these methods in real websites is measured. As far as the research we have done, no previous work mentions or tests the performance of extension detection methods. However, it is crucial to know what methods can be implemented alongside real websites without harming other functionalities. We consider this analysis crucial.

For this performance analysis, we test each method's performance by measuring the time it takes the browser to check for every extension, or the *script execution time*. This is achieved by checking the session date (which contains the time) in the beginning of script execution through the JavaScript method *"Date.now()"*. When all extensions are checked, we calculate the difference between the date in the beginning and at the end of execution.

### 6.2.1 Setup

All methods are implemented differently and therefore the performance cost of a method may not depend mostly on the number of extensions it has to check. For each method, several tests are performed, always varying the variable on which the performance cost of the method should be most dependent.

In the case of detection using WARs, performance is dependent on the number of extensions because of the number of fetch requests it needs to do (one per extension). To simulate a significantly higher number of extentions, the dataset can simply be duplicated enough times until the desired number of extentions is reached. The fact that there are duplicate extensions in the dataset does not affect the test results because performance is not affected by what WAR the fetch request contains.

In the DOM manipulation detection method, the variable that is changed is the number of alterations that the page undergoes, and that are caught by the *Mutation Observer*. All tests are performed with 20,000 extensions and the changes that are made to the page are always the addition and removal of one *HTML* element, for example one *div* element

Finally, in the detection by *CSS* injection, the number of triggers that are injected into the page are varied throughout the tests. This can be achieved by increasing the number of extensions or by increasing the number of triggers per extension. Triggers can be duplicated as the performance does not vary depending on whether or not the trigger is activated.

It is important to mention that every test was done under the same conditions - same hardware and same internet connection. Also, every test was done five times as the tables show.

### 6.2.2 Detection through WARs

In the detection method using WARs, the *script execution time* is measured for 20,000, 50,000 and 100,000 extensions. The results are displayed in tables 6.4, 6.5 and 6.6 respectively.

Table 6.4: Script execution times for detection of 20,000 extensions using WARs.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Script Execution Time (ms) | 3,403 | 3,295 | 3,265 | 3,328 | 3,293 | 3,317 |

Table 6.5: Script execution times for detection of 50,000 extensions using WARs.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Script Execution Time (ms) | 8,453 | 8,692 | 8,863 | 8,566 | 8,594 | 8,634 |

Table 6.6: Script execution times for detection of 100,000 extensions using WARs.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Script Execution Time (ms) | 17,478 | 17,209 | 17,336 | 17,180 | 17,185 | 17,278 |

The results show that the method performance is highly dependent and increases as the number of extensions increases. The script memory size for the three tests were respectively 2.45MB, 6.14MB and 12.2MB. Overall this method is the one that performs best by comparison. It has the capability of detecting the highest number of extensions while not having a big significant performance cost. As the results from table 6.4 show, it would be possible to implement 20,000 extensions detection while only delaying the execution of the website by 3.3 seconds on average. Having a delay of 3 seconds is not ideal and definitely impacts the user but it's viable and can be justified if there's a need to detect extensions. The impact on memory is also not significant as the script only occupies 2.45MB of memory.

These results justify the priority given to this method because of the high detection rate while having low performance cost comparing to the other two methods.

### 6.2.3 Detection through DOM manipulation

For the DOM manipulation detection method, the tests were run with 10, 100, 500 and 1,000 page alterations. The results are displayed in tables 6.7, 6.8, 6.9 and 6.10 respectively.

Table 6.7: Script execution times for detection of 20,000 extensions using DOM with 10 page alterations.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Script Execution Time (ms) | 78 | 80 | 80 | 81 | 79 | 80 |

Table 6.8: Script execution times for detection of 20,000 extensions using DOM with 100 page alterations.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Script Execution Time (ms) | 525 | 548 | 547 | 546 | 526 | 538 |

Table 6.9: Script execution times for detection of 20,000 extensions using DOM with 500 page alterations.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Script Execution Time (ms) | 2,536 | 2,543 | 2,522 | 2,504 | 2,521 | 2,525 |

Table 6.10: Script execution times for detection of 20,000 extensions using DOM with 1,000 page alterations.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Script Execution Time (ms) | 5,071 | 5,054 | 5,068 | 5,051 | 5,045 | 5,058 |

As the results show, this method's performance directly depends on the number of alterations made to the page. This means that the performance cost of implementing this type of detection depends on the website. The script memory size with 20,000 extensions was 3.76MB. This method's viability heavily depends on the website where it will be implemented. On average, it shows good promise, and can easily detect a large number of extensions without harming the user experience.

The implementation can also be optimized, potentially improving the performance. This optimization could be made on the way patterns are matched, meaning the behaviors collected could potentially be better iterated through.

### 6.2.4 Detection through CSS injection

For the CSS injection detection method, tests were run with various number of triggers: 1,250, 5,000, 25,000 and 100,000. However, the browser runs out of memory when we try to inject more than 1,300 triggers approximately. The information on script size variability depending on the number of triggers is displayed in table 6.11.

Table 6.11: Space occupied in memory by the script with different number of triggers.

| **Number of Triggers** | **Memory Size** |
|---|---|
| 1,250 | 502KB |
| 5,000 | 1.96MB |
| 25,000 | 9.84MB |
| 100,000 | 39.4MB |

Because of the need to inject entire *HTML* elements, the information stored in the *JSON* needed to implement detection is very large. This makes the script size increase significantly compared to other methods. This method is simply not viable with a large number of extensions **or** a large number of triggers per extension. Using five triggers per extension, it's only possible to test detection for around 250 extensions. The results of *script execution time* for this scenario are displayed in table 6.12. Even with a very low number of extensions, the method delays the website by almost 3.5 seconds on average. By comparison, this method performs very badly and can not be used to implement massive extension detection. Detection by *CSS* injection should only be used in particular cases where an extension can not be detected by any other method. The results justify placing this method last in priority.

Table 6.12: Script execution times for detection of 250 extensions using CSS with 1,250 triggers.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Script Execution Time (ms) | 3,484 | 3,482 | 3,520 | 3,493 | 3,469 | 3,490 |

## 6.3 Summary

After analysing the detection efficiency for each method implementation, we can conclude that detection through WARs is the better method. It presents the highest percentage of detection rate by almost double the amount of extensions that *detection through DOM manipulation* was able to identify. On the other side, detection through CSS injection presented poor results in comparison to other methods. Nonetheless, these results were somewhat better than what was achieved in previous implementations. DOM and CSS detection showed better results while detection by WARs lowered the detection rate compared to previous implementations. This is because of the new manifest format which enables extensions to hide in domains were they are not active.

A performance analysis was also conducted for each implemented method. This performance analysis focused on the *script execution time*, meaning the time it took each method to check for every extension in the dataset. Multiple scenarios and tests were made to each method to see how performance would vary depending on a certain variable.

In the end, detection through WARs proved to be the most efficient method. It is capable not only of detecting the biggest amount of extensions, as well as do it in a way that minimizes the performance cost. Detection through DOM manipulation performed reasonable well, but worst than detection by WARs. It should only be used to detect extensions that do not declare WARs. However, detection through CSS injection is by far the worst in terms of performance. When trying to inject large amounts of triggers, the browser runs out of memory and is not even able to load. It can only be used to detect very small groups of extensions and in cases where they cannot be detected using any other method.

# Chapter 7

# Final Script Implementation

This chapter serves to describe how the various methods were combined into a single detection script. In Section 7.1 the development process and the script implementation is described and in Section 7.2 the performance results are presented. Finally, in Section 7.3 a summary and conclusions are made about this chapter's information.

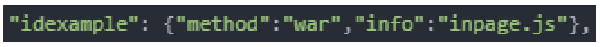
## 7.1 Implementation

By this point of development, every single method is individually implemented. Now, they need to be combined into a single script. The first step is to order each method by priority of detection meaning that, in cases where a particular extension can be detected by two or three methods, we know which one is used in the script to identify the extension. In this work, the order was defined by the performance analysis and detection rate. The order of priority is as follows:

1. WARs
2. DOM
3. CSS

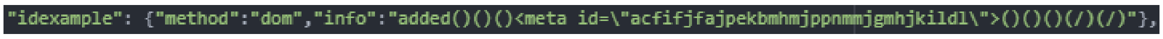
This means that every extension that is capable of being identified using WARs, is going to be detected using that method in the final script. Then, every extension that is identified using DOM manipulation, but not by WARs, is detected with DOM in the final script. And, finally, every extension that can't be identified by WARs or DOM, will be detected using CSS if possible. This ensures that the best method is always used to identify each extension.

The information on how to detect each dataset extension is stored into a *JSON* file. Each entry on the file indicates what method should be used to make the detection, and then the information needed to test it (depending on the method). Figures 7.1, 7.2 and 7.3 display respectively an example of one *JSON* entry for a WARs, DOM and CSS detectable extension.



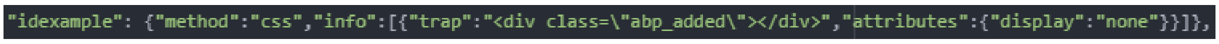
```
"idexample": {"method":"war","info":"inpage.js"},
```

Figure 7.1: Example of an entry in the JSON file of one extension detectable by WARs.



```
"idexample": {"method":"dom","info":"added()()()<meta id=\"acfifjfajpekbmhmjppnmmjgmhjkildl\">()()()(/)(/)"},
```

Figure 7.2: Example of an entry in the JSON file of one extension detectable by DOM.



```
"idexample": {"method":"css","info":[{"trap":"<div class=\"abp_added\"></div>","attributes":{"display":"none"}}]},
```

Figure 7.3: Example of an entry in the JSON file of one extension detectable by CSS.

The full *JSON* file with all the information needed to detect the dataset extensions is created using this priority order and format. The way by which the extension presence is tested in the final script is the same as described for each individual method implementation. The final step is deciding how to "feed" this *JSON* file to the script. How can the script access this information? This can be achieved by two methods:

- The file can be stored on an external server or;
- The file data can be stored in string format in the script code itself.

Each of the methods has pros and cons. The best case scenario is to minimize the performance cost for the script execution. Storing the file on an external server can delay the script execution because of the fetching of the data. On the other hand, storing the file in string format increases the script size significantly. For this dissertation, the *JSON* file was stored in the script itself in string format. In future work it would be potentially interesting to analyse which option would be best.

The *JSON* file is encoded using the *encodeURIComponent()* method and stored into a variable in string format. At the start of script execution, the method *decodeURIComponent()* is used to decode the information and use it to test detection for each extension.

## 7.2 Results

To analyse the performance of the combined script, we measure the loading time of three different websites with and without the detection script, in order to draw conclusions about the impact that the detection of extensions may have on the user experience. For this, a proxy provided by the company JScrambler was used. A proxy server serves as an intermediary in the user's connection to the internet. This way, it is possible to inject a script in the domain that we intend to test. The three websites chosen were: YouTube, Amazon and Ebay.

By analysing the results, we can conclude that, overall, implementing on-site extension detection is heavily taxing on the website's performance. At this point it should only be applied if extremely necessary and in cases where the user experience is not the top priority. However, it is still viable for a not very large number of extensions, specially using the WARs and DOM methods. The *CSS* injection method can't be used to implemented large scale extension detection. It should be used in combination with other techniques to detect particular extensions that other methods can't detect.

The final results on the performance of the combination of the three methods using the implementation described in Section 7.1 are displayed in tables 7.1, 7.2 and 7.3 where we can compare the websites loading times with and without the detection script. It is important to mention that every test was carried out using the proxy, even the tests without the detection script being injected. By having a proxy in place, the connection is delayed and so loading times increase. In real case scenarios the loading times would be lower.

Table 7.1: YouTube website loading times with and without detection script.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Website Loading Time without Detection Script (ms) | 2,840 | 2,656 | 2,759 | 2,682 | 2,741 | 2,736 |
| Website Loading Time with Detection Script (ms) | 5,079 | 5,709 | 6,800 | 6,297 | 5,395 | 5,856 |

Table 7.2: Amazon website loading times with and without detection script.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Website Loading Time without Detection Script (ms) | 3,057 | 3,065 | 3,045 | 3,045 | 3,076 | 3,058 |
| Website Loading Time with Detection Script (ms) | 6,879 | 6,578 | 7,019 | 6,474 | 6,598 | 6,710 |

Table 7.3: Ebay website loading times with and without detection script.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | Average |
|---|---|---|---|---|---|---|
| Website Loading Time without Detection Script (ms) | 2,112 | 2,223 | 2,535 | 2,482 | 2,111 | 2,293 |
| Website Loading Time with Detection Script (ms) | 3,004 | 3,146 | 3,090 | 3,140 | 3,298 | 3,136 |

As the results show, the website's loading times increase significantly and it would definitely be noticeable. Results vary depending on the website. For example, on *Ebay* the script performance cost is not as significant as on the other websites. There are multiple reasons that can influence this but it is mainly because of the number of alterations made to the page.

## 7.3 Summary

A script that combines three different extension detection methods was implemented. This is mainly achieved by the way in which information is stored in the *JSON* containing all the information needed to test detection. The priority given to the methods in cases where one extension is detectable by multiple methods is crucial. Better methods should always be used when possible.

To test this implementation, the script was injected into real complex websites to see how the detection of a dataset containing 1,000 extensions would harm the website's performance. Although the results are far from perfect as the delay is noticeable, it proved to be a viable method.

# Chapter 8

# Conclusions

In this chapter we finalize this dissertation work by answering the raised questions in the beginning and making some conclusions. In Section 8.1 we present the work contributions and conclusions and in Section 8.2 we enumerate what can be improved and possibly worked on in the future.

## 8.1 Contributions

This work aimed to implement and study the efficiency of multiple extension detection techniques. These techniques should be capable of detecting both stealthy and non-stealthy extensions. For that, an extension detection script was developed that combines three different methods: detection by WARs, *CSS* injection and *DOM* manipulation. In the end, it was found that the combination of these techniques was able to detect a very high number of extensions for the considered dataset.

Another objective was to study the performance cost of each method, and understand the feasibility of applying them on real websites without harming the users' experience. It was concluded that, at the moment, it is possible and feasible to implement extension detection, but the difference is noticeable.

The detection method using *WARs* proved to be the best method, not only in detection rate, but also in terms of performance.

The *DOM* detection method shows inferior results and is very prone to detection errors and false positives. However, the detection rate can be improved if the behavior of extensions is collected on the same web page where the detection is performed. This explains the superior detection rate values compared to previous implementations. Performance varies depending on the website. In cases where the website makes few changes to the page, the method proves to be quite efficient. However, if the website makes a lot of changes, the difference in execution starts to be quite significant. These two methods are by far the two best options for implementing on-site extension detection.

On the other hand, the *CSS* injection detection method turned out to have the worst results out of the three developed methods. Not only did it have low results in terms of detection rate, but it also proved to be very unreliable in terms of performance. This method should only be used for low numbers of extensions and in cases where these extensions cannot be detected using any other method. Even when implementing the detection of very few extensions, this method has a quite significant impact on website performance.

## 8.2 Future Work

As for future work, it would be interesting to try to optimize the *CSS* injection and *DOM* manipulation methods. Regarding the *CSS* injection method, the way the information is stored in the script could be changed and instead of storing it in a variable in the form of a *string*, it could be stored on an external server and then, instead of injecting all triggers at once, they could be added to the page in waves. In the case of *DOM* manipulation, the logic of the algorithm that implements the detection test could be optimized.

Furthermore, it would be interesting to test the efficiency of the techniques in scenarios with multiple extensions installed, thus testing the robustness of each method.

# References

[1] Chinmay Agarwal, Medhavini Kulshrestha, Himanshu Rathore, and Kamalakannan J. Security verification in web browser extensions. *SSRN Electronic Journal*, 2018.

[2] Anupama Aggarwal, Saravana Kumar, Ayush Shah, Bimal Viswanath, Liang Zhang, and Ponnurangam Kumaraguru. Spying browser extensions: Analysis and detection. 12 2016.

[3] Ahmet Buyukkayhan, Kaan Onarlioglu, William Robertson, and Engin Kirda. Crossfire: An analysis of firefox extension-reuse vulnerabilities. 02 2016.

[4] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1687–1700, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *2009 Annual Computer Security Applications Conference*, pages 382–391, 2009.

[6] Marat Dulin. Fast and low memory CSS selector parser. Parses CSS selector into object-model. https://github.com/mdevils/css-selector-parser#readme. [Online; accessed 2022-04-14].

[7] Ugo Fiore, Aniello Castiglione, Alfredo De Santis, and Francesco 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, 2014.

[8] Statcounter GlobalStats. Browser market share worldwide. https://gs.statcounter.com/browser-market-share, 2021. [Online; accessed 2022-02-19].

[9] Google. About Manifest V2. https://developer.chrome.com/docs/extensions/mv2/. [Online; accessed 2022-09-03].

[10] Google. Google Chrome, The Browser built by Google. https://www.google.pt/intl/en-US/chrome/. [Online; accessed 2022-03-01].

[11] Google. Welcome to Manifest V3. https://developer.chrome.com/docs/extensions/mv3/intro/. [Online; accessed 2022-09-03].

[12] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *2011 IEEE Symposium on Security and Privacy*, pages 115–130, 2011.

[13] Gabor Gyorgy Gulyas, Doliere Francis Some, Nataliia Bielova, and Claude Castelluccia. To extend or not to extend: On the uniqueness of browser extensions and web logins. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, WPES'18, page 14–27, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. The most dangerous code in the browser. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[15] Palak Jain. Survey on web browser and their extensions. In *Iconic Research And Engineering Journals*, volume 2, pages 41–55, 2018.

[16] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 641–654, San Diego, CA, August 2014. USENIX Association.

[17] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *NDSS*, 2020.

[18] Ankit Kariryaa, Gian-Luca Savino, Carolin Stellmacher, and Johannes Schöning. Understanding users' knowledge about the privacy and security of browser extensions. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 99–118. USENIX Association, August 2021.

[19] Aram Kocharyan. CSS-JSON Converter for JavaScript. `https://github.com/aramk/CSSJSON`. [Online; accessed 2022-04-12].

[20] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *CoRR*, abs/1905.01051, 2019.

[21] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2507–2524. USENIX Association, August 2021.

[22] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012.

[23] Microsoft. Microsoft Edge. `https://www.microsoft.com/en-us/edge`. [Online; accessed 2022-03-01].

[24] Mozilla. Firefox Browser. `https://www.mozilla.org/en-US/firefox/new/`. [Online; accessed 2022-03-01].

[25] Krishna.V. Nair and Elizabeth RoseLalson. The unique id's you can't delete: Browser fingerprints. In *2018 International Conference on Emerging Trends and Innovations In Engineering And Technological Research (ICETIETR)*, pages 1–5, 2018.

[26] Charlie Obimbo, Yong Zhou, and Randy Nguyen. Analysis of vulnerabilities of web browser extensions. In *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 116–119, 2018.

[27] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. *You've Changed: Detecting Malicious Browser Extensions through Their Update Deltas*, page 477–491. Association for Computing Machinery, New York, NY, USA, 2020.

[28] Peerigon. Unzip chrome extension files. https://github.com/peerigon/unzip-crx#readme. [Online; accessed 2022-04-02].

[29] Sudakshina Singha Roy and K. P. Jevitha. Cbeat: Chrome browser extension analysis tool. In Sabu M. Thampi, Gregorio Martínez Pérez, Carlos Becker Westphall, Jiankun Hu, Chun I. Fan, and Félix Gómez Mármol, editors, *Security in Computing and Communications*, pages 364–378, Singapore, 2017. Springer Singapore.

[30] Florian Schaub, Aditya Marella, Pranshu Kalvani, Blase Ur, Chao Pan, Emily Forney, and Lorrie Cranor. Watching them watching me: Browser extensions impact on user privacy awareness and concern. 01 2016.

[31] Broadband Search. Key internet statistics to know in 2022 (including mobile). https://www.broadbandsearch.net/blog/internet-statistics, 2022. [Online; accessed 2022-02-19].

[32] Hossain Shahriar, Komminist Weldemariam, Mohammad Zulkernine, and Thibaud Lutellier. Effective detection of vulnerable and malicious browser extensions. *Computers & Security*, 47:66–84, 2014. Trust in Cyber, Physical and Social Computing.

[33] Simov. Google Chrome Web Store HTTP Client. https://github.com/simov/chrome-webstore. [Online; accessed 2022-03-12].

[34] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, page 329–336, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Dolière Francis Somé. Empoweb: Empowering web applications with browser extensions. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 227–245, 2019.

[36] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *The World Wide Web Conference*, WWW '19, page 3244–3250, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 941–956, 2017.

[38] Statista. Adblocking: penetration rate 2021, by country. https://www.statista.com/statistics/351862/adblocking-usage/, 2021. [Online; accessed 2022-02-17].

[39] Steve Walker. 2020 Chrome Extension Performance Report. https://www.debugbear.com/blog/2020-chrome-extension-performance-report. [Online; accessed 2022-03-18].

[40] Qin Wang, Xiaohong Li, and Bobo Yan. A browser extension vulnerability detecting approach based on behavior monitoring and analysis. In Vasil Khachidze, Tim Wang, Sohail Siddiqui, Vincent Liu, Sergio Cappuccio, and Alicia Lim, editors, *Contemporary Research*

*on E-business Technology and Strategy*, pages 259–270, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[41] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 1286–1295, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee.

[42] Bin Zhao and Peng Liu. Private browsing mode not really that private: Dealing with privacy breach caused by browser extensions. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 184–195, 2015.