BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PREVENTION OF PASSIVE BROWSER FINGERPRINTING

PREVENCE PŘEDÁVÁNÍ PASIVNÍHO OTISKU PROHLÍŽEČE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JOZEF HRUŠKA**
AUTOR PRÁCE

SUPERVISOR **Ing. LIBOR POLČÁK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

Abstract

This diploma thesis aims to design a new protection layer against passive browser fingerprinting, which complements existing protection against active fingerprinting in the JShelter web extension. The protection layer changes the values of selected HTTP headers and modifies the web browser environment to avoid inconsistencies.

Abstrakt

Cílem této diplomové práce je navrhnout novou vrstvu ochrany proti pasivnímu otisku prohlížeče, která komplementuje již existující ochranu proti aktivnímu otistku ve webovém rozšíření JShelter. Navržená ochranná vrstva upravuje hodnoty vybraných HTTP hlaviček a zároveň modifikuje protředí webového prohlížeče aby zamezila případným nekonzistencím.

Keywords

fingerprint, browser fingerprint, passive browser fingerprint, privacy protection, web extension, JShelter, JavaScript, HTTP

Klíčová slova

digitální otisk, otisk prohlížeče, pasivní otisk prohlížeče, obrana soukromí, webové rozšíření, JShelter, JavaScript, HTTP

Reference

HRUŠKA, Jozef. *Prevention of Passive Browser Fingerprinting.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Libor Polčák, Ph.D.

Prevention of Passive Browser Fingerprinting

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jozef Hruška
May 8, 2023

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).

Contents

Chapter 1

Introduction

Although more than 30 years have passed since the invention of the web back in 1989 [12], it is still an integral part of our lives, possibly more than ever before. People worldwide use the web daily to work, learn, consume content and connect with others. Every day our smartphones, tablets, laptops, or even smartwatches and fridges connect to the web.

This diverse ecosystem of devices created a need to identify users who use them uniquely. The ability to uniquely identify a user on a single device or across multiple devices allows applications to personalize the content they provide or improve their security. One of the methods of user identification is browser fingerprinting, the collection of data from the browser to create a fingerprint of the device. Just as a person's fingerprint can uniquely identify that specific person, given that we have enough data, we can uniquely identify a device and its user and act on this information.

However, gathering this kind of data can be considered a violation of privacy [11], which is the primary motivation behind creating privacy-first browsers and browser extensions to fight against fingerprinting. One of the browser extensions set on improving privacy is JShelter[1] which focuses on preventing active fingerprinting.

Active fingerprinting is a method of obtaining a fingerprint, typically by running JavaScript code to collect data from browser APIs [18]. The other variant is passive fingerprinting, in which the data is collected solely from network traffic, for example, HTTP headers.

This work aims to design and implement a browser extension to prevent the collection of passive fingerprints by altering data in HTTP requests and explore possibilities of integrating this protection into JShelter. It is important to note that the altered data must be consistent with the data obtainable through active fingerprinting methods. Otherwise, the fingerprinting application could recognize an attempt to mitigate the effects of fingerprinting.

Fingerprinting is an ever-evolving field. Chapter 3 describes its nature, history, and different approaches to collecting a fingerprint. The last two sections of chapter 3 provide an outlook on threats and opportunities fingerprinting brings to the table because, as we know, nothing is purely black and white.

As this work explores the capabilities of mitigating fingerprinting with the help of browser extensions, the following chapter **??** gives the reader an insight into the inner working of extensions.

[1] JShelter is an anti-malware browser extension to mitigate potential threats from JavaScript, including fingerprinting [3] (Available at `https://jshelter.org/`).

Finally, the last chapter 4 proposes a design of passive fingerprinting protection that works consistently with active solutions to keep user-observable side effects to a minimum.

1.1 Definitions

- **Fingerprint**: A collection of device-specific characteristics gathered for the purpose of identification.
- **Fingerprinting**: The process of collecting the device information (data) to compute a fingerprint. Terms fingerprinting, device fingerprinting, and browser fingerprinting will be used interchangeably in the following chapters.
- **Fingerprinter**: An application performing fingerprinting, typically on a third-party remote server.

Chapter 2

Web browsers and related technology

Web browsers are a significant part of the everyday life of people worldwide. We use browsers for work, learning, or entertainment across various operating systems and device form factors.

This chapter describes the parts of the browser ecosystem relevant to the topic of this work. The first section 2.1 briefly explains web browsers' history, highlighting a few important milestones for browsers as they are known today. It is essential to be aware of the historical development as the constant rush to add new features and possibilities to browsers has played a significant role in giving rise to the threat of fingerprinting.

The following sections, 2.2 and 2.3, examine JavaScript and Hypertext Transfer Protocol (HTTP) as both are cornerstones of this work. Both sections provide an outlook on the role of JavaScript and HTTP in the World Wide Web.

The last section 2.4 discusses browser extensions, versatile tools that enable users to tailor their browsing experience to their unique needs and preferences. As this thesis aims to build a browser extension, the section goes more in-depth, providing a comprehensive understanding of browser extensions, their components, and technological advancements that contributed to their present state.

2.1 History of web browsers

Web browsers have come a long way since the early days of the World Wide Web. Web browsers have come a long way since the early days of the World Wide Web. From their humble beginnings as basic applications for viewing text-only web pages, modern browsers have evolved into sophisticated and versatile platforms, bridging the gap between operating systems and the web. Following paragraphs outline the evolution of browsers, highlighting a few of the most significant features added over the years.

The first browser, named WorldWideWeb[1], was created at CERN by Tim Berners-Lee only a year after he published the first World Wide Web proposal [12]. The very early browser aimed to demonstrate how the user-facing layer of the WWW (browsers) could work in the future. The browser was only available on the NeXTSTEP[2] operating system and had limited features.

[1] https://worldwideweb.cern.ch
[2] https://en.wikipedia.org/wiki/NeXTSTEP

One of the earliest features added to browsers was support for images. The Mosaic browser[3], released in 1993, was the first to allow users to view images on the web, followed by support for tables, which allowed for more complex web page layouts.

As the web gained popularity, developers needed a way to make web pages more interactive and dynamic. The birth of JavaScript programming language addressed this need. JavaScript was first introduced in 1995[4] by Netscape Communications Corporation to add interactivity to static HTML pages. Since then, it has become one of the world's most widely used programming languages, with a vast ecosystem of libraries and frameworks[5].

Nowadays, web browsers expose a variety of JavaScript APIs that allow web developers to interact with the browser and its underlying components, such as the operating system or even the hardware. As highlighted by Shauer in 2021, modern browsers now provide access to over 100 JavaScript APIs [28]. While it is true that these APIs gave the developers much-needed possibilities, they come at the cost of compromised privacy, described in detail in the following chapters.

2.2 JavaScript

In its original form, JavaScript is a client-side language executed in the user's web browser rather than on a web server [15]. It allows applications to respond to user actions in real time without making a round trip to the server. However, the rise of popularity and the fact that JavaScript is a relatively developer-friendly language meant that, eventually, runtimes that allowed running JavaScript outside the browser emerged. The most popular JavaScript runtimes include Node.js[6], Deno[7], or Bun[8].

2.2.1 ECMAScript

As the web was rapidly gaining popularity, developers started using JavaScript to create dynamic and interactive web pages. However, there was no standardization for the language, meaning different browsers implemented JavaScript differently. This made it difficult for developers to create web applications that worked consistently across different platforms and browsers [15].

In 1996, Netscape Communications Corporation, the company behind the Netscape Navigator web browser, submitted JavaScript to ECMA International[9]. This standards organization develops and maintains standards for several technologies, including programming languages. The goal was to create a standardized language version that could be implemented across different platforms and browsers.

ECMA International established a committee to develop a standardized scripting language, which was initially called ECMA-262. The standard's first version, ECMAScript 1, was released in June 1997. Since then, ECMAScript has undergone several revisions, the latest being ECMAScript 2022. Each standard version adds new features and capabilities to the language while maintaining backward compatibility with previous versions [15].

[3] https://en.wikipedia.org/wiki/Mosaic_(web_browser)
[4] https://en.wikipedia.org/wiki/Netscape_Navigator_2
[5] https://www.npmjs.com/
[6] https://nodejs.org/
[7] https://deno.land/
[8] https://bun.sh/
[9] https://www.ecma-international.org/

Although ECMAScript is technically the correct name for the language, JavaScript is used in the following chapters as the industry uses this name predominantly.

2.2.2 Web APIs

Web applications can use JavaScript Web APIs to interact with various web browser features and functionality. The number of available APIs depends on the browser. For instance, at the time of the writing, Firefox featured more than 100 Web APIs[10], some of which were experimental, and more are likely to be added. Here is a short selection of Web APIs important to the topic of this thesis [21, 28]:

- **Canvas API**: Provides a way to dynamically create and manipulate graphics and animations in a web page using JavaScript.
- **Navigator**: Provides information about the web browser or user agent in which the code runs.
- **Web Audio API**: Provides advanced audio processing and synthesis capabilities, allowing developers to create interactive audio experiences on the web.
- **WebGL**: Allows for creating and rendering interactive 3D and 2D graphics within a web browser using the computer's graphics hardware capabilities.

All the APIs listed above are commonly used in fingerprinting to identify devices and their users uniquely [18]. Fingerprinting, the main focus of this thesis, is described more in-depth in the following chapter.

2.3 Hypertext Transfer Protocol (HTTP)

HTTP is a request-response protocol that allows web browsers to request resources from web servers and receive responses containing the requested resources [25]. HTTP is an essential application-layer protocol for transmitting web pages, images, videos, and other resources across the World Wide Web.

After a user types a URL into a web browser's address bar, the browser sends an HTTP request to the server hosting the resource. This request typically contains information such as the request's method, the URL of the requested resource, and any HTTP headers or other metadata associated with the request. The server then resolves the request and sends a response containing the response status code and body.

Web applications can request browsers to send HTTP requests on their behalf with the help of 'XMLHttpRequest' (XHR)[11] or 'Fetch' API[12] APIs. These APIs enable web applications to communicate with servers asynchronously, allowing for dynamic content updates without needing a full page refresh. To optimize performance, web applications often initially transfer only a minimal amount of HTML and JavaScript with the first request. Once the basic structure, or „skeleton," is loaded, the application can request and load additional resources, such as images, stylesheets, and more complex scripts.

[10] https://developer.mozilla.org/en-US/docs/Web/API
[11] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[12] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

2.3.1 HTTP methods

The HTTP specification [27] defines various methods applications use to specify the desired action to be performed on a resource. Among the most common HTTP methods are [5]:

- **GET**: Requests a representation of the specified resource.
- **POST**: Submits data to be processed by the specified resource.
- **PUT**: Updates the specified resource with new data.
- **DELETE**: Deletes the specified resource.
- **HEAD**: Requests the headers of the specified resource without the actual data.
- **OPTIONS**: Requests the supported communication methods for the specified resource.

From the above listed, all but the POST method are idempotent, meaning the server's state will remain unchanged no matter how many identical requests the server receives.

2.3.2 HTTP headers

HTTP headers[13] are key-value pairs sent by the client and server in an HTTP request or response. They are used to transfer additional metadata about the request or response, such as the type of content, encoding, caching policies, and authentication or authorization information. An HTTP header consists of a case-insensitive name and a value separated by a colon (":") [27].

Among the most common HTTP headers are [5]:

- **User-Agent**: Provides information about the client's user agent (typically a browser), operating system, and device.
- **Accept**: Indicates the content types the client is willing or able to receive.
- **Accept-Language**: Indicates the natural language and locale that the client prefers.
- **Authorization**: Contains credentials for authentication and authorization.

As previously mentioned, web applications can utilize specific Web APIs (e.g., the Fetch API) to task the browser to send HTTP requests on their behalf. During this process, the browser automatically appends default headers to the request. However, web applications can add their own headers or modify the default headers to suit their needs.

HTTP headers play an integral role in browser fingerprinting, as they can be used to passively identify and track users based on their browser settings and preferences [18]. By examining a specific set of HTTP headers, including browser type, version, and user's operating system, trackers can create a unique fingerprint for each device and its user. This fingerprint can then be used to monitor user behavior and browsing habits. More on the role of HTTP headers in passive fingerprinting can be found in section 3.3.

[13] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

2.4 Browser extensions

Browser extensions, or add-ons, can modify and enhance the capability of a browser [6]. Users install extensions for various reasons, such as to increase security or productivity, remove ads, or modify the browser UI to their liking. Each browser implements extensions slightly differently, but to a large extent, most browsers follow the WebExtensions API standard.

This section describes the WebExtensions API standard and the recently introduced Manifest V3, which significantly affects the ecosystem of browser extensions by introducing new restrictions and altering certain extension capabilities.

2.4.1 WebExtensions API

WebExtensions API, first introduced by Google in chromium-based browsers, is a set of special-purpose APIs the browser provides to installed extensions. Firefox[14], Safari[15], and Opera[16] have become widely compatible with the WebExtensions API, making it the standard across all major web browsers. This broad adoption allowed developers to create cross-browser extensions with minor changes to the original codebase.

WebExtensions API provides a wide range of functionality for building browser extensions, including access to browser tabs and windows, manipulating web pages and user settings, and interacting with other extensions and websites [2]. This API differs from the traditional Web APIs described in the previous section. Web applications use Web APIs to interact with the content and functionality of a web page. In contrast, extensions use the WebExtensions APIs to interact with the browser itself. Some of the interfaces provided by the WebExtensions API are [2]:

- **Cookies API**: Allows to query and modify cookies. Extensions can also configure a callback function browser calls whenever a particular cookie changes.
- **DeclarativeNetRequest API**: Allows to block or modify network requests by specifying declarative rules. This API allows extensions to modify requests without giving them direct access to their content.
- **History API**: Allows interaction with the browser's record of previously visited pages. Extensions can add, remove, and query for URLs in the browser's history.
- **Tabs API**: Allows interaction with the browser's tab system. Extensions can create new, modify, or rearrange existing tabs.

The WebExtensions API is easy to use and provides consistent interfaces and functionality across different browsers. Lately, a push for increased security and privacy rules for extensions resulted in Manifest V3, described more in-depth in the following section.

2.4.2 Browser extension architecture

Extensions, in their essence, are lightweight software modules that are running in the browser. Every extension acts independently from the browser and all other installed extensions. Like regular web pages, extensions are developed using HTML, CSS, and JavaScript.

[14] https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions
[15] https://developer.apple.com/documentation/safariservices/safari_web_extensions
[16] https://dev.opera.com/extensions/apis

A typical architecture of a browser extension consists of several components that work together to achieve the desired functionality. These components include [2]:

- **Manifest file**: A mandatory JSON file in the root of the extension source directory that contains metadata about the extension, such as its name, version, description, and permissions. It also defines the various components of the extension and their roles.
- **Service workers (background scripts)**: JavaScript files that run in the background and handle the core logic of the extension. They manage persistent states, listen for browser events, and interact with Web APIs.
- **Content scripts**: JavaScript files that run in the context of a web page and have access to the DOM of that page. They enable the extension to read or modify the content of web pages and interact with user actions. Content scripts run in an isolated environment, limiting their access to the same-origin policy of the web page.
- **Popups**: An optional user interface that provides a quick and easy way for users to interact with the extension. It usually appears when a user clicks on the extension's icon in the browser toolbar. Popups are transient and close when they lose focus.
- **Options**: An optional user interface that allows users to configure the settings and preferences of the extension. The options page is accessible through the browser's extension management interface.

In summary, the typical architecture of a browser extension involves a Manifest file that points to other files this extension defines.

Manifest file

The Manifest file is the only file that must be present in every extension. The file must have the name `manifest.json` and be in the extension's root directory. Out of the wide range of configuration options, only three are mandatory, according to the Manifest file format documentation [17]:

- **manifest_version**: An integer value specifying the version of the Manifest file. Since June 2022, the only allowed value by the Chrome Web Store[18] has been "3" for all newly added extensions. The January 2024 deadline, when the Chrome Web Store stops accepting updates to existing Manifest V2 extensions, has been postponed, meaning there is no official date for the end of life of Manifest V2.
- **name**: A plain text string identifying the extension in the Chrome Web Store and the Browser UI.
- **version**: A version string consisting of one to three integers separated by dots. Version numbers must be strictly incremental as the browser uses them to decide whether an extension should be updated.

Apart from the three mandatory fields, the Manifest file supports a wide range of other configuration options, including:

[17] https://developer.chrome.com/docs/extensions/mv3/manifest/
[18] https://chrome.google.com/webstore/category/extensions

- **action**: An object defining the extension's button in the browser's toolbar. Clicking on this button opens the popup if the extension defines it. Otherwise, clicking this button will dispatch an event to the extension's background scripts.
- **background**: Used to include one or more background scripts. For newer Manifest V3 extensions, this configuration field is used to include service worker scripts. Extension service workers do not run on the main thread, meaning they do not interfere with extension content.
- **permissions**: An array of string keys specifying the granular permissions the extension requires. The browser informs the user what permissions the extension requires during installation.

In January 2022, the Chrome Web Store began transitioning to the new version of the Manifest file, Manifest V3. The Manifest V3 represents one of the most significant shifts in the extensions platform since its launch [2]. According to the team behind the Chrome browser, it is an answer to raising concerns about privacy, security, and performance of browser extensions. The most notable changes compared to its predecessor Manifest V2 are [2]:

- **Service workers**: Manifest V3 replaces background pages[19] with extension service workers, similar to traditional web service workers.
- **Network request modification**: In the outgoing Manifest V2, extensions could intercept and modify requests procedurally. In contrast, in the Manifest V3, extensions have to define rules and ask the browser engine to modify requests on their behalf. The primary motivation behind this change was privacy, as extensions could previously access requests with little or no limitations.
- **Support for Promises**: The Chrome team added Promise support to some WebExtensions APIs so that extension developers can use modern JavaScript features such as promise chains or async/await patterns.

Although welcomed from a privacy-concerned standpoint, these changes made it more difficult for developers to create extensions to protect from fingerprinting, mainly because of the network request modification differences.

```
1  [
2    {
3      "id": 10,
4      "priority": 2,
5      "action": {
6        "type": "modifyHeaders",
7        "responseHeaders": [
8          {
9            "header": "h1",
10           "operation": "remove"
11         },
```

[19] Manifest V2 extensions used background pages to run a single persistent script to manage tasks, or a state of the extension [1].

```
        {
          "header": "h2",
          "operation": "set",
          "value": "v2"
        },
        {
          "header": "h3",
          "operation": "append",
          "value": "v3"
        }
      ]
    },
    "condition": {
      "urlFilter": "headers.com/123",
      "resourceTypes": ["main_frame"]
    }
  }
]
```

Listing 2.1: An example of a declarative rule which modifies selected response headers [2].

Listing 2.1 shows an example of the new declarative rule syntax. The rule modifies request headers, specifically removing a header named `h1`, changing the value of `h2`, and adding a new header, `h3`.

At the time of the writing, Manifest V2 extensions are transitioning to Manifest V3. Although no official date is set[20], the Chrome Web Store will eventually stop accepting updates to old Manifest V2 extensions. For this reason and to avoid confusion, this thesis will only consider Manifest V3 in the following chapters.

Service workers

Extension service workers represent a significant update in how browser extensions handle background tasks, focusing on enhancing security, privacy, and performance. Replacing the previously used background scripts, service workers provide more efficient resource management by reducing memory consumption and CPU usage, better offline capabilities through caching, and enhanced security features for browser extensions, contributing to a more optimized user experience. This shift aligns with the ongoing evolution of web technologies[21] and the need for a more secure and reliable foundation for extension development.

Unlike persistent background scripts, service workers are event-driven, meaning they only run when needed. This reduction in resource consumption improves the overall performance of the browser [2]. Additionally, service workers utilize the new CacheStorage API[22] to cache resources and intercept network requests, enabling them to work efficiently even when the user is offline. This feature can enhance the responsiveness and reliability of extensions, even under poor network conditions.

Enhanced security is another significant benefit of service workers in Manifest V3 extensions. Running in a separate thread from the main browser process and having a limited scope, service workers help reduce the attack surface of extensions. With a more restrictive

[20] https://developer.chrome.com/docs/extensions/migrating/mv2-sunset/
[21] https://developer.chrome.com/docs/extensions/mv3/intro/platform-vision/
[22] https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage

API surface compared to background pages, the risks associated with potential security vulnerabilities are lowered.

Service workers cannot access the DOM API as they are running in a separate thread. Extensions that want to modify the DOM must do so in an alternative API, such as the Offscreen API[23].

Content scripts

Extension content scripts are an essential component of browser extensions that enable interaction with web pages displayed in the browser. They provide a means for the extension to read or modify the content of a web page, access the DOM, and manipulate CSS styles. Content scripts bridge the gap between the isolated extension environment and the web pages it aims to enhance, enabling a wide range of functionality, from modifying the appearance of a website to implementing custom features.

Content scripts run in the context of the web pages they are injected into but maintain a separate JavaScript environment [2]. This separation ensures that content scripts do not directly interact with the JavaScript running on the web page, preventing potential conflicts or security issues. Despite the separation, content scripts can still communicate with the web page's JavaScript using custom events or other messaging mechanisms[24].

To use content scripts in a browser extension, developers must declare them in the extension's manifest file, specifying which scripts to inject and the conditions under which they should be injected, such as the URL patterns of the target web pages. Once configured, the browser extension can seamlessly augment or modify web pages, providing users with customized experiences and extended functionality tailored to their needs.

Popups

Extension popups are standalone HTML documents associated with the extension's action button in the browser toolbar [2]. When a user clicks the action button, the browser shows a popup. Popups are closed when a user clicks either outside the popup or clicks again on the extension's action button.

Like standard web pages, the popup's HTML document can import external CSS and JavaScript files. Unlike standard web pages, however, the popup can access the WebExtensions API exposed by the browser. Of course, the extension can only access the APIs requested in the Manifest file.

Browsers wrap popups in a wrapper element to match the visual styling across all extensions. This wrapper element resizes automatically to fit the content of the popup, and extensions cannot modify their shape or other styles.

[23] https://developer.chrome.com/docs/extensions/reference/offscreen/

[24] https://developer.chrome.com/docs/extensions/mv3/content_scripts/#host-page-communication

Chapter 3

Browser fingerprinting

Fingerprinting is a process of obtaining information about a user's device to construct a digital identifier called a fingerprint. Applications can construct a fingerprint from various characteristics, such as information about the device hardware, operating system, and browser, or seemingly unimportant information, such as installed fonts or a language preference. Given that the selected characteristics are uncommon, individually or in combination with others, applications can track devices until the characteristics change enough to affect the fingerprint.

Browser fingerprinting is a subcategory where the application tries to create a fingerprint from the information available in a browser [18]. As the main focus of this thesis is browsers and browser extensions, the term fingerprinting will be used predominantly over browser fingerprinting for ease of reading. Contrary to other identification techniques like cookies that rely on a unique identifier (ID) directly stored inside the browser, browser fingerprinting is qualified as entirely stateless. It does not leave any trace as it does not require storing information inside the browser [18].

The characteristics to construct a fingerprint may be collected either actively, typically by running a JavaScript code inside a browser, or passively by monitoring the network communication. Most characteristics commonly used for fingerprinting are obtainable only through the browser Web APIs [30]. For this reason, the majority of available anti-fingerprinting solutions focus primarily on mitigating the active fingerprinting.

The first section, 3.1, briefly tells the history of browser fingerprinting, offering insight into its development over time. The following section, 3.2, examines the impacts of fingerprinting, emphasizing the importance of recognizing its dual nature. Then, section 3.3 describes the difference between active and passive fingerprinting, while section 3.4 describes the most common fingerprinting countermeasures. Finally, section 3.5 covers the existing software solutions used to combat fingerprinting.

3.1 History of browser fingerprinting

Laperdrix et al. [18] provide an insight into the history of browser fingerprinting. In 2009, Johnathan R. Mayer investigated [20] if remote servers could exploit the differences presented by browsers to deanonymize or uniquely identify their users. He conducted an experiment where he collected the contents of the `Navigator`, `Screen`, `Navigator.plugins`, and `Navigator.mimeTypes` objects of browsers that connected to the website of his experiment. Once collected, the contents were concatenated, hashed using the MD5 algorithm,

and sent to a remote server for storage. He managed to uniquely identify 1278 out of 1328 clients (96.23%) in this experiment. However, he added that the scale of his experiment was too small to draw any meaningful conclusions and that the hashing-based algorithm masks which properties contributed to the uniqueness.

A year later, in 2010, Peter Eckersley pushed the research of fingerprinting further when he conducted a Panopticlick experiment [14]. He obtained over 470 thousand fingerprints in this experiment, uniquely identifying 83.6%. The number of identified browsers was even higher if browsers supported Flash or Java. Out of these browsers, he was able to identify 94.2% uniquely.

Since these two works were published, fingerprinting became a standard in the industry mainly because of easy-to-use solutions like FingerprintJS[1], which has an open-source core. This broad and rapid adoption of fingerprinting practices brought severe security and privacy implications [11], resulting in the growing popularity of solutions focusing on mitigating fingerprinting.

3.2 Understanding the impacts of fingerprinting

The practice of fingerprinting is widely recognized for its potential invasion of privacy. However, its impacts are not simply positive or negative. This chapter explores the various effects of fingerprinting, including both its benefits and drawbacks. Additionally, real-world examples are provided to illustrate these impacts in practice.

3.2.1 Benefits of fingerprinting

The ability to uniquely identify a device and, by extension, its owner allows third parties to modify the behavior of their services and applications. One essential application of browser fingerprinting is to enhance security and user experiences.

Specifically, applications can utilize fingerprinting for payment fraud prevention and detection [9]. By doing so, they can identify and block users who attempt to use stolen credit, debit, or checking information to make illicit transactions. In this context, fingerprinting serves as an additional layer of protection for both businesses and consumers, ensuring that financial transactions remain secure and trustworthy.

3.2.2 Drawbacks of fingerprinting

Fingerprinting, despite its numerous benefits, has gained popularity primarily due to its potential to violate users' privacy [11]. Tracking individuals over extended periods is possible until the device's characteristics change sufficiently to alter the fingerprint. This pervasive surveillance capability has raised concerns about balancing utility and personal privacy.

When a person is being tracked, applications can display advertisements or products tailored to the user based on their browsing history and online activity. This targeted marketing approach is designed to boost business sales, but it further highlights the potential intrusion of privacy associated with fingerprinting.

[1] https://fingerprint.com

3.3 Approaches to fingerprinting

Based on the type of data collected, browser fingerprinting can be categorized as either passive or active [16]. This section briefly explains both, including examples of inconsistencies caused by incorrect manipulation done by protection tools that the fingerprinting application can detect.

3.3.1 Active fingerprinting

When a fingerprinting application has to explicitly run code to obtain the data to construct a fingerprint, it is considered active fingerprinting [16]. Examples of such code are port scanning executed on a remote machine or JavaScript code executed inside the target device's browser.

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML,
    like Gecko) Version/15.6 Safari/605.1.15
```

Listing 3.1: An example of Safari User-Agent string.

Running a JavaScript code in a browser is a potent tool to collect data about the software and hardware on the device. Modern browsers provide Web APIs to give websites native-like capabilities, often allowing developers to access detailed data about the system's underlying components [30]. A commonly misused Web API is the Navigator[2] interface which represents the state and the identity of the user agent running the script. Another example is the Screen[3] interface which represents a screen, usually the one on which the current window is being rendered. Scripts often collect binary data to complement the textual data collected from other Web APIs. For example, it is typical for a fingerprinting application to use the Canvas API[4] to draw an image to test how the system renders text, emojis, or shapes. Even slight differences in the rendered image can help the fingerprinting application identify the device [30].

Table 3.1 shows a few examples of information obtainable through browser APIs. The fingerprinting applications prefer characteristics that change less frequently or are entirely static. For example, the battery percentage will likely change often, which can affect the resulting fingerprint. On the other hand, the number of logical processors will remain the same for the device unless it is virtualized or the user replaces its CPU.

[2] https://developer.mozilla.org/en-US/docs/Web/API/Navigator
[3] https://developer.mozilla.org/en-US/docs/Web/API/Screen
[4] https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

Name	Source (JavaScript API)	Example value
Available memory	`Navigator.deviceMemory`	`8`
Battery level	`BatteryManager.level`	`0.94`
Cookies enabled	`Navigator.cookieEnabled`	`true`
Number of log. processors	`Navigator.hardwareConcurrency`	`8`
Preferred languages	`Navigator.languages`	`["en-US", "en-GB"]`
PDF display support	`Navigator.pdfViewerEnabled`	`true`
Screen width	`Screen.width`	`1920`
Screen height	`Screen.height`	`1080`
User-Agent string	`Navigator.userAgent`	See listing 3.1.

Table 3.1: An example of data available from JavaScript APIs of a browser.

Network requests include fewer suitable characteristics than active fingerprinting with access to browser Web APIs. However, passive fingerprinting is not observable from the device as there is no need to execute a script in the browser. This lack of observability makes detecting fingerprinting very complicated, which is why this fingerprinting approach might be preferred over active fingerprinting.

3.3.2 Passive fingerprinting

Passive fingerprinting collects information needed to create a browser fingerprint from data observable in network traffic. As the name suggests, this approach does not require running a script directly in the browser, but instead, it only consumes incoming network traffic and picks the information it needs.

Passive fingerprinting typically collects data from HTTP request headers or other parts of incoming network traffic [16]. Table 3.2 shows examples of data directly available in request headers commonly used in passive fingerprinting.

Name	Source (HTTP/HTTPS header)	Example value
Available memory	`Device-Memory`	`8`
Content encoding	`Content-Encoding`	`gzip, deflate, br`
Preferred languages	`Accept-Language`	`en-US,en-GB,en;q=0.9`
Referer address	`Referer`	`https://amiunique.org/`
User-Agent string	`Navigator.userAgent`	See listing 3.1.

Table 3.2: An example of data available in HTTP request headers.

Network requests include fewer suitable characteristics than active fingerprinting with access to browser Web APIs. However, fingerprinting is not observable from the device as there is no need to execute a script in the browser. This lack of observability makes detecting fingerprinting very complicated, which is why this fingerprinting approach might be preferred over active fingerprinting.

3.3.3 Inconsistencies

Fingerprinting countermeasures described in section 3.4 often modify browser characteristics to affect the fingerprint and confuse the fingerprinter. However, some characteristics are fully or partially obtainable by both active and passive fingerprinting. In other words, these characteristics have multiple data sources. When a characteristic has multiple data sources, protections manipulating its value must apply these modifications to all data sources to avoid introducing inconsistent (conflicting) pieces of information. A piece of inconsistent information has a valid but conflicting value between various data sources [30].

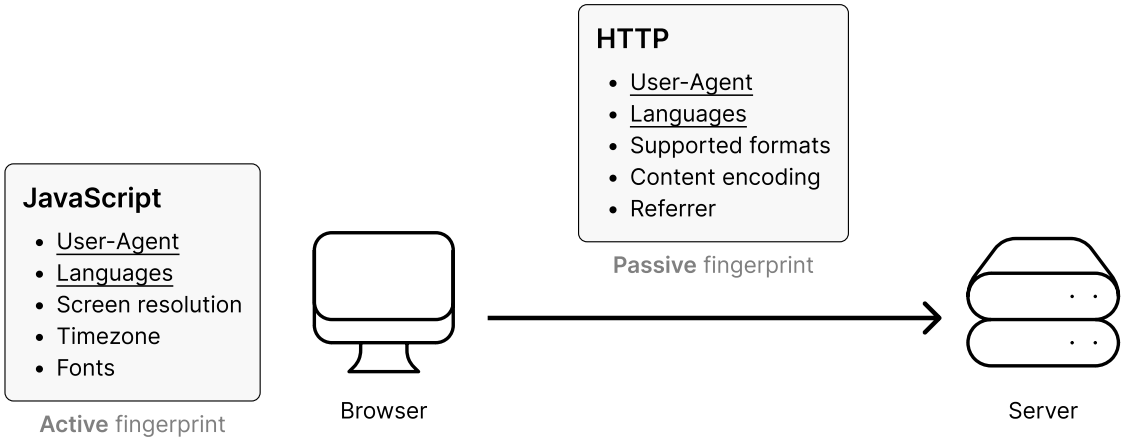


Figure 3.1: A scheme highlighting common properties obtainable with active and passive fingerprinting.

Among the examples shown in tables 3.1 and 3.2 are characteristics with multiple data sources, namely the available memory, preferred languages, and the User-Agent string. Values of specific characteristics, such as the available memory or the User-Agent string, are one-to-one copies meaning the countermeasure algorithm can apply the same transformation to all data sources. However, this is not the case for the values of preferred languages characteristic. Preferred languages transferred in HTTP headers include the quality factor[5] for each language tag, but this value is absent in the version obtainable from the browser Web API. Tables Tables 3.1 and 3.2 illustrate this difference. An algorithm cannot apply the same transformation for all data sources as it has to be aware of these differences in representation.

Suppose a countermeasure algorithm misapplies the transformation to a data source, or the data source is left unchanged entirely. In that case, the fingerprinting application can detect and act on this attempt at fingerprint manipulation.

Additionally, some information can be indirectly derived from other available data. Even if the primary data sources have been modified, specific characteristics can still be inferred from secondary sources or through indirect analysis. For example, by enumerating the installed fonts, one can derive the operating system and, in some cases, also its version [19].

[5]Quality weights or quality values are used in HTTP headers to specify the order of priority in a comma-separated list (https://developer.mozilla.org/en-US/docs/Glossary/Quality_values).

3.4 Fingerprinting countermeasures

Countering fingerprinting may be challenging, but it is feasible to a certain extent. Fingerprinting countermeasures are typically categorized followingly [16, 22]:

- **Creating a homogenous fingerprint**: Fingerprinting attempts to find differences between devices big enough to identify them uniquely. Hypothetically, if every device in the world had the same characteristics (and thus the same fingerprint), identification would not have been possible because each device would be indistinguishable from others. However, every valuable device characteristic exposed to the fingerprinting application must be the same to create absolute homogeneity. If one or more characteristics differ, the anonymity set breaks into smaller ones, making the identification possible again. As Mayer discovered in his work [20], systems with fewer customization options create larger anonymity sets due to the need for more differentiating characteristics.

- **Altering the fingerprint for each domain to prevent cross-domain tracking**: Values of characteristics commonly used to construct a fingerprint are adjusted for each domain, making tracking the device across multiple domains challenging. Algorithms performing these alterations are doing so in a seemingly random way to avoid being detected by the fingerprinting application. Crucially, alterations must be subtle enough not to impact website user experience or functionality significantly. Because of their subtle nature, these alterations are also referred to as "little lies".

- **Altering the fingerprint for each session to prevent cross-session tracking**: Similar to the previous cross-domain countermeasure, but values of characteristics are changed for each browsing session. In other words, the fingerprint is different whenever the browser is closed and reopened again. Compared to the previous cross-domain countermeasure, the fingerprint typically changes less frequently.

- **Detecting and blocking fingerprinting**: Protection tools implementing this countermeasure attempt to detect fingerprinting in real-time, block access to the Web APIs commonly misused for fingerprinting, or limit the page's ability to upload the fingerprint. This approach is strict and potentially intrusive as it can limit the functionality of the visited application, or, in the worst scenario, it could break the page completely. Some tools, most notably the uBlock Origin extension[6], come with a predefined set of URLs to block.

Sometimes, applications implementing these countermeasures combine two or more categories for overall protection (e.g., cross-domain and cross-session tracking). The type of application is also significant. Usually, the closer the algorithm is to the operating system's core, the more capabilities it has to fight fingerprinting. Browser extensions are more restricted than browsers, which are more restricted than the operating systems. Generally, operating systems do not directly combat fingerprinting. However, specific fingerprinting techniques, like port scanning, cannot be addressed by applications like browsers and may require operating system-level solutions.

It is also important to note that it is very complicated to counter fingerprinting fully. The process of collecting the fingerprint can be detected, but it is impossible to verify

[6] https://ublockorigin.com/

if changes made to the data affected the fingerprint. Current protection tools are only mitigating the threat rather than eradicating it completely.

3.5 Existing anti-fingerprinting software

In response to the growing concerns discussed in section 3.2, numerous anti-fingerprinting software solutions have been developed to mitigate this threat [10]. With a focus on web-based applications, these solutions can be integrated directly into browsers or installed as browser extensions.

This section outlines selected existing anti-fingerprinting software solutions, examining their strengths and weaknesses.

3.5.1 Tor browser

Tor[7], built on top of Mozilla Firefox, is specially designed for privacy and anonymity. Among other protections, it incorporates a variety of anti-fingerprinting measures, including:

- **First-party isolation**: To prevent cross-site tracking, Tor segregates cookies, caches, and other browsing data on a per-domain basis by implementing first-party isolation.
- **API removal (restriction)**: By disabling or restricting access to browser features commonly used in fingerprinting, such as WebGL and WebRTC, Tor effectively reduces the identifiable surface area.
- **Homogeneity**: Tor promotes user homogeneity by actively spoofing the User-Agent string to make all Tor Browser users appear identical and enforcing a standard window size across all instances, preventing screen size-based fingerprinting. These measures restrict trackers' ability to distinguish individual users and protect user privacy during browsing sessions.

These countermeasures are sometimes extraordinarily strict and limiting, which is precisely why they are very effective.

3.5.2 Brave browser

Brave[8] is another privacy-focused browser, this time based on the Chromium project. It includes several built-in fingerprinting protection mechanisms:

- **Tracking protection**: Brave enhances user privacy by blocking third-party cookies and tracking scripts, effectively reducing the ability of trackers to monitor user activity across websites.
- **API removal (restriction)**: Brave limits access to browser features often used for fingerprinting, such as battery status and device sensors, to minimize the identifiable information exposed.

[7] https://www.torproject.org/
[8] https://brave.com/

- **Little lies (farbling)**: By randomizing specific browser characteristics, like the Canvas API or available fonts, Brave makes fingerprinting more challenging and helps users blend in with a larger pool of browser configurations.

Brave protections are less severe than Tor's, offering users more flexibility and customization options. However, this approach makes achieving complete homogeneity among users harder, possibly increasing susceptibility to fingerprinting or other privacy-invading techniques.

3.5.3 JShelter

JShelter, evolved from the work of Timko [29] and Červinka, is a browser extension focused on fingerprinting prevention, limitations of rich web APIs, prevention of attacks connected to timing, and learning information about the device, the browser, the user, and surrounding physical environment and location. JShelter is available on three of the most popular browser extension stores – Google Chrome Web Store, Mozilla Add-ons and Opera Add-ons Store.

JavaScript Shield

The pivotal protection feature of JShelter is the JavaScript Shield [4], which modifies the behavior of the JavaScript environment of the browser. JShelter provides fake information to confuse fingerprinters and make webpage-triggered attacks harder or impossible.

The JavaScript Shield internally consists of small independent pieces of code called wrappers, which modify the original behavior of JavaScript APIs available in the browser. The behavior of wrappers can be categorized followingly [4]:

- **Precision reduction**: The original value is unnecessarily precise for most use cases. JavaScript Shield modifies the values so that typical and essential use cases are unaffected.
- **Providing fake information**: Some wrappers provide fake information mostly to confuse fingerprinters. For example, canvas wrappers modify the image so that the exact instructions produce a different result in each session and for each domain.
- **Hiding information**: JavaScript Shield hides information from selected browser APIs that are generally unnecessary or have little use. Depending on the API, it might return an error, an empty value, or block the API completely.

To modify the values of the browser characteristics, the JavaScript Shield uses a technique called *farbling* [16]. This technique is based on the implementation of farbling in Brave [13], which has its roots in previous privacy research, including the PriVaricator [22] and FPRandom [17].

Farbling first deterministically generates a per-session, per-eTLD+1[9] seed. This seed is later used to slightly randomize the output of web APIs typically used for fingerprinting. The way a seed is generated makes cross-site and cross-session tracking difficult for fingerprinting applications.

[9] eTLD+1 is a part of a domain name, including TLD, eTLD and one more domain part (More on https://jfhr.me/what-is-an-etld-+-1/).

Chapter 4

Design proposal

This work aims to design and implement a browser extension on the latest Manifest V3 framework capable of misinterpreting HTTP request headers, information commonly used for passive fingerprinting. Initially, this thesis aimed to implement an additional protective shield of the JShelter extension, which would protect against passive fingerprinting. However, due to JShelter's ongoing migration from Manifest V2 to V3, it was decided to create a standalone extension instead. This new extension aims to demonstrate the feasibility of combating passive fingerprinting within the more restrictive constraints of the Manifest V3 framework.

In this chapter, the first section 4.1 emphasizes the importance of avoiding alterations that could disrupt the user experience. Section 4.2 describes the design of individual protections that modify HTTP headers, discussing the reasoning, presenting examples, and detailing the steps needed to ensure consistency.

4.1 Minimizing undesired side effects

Whenever a user loads a webpage, the browser is responsible for constructing the outgoing HTTP request. Headers included in these requests provide valuable additional information servers use to decide on the response. By default, browsers set the headers according to their best knowledge to maximize speed and compatibility.

Protections that modify header values have to do so carefully. No matter how small the change is, it can affect the server's response. For example, English is the prevalent language used on the web. To make the fingerprint look more homogenous, the protection could modify the `Accept-Language` header to report English as the preferred language instead of Czech, the user's original language. However, the user may need help understanding the content if the server reacts and actually returns a website in English.

On the other hand, the alterations made to header values should be substantial enough to obfuscate the device's fingerprint effectively. Striking the right balance is essential in ensuring the implemented protections can prevent fingerprinters from tracking the device while avoiding unintended consequences or errors. Each protection is designed to prevent such unintentional errors while keeping the capability to confuse fingerprinters.

4.2 Fingerprinting protections

The browser extension comprises several independent modules, each implementing passive fingerprinting protection for a single HTTP header or a group of HTTP headers. This separation of concerns allows users to enable only some protections while keeping others disabled.

Each protection modifies the values of the headers it manages every time the browser starts, safeguarding users from cross-session tracking. However, the extension doesn't update header values when switching between different domains during a single browsing session, which may expose users to tracking within that session (cross-domain).

4.2.1 Accept

The `Accept` request HTTP header indicates which content types the client can understand, expressed as MIME types [5]. It is possible to add or remove MIME types to affect the fingerprint. However, the server may react to this modification and respond with an unexpected and incompatible response body.

```
Accept: text/html, application/xhtml+xml, application/xml;q=0.9, image/webp,
    */*;q=0.8
```

Listing 4.1: An example of Accept header contents [5].

The protection algorithm removes certain MIME types, which do not result in incompatibility, but only a slight performance hit. For example, eliminating modern image formats such as `image/webp` and forcing the browser to use older image formats is possible. This transformation would affect the header value (and, transitively, the device's fingerprint) without potentially breaking the website.

Consistency

Browsers no longer offer a standard API for quickly checking supported MIME types. The `Navigator.mimeTypes` API that partially allowed this check in the past has been deprecated and removed from the relevant web standards [7]. Due to the absence of a standardized API, determining supported MIME types is a complex task for both the protection and fingerprinting applications. As a result, it is not necessary to make any modifications to Web APIs.

Additionally, it's important to mention that a fingerprinting application might keep a record of browser versions and their corresponding supported MIME types. Then, the application could then use this list and the user's browser version to compare the MIME types reported in the 'Accept' HTTP header, potentially uncovering an inconsistency.

4.2.2 Accept-Language

The `Accept-Language` request HTTP header indicates the natural language and locale that the client prefers [5]. It is a list of language tags separated by commas (","). A language tag consists of a 2-3 letter base tag that identifies the language, optionally followed by a dash ("-") and a subtag, usually specifying the country or the region where the language is used. Language tags can optionally have a quality value (weight) represented by a decimal number separated from the tag by a semicolon (";").

```
Accept-Language: fr-CH, fr;q=0.9, en;q=0.8, de;q=0.7, *;q=0.5
```

Listing 4.2: An example of Accept-Language header contents [5].

Modifying the `Accept-Language` header is not straightforward because the web server might respond with content in a different language than the user requested. The protection only makes minor changes and works in two modes:

- **Generalization**: Language regions (subtags) are removed, and only the base language tags are kept unchanged. Listing 4.3 shows the result of generalization for the `Accept-Language` header in listing 4.2. The algorithm also deduplicates language tags and updates quality values after it removes the language regions.
- **Randomization**: Language regions (subtags) are changed for multiregional languages like English or Spanish. Listing 4.4 shows the randomization result for the `Accept-Language` header in listing 4.2.

The reasoning behind these transformations is that the difference between language forms is usually manageable for the users. For example, changing "en-US" (United States English) to "en-GB" (Great Britain English) might get noticed by users, but they would still understand the content.

```
Accept-Language: fr, en;q=0.9, de;q=0.8, *;q=0.5
```

Listing 4.3: The result of the "generalization" mode transformation of listing 4.2.

```
Accept-Language: fr-BE, fr;q=0.9, en;q=0.8, de;q=0.7, *;q=0.5
```

Listing 4.4: The result of the "randomization" mode transformation of listing 4.2.

Consistency

The client-preferred languages are also available through the `Navigator` interface under the `Navigator.languages` property. The languages are ordered the same way as in the `Accept-Language` header but without the quality value (weight). The first language in `Navigator.languages` is also extracted to the `Navigator.language` property. Any change to the `Accept-Language` HTTP header must be reflected in these two properties of the `Navigator` interface.

4.2.3 Device-Memory

The `Device-Memory` request header indicates the approximate amount of available RAM on the client device. The possible values this header can carry are `0.25`, `0`, `5`, `1`, `2`, `4`, and `8` [5]. A preview of this header is displayed in Figure 4.5. Even though this header is still experimental, it is already implemented by the Chrome browser.

```
Device-Memory: 1
```

Listing 4.5: An example of Device-Memory header contents [5].

The protection algorithm randomly picks one of the possible values and uses it as the value of the `Device-Memory` header.

Consistency

The value of the `Device-Memory` header is also accessible through the `Navigator.deviceMemory` property. Any transformation to the `Device-Memory` header must be copied to this property.

4.2.4 User-Agent and client hints

The `User-Agent` request HTTP header is a string that lets servers and network peers identify the application, operating system, vendor, and version of the requesting user agent [5]. The value of this header varies depending on the operating system and browser type and version, making it a valuable fingerprint component.

Figures 4.6 and 4.7 show examples of `User-Agent` strings of Chrome and Safari, which according to Statcounter GlobalStats[1], are the top two browsers with the highest market share of 64.62% and 18.29%, respectively.

```
Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/51.0.2704.103 Safari/537.36
```

Listing 4.6: An example of Chrome User-Agent string [8].

```
Mozilla/5.0 (iPhone; CPU iPhone OS 13_5_1 like Mac OS X) AppleWebKit/605.1.15
    (KHTML, like Gecko) Version/13.1.1 Mobile/15E148 Safari/604.1
```

Listing 4.7: An example of Safari User-Agent string (mobile version) [8].

These examples indicate that some browsers and operating systems have version numbers following the semantic versioning specification [26] or its variation. Semantic versioning in the most recent version 2 is composed of three segments separated by a dot (“„.“”), where:

- The first segment (MAJOR) is incremented when an incompatibility, such as a breaking API change, is introduced.
- The second segment (MINOR) is incremented when new functionality is added while maintaining backward compatibility.
- The last segment (PATCH) is incremented for all other changes, for example, a bug fix or code refactoring.

Based on the observation that some software versions in the `User-Agent` string follow the semantic versioning specification, we can assume that:

1. Software builds with version numbers that differ only in the last segment (PATCH) are expected to have the same functionalities. In other words, the user should not be able to distinguish any difference between them in terms of features and capabilities.
2. A change (either an increment or a decrement) of the (PATCH) segment value would likely affect the device’s fingerprint. However, at the same time, it is unlikely that the change would cause any undesired side effects.

[1]Statcounter GlobalStats collects statistics from more than 1.5 million sites with yearly views ranging from 5 to 6 billion. These statistics are available at https://gs.statcounter.com/.

The protection cannot apply changes at random, as this could introduce invalid version numbers that have yet to be released. Invalid version numbers could be used by the fingerprinting application to discover an attempt at fingerprint manipulation. Instead, the extension keeps a list of previously released versions gathered from the changelog[2] of the specific software product. The algorithm then randomly selects a suitable version that only differs in the **PATCH** segment and should have the same functionality as the original version.

```
Sec-CH-UA: " Not A;Brand";v="99", "Chromium";v="96", "Google Chrome";v="96"
Sec-CH-UA-Mobile: ?0
Sec-CH-UA-Full-Version: "96.0.4664.110"
Sec-CH-UA-Platform: "Windows"
```

Listing 4.8: An example of HTTP Client hints [5].

The `User-Agent` is not the only HTTP header from which the fingerprinter can gain detailed information about the device's browser and platform. The same information can also be extracted from HTTP Client hints. Client Hints are a set of HTTP headers that enable websites to request and receive specific information about a user's device and preferences directly from the browser [5]. Client Hints can be broadly classified into two categories based on their entropy levels:

- **Low entropy client hints**: These headers provide limited variability and do not significantly contribute to unique browser fingerprints. Examples of low entropy headers include `Save-Data`, `Viewport-Width`, and `DPR` (Device Pixel Ratio). These headers usually pose a low risk to user privacy.
- **High entropy client hints**: These headers provide a higher level of variability and may be used to create more accurate browser fingerprints, posing a higher risk to user privacy. Examples of high entropy headers include `Sec-CH-UA` (`User-Agent`), `Sec-CH-UA-Platform-Version`, and `Sec-CH-UA-Full-Version` (full browser version).

From the fingerprinting point of view, the high entropy values are more important because they tend to change less often, if ever. The extension modifies selected client hints carrying the characteristics which can also be found in the `User-Agent` header. This additional layer ensures consistency across all HTTP headers and better protection against fingerprinting.

Consistency

The `User-Agent` string can also be accessed through the JavaScript `Navigator` interface. This value is identical to the one transmitted in the `User-Agent` HTTP header when a client makes a request to a web server. To maintain consistency between the two sources of the `User-Agent` string, applying the same transformation or parsing logic to the value obtained from `Navigator.userAgent` is crucial. This consistency ensures that the information is treated uniformly and helps to prevent a potential discovery by comparing the strings.

[2]Software products typically maintain a list of published versions. For example, the changelog of Chrome is available at https://chromereleases.googleblog.com/search/label/Stable%20updates.]

```
navigator.userAgentData
  .getHighEntropyValues([
       "mobile"
    "platform",
    "uaFullVersion",
  ])
  .then((ua) => {
    console.log(ua);
  });
```

Listing 4.9: An example the User-Agent Client Hints API usage [5]. This API call returns the same values as the HTTP client hints in example 4.8.

To also ensure the HTTP client hints are consistent with the browser environment, the extension must modify the return values of the `navigator.userAgentData`[3] API. Although this API is still marked as experimental, it is already supported by Chromium-based browsers. In fact, the only major browser which doesn't support this API is Firefox.

4.2.5 Extra HTTP header

One of the valuable characteristics used to construct a fingerprint is the list of all headers present in a request. Most requests share standard headers like those mentioned in previous sections. However, it is common for HTTP requests to include less standard, even custom[4] headers.

```
Accept, Accept-Encoding, Accept-Language, Connection, Cookie, Host, User-Agent
```

Listing 4.10: An example of a list of headers present in an HTTP request.

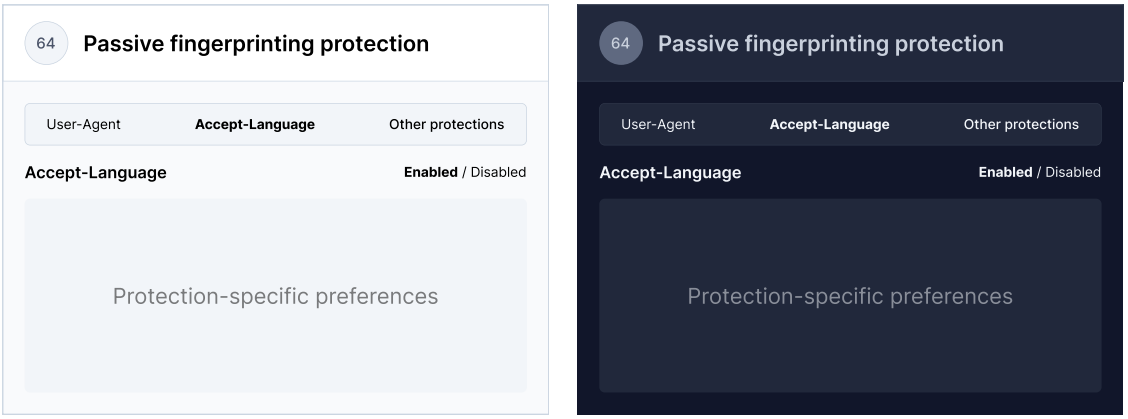
The protection adds a new header to the HTTP request to alter the list and order of headers and transitively the device's fingerprint. Additionally, if the newly added header is custom and not standardized by the protocol, this change would likely not affect the response.

4.3 Graphical user interface

The popup is the only graphical user interface the extension exposes. This view contains preferences users can use to change the behavior of the fingerprinting protections.

[3] https://developer.mozilla.org/en-US/docs/Web/API/User-Agent_Client_Hints_API

[4] Until 2012, developers tended to prefix custom headers with "X-" to differentiate them from standardized headers, but this naming convention has been officially discouraged in RFC6648 [24].



(a) Light mode wireframe.

(b) Dark mode wireframe

Figure 4.1: A comparison of light and dark mode wireframes of the browser extension.

The preferences are split into three tabs (views). This division allows for more prominent controls and descriptive hints that help users understand what each option does. The first two tabs contain preferences for the most influential HTTP headers, `User-Agent` and `Accept-Language`. The last tab groups other, less significant protections, such as `Device-Memory`.

Each protection can be enabled or disabled individually from others. When enabled, a preview field will appear under the protection's preferences. The preview field shows the header value as it will be sent in future outgoing requests and is updated automatically according to the selected preferences.

Modern operating systems have the option to switch the user interface to a dark mode which complements the standard light color scheme. The extension's popup view and every component it renders are made in light and dark variants. The extension automatically switches between these variants based on the operating system's active color scheme. Figure 4.1 displays browser extension wireframes in both light and dark modes.

Chapter 5

Implementation

5.1 Development environment and tools

This section discusses the programming languages, libraries, frameworks, and other tools used to develop the browser extension. The rationale behind choosing these technologies and how they contribute to the effectiveness of the extension is also explained.

5.1.1 TypeScript

TypeScript[1], a statically typed superset of JavaScript, was chosen as the primary programming language. TypeScript provides type safety, which helps catch errors during development and makes the codebase easier to maintain and refactor. Moreover, TypeScript's advanced type features improve code readability and provide better support for modern IDEs, enhancing the development experience.

5.1.2 React

React[2], a popular JavaScript library for building user interfaces, was employed for creating the extension's UI components. React's component-based architecture promotes modularity and reusability, simplifying the development process and ensuring maintainable code. Furthermore, React's virtual DOM enhances the performance of the browser extension by reducing the need for frequent DOM manipulations, which can be resource-intensive.

5.1.3 Radix UI

Radix UI[3], a set of low-level, unstyled, accessible, and composable UI components for React, was integrated into the project to streamline the browser extension's UI development. Using Radix UI components ensures that the extension's UI adheres to accessibility best practices and provides a solid foundation for customization. Thanks to its focus on accessibility, Radix UI offers a consistent user experience across different platforms and devices.

[1] https://www.typescriptlang.org/
[2] https://react.dev/
[3] https://www.radix-ui.com/

5.1.4 Tailwind CSS

Tailwind CSS[4], a utility-first CSS framework, was utilized to style the browser extension's user interface. Tailwind CSS enables rapid UI development by providing predefined CSS utility classes, which can be easily combined to create custom styles. This approach reduces the need for writing custom CSS, leading to a more streamlined development process and a consistent design language across the extension. Although Tailwind CSS offers a wide variety of utility functions, only the ones used by the extension make it to the final bundle, keeping the size low.

5.1.5 Vite

Vite[5], a modern build tool and development server, was selected to handle the project's build process and improve the development experience. Vite offers fast development server startup and on-demand compilation, resulting in a more efficient and productive development environment. Additionally, Vite supports various output formats, making integrating the browser extension into different browsers easier.

The `vite-plugin-web-extension`[6] installed on top of Vite allows for easier creation of cross-browser extensions with support for ES Modules and HMR[7].

5.2 Browser extension components

This section provides a detailed breakdown of the main components that make up the browser extension, explaining their functionality and how they interact with one another.

5.2.1 Storage and data management

The extension uses the `storage` API to store and persist user settings and extension state. This API bears similarities to the conventional Web Storage API[8], with an important distinction being that its scope is not bounded to a domain but is instead associated with the extension itself. In particular, the extension stores its data in the `storage.sync`[9] storage area, automatically synchronizing with all browser instances where the user is signed in.

Each passive fingerprinting protection stores its preferences in a separate part of the storage space. In addition to the protection-specific settings, each protection shares two common fields:

- **enabled**: A boolean flag determining if the protection is enabled, meaning the browser extension should enable the rule responsible for changing the value of a specific HTTP header.
- **value**: A string representing the current value of the HTTP header, incorporating all protection-specific preferences. It is displayed in the popup interface to provide the user with a preview of the particular HTTP header.

[4] https://tailwindcss.com/
[5] https://vitejs.dev/
[6] https://github.com/samrum/vite-plugin-web-extension
[7] https://webpack.js.org/concepts/hot-module-replacement/
[8] https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API
[9] https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/storage/sync

The storage is the centerpiece of the browser extension. Other components, including the service worker and user interface, rely on the storage to synchronize their actions, given that they execute in separate contexts.

5.2.2 Service worker

The browser extension defines only a single service worker script, which serves two primary functions described in the following sections, 5.2.2 and 5.2.2.

Additionally, as a secondary function, when started in the development mode, the service worker also subscribes to the `onRuleMatchedDebug` event, which the browser emits every time a rule defined by the extension is matched and applied. However, this event is only available on Chromium-based browsers.

The service worker does not subscribe to the `onInstalled` event as, by default, every protection is disabled when the extension initializes for the first time. This design decision allows users to choose only the protections they need after installing the extension.

Browser startup initialization

The service worker defines a callback function for the `onStartup`[10] event emitted by the browser on startup to initialize the extension and the associated protections. The callback function first checks if the browser storage is initialized, and if it is, the function regenerates header values for every enabled protection. This regeneration is necessary to ensure the device is protected against cross-session tracking. For the same reason, the extension only defines session header-modifying rules, which are removed after each session is closed, ensuring the same header values are not used in multiple browsing sessions. As the last step of its execution, the callback saves the updated header values to the browser storage.

Modification of Web APIs

The other primary function of the service worker is to ensure the consistency between HTTP headers and the browser environment (Web APIs). To do this, the service worker subscribes to the `onUpdated`[11] event emitted by the browser when a browser tab is updated. Then, it executes a script via the `chrome.scripting`[12] API with the `world` option set to `MAIN`, which ensures the script is run in the primary execution environment instead of the default isolated environment.

The executed script then updates the prototype of the `Navigator` object, modifying the values of `UserAgent`, `language`, `languages`, and `deviceMemory` properties if the user enabled the relevant protections.

5.2.3 User interface

As section 4.3 describes, the popup is the only graphical user interface used by the browser extension. The popup consists of one HTML file, which does not directly render UI elements. Instead, it initializes a React application which takes over the rendering. By delegating the rendering process to the React application, the popup benefits from React's

[10] https://developer.chrome.com/docs/extensions/reference/runtime/#event-onStartup
[11] https://developer.chrome.com/docs/extensions/reference/tabs/#event-onUpdated
[12] https://developer.chrome.com/docs/extensions/reference/scripting/

dynamic and efficient nature, allowing for enhanced performance, maintainability, and ease of development when managing the extension's user interface.

```
useEffect(() => {
  if (typeof unsubscribe.current === 'function') {
    unsubscribe.current();
  }

  void browser.storage.sync.get().then((store: StoreValue | {}) => {
    if (isStoreInitialized(store)) {
      initializeStore(store);
    }

    unsubscribe.current = useStore.subscribe((store) => {
      void browser.storage.sync.set(store);
    });
  });
}, []);
```

Listing 5.1: A code snippet in which the React application initializes its storage and creates a subscription for future changes.

Immediately after the initialization, the React application connects to the extension's storage, copies the content to its internal data storage, and subscribes for future changes. Utilizing an internal storage layer between the application and the browser storage was essential due to the asynchronous nature of the browser storage. If every read or write operation was performed directly on the browser storage, the UI would only be allowed to update once the operation was completed, resulting in a sluggish user experience. The application can mitigate this issue by utilizing an internal storage layer, ensuring a more responsive and seamless interface. Every change to the internal storage is automatically synchronized with the browser storage.

```
useEffect(() => {
  if (enabled && value !== null) {
    updateHeaderRule(Header.ACCEPT_LANGUAGE, value);
  } else {
    removeHeaderRule(Header.ACCEPT_LANGUAGE);
  }
}, [value, enabled]);
```

Listing 5.2: A part of the code that updates the HTTP header-modifying rule when a protection is enabled or disabled.

The popup interface is separated into multiple sections exclusive to specific protections. Each protection section has a header, a switch to enable or disable the protection, and a preview field with the current header value. If protection has custom preferences, these are also colocated with these primary elements. Whenever a user changes protection-specific preferences, the extension automatically updates the `value` field in storage, which propagates to the UI. However, the rule modifying the header value is only updated when the protection is enabled or disabled.

5.3 Fingerprinting protections

This section describes the browser extension's specific strategies to combat passive fingerprinting through HTTP headers. Each method is discussed in detail, accompanied by relevant implementation examples. Furthermore, any deviations from the initial design plan are highlighted and explained throughout this section.

```
export const updateHeaderRule = (header: Header, value: string) => {
  const id = HEADER_RULE_IDS[header];

  void browser.declarativeNetRequest.updateSessionRules({
    removeRuleIds: [id],
    addRules: [
      {
        id,
        priority: 1,
        action: {
          type: 'modifyHeaders',
          requestHeaders: [
            {
              header: header,
              operation: 'set',
              value,
            },
          ],
        },
        condition: {
          urlFilter: '*://*/*',
          resourceTypes: ['main_frame'],
        },
      },
    ],
  });
};
```

Listing 5.3: A utility function that removes the old header-modification rule and defines a new one with a new value.

In order to effectively manage the header-modification rules, a unique identifier is assigned to each HTTP header, serving as the rule's ID that does not change for the complete run of the extension. Whenever a rule needs to be updated, protections call a utility function (shown in listing 5.3) with two arguments - the header's name and its new value.

5.3.1 User-Agent and client hints

The primary purpose of this protection is to modify selected HTTP headers carrying user agent information as defined in section 4.2.4. In contrast with the original design described in the section mentioned earlier, the protection relies primarily on the HTTP client hint headers and not the `User-Agent` HTTP header. This change was introduced during the implementation phase due to the browsers reducing the `User-Agent` header, described in detail in section 5.4.2.

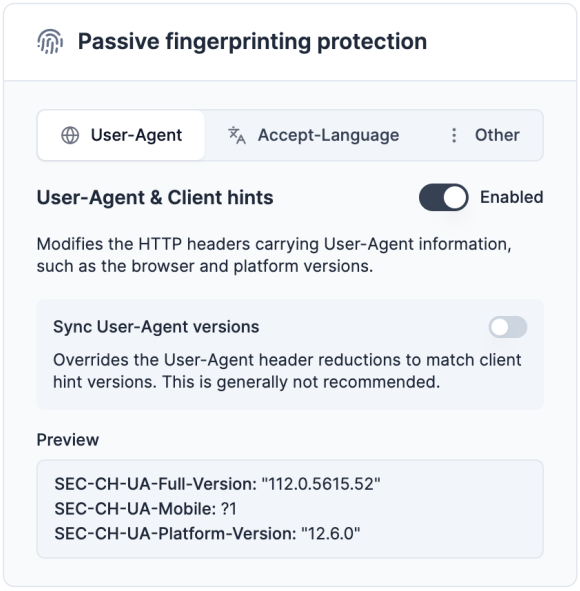


Figure 5.1: A screenshot of the User-Agent protection configuration interface.

When users enable this protection, it modifies the values of three HTTP client hint headers followingly:

- **SEC-CH-UA-Full-Version**[13]: A high-entropy client hint which carries a full (unreduced) browser version. The extension contains a record of released browser versions scraped automatically from changelogs or repositories of said browsers. This list ensures that the protection does not have to generate the version numbers, ensuring the browser always reports a valid version. When a new header value is needed, the protection looks up an alternative version number within the same MAJOR version range as the actual browser version obtained through the `navigator.userAgentData` API. The selected version is then wrapped in quotes and ready to be used as the new header value.
- **SEC-CH-UA-Mobile**[14]: A low-entropy client hint indicating whether the request was made from a mobile device or a desktop that prefers the mobile version of the web page. It is represented as a boolean with a value of `?0` (`false`) or `?1` (`true`). The protection randomly selects one value with a uniform probability.
- **SEC-CH-UA-Platform-Version**[15]: A high-entropy client hint which carries a full (unreduced) operating system version. When the protection needs a new value for this header, it looks up the actual OS version through the `Navigator.userAgentData` API, copies it, and randomly increases or decreases the last PATCH version segment. The resulting version string is then wrapped in quotes and ready to be used as the new header value.

While not recommended, users are provided with the option to synchronize versions found in HTTP client hints with those in the `User-Agent` header, effectively bypassing

[13] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-CH-UA-Full-Version
[14] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-CH-UA-Mobile
[15] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-CH-UA-Platform-Version

the browser reductions. If this option is enabled, the protection looks up the browser and the platform versions in the `User-Agent` header and replaces them with values previously generated for HTTP client hints.

5.3.2 Accept-Language

As section 4.2.2 defines, the browser extension modifies the `Accept-language` header in two modes – randomization and generalization. Both modes start by looking at the contents of `Navigator.languages` and selecting only the language codes with the ISO-639 subtag. These language codes are identified by checking if they include a dash ("-") separating the language tag from the subtag.

The randomization algorithm then checks whether one or more languages have subtag alternatives. If yes, it randomly selects a subtag from all the possible alternatives and replaces the original subtag. The algorithm can also select the original subtag to prevent the fingerprinting application from guessing it by narrowing the options.

On the other hand, the generalization algorithm removes subtags from all language codes leaving only the base language tags (i.e., "en" or "zh").

Upon completion of the transformation algorithms, the main program verifies the validity of the resulting list of languages. Initially, it eliminates any duplicate values that may have been introduced by the generalization or the randomization algorithm, which could select two or more identical subtags. The generalization algorithm always introduces duplicates if the original list of languages includes the same language tag more than once (e.g., "en, en-US, en-GB" becomes "en, en, en" and gets reduced to "en" after deduplication).

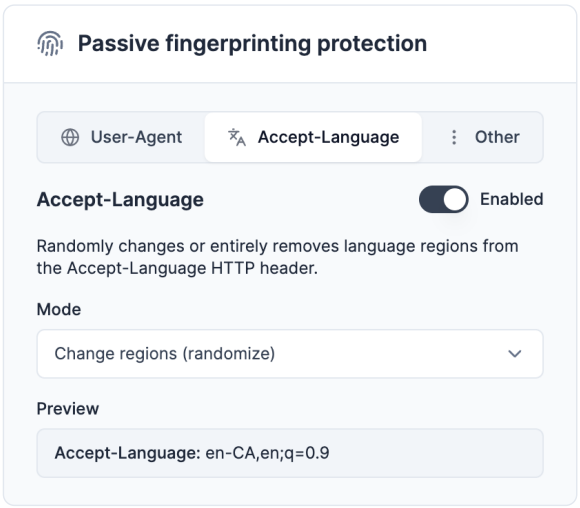


Figure 5.2: A screenshot of the Accept-Language protection configuration interface.

In the final step, the algorithm takes the resulting list of languages, calculates the quality values for each language, and joins them into a single string, using a comma (",") as the delimiter. The quality value is determined by the position of the language in the list of languages. The first language does not have a quality value as it is the primary language and has an implicit value of `1.0`. Each following language has a quality value equal to `1.0 - index * 0.1`. After this step, the resulting string is ready to be used as the value of the `Accept-Language` header.

5.3.3 Device-Memory

In its nature, the protection modifying the `Device-Memory` is very straightforward, and the rationale behind it is explained in section 4.2.3 of the previous chapter.

Initially, the algorithm retrieves the user-configured min and max values from the popup interface. These values represent the minimum and maximum indices within the array of permissible `Device-Memory` values, which include `0.25`, `0.5`, `1`, `2`, `4`, and `8`.

Subsequently, the algorithm randomly chooses a value between the specified min and max indices. The randomized selection introduces an element of unpredictability, making it more challenging for fingerprinters to accurately identify the actual user's device memory. Once the value is selected, the algorithm returns it as the modified `Device-Memory` header, ensuring that the user's actual device memory remains concealed.

5.3.4 Extra HTTP header

According to the design discussed in section 4.2.5, this protection adds a new, non-standard header `Extra` to each outgoing request. The value of this request is determined by the native function `Crypto.randomUUID()`. This function returns a UUID, a 128-bit long value represented as a 36-character string, which is guaranteed to be unique across space and time [23].

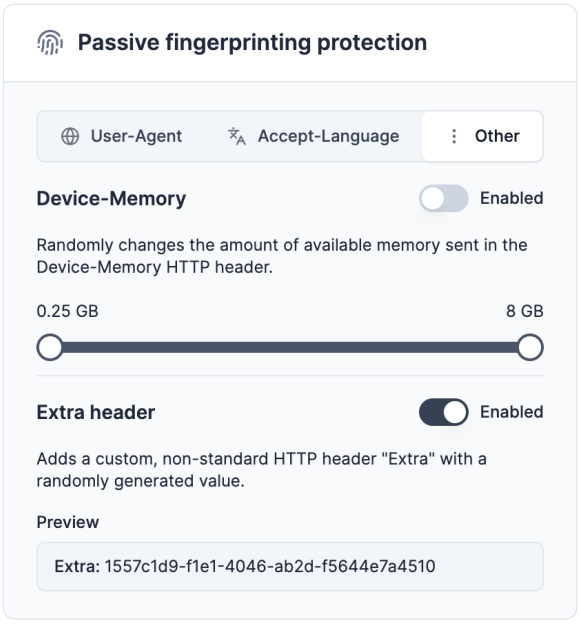


Figure 5.3: A screenshot of the Device-Memory and Extra protections configuration interfaces.

5.4 Challenges and limitations

This section discusses the challenges encountered throughout the implementation process and the corresponding strategies to address them, where applicable.

5.4.1 Manifest V3

During the time of writing, the new Manifest V3 has begun replacing the outgoing Manifest V2. Although the official end-of-life support deadline for Manifest V2 has been postponed, the browser extension discussed in this thesis has been developed utilizing the latest Manifest V3 framework. This design decision led to a series of challenges described in the following paragraphs.

Changes in network request modification

The biggest obstacle was the deprecation of the blocking web request listeners previously available through the `chrome.webRequest` API. Rather than intercepting requests and modifying them in real-time using the `chrome.webRequest` API, extensions must define request-altering rules through the `chome.declarativeNetRequest` API before the browser sends the request. Since the rules are defined beforehand, it is impossible to base the new HTTP headers on the original values sent by the browser. Instead, the extension has to use the available Web APIs to approximate the header values.

The `Accept` header protection designed in section 4.2.1 was not utilized precisely for this reason. Web browsers do not offer a dependable method for recognizing supported content types using a single or a combination of multiple Web APIs. Additionally, there is a high likelihood of unintended consequences resulting from incorrect header value changes, making the protection not worth the risk and, therefore, not included in the implementation.

Stricter content security policy

The extension modifies the `Navigator` interface to ensure consistency between HTTP headers and the browser environment. Traditionally, browser extensions inject a `<script />` element directly into the DOM[16] of the web page. This element contains a JavaScript code that wraps selected APIs and modifies the value of specific properties.

Manifest V3 disallowed specific content security policy (CSP) values[17], making injecting an inline JavaScript code complicated or outright impossible. Instead, the extension's service worker utilizes the new `chrome.scripting` API to execute a script whenever the browser loads a new web page. The script is set to execute with the `injectImmediately` option, meaning the browser will attempt to inject it as soon as possible. However, it is not guaranteed to execute before other scripts, which might be used for fingerprinting.

Limited amount of resources

Manifest V3 represents a relatively recent advancement in browser extension frameworks. Consequently, the resources and documentation detailing its inner workings, best practices, and nuances may not be as comprehensive as desired. This lack of resources led to somewhat more challenging development.

Despite this limitation, the adoption of Manifest V3 in this project demonstrates the possibilities for future development of Manifest V3 browser extensions.

[16] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

[17] https://developer.chrome.com/docs/extensions/migrating/improve-security/#remove-unsupported-csv

5.4.2 Chrome's User-Agent reduction

Beginning with Chrome version 101, the browser replaced parts of the User-Agent string with static values to protect the user's privacy. Version 101[18] replaced the MINOR, BUILD, and PATCH version segments with zeros, only leaving the MAJOR segment displaying the actual version (i.e., `101.0.0.0`). Next, in version 107[19], the browser replaced the desktop OS version and CPU information with a fixed value for the platform. The latest reduction happened in version 110[20], where the browser started using a fixed OS version and device model value for Android devices.

While these changes are certainly welcomed from a privacy standpoint, these reductions complicated the implementation of the `User-Agent` modifying protection, as any change made to this header would be instantly detectable by fingerprinting applications, potentially compromising the effectiveness of the privacy protection. For this reason, the `User-Agent` protection primarily focuses on modifying HTTP client hints headers instead of the `User-Agent` string directly. Despite this, the extension permits users to bypass the browser's `User-Agent` reduction but discourages them from doing so.

[18] https://groups.google.com/a/chromium.org/g/blink-dev/c/dcTStiBZVoQ/m/KyomPLOnAwAJ
[19] https://groups.google.com/a/chromium.org/g/blink-dev/c/kC-AeZ1fSdY/m/a_ICuXZRBQAJ
[20] https://groups.google.com/a/chromium.org/g/blink-dev/c/zVOEHwgyyu4/m/8KljdSN9AQAJ

Chapter 6

Testing

The primary goal of this work is to build a browser extension that protects users against passive fingerprinting through HTTP headers. However, as passive fingerprinting is not transparent to the fingerprinted device, it is not trivial to assess the effectiveness of the browser extension. Any changes to HTTP headers may or may not affect the resulting fingerprint. Moreover, the absence of standardized passive fingerprinting tools further complicates the assessment of the extension's usefulness.

For the abovementioned reasons, testing solely focuses on the experimental observation of changes made to HTTP headers without assessing the effectiveness of the browser extension. While this strategy does not directly measure the extension's overall effectiveness in preventing passive fingerprinting, it provides valuable insights into the extension's potential to modify HTTP headers in a manner that could mitigate passive fingerprinting.

The subsequent sections of this chapter provide an in-depth description of the testing process with the evaluation of results and insights into potential improvements.

6.1 Testing methodology and environment

The testing objective outlined in this section is to showcase the impact of the browser extension on the browser environment. Two separate experiments have been conducted to achieve this, each focusing on a different metric.

The testing environment was identical for both experiments. In order to simulate a user opening a browser and visiting a web page, the experiments utilized the Puppeteer[1] package. This package is suitable for testing the extension's behavior as it provides a high-level API to control a Chromium browser. The Chromium browser was selected for these experiments as the family of Chromium-based browsers currently has the highest usage share. The host system on the platform where the experiments were executed was MacOS in version 12.6.3. Default header values (unchanged by the extension) are displayed in Table 6.1. Please note that the `User-Agent` header value is absent due to its length. The `Extra` header is also absent, as only the browser extension sends it.

[1] https://pptr.dev/

Header name	Value
Accept-Language	en-US,en;q=0.9
Device-Memory	8
SEC-CH-UA-Full-Version	112.0.5614.0
SEC-CH-UA-Mobile	?0
SEC-CH-UA-Platform-Version	12.6.3

Table 6.1: A list of default HTTP header values (unchanged by the extension).

Next, a simple Node.js application was deployed to a remote server. This application listens on the HTTP port (80) for all incoming requests. When it receives a request, it immediately returns an HTML response containing a table with all headers and header values present in the request. Both experiments use this app to gather the test data.

6.1.1 Experiment 1: HTTP header values

The objective of the first experiment is to observe how the extension changes the values of HTTP headers. As the extension's primary function is to counter cross-session passive fingerprinting, this experiment collects and compares values between multiple browsing sessions. The experiment's algorithm can be described as follows:

1. The algorithm opens a new instance of a Chromium browser. The browser is pre-installed with the browser extension, specially modified to enable all protections when installed.
2. Next, it waits for a static timeframe of 500 milliseconds. This delay gives the extension time to initialize the header-modifying rules as the service worker runs in parallel to the primary browser process. There is no API exposed by Puppeteer which could be used to check if the rules are already in place. However, the static delay proved to be sufficient.
3. Once the delay ends, the algorithm opens a new page (tab) and loads the remote Node.js server application. A delay of 2000 milliseconds is set, after which the page load throws an exception. This exception is caught, and the data from this experiment are not written, meaning it effectively filters out network-related anomalies.
4. As the last step, the algorithm collects the values of HTTP request headers, which are returned by the Node.js application, and stores the data in a CSV file.

The steps of the algorithm above are repeated over 200 times, collecting HTTP header values in each iteration.

6.1.2 Experiment 2: Response times

As the extension defines declarative header-modifying rules to modify the outgoing request, the browser has to block the request until all matching rules are applied. Depending on the number of rules, this process could increase the time user has to wait for a response.

The goal of the second experiment is to get an understanding of the performance implications of using the browser extension. The experiment measures request-response times in

two runs, one with the extension enabled and the other with the extension disabled. The algorithm of both runs is similar to the first experiment, with a few modifications:

- In the run with the extension disabled, the first and second steps are omitted, implying the extension is not only deactivated but absent from the browser installation. The static delay is, therefore, also not necessary.
- Instead of collecting header values, the algorithm measures the time from the navigation start (`navigationStart` event) until the navigation end (`navigationEnd` event). The measured time is then stored in a CSV file.

In both experiment runs the algorithm is repeated 100 times, collecting request-response times with and without the extension.

6.2 Evaluation of results

This section presents the data collected in the previously conducted experiments and evaluates their results. Examination of the gathered data provides insights into the browser extension's effectiveness and impact on the browser environment and performance.

6.2.1 Experiment 1: HTTP header values

In the first experiment, the algorithm collected header values from 208 HTTP requests. As the previous section 6.1 describes, this experiment examines the variation among all the requests.

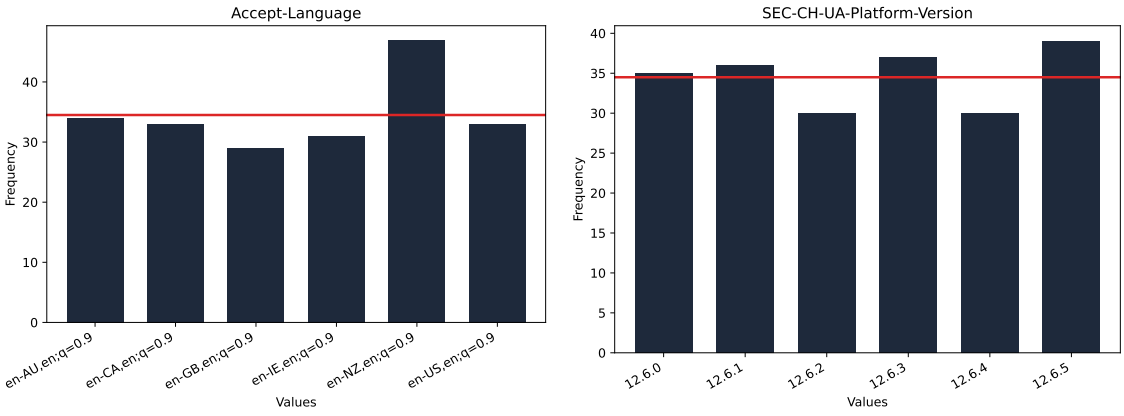


Figure 6.1: Values of `Accept-Language` and `SEC-CH-UA-Platform-Version` headers gathered in the experiment.

Figure 6.1 above displays frequency charts of the values found in the `Accept-Language` and `SEC-CH-UA-Platform-Version` headers. Since the values are generated randomly, they should follow a uniform distribution with a frequency of `N/M`, where `N` (208) represents the number of requests and `M` represents the number of unique values. The horizontal red line visualizes the ideal normal distribution in both charts.

Values of both headers do not significantly deviate from the normal distribution. The extension changes header values randomly enough, making detecting its presence difficult for a fingerprinting application.

6.2.2 Experiment 2: Response times

In both runs of the second experiment (with the extension enabled and disabled), the algorithm sent precisely 100 requests to the server while measuring the response-request time. The results of both runs are presented in a scatter chart visible in Figure 6.2.

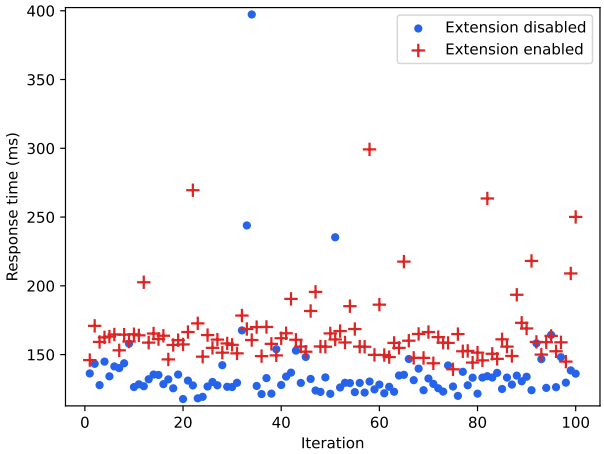


Figure 6.2: A screenshot of the Accept-Language protection configuration interface.

As anticipated, the response times of requests made with the extension disabled (represented in blue) are consistently lower than those with the extension enabled (represented in red). The outcome aligns with the expectations, given that the extension works by defining request-blocking rules.

However, the average difference of 37.03 milliseconds between runs is quite substantial, given that the browser used in experiments had only a single extension installed. With all protections enabled, the extension defines up to 7 rules, one for each HTTP header it modifies. Considering the extension's user could have other request-modifying extensions installed, the total time the request is slowed down could be significant.

Chapter 7

Conclusion

In this era of remarkable technological advancement, we enjoy conveniences at the cost of our privacy. For this reason, privacy-enhancing tools are increasingly popular and necessary.

The original goal of this master's thesis was to explore the ways of passive fingerprinting and implement an additional protective shield for the JShelter extension to protect against this type of user tracking. However, due to the ongoing migration of JShelter to Manifest V3, the focus of the work shifted to building a standalone extension that would showcase potential integration opportunities with JShelter.

The implemented extension successfully modifies selected HTTP request headers, focusing on minimizing the unintended user-facing side effects. After changing the header values, the extension modifies relevant Web APIs to ensure consistency between the headers and the browser environment. The testing phase showed that the extension executes the header modifications steadily as desired, despite the expected increased delay.

Unfortunately, the fact that browsers are still adopting the latest Manifest V3 complicated the implementation more than initially expected. Although welcomed, the privacy-oriented changes in Manifest V3 made it more complicated for extensions that want to push privacy protection even further. This slow adoption also caused the extension to be only supported on Chromium-based browsers, as the support on other browser families is often unstable or experimental.

That said, the browser extension showed the potential to mitigate passive fingerprinting, and I am optimistic that, combined with active fingerprinting countermeasures, it would be beneficial to preventing users' privacy.

In light of the challenges faced during the development of this browser extension, future improvements could focus on enhancing its effectiveness and expanding its compatibility. One promising direction is to implement cross-domain tracking countermeasures, as this would significantly interfere with the ability of third-party trackers to follow users across different websites, further protecting their privacy. Additionally, as new techniques of fingerprinting leveraging CSS emerge, exploring the potential to mitigate these threats would contribute to a more comprehensive and robust privacy protection solution. Last but not least, future versions of the extension could merge suitable header-modifying rules to reduce the slowdown of outgoing requests.

Bibliography

[1] *Chrome Developers - Manifest V2* [online]. Google LLC [cit. 2022-12-30]. Available at: https://developer.chrome.com/docs/extensions/mv2/.

[2] *Chrome Developers - WebExtensions API reference* [online]. Google LLC [cit. 2022-12-30]. Available at: https://developer.chrome.com/docs/extensions/mv3/.

[3] *JShelter* [online]. JShelter [cit. 2022-12-30]. Available at: https://jshelter.org/.

[4] *JShelter - JavaScript Shield* [online]. JShelter [cit. 2022-12-30]. Available at: https://jshelter.org/levels/.

[5] *Mozilla Developer Network* [online]. Mozilla Corporation [cit. 2022-12-30]. Available at: https://developer.mozilla.org/.

[6] *Mozilla Developer Network - Browser extensions* [online]. Mozilla Corporation [cit. 2022-12-30]. Available at: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions.

[7] *Mozilla Developer Network - Navigator Web API* [online]. Mozilla Corporation [cit. 2023-01-9]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Navigator.

[8] *Mozilla Developer Network - User-Agent HTTP header* [online]. Mozilla Corporation [cit. 2022-12-30]. Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent.

[9] *Use Cases* [online]. FingerprintJS, Inc. [cit. 2022-12-30]. Available at: https://fingerprint.com/use-cases/.

[10] A. Datta, J. L. and Tschantz, M. C. Evaluating Anti-Fingerprinting Privacy Enhancing Technologies. In:. 2019 [cit. 2023-05-08]. DOI: 10.1145/3308558.3313703. Available at: https://arxiv.org/abs/1905.01051.

[11] Article 29 Working Party. Opinion 9/2014 on the application of Directive 2002/58/EC to device fingerprinting. [online]. European Commission. [cit. 2022-12-30]. Available at: https://ec.europa.eu/justice/article-29/documentation/opinion-recommendation/files/2014/wp224_en.pdf.

[12] Berners Lee, T. *Information Management: A Proposal* [online]. 1989 [cit. 2022-12-30]. Available at: https://www.w3.org/History/1989/proposal.html.

[13] Brave Privacy Team. *Brave browsers - Fingerprinting defenses 2.0* [online]. Brave Software, Inc. [cit. 2023-01-10]. Available at: https://brave.com/privacy-updates/4-fingerprinting-defenses-2.0/.

[14] Eckersley, P. How Unique is Your Web Browser? In:. Berlin, Heidelberg: Springer-Verlag, 2010 [cit. 2022-12-30]. PETS'10. ISBN 3642145264.

[15] Flanagan, D. *JavaScript: the Definitive Guide : Activate Your Web Pages*. O'Reilly Media, Incorporated, 2011 [cit. 2023-02-19]. Available at: https://ebookcentral.proquest.com/lib/vutbrno/detail.action?docID=686420.

[16] L. Polčák, G. M. R. H. and McMahon, M. *JShelter: Give Me My Browser Back*. arXiv, 2022 [cit. 2022-12-30]. DOI: 10.48550/ARXIV.2204.01392. Available at: https://arxiv.org/abs/2204.01392.

[17] Laperdrix, P., Baudry, B. and Mishra, V. FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In: *ESSoS 2017 - 9th International Symposium on Engineering Secure Software and Systems*. Bonn, Germany: [b.n.], July 2017 [cit. 2023-01-6]. Available at: https://hal.inria.fr/hal-01527580.

[18] Laperdrix, P., Bielova, N., Baudry, B. and Avoine, G. Browser Fingerprinting: A survey. arXiv. 2019, [cit. 2022-12-30]. DOI: 10.48550/ARXIV.1905.01051. Available at: https://arxiv.org/abs/1905.01051.

[19] M. Schwarz, F. L. and Gruss, D. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In:. 2019 [cit. 2023-05-08]. Available at: https://www.ndss-symposium.org/ndss-paper/javascript-template-attacks-automatically-inferring-host-information-for-targeted-exploits/.

[20] Mayer, J. R. *Any person... a pamphleteer*. 2009. Master's thesis. Princeton University. Available at: https://jonathanmayer.org/publications/thesis09.pdf.

[21] Mueller, J. P. *Security for Web Developers : Using JavaScript, HTML, and CSS*. O'Reilly Media, Incorporated, 2015 [cit. 2023-03-05]. Available at: https://ebookcentral.proquest.com/lib/vutbrno/detail.action?docID=4333783.

[22] Nikiforakis, N., Joosen, W. and Livshits, B. PriVaricator: Deceiving Fingerprinters with Little White Lies. In: *Proceedings of the 24th International Conference on World Wide Web*. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2015 [cit. 2023-01-6]. WWW '15. DOI: 10.1145/2736277.2741090. ISBN 9781450334693. Available at: https://doi.org/10.1145/2736277.2741090.

[23] P. Leach, M. M. and Salz, R. *A Universally Unique IDentifier (UUID) URN Namespace* [Internet Requests for Comments]. RFC. RFC Editor, july 2005 [cit. 2023-04-8]. Available at: https://www.rfc-editor.org/rfc/rfc4122.

[24] P. Saint Andre, D. C. and Nottingham, M. *Deprecating the "X-" Prefix and Similar Constructs in Application Protocols* [Internet Requests for Comments]. RFC. RFC Editor, june 2012 [cit. 2023-01-6]. Available at: https://www.rfc-editor.org/rfc/rfc6648.

[25] Pasquali, S. *Mastering Node.js*. Packt Publishing, Limited, 2011 [cit. 2023-02-19]. Available at: https://ebookcentral.proquest.com/lib/vutbrno/detail.action?docID=1389335.

[26] Preston Werner, T. *Semantic Versioning Specification* [online]. [cit. 2022-12-30]. Available at: https://semver.org.

[27] R. Fielding, M. N. E. and J. Reschke, E. *HTTP Semantics* [Internet Requests for Comments]. RFC. RFC Editor, june 2022 [cit. 2023-01-6]. Available at: https://www.rfc-editor.org/rfc/rfc7231.

[28] Schauer, M. *Oblíbenost JavaScriptových API internetového prohlížeče*. Brno, CZ, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/23312/.

[29] Timko, M. *Vylepšení rozšíření pro omezení volání JavaScriptu*. Brno, CZ, 2019. [cit. 2022-12-30]. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/21824/.

[30] Vondráček, T. *Získávání informací o uživatelích na webových stránkách*. Brno, CZ, 2021. [cit. 2022-01-08]. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/23972/.