COMPARISON OF PRIVACY PRESERVING TOOLS IN WEB BROWSERS AND EXTENSIONS

POROVNÁNÍ NÁSTROJŮ ZAMEZUJÍCÍCH SLEDOVÁNÍ UŽIVATELE PROHLÍŽEČE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	**Bc. VOJTĚCH FIALA**
SUPERVISOR VEDOUCÍ PRÁCE	**Ing. LIBOR POLČÁK, Ph.D.**

BRNO 2025

Master's Thesis Assignment

163497

Institut:	Department of Information Systems (DIFS)
Student:	**Fiala Vojtěch, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Computer Networks
Title:	**Comparison of Privacy Preserving Tools in Web Browsers and Extensions**
Category:	Web
Academic year:	2024/25

Assignment:

1. Study web browsers, privacy enhancing browser extensions. and techniques that prevent user tracking.
2. Learn options of controlled web browsers and extension testing.
3. Design an environment for repeatable and deterministic testing of the methods found to prevent user tracking.
4. Implement the proposed environment.
5. Test selected tools that you studied in point 1.
6. Evaluate achieved results and suggest possible future directions of the project.

Literature:

- PETRÁŇOVÁ, Jana. Porovnání webových rozšíření zaměřených na bezpečnost a soukromí. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- PETRÁŇOVÁ, Jana. Běhová prostředí pro testování činnosti rozšíření pro webový prohlížeč. Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- SMATANA, Andrej. Evaluation of Little Lies in Browser Fingerprinting. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

Requirements for the semestral defence:
Points 1-3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Polčák Libor, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	21.5.2025
Approval date:	22.10.2024

Abstract

The prevalent use of user tracking on the Web has stimulated the development of privacy-preserving tools such as browser extensions and privacy-oriented browsers. However, comparing their effectiveness remains challenging due to the dynamic and complex nature of the Web. This work focuses on evaluating and comparing tools that prevent user tracking in web browsers, specifically those tools that block web requests. Consistent conditions are ensured across tests by capturing and replaying real web traffic in a controlled environment. The proposed system utilizes directed trees to recreate request structure and identifies blocked connections, providing deeper insights into indirectly blocked requests. The results reveal significant differences in blocking effectiveness across the tested tools, with substantial discrepancies observed even when using the same extension in Chrome and Firefox.

Abstrakt

Rozšířené sledování uživatelů na webových stránkách vedlo k vývoji nástrojů na ochranu soukromí, jako jsou speciální prohlížeče či jejich rozšíření. Srovnání efektivity těchto nástrojů je však náročné kvůli dynamické a komplexní povaze webových stránek. Tato práce se zabývá testováním a srovnáním nástrojů, které brání sledování uživatelů v prohlížeči. Konkrétně se zaměřuje na nástroje, které blokují webové požadavky. Zachycením a opětovným přehráním reálného webového provozu v kontrolovaném prostředí jsou zajištěny konzistentní podmínky pro testování. Navržený systém využívá orientované stromy k rekonstrukci struktury zachycených požadavků a identifikaci zablokovaných spojení, což umožňuje detailně zkoumat nepřímo zablokované požadavky. Výsledky ukazují výrazné rozdíly v efektivitě blokování mezi jednotlivými nástroji, přičemž významné odchylky byly zaznamenány i mezi verzemi stejného rozšíření pro prohlížeče Chrome a Firefox.

Keywords

content blocking, content blockers, ad blocking, ad blockers, browser fingerprinting, comparison, evaluation, privacy, privacy preserving tools, anti-tracking, request trees

Klíčová slova

blokování obsahu, blokátory obsahu, blokování reklam, blokátory reklam, otisk prohlížeče, srovnání, vyhodnocení, soukromí, nástroje zajišťující soukromí, ochrana proti sledování, stromy požadavků

Reference

FIALA, Vojtěch. *Comparison of Privacy Preserving Tools in Web Browsers and Extensions.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Libor Polčák, Ph.D.

Rozšířený abstrakt

Sledování uživatelů na internetu je dnes běžnou praxí sloužící primárně pro účely navýšení relevance zobrazovaných reklam. Kromě relevantních reklam to ale pro uživatele znamená, že třetí strany mají přehled o jejich aktivitách na internetu, například o stránkách, které navštěvují. Mnoha uživatelům je však ztráta soukromí nepříjemná a proto volí nástroje, jakými lze toto sledování omezit. Mezi nejčastěji používané patří různé blokátory reklam a nevyžádaného obsahu.

Nástrojů blokujících obsah je k dispozici mnoho. Jejich efektivita z hlediska toho, kolik nevyžádaného obsahu dokáží zablokovat, se ovšem liší. Používané nástroje často poskytují uživatelům statistiky, kolik domén či síťových požadavků bylo na navštívené stránce zablokováno. Neexistuje však ucelená metrika, kterou by se poskytované statistiky řídily a je tedy obtížné nástroje mezi sebou srovnávat.

Tato diplomová práce se zaměřuje na zmíněné blokátory obsahu a klade si za cíl navrhnout a implementovat systém, který umožní opakovatelné a determinstické srovnání jak existujících, tak i budoucích nástrojů blokujících obsah. Kromě toho, že navržený systém dokáže srovnávat počty zablokovaných požadavků různými blokátory, dokáže aproximovat i jejich efektivitu z hlediska zamezování sledování uživatelů. Pro jednotlivé vyžádané zdroje jsou zaznamenány přístupy k webovým API, které lze potenciálně zneužít pro sledování uživatele prohlížeče. Na základě toho, které požadavky testovaný nástroj zablokuje, lze zjistit, kolika potenciálním pokusům o sledování bylo touto blokací zamezeno.

Různé požadavky jsou různě důležité. To znamená, že jeden požadavek může tranzitivně vyvolat další požadavky, zatímco jiný požadavek nevyvolá žádnou další komunikaci. Tato diplomová práce umožní srovnávat nástroje i z hlediska tranzitivně zablokovaných požadavků zachováváním návaznosti mezi těmito požadavky. Návaznost je replikována s využitím orientovaných stromů, které reprezentují strukturu požadavků.

Navržený systém pro porovnání blokátorů obsahu a vyhodnocení jejich efektivity je složen ze čtyř částí. Úkolem první části je vytvořit dataset, který je použit pro pozdější analýzu chování blokátorů. Nástroj prochází webové stránky a na nich sbírá tři druhy dat – síťové požadavky, DNS odpovědi na tyto požadavky a volání webových API potenciálně zneužitelných pro sledování uživatele s využitím nástroje *JShelter Fingerprint Detector*. V rámci druhé části systém vytváří orientované stromy, které reprezentují strukturu, kterou požadavky na navštívených stránkách měly. Přiřazuje k jednotlivým požadavkům také zaznamenaná volání webových API.

Třetí část systému simuluje blokování požadavků. Nástroj nainstaluje do simulovaného prohlížeče vybraný blokátor obsahu a navštíví vlastní webový server, který zopakuje všechny zaznamenané požadavky. Na každý požadavek je zopakována zaznamenaná DNS odpověď, kterou poskytuje vlastní *BIND 9* DNS server. Následně systém extrahuje z prohlížeče informace o tom, které stránky byly zablokovány. Je podporován prohlížeč *Google Chrome*, libovolné prohlížeče založené na *Chromiu* a také prohlížeč *Mozilla Firefox* a libovolné jejich rozšíření ve formátu *.crx* pro Chromium a *.xpi* pro Firefox.

V rámci poslední části systém zpracuje výsledky simulace a zaznamenané blokace aplikuje do vygenerovaných stromů požadavků. Tyto stromy jsou následně analyzovány a jsou spočítány metriky popisující přímé i tranzitivní vlastnosti testovaného blokátoru.

S využitím implementovaného systému byly otestovány a porovnány vybrané blokátory obsahu. Nástroj byl validován s využitím jednotkových testů a bylo dosaženo pokrytí kódu *100 %*. Byly také identifikovány limitace navrženého nástroje. Hlavní limitace spočívají v nedostatcích orientovaného stromu, který byl pro reprezentaci struktury požadavků zvolen. Další limitace jsou důsledkem zvolené metody simulování požadavků.

Comparison of Privacy Preserving Tools in Web Browsers and Extensions

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Libor Polčák, Ph.D. The supplementary information was provided by Ing. Ondřej Malačka and Iskander Sanchez-Rola, PhD. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis. I acknowledge the use of generative AI tools to improve wording and sentence structure in the thesis.

. . . .
Vojtěch Fiala
May 4, 2025

Acknowledgements

I would like to thank my supervisor, Ing. Libor Polčák, Ph.D., for his invaluable guidance, advice, detailed feedback, and patience. I would also like to thank Iskander Sanchez-Rola, PhD., for his feedback, as well as Ing. Ondřej Malačka for both his feedback and the opportunity to work with the people from GenDigital. And last but not least, I would like to thank my family and friends for their support throughout my college years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, tracking is widespread across the web. Over 90 % of websites incorporate some form of user tracking [9]. Among the 10,000 most popular websites, more than 2,000 include 10 or more trackers[1]. This pervasive presence of tracking raises valid privacy concerns, as users may not want third parties to monitor their browsing habits.

Website owners are highly motivated to track user activity online to understand better who visits their pages. Advertisers, in particular, rely on tracking to identify who sees their ads, allowing them to serve more targeted content and increase the likelihood of a purchase being made by the viewer [8].

A variety of privacy-enhancing tools has been developed to combat the tracking issue. These tools include browser extensions and specialized browsers designed to reduce or prevent tracking. Many of these tools block outgoing HTTP requests to known tracking domains, enhancing privacy and slightly improving page load times and data usage [7, 24].

However, because these tools are developed independently, they vary significantly in their effectiveness. As a result, users must choose between different tools, often without clear guidance on which is most suitable for their needs. This thesis aims to help with that decision by focusing on content-blocking tools that block requests to potentially privacy-invasive trackers and comparing their performance.

This thesis introduces a deterministic, repeatable testing environment to ensure a fair and reliable comparison. It accounts for the structure of websites, recognizing that a single request may lead to several others. Since blocking one request can indirectly block others, the thesis models these relations using directed trees to maintain the underlying request structure.

Chapter 2 provides background on tracking techniques and methods used to counter them. Chapter 3 reviews existing evaluation tools and comparison strategies. New evaluation methods are proposed in Chapter 4 based on the existing approaches. The implementation of the proposed evaluations is described in Chapter 5. Chapter 6 presents and analyzes the evaluation results. Finally, Chapter 7 outlines the limitations of the proposed approach and discusses possible directions for future work.

[1] https://www.ghostery.com/whotracksme

Chapter 2

User Tracking and Countermeasures

When a person visits a website, their activity is tracked for various reasons. These reasons can range from enhancing the user experience by remembering display preferences (e.g., selecting white or dark display mode) to delivering personalized advertisements tailored to the user's interests. The pieces of code responsible for tracking user activity are referred to as *trackers* [4]. While the internal mechanisms of each tracker may differ, their fundamental objective remains the same: identifying the user who accessed the page.

This chapter introduces several tracking methods. Each method is presented along with possible ways to mitigate them. Afterward, the chapter explains general methods of tracking prevention and presents selected privacy-oriented web browsers and extensions.

2.1 Tracking Techniques

A wide variety of techniques exists to track users across the Internet. These techniques typically operate without the user's knowledge. They can be classified into two basic categories, namely *stateful* and *stateless*, according to how they work [19].

Stateful tracking techniques are methods that store information, such as user identification, on the user's computer for the specific purpose of tracking. Upon subsequent visits to the page, the identification is sent to the server, allowing the page, for instance, to recall that the user is already logged in.

On the other hand, stateless techniques attempt to uniquely identify the user without explicitly storing any data on the user's computer. The identification is based solely on the user's computer and browser properties, which are unintentionally shared with the remote server. This section presents specific techniques used to identify users. Furthermore, it explains how to counter these techniques and how the usage of countermeasures might affect the functionality of web pages.

2.1.1 Cookies

Cookies [3] are small data units stored on the user's device to provide the stateless Hypertext Transfer Protocol (*HTTP*) with a method to retain information across multiple visits. Only the originating site can access data stored in its cookies. Cookies can be classified as either *persistent* or *session-based.* Persistent cookies remain stored on the device even after the browser is closed and expire after a predefined period. In contrast, session-based cookies are only stored until the browser or website is closed.

Cookies can enhance user experience by storing user preferences, eliminating the need for manual adjustments to the page on each visit. Another potential application is maintaining user authentication status, preventing users from having to re-enter credentials on every page visit. The use cases mentioned above represent the intended functionality of cookies. However, cookies can also be misused for tracking users by storing a unique identifier that serves no other purpose beyond user recognition. This identifier is provided to the page each time the user revisits it.

First and Third-Party Cookies

Cookies can be classified into two categories – those originating from a *first-party* and those originating from a *third-party* [12, 23]. When a user accesses a domain that employs cookies, the cookies set by that particular domain are called first-party cookies. However, a page may also contain resources originating from third parties. Suppose a page incorporates elements with third-party content originating from another page with a different domain name, which sets its own cookies. In that case, these cookies are designated as third-party cookies. Many elements may be used to load third-party content. Two example elements that can utilize third-party content are the *img*[1] element, which can show remotely-hosted images, and the *iframe*[2] element, which allows a page to contain another page [20, 23].

Cookie Blocking

Many browsers block all third-party cookies by default [23], as they are primarily employed to track user activity across multiple websites to offer more precise targeted advertising. Nevertheless, trackers can overcome this issue by utilizing first-party cookies [23]. For example, a page may share its first-party cookie with a third party by appending the identifier as a part of an HTTP request sent to a third-party tracker. However, linking arbitrary identifiers to a single user would be difficult. Identifiers such as the user's email address handle this problem. Users provide their email address to a web page to log in, and later, when they log in using the same email on another site, both sites may share the email address with a third-party tracker, which allows the tracker to recognize that the user has visited both sites.

One way to prevent first-party cookie tracking is to block all cookies entirely. However, this approach may severely limit the functionality of many websites. Alternatively, it would be possible to permit the usage of cookies but to delete them each time the browser is closed. However, this approach might not work for users who rarely close their browsers.

2.1.2 Evercookies

Evercookies [5] function differently from conventional cookies. Upon removing a conventional cookie, it cannot be recovered unless the originating site sets it again. Removing evercookies is significantly more challenging since they use unconventional storage mechanisms. If one instance is deleted, as long as at least one other instance remains, it may restore itself to all the storage mechanisms from which it was deleted.

[1] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img
[2] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

Evercookies exploit browser mechanisms to store themselves that were not designed for such purposes. Consequently, it becomes more challenging for users to remove the tracking data, as they may not be aware that it is being stored through such mechanisms. These mechanisms may include classic HTTP cookies, browser local storage, DOM properties, or browser cache. Historically, these storage methods included Flash cookies or Silverlight [5]. Removing data from all possible storage locations or limiting access to specific storage mechanisms is necessary to mitigate the evercookies issue.

2.1.3 Browser Fingerprinting

Browser fingerprinting involves collecting data from the user's browser or the HTTP communication [19]. Contemporary fingerprinting techniques rely heavily on browser APIs to extract distinctive information, which is then employed to identify the user accurately. The final fingerprint is derived from the collective data obtained through this process. To prevent fingerprinting, three distinct approaches [26] may be employed:

- Creating homogenous fingerprints: All generated fingerprints are identical, and no individual can be considered distinctive, thus thwarting any potential for tracking.
- Modifying fingerprints on each visit: All fingerprints are modified (e.g., by randomization) to be different on every visited domain, preventing trackers from linking the fingerprints to a single user.
- Blocking fingerprinting attempts: Properties commonly utilized for fingerprinting are monitored, and should they be utilized by a tracker, network communication with such tracker is blocked. Another method is pre-emptively blocking all communication with known trackers.

Fingerprinting techniques can be divided into two categories [26]: *passive fingerprinting*, which relies on the extraction of information from the HTTP headers or the communication between a user and a server, and *active fingerprinting*, which extracts information from various browser APIs by using JavaScript or other technologies.

2.1.4 Passive Fingerprinting

The term *passive fingerprinting* is derived from the fact that the fingerprinting process does not require the utilization of any additional means of data collection. Upon accessing a website via a web browser, the user initiates an HTTP request, which is then sent to the server. However, an HTTP packet also contains other data, such as the user's IP address. The HTTP and other protocol data are transmitted each time the user connects to a site, regardless of the user's intention. Without this data, the user would be unable to access the site. A passive fingerprint may, for example, be comprised of the following elements [19, 36]:

- IP Address: Used to identify a device in the network layer of TCP/IP model.
- User-Agent[3]: HTTP header used to identify client's browser, operating system, and other information for compatibility purposes.

[3] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent`

- Accept[4]: HTTP header to indicate what content the client can process.
- Accept-Language[5]: HTTP header to indicate the client's preferred language settings.

In real-world applications, trackers collect a greater quantity of such data. Given that all of this data originates from the client, it is relatively straightforward to render them unusable, should the possible consequences be disregarded, by simply modifying them to another value. Such consequences may arise, for example, from modifying the IP address. If a client were to alter their IP address when sending a packet, simply changing it without implementing any additional modifications would break the routing protocols used to determine the destination for each packet, leading to communication failure.

To alter the IP address, tools such as Virtual Private Networks (*VPNs*) [14] can be utilized. Rather than communicating directly with the website (or any other entity), all data is redirected to a VPN provider. This provider then effectively acts as an intermediary, directing each packet to its intended destination, thereby preventing third parties from discerning the original IP address, which the VPN provider's IP address would instead replace.

It can be reasonably assumed that correctly modifying an IP address does not significantly change a site's functionality. However, modifying HTTP headers may have a more pronounced impact on the behavior of a site. For instance, modifying the Accept-Language header, should the site support it, may result in the site being displayed in a different language.

2.1.5 Active Fingerprinting

Unlike passive fingerprinting, active fingerprinting involves executing code within the user's browser. Actively acquiring browser data involves misusing various technologies provided by web browsers to create a fingerprint. One example is using the JavaScript *navigator*[6] interface to collect information about the device. This subsection presents several technologies that can be employed in the fingerprinting process.

Canvas Fingerprinting

A canvas element[7] is a programmable area that can be populated with graphics or text by the programmer. However, the same graphics drawn onto the element may appear differently, depending on the user's system configuration, such as the installed Graphical Processing Unit (*GPU*). This slight difference is the basis of canvas fingerprinting [22]. The canvas element also provides programmers with multiple API methods that can be employed in the process of fingerprint creation. These include `getImageData()`[8], which returns information about each pixel of the image, and `toDataURL()`[9], which returns an encoded representation of the canvas content. Extracting these data provides a strong and consistent fingerprint independent of other fingerprinting techniques.

[4] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept
[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Language
[6] https://developer.mozilla.org/en-US/docs/Web/API/Navigator
[7] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas
[8] https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/getImageData
[9] https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/toDataURL

The canvas fingerprinting methods work with *WebFonts*[10], which provide web pages with a simple method to include fonts [22]. Even when using the same browser and operating system on multiple machines, the results of rendering text on the canvas can vary due to minor differences in rendering engines. These subtle variations are what contribute to the generation of unique fingerprints. Acar et al. [1] revealed that this method could be extended even further by, for example, incorporating an additional graphic element or overlaying another text on the original.

Regarding countermeasures, the most straightforward approach with the most significant drawbacks is to disable the element, or even JavaScript itself, entirely. However, the canvas element's missing support may be an identifiable event. If the user wishes to retain the canvas functionality, they would need to utilize more sophisticated techniques to modify the results returned by the methods associated with the canvas element. These techniques, for instance, involve making minor random alterations to the RGB values of chosen pixels, which, in the ideal scenario, should not even be perceptible to the user [30].

WebGL Fingerprinting

WebGL[11] technology allows users to render three-dimensional images. Fingerprinting based on this technology is similar to the Canvas fingerprinting described above, as it also employs the canvas element, thus enabling the same techniques to be utilized for fingerprint calculation [22].

Yinzhi et al. [6] have demonstrated that a significant proportion of WebGL rendering features can enhance fingerprint quality. Additionally, WebGL itself provides a method[12] for retrieving data about the user's GPU, which may potentially constitute an additional component of the fingerprint. The techniques for preventing WebGL fingerprinting and their impact on the user experience remain the same as those employed for canvas fingerprinting. Additionally, some WebGL methods could be prevented from functioning, such as making it impossible for the page to acquire additional information about the user's system, such as the mentioned GPU.

Font Fingerprinting

Fifield and Egelman [16] have demonstrated that individual letters or symbols utilizing the same font may be rendered differently depending on the browser used. Even when using the same browser and operating system, discrepancies can be observed depending on the system configuration. This method involves increasing the size of fonts drawn in the background and taking measurements of individual symbols. While it is less accurate than previous fingerprinting methods, it can still be used as a part of the final fingerprint. Mitigation may include randomizing the reported properties of shown symbols [16]. Another method would be disabling JavaScript, which is used for measurements or obtaining the list of fonts, and complete disabling of Cascading Style Sheets (*CSS*).

Another way to fingerprint users based on fonts is by obtaining the list of fonts installed on the user's system. Eckersley [13] has shown that the installed fonts can be extracted using JavaScript and present a highly identifiable metric. It can be mitigated by forcing the web browser to work only with a predefined set of fonts shipped with the browser, thus limiting variability.

[10]https://developer.mozilla.org/en-US/docs/Learn/CSS/Styling_text/Web_fonts
[11]https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
[12]https://developer.mozilla.org/en-US/docs/Web/API/WEBGL_debug_renderer_info

The mitigation methods may impact the user experience. For example, disabling both JavaScript and CSS would result in a significant change in the visual appearance of the majority of websites while also limiting their functionality. Similarly, using a predefined set of fonts may adversely affect the visual appearance of certain pages designed with a specific font in mind.

Cascading Style Sheets Fingerprinting

CSS can be utilized to recognize the browser the user is using. Takei et al. [34] have demonstrated that by employing CSS in conjunction with remotely hosted content, it is feasible to find distinctive properties that are specific to different browsers. This method defines CSS properties, such as `background-image`[13], with data hosted remotely – an URL of an image in this case. Because the implementation of CSS may differ across browsers, some CSS properties may not be supported. Different browsers then send the server requests for different combinations of supported elements. The server that hosts these data can then analyze received requests and determine which browser they came from.

An alternative approach is the utilization of media queries, which specify the width of the user's device and send requests to the server based on the user's screen size. This serves as an additional method for identifying the user [34]. Another potential strategy is the usage of distinct fonts for the rendering of text. Initially, the font would try to be sourced locally. If this were not feasible because the font was absent, a request to the server would be initiated, which could serve as an additional identification method by differentiating the fonts the user has installed.

In contrast to earlier techniques, the deactivation of JavaScript does not impact the viability of this fingerprinting approach. The previously mentioned method of limiting supported fonts described in the Font Fingerprinting subsection may diminish the accuracy of this fingerprinting technique. However, it only renders it somewhat ineffective. The sole option would be to prevent the execution of CSS entirely. However, this would inevitably result in the disruption of the majority of web pages' functionality and severely impact user experience.

AudioContext Fingerprinting

AudioContext[14] is part of the *Web Audio API*[15]. This API controls audio within the web browser, for example, by selecting audio sources or modifying audio properties. AudioContext is used as a graph representation of multiple audio sources and offers methods for manipulating the audio processing.

The methods provided by AudioContext can be utilized to create a unique fingerprint of a given device [17]. This fingerprinting method is possible because each browser has a distinct implementation of AudioContext. A potential fingerprinting method may be based, for instance, on measuring the latency between the output of AudioContext being passed to the system audio.

The most straightforward method for preventing this fingerprinting is entirely disabling the Web Audio API. However, this may break some websites that rely on this API for functionality. An alternative approach is similar to the previously described method for

[13]`https://developer.mozilla.org/en-US/docs/Web/CSS/background-image`
[14]`https://developer.mozilla.org/en-US/docs/Web/API/AudioContext`
[15]`https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API`

mitigating canvas fingerprinting and involves introducing random noise into the output [29].

Extension Fingerprinting

The presence of browser extensions can be used as another fingerprinting method. Several methods can be employed to confirm whether or not an extension is present. Many browser extensions modify the visited page, for instance, by removing unwanted advertisements or tracking elements. Starov et al. [32] have demonstrated that such alterations can be used as a fingerprint. If an extension removes a Hypertext Markup Language (*HTML*) element, the Document Object Model (*DOM*) is altered. The same is true should an extension add a new element into the DOM to, for example, provide additional functionality. This fingerprinting method is based on the analysis of the DOM. As different users employ different extensions, the changes they introduce to the DOM may serve as a fingerprint, providing further information towards the final identification of the user.

A method of preventing this fingerprinting technique would be introducing random, artificial changes to the DOM structure [32], making it difficult to distinguish whether the changes are caused by a specific extension or only artificially mimicked. A downside is that it could result in a decline in user experience.

2.2 Pre-Emptive Tracking Prevention

There are multiple methods for blocking tracker functionality. Some of them were already presented in the previous Section 2.1. The methods described in this section work not by blocking specific user identification methods, which were explained in the previous section, but rather by preventing trackers from launching in the first place. Due to their broad approach, however, they may occasionally block more than just trackers, or fail to block some entirely.

One common strategy involves identifying the domains that host trackers and blocking any outgoing HTTP requests to them. Another method focuses on disabling JavaScript, dynamic browser functionality, or specific HTML elements to limit how trackers can operate.

2.2.1 Content Blocking

Before code embedded in a web page can execute, it must first be downloaded. This begins with an HTTP request to the server hosting the content. The server responds by sending the web page along with all of its resources, including scripts and other executable code.

Content blocking works by filtering potentially malicious outgoing HTTP requests. If a request is identified as targeting a a Uniform Resource Locator *URL* known to host tracking elements, it is blocked [31]. This prevents scripts and other resources from being loaded, often without noticeably affecting the core functionality of the website. However, results may vary. If a page component includes both useful content and a tracking mechanism, blocking that component can break the page. Likewise, if a site relies heavily on third-party resources and those are blocked, it may become unusable.

This technique is valued for its simplicity. It allows users to block content from known trackers and is also used by ad blockers to prevent the display of unwanted advertisements. Since many ads include tracking components [20], even basic ad blockers, those not specifically designed to enhance privacy, can indirectly offer enhanced user privacy by preventing trackers from loading.

The decision to block or allow requests typically depends on predefined lists. While a *whitelist* approach of only allowing traffic to approved sites is technically possible, it is not practical for everyday use. Instead, *blacklists* are used, which contain patterns or rules used to detect and block unwanted traffic.

Existing Blacklists

Given the considerable number of trackers and other unwanted content, capturing them in a single list is not feasible – not only would it affect performance, but maintaining such a list would be impractical. Consequently, multiple blacklists exist, often maintained by individuals who usually use their intuition or experience rather than a systematic approach [31]. This means that what one person considers unwanted, another might view as harmless.

Some blacklists are focused on a specific subset of unwanted domains, for example, domains from particular countries, while others take a more general approach. Examples of general blacklists include *EasyList*[16] and *EasyPrivacy*[16]. While EasyList is targeted towards blocking advertisements and other such unwanted content, EasyPrivacy is explicitly oriented towards blocking trackers. These lists can become quite extensive – as of October 2024, EasyList had over *71,000* entries.

Nevertheless, the implementation of address filtering has its own set of challenges. To best utilize content blocking, it is essential to have a comprehensive understanding of the specific pages and domains that should be blocked. Otherwise, even valid non-tracking content might be blocked. However, it is impossible to create a blacklist of all addresses that contain undesirable content. Consequently, if a page contains a tracker but is not included in the blacklist, it is not blocked. Because of that, it is essential to update the blacklists regularly. Furthermore, the blacklists must be updated if a tracker or other unwanted content changes its address.

2.2.2 Element and JavaScript Blocking

Even if a page contains no trackers, they may still be present because of elements such as *iframes* or others. This third-party tracking can be mitigated by blocking such HTML elements, thus preventing any code inside them from executing. Trackers also use JavaScript and other dynamic browser functionalities, as described in Subsection 2.1.3. However, differentiating between a tracker and a legitimate code is complicated. One way to solve this problem is to block all JavaScript [31] and other such dynamic browser functionality. Blocking all JavaScript could, however, pose a problem because according to a survey[17], as of October 2024, *98.8 %* of all pages available on the Internet use JavaScript in some way. Thus, blocking all JavaScript code would probably impact the functionality of many sites, rendering them unusable.

[16] https://easylist.to/
[17] https://w3techs.com/technologies/details/cp-javascript

2.3 Anti-Tracking Tools

This section presents a selection of existing tools that can be used to block fingerprinting methods described in previous Sections 2.1 and 2.2. These tools are in the form of both browser extensions and browsers with natively implemented anti-fingerprinting measures.

2.3.1 Browsers

The presented browsers all offer a variety of privacy-enhancing measures by default. Consequently, installing additional privacy-oriented extensions is not as crucial as in different browsers that lack native anti-tracking measures. All the presented browsers are free of charge, meaning there is no cost associated with using them. However, some may offer additional functionality for a payment.

Brave

Brave[18] is a privacy-oriented browser that natively provides various anti-tracking and ad-blocking techniques. The browser is developed by Brave Software Inc., a privately held company. Brave is open-source and based on the Chromium[19] browser, which is itself open-source. While Brave is free by default, it also offers a paid built-in VPN. The browser's default settings block trackers, third-party cookies, and fingerprinting techniques. Additionally, the browser can be configured to block JavaScript, upgrade HTTP connections to HTTPS when available, or block all cookies.

To prevent the display of advertisements and the tracking of user activity, Brave employs a content-blocking method described in Subsection 2.2.1. Furthermore, Brave randomizes browser API calls to prevent the functioning of specific API-based fingerprinting techniques, such as canvas fingerprinting or WebGL fingerprinting, as discussed in Section 2.1.

Avast Secure Browser

Avast Secure Browser[20] is a browser that prioritizes the protection of user privacy and security. It is a proprietary product developed by Gen Digital Inc., utilizing the Chromium browser as its core. Avast Secure Browser is provided free of charge, while a paid *Avast Secure Browser Pro* version is also available, offering improved functionality such as a built-in VPN. It incorporates content-blocking and anti-tracking capabilities and includes an extension guard, preventing the installation of untrusted extensions. One of the filtering lists used is EasyList. Furthermore, it performs randomization and generalization of API responses for elements such as canvas or the navigator interface.

Mozilla Firefox

Mozilla Corporation, a subsidiary of the non-profit Mozilla Foundation, is responsible for the development of the *Firefox*[21] browser. The software is open-source, with the code available online. It is based on the Gecko engine, which the Mozilla Foundation also develops. It offers third-party cookie blocking, crypto-mining software blocking, and fingerprinting

[18] https://brave.com/
[19] https://www.chromium.org/
[20] https://www.avast.com/secure-browser
[21] https://www.mozilla.org/firefox/

protection by default. The fingerprinting protection is based on blocking requests to third parties and may be configured further to alter specific browser APIs.

2.3.2 Extensions

Many privacy-oriented extensions are currently available for download. For this thesis, a selection of them has been chosen for comparison. As of October 12, 2024, all selected extensions are available for download from the Chrome Web Store and the Mozilla Addons page. All the described extensions are free to download, although some may offer paid upgrades.

uBlock Origin

uBlock Origin[22], developed by Raymond Hill, is a content-blocking extension. It is an open-source software. It employs the content-blocking method with numerous pre-configured filter lists, including EasyList and EasyPrivacy. Consequently, it can block advertisements, trackers, pop-ups, and other undesired content.

It will soon be removed from the Chrome Web Store and cease to be functional on Chromium-based browsers due to its failure to comply with the latest Manifest v3[23] rules as set out by Google. A Manifest v3-compliant version, *uBlock Origin Lite*, is a replacement with limited functionality. However, the original will remain available on Firefox.

Privacy Badger

Privacy Badger[24] is designed to block third-party trackers. It is open-source software developed by the non-profit organization Electronic Frontier Foundation. The software is primarily designed to block trackers and does not interfere with advertisements unless they contain trackers. The software is provided with a set of pre-existing lists of trackers, which are blocked by default. This list can be extended algorithmically through the analysis of trackers, and only when a tracker is identified as such, it is blocked.

Adblock Plus

The primary function of *Adblock Plus*[25] is to facilitate the blocking of advertisements. A premium version is available for purchase and offers additional features. It is open-source and developed by Eyeo GmbH. It employs the content-blocking method, which may offer limited tracker-blocking functionality as was described in Section 2.2.1, thus partially enhancing the user's privacy. However, Adblock Plus does not block all ads. By default, the developers have chosen some *acceptable* ads that are allowed.

[22] https://ublockorigin.com/

[23] https://github.com/uBlockOrigin/uBlock-issues/wiki/About-Google-Chrome's-%22This-extension-may-soon-no-longer-be-supported%22

[24] https://privacybadger.org/

[25] https://adblockplus.org/

Ghostery

Ghostery[26] is an extension designed to prevent tracking and block advertisements. Ghostery GmbH develops the software. The code is open-source. Ghostery can block advertisements, cookies, trackers, and other content, including various pop-ups.

JShelter

JShelter[27] is a privacy-oriented extension developed by Libor Polčák and Giorgio Maone. By default, JShelter provides anti-fingerprinting methods, such as randomizing the canvas fingerprint. Additionally, the software employs a heuristic monitoring system to detect the use of browser APIs commonly associated with fingerprinting techniques. Upon detecting such fingerprinting activity on a given website, a notification is generated and displayed on the user's computer. JShelter can be configured to prevent further HTTP requests to such websites. However, Manifest v3 restricts its functionality on Chromium-based browsers.

[26]https://www.ghostery.com/
[27]https://jshelter.org/

Chapter 3

Existing Solutions for Evaluating Anti-Tracking Tools

Establishing a stable and reproducible testing environment is essential to ensure fair and meaningful comparisons between anti-tracking tools. Such an environment should include an automated system that minimizes human interaction and allows for fair and repeatable evaluations of the anti-tracking capabilities of different browsers and extensions. Since trackers are loaded on web pages accessed through the browser, the environment must replicate user activity, such as visiting the tracking sites with an automated browser. This chapter presents an overview of existing tools used to compare the anti-tracking capabilities of privacy-enhancing tools.

3.1 Evaluation Environment

The purpose of a dedicated environment is to make the results easily replicable. There are two ways to set up such an environment: following a detailed set of setup instructions or deploying a pre-configured environment that is ready to use. Pre-built environments typically come in two forms: through *virtualization* or *containers*.

3.1.1 Virtualization

One of the ways to create a replicable environment is by using virtualization [2]. An example of virtualization software is *VMware Workstation Pro*[1]. Virtualization allows multiple virtual machines to run concurrently on a single physical machine. Virtual machines are emulated physical computers that do not need their own hardware to run but instead rely on the hardware of the host machines. Host hardware can be shared among multiple virtual machines.

Virtual machines are controlled by a *hypervisor* – software designed to provide a level of abstraction from the host system. It also manages the resources allocated to each virtual machine and prevents virtual machines from affecting each other by isolating them. Figure 3.1 (left) visualizes the virtualization process. Each virtual machine can run its own operating system, such as *Linux* or *Windows*. Besides that, each virtual machine can be configured for specific purposes, such as running a web server or, for this thesis, a compar-

[1]https://www.vmware.com/products/desktop-hypervisor/workstation-and-fusion/

ison of anti-tracking tools. While virtualization is viable, using a whole virtual operating system may be unnecessarily complex for certain purposes.

3.1.2 Containers

Another method of creating environments is through the use of containers [33]. An example of container software is *Docker*[2]. Containers are software that can be executed to run an application (or multiple applications) they contain. The code and configuration of each application are included in the container, together with libraries the application uses and other required dependencies. Containers rely on operating system-level virtualization, isolating processes from each other and controlling what resources each process can access. Figure 3.1 (right) shows the container technology. A practical difference from the previously mentioned virtualization approach is that containers do not need their own operating system. Instead, they rely on the host computer's operating system, which makes them less isolated than virtual machines but offers better performance. Relying on the host operating system makes them more lightweight in terms of both actual size and the overhead, which is lesser than with virtualization.

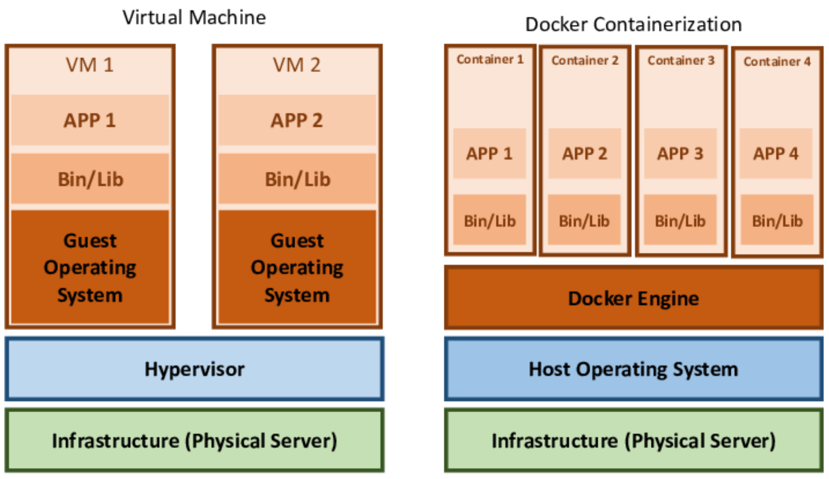

Figure 3.1: Illustration[3] of virtualization (left) and containers (right).

3.2 Existing Evaluation Methods

This section presents several tools and methods that are used to evaluate the effectiveness of anti-tracking technologies. These tools and methods commonly rely on browser automation to simulate real user activity, such as visiting websites or interacting with elements on a page. Visiting websites is necessary to trigger tracking scripts and collect data on whether trackers are blocked or whether a unique and consistent fingerprint can still be generated. A widely used tool for browser automation in this context is *Selenium*[4].

[2]https://www.docker.com/

[3]"Docker Container vs. Virtual Machine" by *Mehr, I. E. et al.* is licensed under CC BY 4.0. Retrieved from https://www.researchgate.net/figure/Docker-Container-vs-Virtual-Machine_fig2_369061128/ (accessed on April 16, 2025).

[4]https://www.selenium.dev/

3.2.1 Straightforward Approach

The first presented method for benchmarking anti-tracking browser extensions was proposed and implemented by Traverso et al. [35] in 2017. Their method compares anti-tracking performance based on the number of known trackers contacted during browsing. Their setup uses Selenium to simulate the Firefox browser. A list of URLs is used to specify which pages to visit. Each page is visited ten times, with the browser cache cleared after every visit to ensure consistency. Different privacy-focused extensions are installed and tested one at a time, each forming a unique configuration for the crawl. All HTTP communication between the browser and the websites is logged for analysis. The collected data includes the number of bytes downloaded, page load times, third-party requests, and contacts with known trackers. These statistics are then used to compare how effectively each extension blocks tracking. The process moves to the following URL only after the current page fully loads, as indicated by the JavaScript `onLoad` event. If a page fails to load correctly, it is skipped and not included in the results.

3.2.2 Open-WPM

Open-WPM [15], released in 2016 and still actively maintained as of 2024, is an open-source tool designed for large-scale web privacy measurements, with its code available at GitHub[5]. The tool is built to simulate the user's activity and record data from the simulated interactions. This data may include cookies, response metadata, and script activity, indicating that the tool can observe stateful and stateless tracking methods. In addition, OpenWPM can store the state of the browser over multiple measurements, allowing it to observe tracking techniques such as cookie respawning, which may be used by evercookies which were described in Subsection 2.1.2. OpenWPM consists of three modules: multiple *browser managers*, *task manager*, and *data aggregator*. Figure 3.2 shows the system's design.

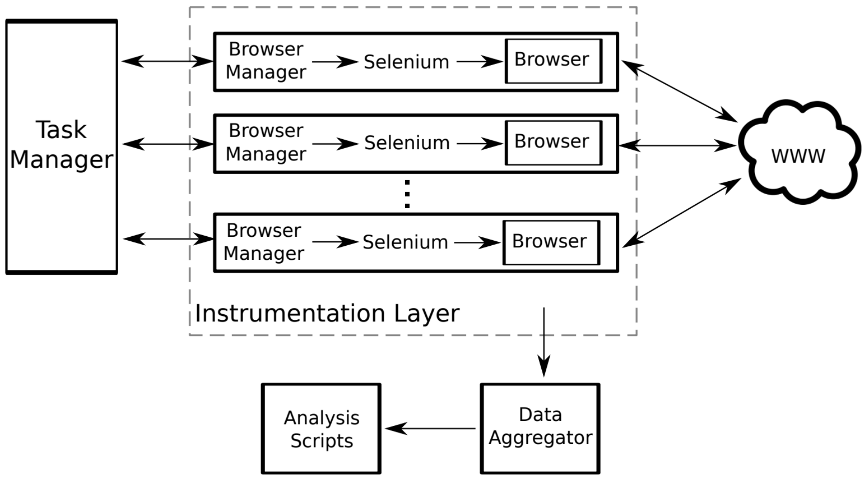

Figure 3.2: High-level design of OpenWPM, retrieved from [15].

Browser managers run separate instances of full-fledged *Firefox* web browsers via Selenium. Managers are run in parallel to improve performance. When a new browser manager is created, a specified configuration is applied. This allows each session to use a different

[5]`https://github.com/openwpm/OpenWPM`

configuration, such as a different combination of browser extensions, which may themselves be further configured. In addition, each manager can be configured to perform different tasks. These tasks may consist of, for example, visiting different websites. These managers can also be launched in headless mode, using a virtual frame buffer for the graphical interface without displaying it to the user.

Task manager controls multiple instances of browser managers. Commands for each browser are stored in an execution thread unique to each browser. If the browser manager encounters an error, such as a timeout or crash, the current state of the browser is archived. The archived state allows the browser to be restarted later in the same state (e.g., with cookies stored from visited sites).

Data aggregator stores data received from browser managers. This includes compressed web content, such as all types of cookies (e.g., regular cookies or Flash cookies on legacy systems), HTTP (and HTTPS) request and response headers, and finally, access to various JavaScript prototypes such as storage, canvas, or navigator. The data is stored in a structured way, allowing it to be traced back to a specific browser and page from which it originated. Upon completion, this data can be analyzed to determine, for example, which extension blocked the most tracking attempts.

3.2.3 PETInspector & FPInspector

PETInspector and *FPInspector* [11] are used together to form a benchmarking system, where FPInspector evaluates the results provided by PETInspector. *PET* stands for *Privacy Enhancing Technology*, and *FP* stands for *Fingerprint*. It was developed in 2019 but has not been updated since. PETInspector is available on GitHub[6]. The system can be used to compare different privacy-enhancing extensions and browsers. PETInspector serves as a tool to experimentally calculate the anti-fingerprinting capabilities of a given privacy-enhancing tool, which then forms a mask of how the tool behaves. This mask is then used with actual observed fingerprints to simulate how a browser would behave if everyone were using the given privacy-enhancing tool. The resulting traceability is then measured, which is the purpose of the FPInspector. This experimental and observational approach forms the hybrid benchmarking system. The output of the system is the effectiveness of a given tool.

The system consists of four parts. These parts are the *client simulator*, *fingerprinting server* and *analysis engine*, which form the PETInspector, and the FPInspector, which is the fourth part. This is illustrated in the Figure 3.3.

Client simulator simulates users via Selenium. The simulator allows the configuration of different fonts, time zones, and other properties. Each simulated user has various privacy-enhancing tools installed. Additionally, a client with no privacy-enhancing tools is used to isolate each tool's changes. These simulated users visit the fingerprinting server, triggering various fingerprinting scripts. The visits consist of accessing different pages, idling on the pages, reloading them, and browsing across sessions separated by 45 minutes of downtime.

The fingerprinting server uses various fingerprinting techniques to compute the fingerprints of visiting clients. The analysis engine uses information from both the client simulator and the fingerprinting server to measure how a particular client configuration affects a fingerprint by observing whether the data the browser passed to the fingerprinting server was changed by generalization or randomization. Suppose the result of the fingerprinted prop-

[6]https://github.com/tadatitam/pet-inspector

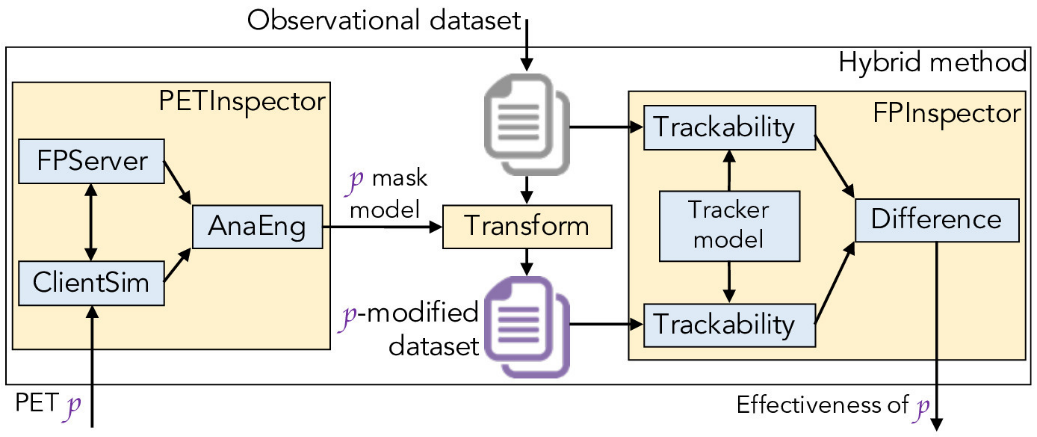

Figure 3.3: Benchmarking system design showcasing how *PETInspector* and *FPInspector* interact, retrieved from [11].

erty is altered by the privacy-enhancing tool, either by generalization or randomization. In that case, the property is modeled as masked in the generated mask and unmasked if not changed. This approach overestimates the effectiveness because it treats any change to the property values as unrecoverable, rendering the property useless for tracking.

FPInspector receives two fingerprint datasets as input. The first dataset consists of actual measured fingerprints with no privacy-enhancing tool installed. The second dataset consists of simulated fingerprints, which show what the first dataset would look like if everyone used a particular privacy-enhancing tool. The masks created by PETInspector are used to obtain the second data set. These datasets are then compared for uniqueness to calculate effectiveness.

3.2.4 Browser Extension Testing Environment by Jana Petráňová

This Browser Extension Testing Environment [25] has built-in methods for testing privacy-oriented browser extensions. It does not directly compute the effectiveness of a given privacy-oriented tool but returns information about how it affects tracking. It has no official name. The environment was proposed as part of the master's thesis by Jana Petráňová [25]. While the environment is mainly designed to test the *JShelter* extension, it also supports testing other privacy-oriented extensions. The extensions can be tested in two supported browsers: Google Chrome and Firefox. It uses the designs and implementations proposed by *Open-WPM* and *PETInspector* described above. The environment consists of two main parts: *integration testing* and *system testing*. Its high-level design, together with the technologies used, is shown in Figure 3.4.

Integration testing is based on the PETInspector tool, described in Subsection 3.2.3. It uses Selenium to simulate a browser client with privacy-oriented extensions installed. This client visits a locally hosted website that uses several tracking techniques to generate a fingerprint. The server can be customized, though a default setup is provided. The server collects various attributes from the browser, which are stored for later analysis to understand how a given extension affects the results returned by various browser methods.

System testing measures how different extensions impact visited web pages. The Selenium-controlled browser visits a list of predefined pages, both with and without the extensions installed. For each visit, the console output is logged, and a screenshot is taken before and after applying the extension. These outputs are then compared to measure how much the extension changed the visual layout and behavior of the page.

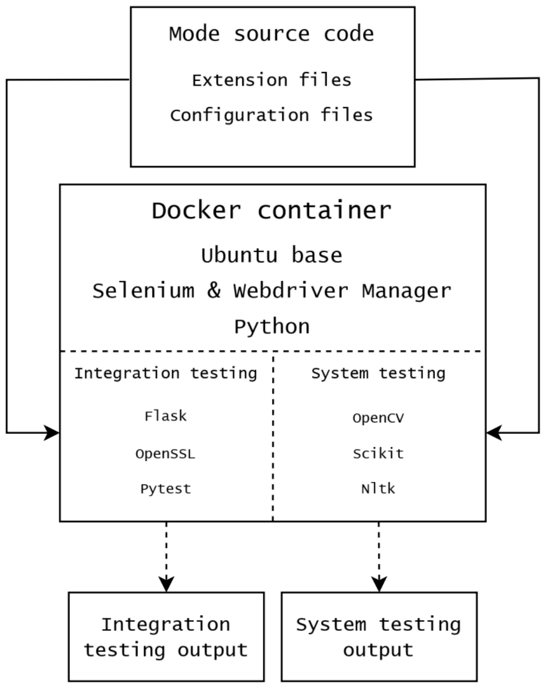

Figure 3.4: Architecture of the testing environment for anti-tracking tools, retrieved from [25].

Chapter 4

Designing Privacy Enhancing Tools Comparison Environment

As outlined in Chapter 2, multiple methods for user tracking exist. Consequently, privacy-preserving tools that prevent tracking have been developed. However, these tools are mainly independent of each other and thus differ in both the anti-tracking methods they provide and their effectiveness.

Since not all anti-tracking tools have their source code publicly available, comparing their functionality by studying the code is impossible. Consequently, specialized solutions have been developed to compare their effectiveness. Several of these solutions are described in Chapter 3. The main objective of this thesis is to improve existing comparisons of anti-tracking tools. This chapter presents a testing environment and the comparisons that will be performed since they are the primary focus of this thesis.

4.1 Environment and Comparison System Design

To design an environment for the comparisons, it is first necessary to establish the requirements. The basic requirements are summarized in the following list:

- The environment should be documented and easy to replicate.
- The environment should support multiple anti-tracking tools.
- The environment should support the addition of new tools developed in the future.
- Comparison methods should be clearly defined, accompanied by their respective limitations.
- Comparison methods should be deterministic, making the results replicable.

One of the requirements for the environment is to ensure its replicability, which can be accomplished by employing the technologies presented in Section 3.1. However, the environment could also be set up manually by following steps that prepare a system to support the comparison methods. Furthermore, the environment should support multiple anti-tracking tools, including existing tools presented in Section 2.3, and additional tools that may emerge. Overall, the environment only serves as a container for the proposed comparison methods.

Selected methods of comparing anti-tracking tools are described in Section 3.2. *OpenWPM* is a tool often used in existing anti-tracking research [24, 18, 21]. While OpenWPM is explicitly designed for web privacy research and is still maintained even in 2024, it lacks one crucial feature: the support of other browsers. OpenWPM can only be used with the Firefox browser, which is insufficient for this thesis, as it also aims to compare the anti-tracking performance of different browsers presented in Section 2.3. Alternative comparison tools are usually outdated or incompatible with the stated requirements. Thus, a custom system needs to be designed. However, the system adapts and integrates certain concepts and approaches from existing solutions.

4.2 Proposed Evaluation Methods

Following a discussion with the thesis supervisor, this work focuses only on evaluating content blockers, i.e., privacy-enhancing tools that block network requests for content that may also include trackers. This section describes two proposed evaluation methods. There are many approaches to evaluate the effectiveness of content blockers. However, no single comparison can definitively determine which tool is the most effective. While the methods presented here aim to provide meaningful insights, it is essential to understand the inherent limitations of each approach.

4.2.1 Ensuring Determinism Across Evaluations

It is necessary to compare the content blockers on the same dataset for a fair and deterministic comparison. Simply visiting a set of pages multiple times to test different content-blocking tools does not ensure determinism, as the pages may request different content each time they are visited.

Because of this variability, comparing the number of blocked requests across multiple page visits can be misleading. One tool might appear more effective simply because it encountered more blockable requests, not because it is inherently better at blocking them. For instance, a request to a specific tracker might be present in one test run but not in another, which would unfairly skew the results.

This issue of non-determinism can be partially mitigated by increasing the number of page visits. For example, Mazel et al. [21] visited each page multiple times and averaged the results to smooth out inconsistencies. However, this still does not guarantee determinism.

A truly deterministic approach involves capturing and saving the network traffic from each visited page to build a consistent dataset. Once this dataset is created, it can be replayed with different content-blocking tools under the same conditions. This allows for a deterministic comparison, as each tool can block the same set of requests.

4.2.2 Evaluation of Requests Blocked by Content-Blocking Tools

The first evaluation method measures how effectively each content blocker prevents outgoing network requests by calculating the number of blocked requests from a given dataset. It also considers the structure of the requests, accounting for cases where one request depends on another.

Performance Metrics

A simple comparison might rely on the built-in reports from each tool. However, such an approach is unreliable since the reports may not be precise, or the tool might not provide any reports at all. Thus, such a comparison is not feasible. Instead, a custom evaluation system is used, applying uniform metrics across all tools. Requests are replayed using the previously mentioned dataset, and the number of requests blocked by each tool can be directly compared, as the blockings occurred on the same data.

Tree-like Request Structure

However, a simple comparison of the number of blocked requests still presents a problem: web requests often have a tree-like structure. Therefore, some requests act as the entry point of other requests, meaning that blocking one parent request may prevent many child requests. For example, suppose a user visits page A, which includes a tracker from page B. B may then load additional content from C and D. Consequently, when a user visits page A, pages B, C and D are also requested.

Two requests are marked as blocked if one content blocker blocks C and D. Another tool might only block page B, thereby also transitively preventing requests for C and D. However, this second tool only blocked one request, even though its action effectively blocked three requests. A simplistic count would incorrectly suggest that the first tool was more effective in such a scenario.

The evaluation preserves the directed request tree to address the abovementioned issue. Figure 4.1 illustrates a simple request tree. The tree structure has been used to model request relations before. For example, Sanchez-Rola et al. [28] used a tree-like structure to model how requests create cookies. The parent-child relations between requests are obtained from the network logs. If a parent request is blocked, all its children are also considered blocked, which more accurately reflects the impact of blocking a request.

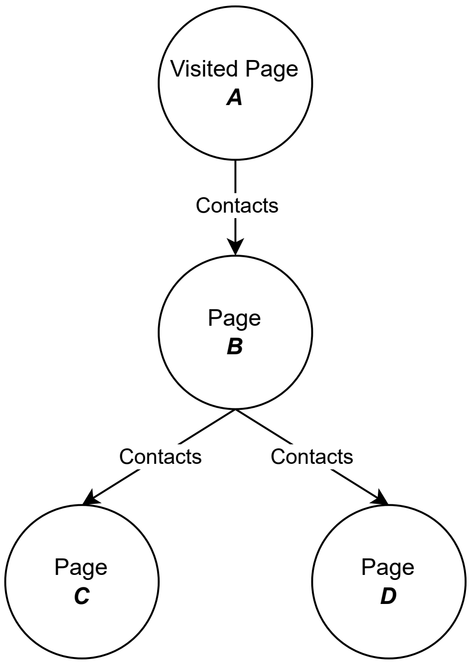

Figure 4.1: Illustration of a tree-like request chain which allows for accurate measurement of number of requests blocked.

Limitations

Despite its advantages, this evaluation has limitations. It only reflects behavior based on the pages and resources that were observed. Consequently, it is impossible to determine how the extension would behave on unvisited pages, which may contain different requests.

Another limitation lies in the subjectivity of filter lists, as discussed in Section 2.2.1. These lists are created by people, often with different priorities. For example, a tool might block more requests simply because its filter lists include more pages, even though other people would not block these requests. Therefore, a higher number of blocked requests does not necessarily imply one tool is better than another. A deeper analysis would require manual inspection of blocked requests to ascertain the validity of the blocking action. Even so, this evaluation method still offers insight into the general behavior of different content-blocking tools.

4.2.3 Evaluation of Anti-Tracking Capabilities of Content-Blocking Tools

The second method builds on the previous one by focusing on the anti-tracking capabilities of the content blockers. It measures how many JavaScript fingerprinting API calls were blocked due to blocking the resources that attempted to initiate them. These calls can reveal unique information about the user's browser and device, as discussed in Subsection 2.1.3.

Observing Fingerprinting API Calls

The JShelter browser extension, presented in Subsection 2.3.2, can be used to capture the API calls. JShelter includes a Fingerprint Detector based on Marek Saloň's master's thesis [27], which can detect the calls of APIs misused for fingerprinting in real-time. These API calls represent potential fingerprinting attempts since they may be used in the fingerprint calculation process.

No content blockers are active during the dataset creation, and JShelter's protections are disabled to avoid interference. The Fingerprint Detector logs fingerprinting API calls on each visited page, which can be matched with the associated network traffic logs.

Performance Metrics

This evaluation intends to quantify the anti-tracking capabilities of a given content-blocking tool. It uses the same tree-like request structure described in the previous evaluation in Subsection 4.2.2, but with additional information. Each node (i.e., each contacted resource) is assigned the number of fingerprinting API calls it caused during the visit. This allows using request trees and their transitive properties to obtain the number of fingerprinting attempts that could be prevented by blocking a specific request.

For example, if page A loads page B, which then loads C and D, and fingerprinting attempts are observed on all of them, a content blocker that blocks B would also prevent the subsequent calls from C and D as they would not be loaded. This would reduce the total number of fingerprinting API calls by all the calls caused by pages B, C and D. Figure 4.2 illustrates a request tree with associated fingerprinting attempts.

Limitations

This method inherits the limitations of the first evaluation. Additionally, it cannot confirm whether observed fingerprinting API calls were used for tracking. These APIs have

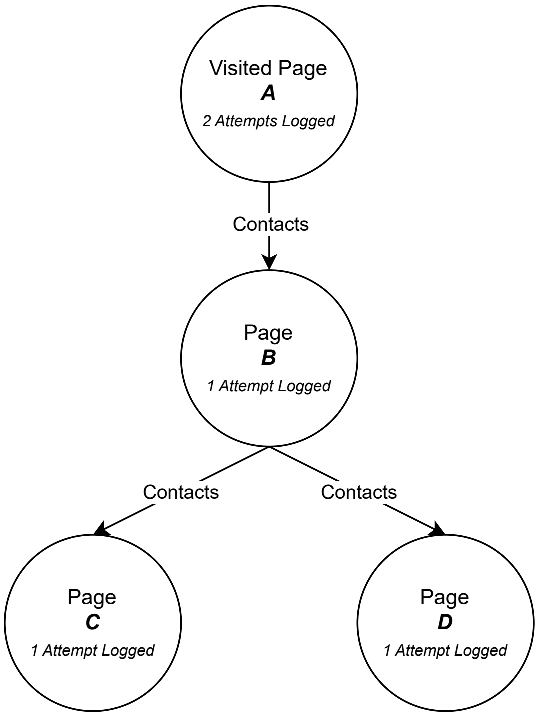

Figure 4.2: Illustration of a request tree where each node is assigned its API calls that can be misused for fingerprinting.

legitimate uses, so not all calls represent tracking behavior. Consequently, blocking more API calls that are potentially usable for fingerprinting might not be inherently better than blocking less. However, this evaluation still provides a way to compare how well different tools prevent potential fingerprinting, complementing the first request-based evaluation with a more privacy-oriented metric.

Chapter 5

Implementation of the Proposed Evaluation Methods

This chapter describes the implementation of the two evaluation techniques proposed in Section 4.2. The two evaluations are implemented as a single, comprehensive evaluation. The source code is available on GitHub[1]. The chapter outlines the technologies used and describes the implementation logic. In this chapter, various limitations are mentioned. Additional limitations are presented in Chapter 7.

To support comparison of privacy-oriented browsers, such as the Avast Secure Browser, which is available only on *Mac* and *Windows* desktop operating systems, *Windows 11* was selected as the operating system for the evaluation of the content-blocking tools. *Docker* was not used to containerize the system since running a Windows OS inside a Docker is not officially supported. Instead, the implementation can be run natively on a Windows machine or within the included virtual environment, provided the host machine supports nested virtualization since Docker is utilized to run a custom DNS server. The implementation is written in *Python*[2], chosen for its easy access to many libraries.

Given the objective of simulating user behavior, such as visiting web pages using a browser with or without privacy-preserving extensions, *Selenium* is employed for automation. Additional technologies used in the evaluation are discussed in their respective sections. The program can operate on either previously stored traffic or newly acquired data to ensure reproducibility and determinism across multiple evaluations. Figure 5.1 depicts the high-level overview of the evaluation system.

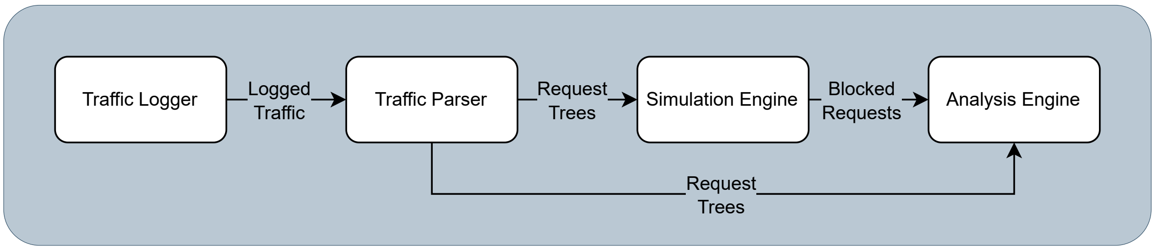

Figure 5.1: High-level overview of content-blocking tool evaluation system.

[1] https://github.com/Aenariss/DIP
[2] https://www.python.org/

The overview shows that the system consists of four primary components, each independently executable. The modularity ensures that the system can, for example, be configured to collect traffic data without performing any subsequent parsing.

5.1 Traffic Logger

The objective of the traffic logger is to collect data from designated sites or to load previously gathered data. The collected data for each page are *network traffic*, *fingerprinting API calls* and Domain Name System *(DNS)*[3] *responses*.

Logged network traffic allows the evaluation of a content-blocking tool's effectiveness by measuring the number of blocked requests. Observed fingerprinting API calls are used to determine a content-blocking tool's inherent anti-tracking capabilities. Additionally, DNS responses are utilized to ensure determinism.

Users can specify the pages from which the traffic logger should collect data to maximize configurability. The collection process is performed for each specified page and consists of the following steps:

- Start the DNS Response Monitor to capture DNS responses.
- Create a Selenium-driven Google Chrome browser instance[4].
- Load a custom configuration of the JShelter extension into the browser.
- Visit the specified page and collect data for user-defined duration. If the page fails to load within the user-specified maximum time, skip it and continue with the next.
- Quit the browser and the DNS Response Monitor and save the logged data.

5.1.1 DNS Responses and Logs

Logging DNS responses allows the evaluation to save the IP address and possibly *CNAME*s associated with each request. This data can be utilized to repeat the response later, ensuring determinism, since some content-blocking tools can use the IP address associated with a domain to block the request[5]. The DNS Monitor uses the *Scapy*[6] Python module.

CNAME Cloaking

Another reason for logging DNS responses is to address the issue of CNAME cloaking [10]. Figure 5.2 depicts how a DNS resolution should proceed for domain `tracker.example.com`. Suppose our content-blocking tool includes a rule that blocks `tracker.example.com`. In that case, the content-blocking tool would block the request before any DNS query could even be created.

To illustrate the CNAME cloaking issue, suppose a page requests data from domain `not-a-tracker.example.com`. Under normal circumstances, the DNS server would return an *A* record, as in the previous Figure 5.2. However, if a CNAME record exists for the domain instead of an A record, the resolution of `not-a-tracker.example.com` could,

[3]https://www.ietf.org/rfc/rfc1035.txt

[4]Traffic logging could theoretically work with other Chromium-based browsers, but traffic logger is implemented to utilize Google Chrome

[5]Tested with uBlock Origin 1.63.0 using a custom filter in Firefox. Also tested in Brave Browser (Chromium-based) with the same configuration – only works in Firefox.

[6]https://scapy.net/

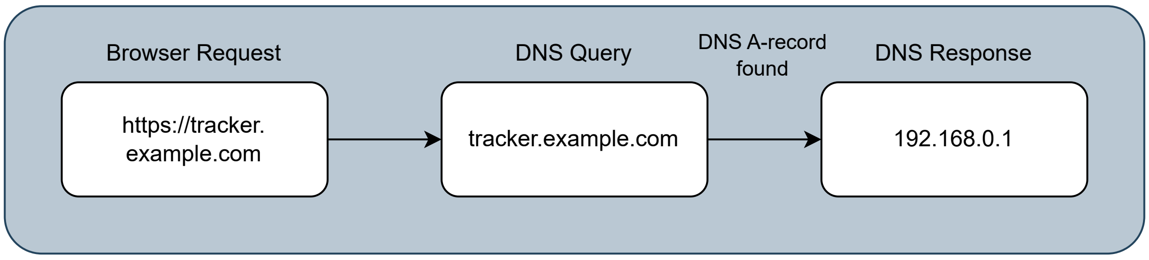

Figure 5.2: DNS resolution for domain `tracker.example.com` with no CNAME records present.

in reality, lead to `tracker.example.com`, which only then leads to the A record. Figure 5.3 depicts this situation.

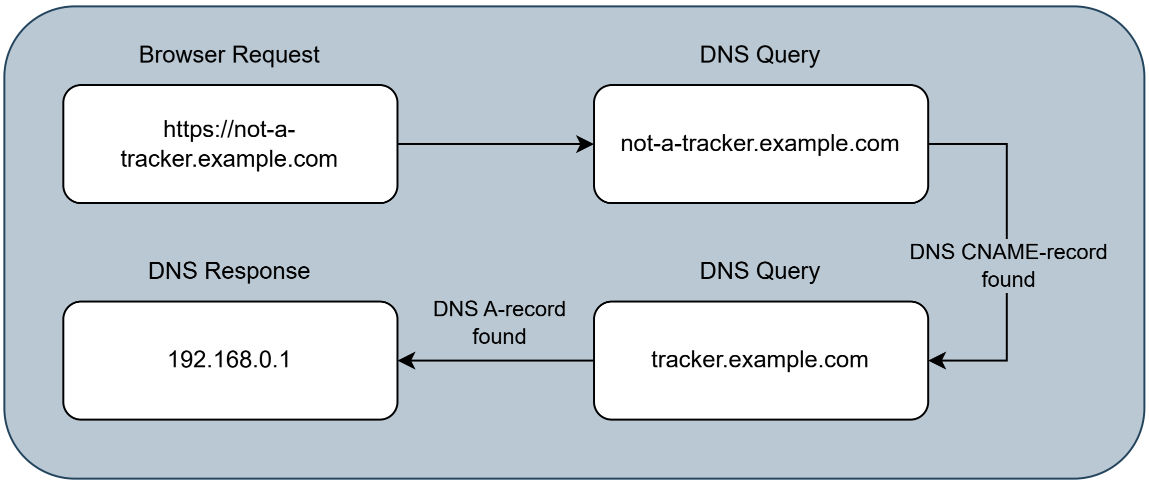

Figure 5.3: DNS resolution for domain `not-a-tracker.example.com` with a CNAME record present.

Suppose that the tested content-blocking tool does not block `not-a-tracker.example.com`, even though it resolves into the unwanted `tracker.example.com`. If the tool does not see the intermediate step in the resolution process, it does not block the request. CNAME responses are logged to replicate these situations. To replicate the DNS responses later, all A and CNAME responses are stored for each domain and its subdomains. Examples of observed DNS responses are shown in Listing A.1 in Appendix A.

5.1.2 Default JShelter Version

As previously mentioned, *JShelter* is used to collect fingerprinting logs for each page. By default, JShelter includes functionality to protect users from browser fingerprinting. However, since this would affect visited sites, it must be disabled.

In the pre-compiled version of JShelter, fingerprinting API calls are only logged should the user explicitly choose to do so. The button to start the logging is located inside a submenu, which is located in the extension menu. The submenu location is shown in Figure 5.4. After activation, tracing the callers causes a page reload and the submenu to populate with observed callers.

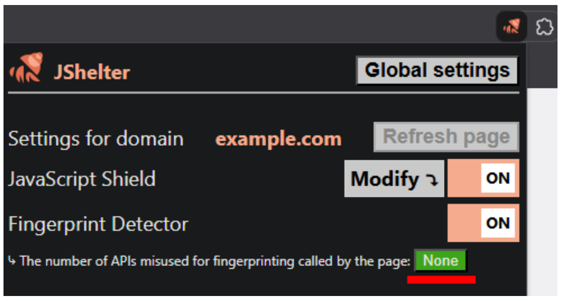
JShelter
Global settings
Settings for domain example.com
Refresh page
JavaScript Shield
Modify
ON
Fingerprint Detector
ON
↳ The number of APIs misused for fingerprinting called by the page: None

Figure 5.4: JShelter Fingerprint Detector button is located in the extension menu. A red line underscores the button. The menu must be opened to track callers of potentially fingerprintable APIs.

However, the page reload breaks the network logging. It leads to repeated entries in the logs, making it difficult to distinguish the original request tree. Additionally, if the initially visited page redirects the user to another page, the reload only affects the final page. Therefore, the repeated network logs would be incomplete since the original redirects would be lost.

Another issue with the default version of JShelter is that it exposes the caller data inside a submenu, which can only be opened by manually clicking the extension button. The submenu location is shown in Figure 5.4. Since Selenium does not support interacting with extension menus[7], automating the clicking process is not feasible.

5.1.3 Custom JShelter Version

I created a custom version of JShelter 0.19 with all anti-tracking functionality disabled to address the limitations of the default JShelter version. In the custom version, API caller tracking is enabled by default. Therefore, each time a page is visited, JShelter automatically tracks the callers by inserting the required JavaScript wrappers and populating its internal callers list. To obtain the fingerprinting data, JShelter is modified to automatically download a JSON fingerprinting report *five* seconds after a page was accessed. Therefore, a browser must remain on the page for at least five seconds to successfully download the report.

However, this approach has one severe limitation – JShelter sometimes fails to insert the wrappers correctly. The issue is present only in Selenium, which is used to create the dataset. It makes the results in the fingerprinting report empty. Testing with a newer version of JShelter (0.20) resulted in an even higher failure rate in the Selenium environment. This ultimately led to using the previous version 0.19.

Despite the abovementioned limitation, the JShelter Fingerprint Detector can still be used, albeit with limited results. Since all content-blocking tools are evaluated on the same dataset, missing fingerprinting data for some pages is equivalent to those pages not using any fingerprinting API calls. That is, in turn, equivalent to a smaller dataset being used.

5.1.4 Network Logs

I selected *Google Chrome* for traffic logging as it provides a convenient way to obtain network logs. Network logs are later used to create request trees. For each specified page, a new

[7]https://github.com/seleniumhq/selenium-google-code-issue-archive/issues/7805

instance of Google Chrome is created and configured to log performance events[8], including network logs. Listing 5.1 shows the command to enable performance logging.

```
chrome_options.set_capability("goog:loggingPrefs", {"performance": "ALL"})
```

Listing 5.1: Command to configure the browser to log performance events

Afterward, network logs are obtained by parsing each record and identifying whether it matches a `Network.requestWillBeSent`[9] event. If the record matches this event, the traffic logger saves it. Internal browser requests (e.g., requests starting with `chrome://`) are not included. For each `requestWillBeSent` event, three key attributes are logged:

- `documentURL`, renamed as `requested_for`: The URL of the document the request is loaded for.
- `request.url`, renamed as `requested_resource`: The URL of the requested data.
- `initiator`: The entity that initiated the request.

5.2 Traffic Parser

The traffic parser is responsible for reconstructing request trees from the logged data. The trees are created from the *fingerprinting logs* and *network logs*. The process is summarized in the following steps:

- Load fingerprinting groups from the JShelter FPD settings.
- Retrieve all wrapped APIs from the FPD settings and assign each API a primary group. The current implementation requires the FPD logs to originate from a Chromium-based browser.
- Parse fingerprinting logs for each visited site and count fingerprinting API calls of each resource. Assign the fingerprinting API calls to the primary groups.
- Construct a request tree for each network log. Assign fingerprinting API calls to each resource in the tree.

5.2.1 Fingerprinting Groups and APIs

The goal of parsing fingerprinting logs is to determine the number of fingerprinting API calls caused by each resource. According to the JShelter FPD groups configuration file[10], the API calls are divided into three main groups. Information about the group configuration and other parts of JShelter Fingerprint Detector is explained in great detail in the original thesis by Marek Saloň [27].

Each wrapped fingerprinting API belongs to a specific group, but the configuration file specifies many groups. If all groups were used as-is, the fingerprinting results would be overly detailed and challenging to understand. Each main group has multiple associated subgroups, which may also have their own subgroups. To improve readability, only the three main groups are used. These groups are:

[8] https://developer.chrome.com/docs/chromedriver/logging/performance-log

[9] https://chromedevtools.github.io/devtools-protocol/tot/Network/#event-requestWillBeSent

[10] https://pagure.io/JShelter/webextension/blob/main/f/common/fp_config/groups-lvl_0.json, accessed on March 4, 2025

- *BrowserProperties*: APIs that use browser properties representing the state of the browser, such as the *Navigator* object.
- *AlgorithmicMethods*: APIs that need to be dynamically calculated, such as the *Canvas* API.
- *CrawlFpInspector*: APIs identified by Iqbar et al. [18] as mostly used for fingerprinting purposes. It consists of APIs from both previous categories and some new ones.

As mentioned, each wrapped API belongs to a specific group. The APIs are defined in the JShelter FPD wrapper configuration. Each observed API's associated subgroup is changed to one of the three primary groups. For example, the API `Navigator.prototype.userAgent` belongs to a group *NavigatorBasic*, which in turn belongs to the primary group *BrowserProperties*. Therefore, the group assigned to this API is changed to *BrowserProperties*. Should the API also be included in the *CrawlFpInspector* category, this group, too, is associated with that API. This way, no group information is lost, only reduced.

The configuration for both groups and wrapped APIs was obtained directly from the JShelter project[11]. This ensures that future updates to JShelter FPD can be easily integrated by replacing the configuration files.

5.2.2 Associating Fingerprinting API Calls to a Resource

For each fingerprinting API call, the complete call stack is logged. However, only the *final caller* in the call stack is used to associate a call with a resource. For example, if script A calls script B, which then calls script C, and C uses a potentially fingerprintable API, this call is attributed to script C. If the callers of an API are not listed, the calls are skipped. Such calls have been observed to come mainly from the property `CSSStyleDeclaration.prototype.fontFamily`, which often reports thousands of calls but no callers.

5.2.3 Request Trees

Request trees are a vital part of the evaluation process since they facilitate the transitive properties of the evaluation. Request trees are created using both network and fingerprinting logs. The trees have a limitation, however. If a resource is requested multiple times, duplicate nodes are created, which distorts the analysis results. The duplication issue is explained further in Section 7.2. To partially address this issue, each resource in the tree can include any given follow-up resource only once among its list of children. Although this reduces the distortion caused by duplicate resources requested by the same parent, it also introduces a different kind of inaccuracy, as these resources are excluded from the analysis.

The process of associating fingerprinting statistics to each resource has already been discussed; the statistics are assigned to the corresponding node in the tree. If a resource is already present in the tree, the new node's fingerprinting API calls are assigned twice, distorting the results further. Each node in the tree has the following attributes:

- `resource`: The resource URL represented by the node.
- `children`: A list of nodes requested by this resource.
- `fp_attempts`: The number of fingerprinting API calls made by this node.

[11] https://pagure.io/JShelter/webextension/blob/main/f/common/fp_config/

- `blocked`: Whether this node was blocked by a content-blocking tool. It is only used during the analysis phase (described in Section 5.4).
- `root_node`: Whether the node is a *root* node or not.
- `parents`: The resources that requested this node.

The request trees are constructed using three attributes from network logs – `requested_for`, `requested_resource` and `initiator`. The network logs are parsed chronologically. During the construction process, two types of nodes can be observed, as illustrated in Figure 5.5, which represents a real example[12].

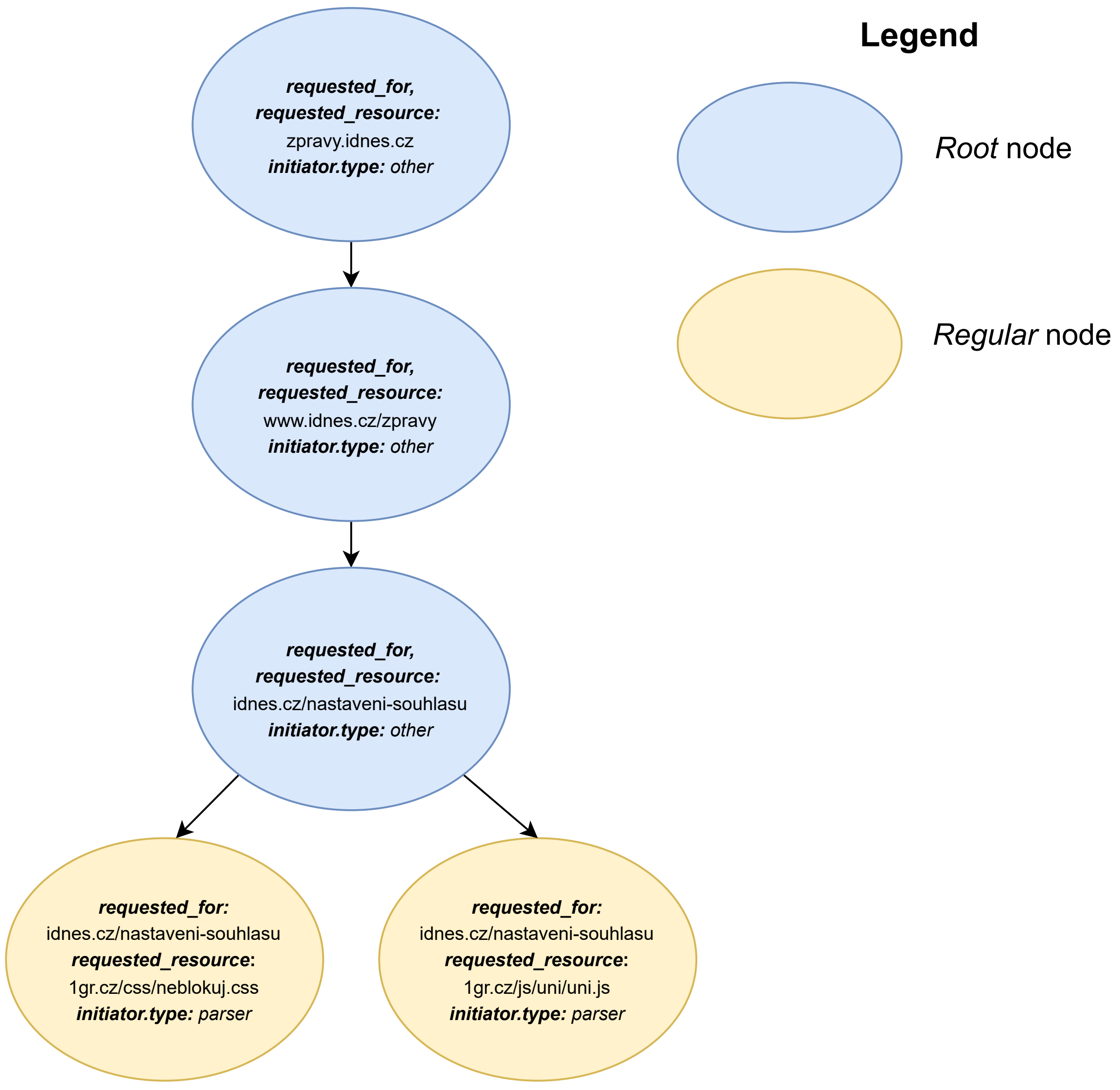
Legend
Root node
Regular node
requested_for, requested_resource: zpravy.idnes.cz ***initiator.type:*** *other*
requested_for, requested_resource: www.idnes.cz/zpravy ***initiator.type:*** *other*
requested_for, requested_resource: idnes.cz/nastaveni-souhlasu ***initiator.type:*** *other*
requested_for: idnes.cz/nastaveni-souhlasu ***requested_resource:*** 1gr.cz/css/neblokuj.css ***initiator.type:*** *parser*
requested_for: idnes.cz/nastaveni-souhlasu ***requested_resource:*** 1gr.cz/js/uni/uni.js ***initiator.type:*** *parser*

Figure 5.5: Illustration of *root* and *regular* nodes. For *root* nodes, `requested_resource` and `requested_for` must match while also having `initiator.type` as *other* and `initiator.url` not set. Regular nodes are all other nodes.

[12]The initially accessed URL was `https://zpravy.idnes.cz` on March 12, 2025. Part of the observed structure was omitted, and URL addresses were shortened for clarity.

5.2.4 Root nodes

The *root* nodes are characterized by the `requested_for` and `requested_resource` attributes being identical and the `initiator.type` being *other*. `initiator.url` must also not exist. These nodes represent main pages loaded in the browser (e.g., pages with URL shown in the address bar), including any redirects leading to the final page (i.e., the page URL of which is shown in the address bar). The last node in the sequence of root nodes is used as the parent node of all *anonymous* nodes (i.e., the nodes with no known parent). All unidentifiable fingerprinting API calls are also assigned to the last root node.

Root nodes have the previous root node set as their parent. If no previous root node exists, it was the very first root node observed – in such case, the node has no parent and represents the root of the whole tree. When one of the root nodes is loaded multiple times, i.e., two `Network.RequestWillBeSent` events have the exact attributes `requested_for`, `requested_resource` and `initiator.type`, the second such event is added as a child of the current root node as a *regular* node. The duplicate node has its associated fingerprinting API calls set to zero since they are already associated with the primary node.

5.2.5 Regular Nodes

Any node that does not match the specific characteristics of a *root* node is classified as a *regular* node. The root nodes have special parentage rules described above. The traffic parser uses three methods to determine the parentage of the regular nodes. However, since the trees may contain duplicate nodes, child nodes may be assigned to multiple parents, leading to redundant parent-child relations, which affect the subsequent analysis. The methods to determine the regular nodes parentage are described in the following subsections.

Initiator URL

When the attribute `initiator.url` is available, the tree is searched for matching parent nodes (i.e., nodes with the `resource` attribute matching this `initiator.url`).

Initiator Call Stack

When `initiator.url` is not available, the final record of the *JavaScript call stack* is utilized. Each record in the call stack has `url` specified, which represents the URL of the resource that caused the event. Nodes representing this URL are selected as parents.

This approach has another limitation in addition to the one described above. The limitation is that calls may have come from a dynamically generated code, such as from `eval`[13], which would show the `url` attribute as an empty string. This situation is illustrated in Listing 5.3. It was caused by calling `eval` as shown in Listing 5.2. In this case, the call stack is traversed until a valid URL is obtained. This URL is then set as the parent, even though it may not reflect the actual parent. The current root node is selected as the parent if no valid URL exists.

```
eval("const script2 = document.createElement('script'); script2.src = '/
    script2.js'; document.body.appendChild(script2);")
```

Listing 5.2: Loading a script using *eval* which resulted in empty *url* parameter in the call stack.

[13]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

```
"initiator": {
    "stack": {
        "callFrames": [
            {
                "functionName": "eval",
                "url": ""
            },
            {
                "functionName": "",
                "url": "http://localhost:8000/script1.js"
            }
        ]
    },
    "type": "script"
}
```

Listing 5.3: Example of empty `url` attribute in the call stack caused by the use of `eval`, which was called by *script1.js*, but the empty `url` attribute of the record does not reflect it. The first record in the list of call frames represents the final call. This presented call stack is edited and does not contain all logged attributes.

Other

If none of the above is applicable, the current root node is selected as the parent. In case a valid parent could not be found for any reason (e.g. because an *iframe* with defined `srcdoc`[14] property was opened, so the `initiator.url` was specified as `about:srcdoc`), the parent is also set as the current root node.

5.3 Simulation Engine

The simulation engine simulates which of the requests from the initial dataset would have been blocked by a specified content-blocking tool. The simulation attempts to partially replicate the conditions of the initial visit. To understand what requests a given content-blocking tool would block, several steps are taken:

- Set up the simulation. The simulation includes a custom DNS server to replicate DNS responses and a custom web server to request all the resources.
- Configure a client with the content-blocking tool to be tested. Visit the custom web server with the client.
- Obtain results indicating what requests were blocked.

Each of these steps is described in greater detail in the following subsections. The simulation engine also creates firewall rules to prevent requests from leaving the machine to save network bandwidth.

[14] https://developer.mozilla.org/en-US/docs/Web/API/HTMLIFrameElement/srcdoc

5.3.1 Custom DNS Server

A custom DNS server is utilized to resolve DNS requests as they were observed to reproduce the conditions present during the initial visit. It allows the tested tools to block requests depending on the IP address to which they were resolved. As mentioned in Subsection 5.1.1, this step also allows the tested tools to use methods allowing them to block requests utilizing CNAME cloaking. However, CNAME cloaking can only be prevented in Firefox since Chromium does not provide the necessary API.

I chose *BIND 9*[15] server for this purpose. It runs inside a Docker container, built using an existing image[16]. To configure the system to use the custom DNS server as its default, the `Set-DnsClientServerAddress`[17] Powershell command is executed from within the program. After the simulation ends, the same command is used to reset Windows settings back to the default state, using the DNS server address provided by Dynamic Host Configuration Protocol *(DHCP)* – however, any custom configuration used before the simulation launch is lost in the process.

Zone Files

Zone[18] files are used to define how domain names should be resolved. A separate zone file is created for each observed second-level domain[19].

All DNS logs are merged to generate zone files correctly, and duplicates are removed. The merging is possible since the DNS responses remain valid for a specified period of time. Therefore, during dataset creation, repeated DNS queries should return identical responses. However, a DNS reply may have changed during dataset creation. The merging erases this change, slightly limiting the *correctness* of DNS replies.

If a DNS response contains multiple IPv4 addresses, only the *first* address is used in the zone file. This ensures that all queries for a given domain are resolved into the same address. Similarly, if a response contains multiple CNAMEs, only the first CNAME is added to the zone file. However, no information is lost, as artificial records for all observed CNAMEs are present in the logs. These artificial records ensure that every CNAME has its corresponding zone file, as illustrated in Listing A.1 in Appendix A[20].

The simulation engine appends a record to custom `named.conf` file for each zone file. `named.conf` is used to configure the *BIND 9* server to use the created zone files. After all the zone files are created and included in the `named.conf`, the zone files and the `named.conf` are uploaded to the Docker container, and the server is restarted to apply the changes.

[15]https://www.isc.org/bind/

[16]https://hub.docker.com/r/internetsystemsconsortium/bind9

[17]https://learn.microsoft.com/en-us/powershell/module/dnsclient/set-dnsclientserveraddress

[18]https://datatracker.ietf.org/doc/html/rfc1035

[19]e.g., for observed address `tracker.example.com`, zone file `example.com` is created, containing a record for `tracker`. The record can either be a CNAME pointing to another domain or an A record specifying an IPv4 address.

[20]The accessed domain was `ads.pubmatic.com`. Response for this domain only was observed. The other records were automatically added to be used with CNAMEs. Each of the primary records has its own zone file.

5.3.2 Custom Web Server

The custom web server is responsible for repeating the requests for all resources observed during dataset creation. *Flask*[21] is used to build a simple locally hosted webpage that dynamically loads recorded resources using JavaScript `fetch`[22].

To monitor request completion, `fetch` is overridden to track how many resources have already been loaded and how many are still waiting. These values are stored in the *window*[23] object, allowing the connected client to obtain these values to determine whether all requests have been loaded.

However, calling `fetch` to load tens of thousands of resources, depending on the size of the dataset, may be too intensive in terms of required computing power. Thus, the requests are fetched in batches to improve performance.

5.3.3 Firewall Rules

To minimize network bandwidth usage and to prevent unnecessary *spam* towards the servers that host the observed content, firewall rules are configured to block all outgoing `TCP` traffic on ports `80` and `443`, effectively blocking all outgoing *HTTP* and *HTTPS* requests. After the simulation ends, these firewall rules are removed.

An additional benefit of this approach is that since all requests are blocked immediately due to the firewall, the simulation can proceed much faster since the simulation engine does not need to wait for the server to answer. Content-blocking tools apply their blocking rules even before a request is sent[24]. Thus, if a request is blocked by a content-blocking tool, it happens before the firewall may intercept it, ensuring correct results.

5.3.4 Client Setup

A Selenium driver is created based on the user-specified configuration to load a selected content-blocking tool. Users can choose whether to use Firefox or Google Chrome. Users can also use a *custom* browser. This browser, however, needs to be compatible with Selenium Chrome WebDriver. If a custom browser is used, users also need to specify the binary file that launches the custom browser and the correct *Chromedriver*[25] path. Unfortunately, selecting the correct Chromedriver using *Selenium Manager*[26] does not work when using a custom browser. Depending on the browser used, users may also need to specify the path to the *profile*. This setting is necessary for Avast Secure Browser.

Users can specify the list of extensions to be loaded into the selected browser. Any extension in the *.crx* format for Google Chrome and *.xpi* format for Firefox is supported. If the user selects Google Chrome, they can specify the version to be used, which is loaded using Selenium Manager. For Firefox, users can choose to disable its inherent anti-tracking functionality.

Some tools require manual activation after browser launch. Automating this process is not viable due to UI changes or using an unknown content-blocking tool. After the browser

[21]`https://flask.palletsprojects.com/en/stable/`

[22]`https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API`

[23]`https://developer.mozilla.org/en-US/docs/Web/API/Window`

[24]This has been tested with extensions for both Google Chrome and Firefox browsers. uBlock Origin can also filter content later, but at that point, an initial request would have already been sent `https://github.com/gorhill/uBlock/commit/6acf97bf51`.

[25]`https://developer.chrome.com/docs/chromedriver`

[26]`https://www.selenium.dev/documentation/selenium_manager/`

is launched, users are given time to enable all the settings their configuration may require. Initializing some tools also takes time, and the wait period provides it.

5.3.5 Server Visit and Result Collection

The firewall and DNS server are activated just before the client connects to the server. Activating them at this point ensures that the browser or extension has enough time to download everything it may need to function correctly.

As mentioned, during the visit, the client fetches all observed resources. After all of the resources have been loaded, the client leaves the server. The loading status is indicated by variables stored in the *window* object. After all requests are complete, the logs from the visit are saved. Depending on the type of browser used, the logs can have different form and purpose. However, both are used to establish what resources have been blocked.

Google Chrome

For Google Chrome, blocked requests are logged in the console. If a request were blocked by a content-blocking tool, it should show in the console as `net::ERR_BLOCKED_BY_CLIENT` or `net::ERR_BLOCKED_BY_ADMINISTRATOR` error, depending on the implementation. Either way, both these errors mean the request was not sent. Since the logs contain all required information, they are extracted from the Selenium driver and saved.

Mozilla Firefox

For Firefox, the situation is more complex. Unlike Google Chrome, Firefox does not log the blocked requests into the console. Instead, the information is available in the *Network Monitor*[27]. The problem with Firefox is that this data is not easily accessible to Selenium.

I developed a custom extension to work around the problem. This extension utilizes *webRequest*[28] API to track which requests were successfully sent. Specifically, a listener is added to the `onSendHeaders` event. For each such event, the URL of the requested resource is logged. After all requests have been loaded, Selenium accesses and saves the logs. Blocked requests are not logged since they never trigger the `onSendHeaders` event. Based on this logic, blocked requests are computed as the difference between the list of all resources and the list of logged resources.

5.4 Analysis Engine

The analysis engine uses the results from the simulation to assess the behavior of a selected content-blocking tool. Request trees are analyzed one by one. If a resource was blocked during simulation, this block is projected into the request tree. After all the trees have been analyzed, the partial results are aggregated to form the final evaluation result.

As previously mentioned, simulation logs determine which pages the selected content-blocking tool blocked. The logs are parsed differently, depending on whether they originate from Google Chrome-based browser or Firefox, as explained in Subsection 5.3.5 – however, in both cases, the log parsing returns a list of pages that have been blocked during the simulation.

[27]https://firefox-source-docs.mozilla.org/devtools-user/network_monitor/
[28]https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest

The analysis computes several metrics that describe the behavior of the tested content-blocking tool. The results include both content-blocking and anti-tracking performance. The anti-tracking performance is based on the number of prevented potential fingerprinting API calls. All metrics are explained using the example request tree in Figure 5.6.

In the figure, each node is marked as *blocked*, *transitively blocked* or *loaded*. A *blocked* node is one that a content-blocking explicitly blocked. A *transitively blocked* node was indirectly blocked due to its parent node being blocked. A *loaded* node represents a resource that was not blocked. Each node is labeled with the corresponding resource address and the number of observed fingerprinting API calls for which this resource is responsible.

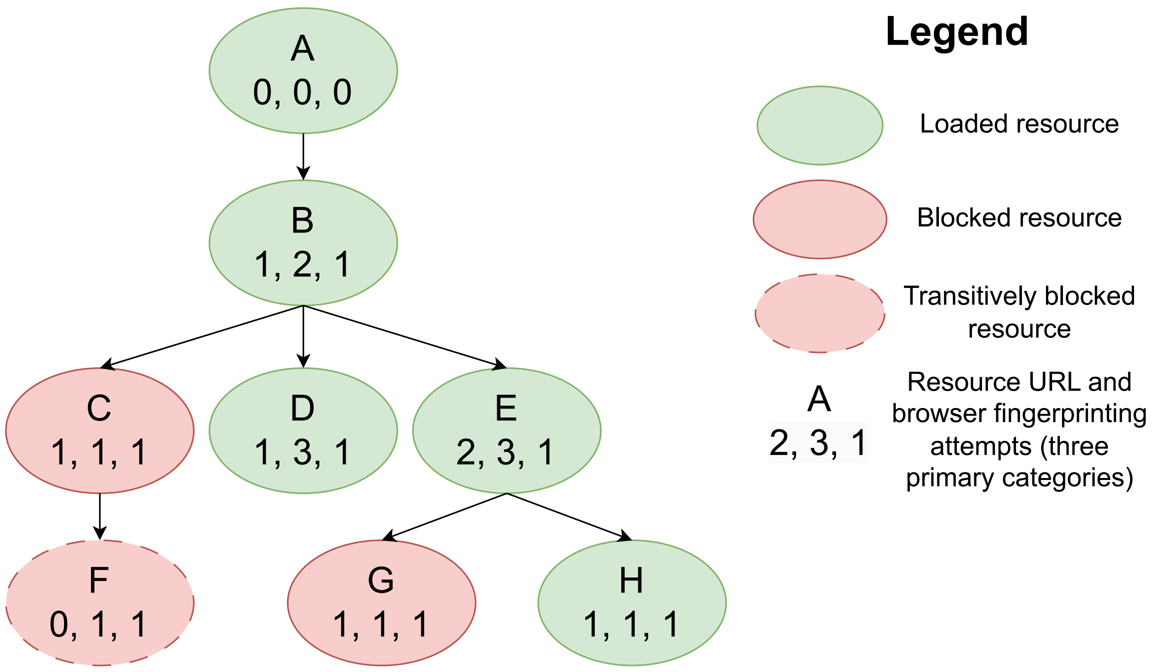

Figure 5.6: Visualization of a request tree with assigned fingerprinting API calls and block statuses as a visual aid to explain the measured metrics. Each node contains its URL (represented by a single letter in the node) and number of observed fingerprinting API calls, split into three primary categories as described in Subsection 5.2.1.

5.4.1 Requests Analysis

For the request blocking analysis, most metrics consist of only a single number, with one exception. All the metrics are explained below in the same format as in the created results file.

requests_observed This metric should be consistent for all tools tested on the same dataset. It represents the total number of requests that were observed in the dataset. In Figure 5.6, this metric corresponds to the number of nodes, 8.

requests_blocked_directly Directly blocked requests are those that were explicitly blocked by a content-blocking tool. None of their parent nodes is blocked. This metric represents the total number of such nodes in the request tree – in Figure 5.6, it is 2.

requests_blocked_transitively Transitively blocked requests are those blocked due to any preceding parent node being blocked. Since the parent was blocked, these requests would have never occurred if the tested browser visited the page. In Figure 5.6, there is only one such node, F, which was blocked because C, its parent, was blocked. Thus, this metric would be 1.

requests_blocked_in_total The sum of *requests_blocked_directly* and *requests_blocked_transitively*. For figure 5.6, this number is 3.

The following metrics are experimental, meaning their results are not as easy to grasp as the previous metrics. They primarily measure the *aggresiveness* of a content-blocking tool, with more aggressive tools blocking parent requests before any child request can be loaded. The metrics may help distinguish between similarly performing tools and quantify the tool's aggressiveness.

blocked_requests_with_followup_requests This metric represents the number of directly blocked requests that have at least *one* child node associated, i.e., they were responsible for at least one follow-up request. In Figure 5.6, there is only one node that was blocked that also has at least one child node – node C, making this metric 1.

average_request_block_level The average request block level represents the average level at which blocking occurred in the tree. Only the first block in each subtree is considered. The root of the tree is considered as level 1. In Figure 5.6, node A would be level 1, node B would be level 2, nodes `C, D, E` would be level 3 and nodes `F, G, H` would be level 4. Since only the first blocked node in each subtree is taken into account, this number is counted from nodes C and G, which is $\frac{3+4}{2} = 3.5$. This metric is only counted for websites that include at least one blocked request.

blocked_subtrees_data This metric is composed of five numbers. It describes blocking of primary *subtrees* in the request tree. In the context of this metric, subtrees are defined by the following procedure: A given request tree is traversed until the first node with at least *two* child nodes is found. Each child node then serves as a root of its primary subtree.

In Figure 5.6, the tree is traversed until node B is reached. Since node B has *three* children, nodes `C, D` and E, these child nodes are used as root nodes of three primary subtrees. Five numbers are measured as part of this metric:

subtrees_fully_blocked The number of primary subtrees blocked at the root level. In Figure 5.6, only subtree with root node C is fully blocked, making this metric 1.

subtrees_partially_blocked The number of primary subtrees that include a blocked node anywhere but at the root level. In Figure 5.6, only subtree composed of nodes `E, G` and H is partially blocked since one of its nodes, the node G is blocked, making this value 1.

subtrees_not_blocked The number of primary subtrees with no blocked nodes. In Figure 5.6, only subtree D includes no blocked nodes. Thus, the metric is 1.

subtrees_in_total The total number of primary subtrees observed. It is the sum of the three metrics above – subtrees not blocked anywhere and those blocked partially or fully. In Figure 5.6, this metric is 3. This metric should be consistent for all tools evaluated on the same dataset.

trees_with_blocked_root_node The number of times the whole request tree was blocked due to a *root* node being blocked. In this context, *root* nodes mean those described in Subsection 5.2.4. If any root node were blocked in a tree, this metric for such a tree would be 1. No root blocking occurs in Figure 5.6, making this metric 0. However, if node A or B were blocked, this metric would be set to 1, and all three primary subtrees would be considered fully blocked. If a request tree was blocked because a root node was blocked, the page would be inaccessible, and the user would be presented with an error page. The user must manually interact with the error page to access such a page.

5.4.2 Fingerprinting Analysis

The fingerprinting analysis consists of four metrics, listed in the same format as in the produced results file. As described in Subsection 5.2.1, the fingerprinting API calls are split into three categories. Therefore, each fingerprinting metric is composed of three numbers, with each number describing the corresponding category – *BrowserProperties*, *AlgorithmicMethods* and *CrawlFpInspector*.

fpd_attempts_observed This metric should be consistent for all content-blocking tools evaluated on the same dataset. It represents the total number of fingerprinting API calls observed across all pages in the dataset. In Figure 5.6, this metric is the sum of attempts logged for each category for each node – $(7, 12, 7)$.

fpd_attempts_blocked_directly Directly blocked fingerprinting attempts represent the fingerprinting API calls associated with nodes that were blocked directly, as described above. In Figure 5.6, this metric is the sum of fingerprinting API calls observed for nodes C and G, which is $(2, 2, 2)$.

fpd_attempts_blocked_transitively Transitively blocked fingerprinting attempts are the API calls associated with transitively blocked nodes, as explained above. If a parent node was blocked, none of its child nodes would be loaded, and their corresponding fingerprinting API calls would be prevented. In Figure 5.6, this metric is $(0, 1, 1)$, which corresponds to the F node, which is the only transitively blocked node.

fpd_attempts_blocked_in_total The number of fingerprinting attempts blocked in total is the sum of *fpd_attempts_blocked_directly* and *fpd_attempts_blocked_transitively*. In Figure 5.6, this metric is $(2, 3, 3)$.

Chapter 6

Results of the Evaluation and Comparison of Selected Tools

This chapter presents the results of the evaluation process described in Chapter 5. The results are displayed in tables and discussed. The evaluated tools are described in Section 2.3, except for JShelter, which is not evaluated as it is not a content-blocking tool but is used only for dataset creation. The evaluation uses the metrics presented in Subsections 5.4.1 and 5.4.2. All tools were tested in their default state. The only manual interaction involved enabling the Ghostery extension, activating the content-blocking functionality of Avast Secure Browser after it was installed, and deactivating its *Web Shield* functionality that interfered with the test page.

The implementation was validated through unit testing using the *unittest*[1] Python module. Code coverage analysis, performed with the *coverage*[2] module, reported that 100 %[3] of the code was executed during testing.

Logging times were recorded across a sample of 100 web pages using the `Measure-Command`[4] utility to assess the performance of the data collection process. The measurement was performed on a system with an 8-core *Ryzen 7 9800X3D* CPU and *32 GB* RAM. The traffic logger was configured to remain on a page for 7 seconds, with a timeout set to 10 seconds. Of the 100 pages tested, 99 loaded successfully, and the entire collection process took 2,574 seconds. This corresponds to an average of 25.74 seconds per page to complete all data collection, including overhead from browser automation and traffic capture.

The dataset was collected on March 18, 2025 and consists of 937 pages, with fingerprinting statistics successfully gathered from 833 of them. The dataset is based on pages obtained from the *Top 1000 Websites – DataForSEO*[5] list for Czechia. The evaluated browsers (i.e., browsers with content-blocking capabilities and no additional content-blocking extensions present) are:

[1]https://docs.python.org/3/library/unittest.html

[2]https://coverage.readthedocs.io/en/7.8.0/

[3]The coverage configuration excluded the `if __name__ == "__main__"` block, which only runs when the script is executed directly (e.g., with `python start.py`). This block only launches the initial start function, which was tested separately. If this block were included, overall coverage would be only 99 %.

[4]https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/measure-command

[5]https://dataforseo.com/free-seo-stats/top-1000-websites

- **Avast Secure Browser**, version 136.0.29662.17. The development version was used due to compatibility issues with Selenium. Chromedriver 136 was used for automation.
- **Brave browser**, version 1.76.82. Chromedriver 134 was used for automation.
- **Mozilla Firefox**, version 136.0.3.

All extensions were tested in both Firefox and Google Chrome, the only exception being uBlock Origin, which is not available in Google Chrome anymore due to its lack of Manifest 3 support. uBlock Origin Lite was used instead. The extensions evaluated in Google Chrome are:

- **Adblock Plus**, version 4.15.0.
- **Ghostery**, version 10.4.25.
- **Privacy Badger**, version 2025.1.29.
- **uBlock Origin Lite**, version 2025.3.2.1298.

For Firefox, the evaluated extensions are:

- **Adblock Plus**, version 4.15.0.
- **Ghostery**, version 10.4.25.
- **Privacy Badger**, version 2025.1.29.
- **uBlock Origin**, version 1.62.0.

6.1 Effectiveness in Blocking Requests

This section presents the results of the request-blocking evaluation. The metrics used for analyzing requests are detailed in Subsection 5.4.1. A total of *100,753* requests were analyzed. Blocked requests serve as a metric for evaluating the effectiveness of content-blocking tools, as it reflects their capability to prevent the loading of third-party content, tracking scripts, and other potentially unwanted resources.

However, a higher number of blocked requests does not necessarily indicate better performance, as some of the blocked content may not have been considered undesirable by all users. Additionally, the results do not take into consideration the reaction of the page to blocking the requests, as described in Section 7.1. The results are also affected by the duplication issue described in Section 7.2.

The analysis of the results begins with statistics on blocked root nodes, as this metric highlights an important limitation of the evaluation method. Root nodes are described in Subsection 5.2.4. The evaluation is designed to work correctly with tools that rely on static filter lists. However, if a tool uses another method, such as heuristic-based blocking, the results may not be entirely accurate due to how the simulation is designed. The issue is further discussed in the next chapter, in Section 7.3.

6.1.1 Trees With Blocked Root Node

Avast Secure Browser, Brave browser, and the Privacy Badger extension blocked some pages at the root level. However, manual verification showed that these pages loaded properly in an actual browser visit. This suggests that these tools use internal heuristics or dynamic filtering rules that triggered the blocking behavior based on how the simulation loaded resources. Privacy Badger, for example, explicitly states that it uses an algorithmic approach, which aligns with this behavior. The issue might be present even with other tools. However, it is not apparent from the results.

Due to the abovementioned issue, this metric is *not reliable* for these tools. Furthermore, when these tools blocked root pages in the simulation, the analysis engine subsequently marked all requests within the corresponding request trees as blocked, which distorted the results of all subsequent comparison. Removing the problematic pages from the dataset would not fix the issue, as the tools may have also blocked other requests they normally would not, or conversely, they may have missed requests they otherwise would have blocked – the results would remain distorted. Consequently, all evaluation results for these tools, particularly for Privacy Badger, are unreliable. Due to this distortion, the results of Privacy Badger are excluded from the comparison and are not commented on, but they are included in the tables for completeness. Avast Secure Browser and Brave are still included, as their result distortion might not be as severe. Table 6.1 shows the number of root nodes blocked by each tool. Apart from the three mentioned, no other tools blocked any root nodes.

Tool	Trees With Blocked Root Node
Avast Secure Browser	3
Brave browser	1
Firefox browser	0
Chrome Adblock Plus	0
Firefox Adblock Plus	0
Chrome Ghostery	0
Firefox Ghostery	0
Chrome Privacy Badger	31
Firefox Privacy Badger	31
Chrome uBlock Origin Lite	0
Firefox uBlock Origin	0

Table 6.1: Results of the evaluation of the number of trees with blocked root node per tool. The results of Privacy Badger are unreliable as the tool uses an algorithmic approach to block requests. Similarly, Avast Secure Browser and Brave browser blocked several root nodes incorrectly, as manual verification showed that in regular browsing session, those pages are not blocked.

6.1.2 Directly and Transitively Blocked Requests

Table 6.2 summarizes the number of requests blocked directly, transitively, and in total. Firefox's built-in content blocking functionality blocked the fewest requests by a considerable margin, indicating significantly weaker default performance. This outcome can be attributed to Firefox not blocking tracking content by default[6].

[6]https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop#w_what-enhanced-tracking-protection-blocks

Across all evaluated tools, more requests were blocked transitively than directly, with the exception of the Firefox browser. Furthermore, all evaluated extensions blocked an equal or higher number of requests in Firefox compared to their Chrome counterparts when considering transitively blocked requests.

Adblock Plus exhibited identical performance in both browsers. In other extensions, performance across browsers slightly varied. Interestingly, uBlock Origin Lite in Chrome blocked more requests in total than the regular uBlock Origin extension in Firefox. In practice, blocking more requests should reduce page load times and network bandwidth usage, as blocked resources are not downloaded.

Tool	Requests Blocked Directly	Requests Blocked Transitively	Requests Blocked in Total
Avast Secure Browser	4,484	8,375	12,859
Brave browser	9,513	13,908	23,421
Firefox browser	147	17	164
Chrome Adblock Plus	4,132	6,011	10,143
Firefox Adblock Plus	4,132	6,011	10,143
Chrome Ghostery	9,286	13,425	22,711
Firefox Ghostery	9,133	15,596	24,729
Chrome Privacy Badger	8,476	14,519	22,995
Firefox Privacy Badger	8,491	14,520	23,011
Chrome uBlock Origin Lite	9,238	15,070	24,308
Firefox uBlock Origin	9,202	15,085	24,287

Table 6.2: Results of the evaluation of requests blocked directly, transitively, and in total. The low Firefox results are caused by it not blocking tracking content by default.

6.1.3 Blocked Requests with Follow-Up Requests

Table 6.3 presents the number of blocked requests that triggered at least one follow-up request. Firefox's performance was again the weakest, consistent with previous results. In contrast, Ghostery for Firefox blocked the highest number of parent requests that caused further network activity, followed by Brave.

Avast Secure Browser and Adblock Plus blocked fewer parent requests compared to the best-performing tools, indicating a more conservative blocking strategy. For example, Adblock Plus blocked only *41* % of the total number of requests blocked by Ghostery for Firefox but only *31.7* % of the parent requests. Similarly, Avast Secure Browser blocked *52* % of Firefox Ghostery's total blocked requests but only *38* % of the parent requests. These results suggest that Adblock Plus and Avast Secure Browser allow more initiating requests to pass through, potentially resulting in more unwanted content being partially loaded.

6.1.4 Average Request Block Level

In addition to the number of blocked parent requests, Table 6.3 also presents the average request block level. As previously noted, Firefox blocked the fewest requests overall.

Additionally, these requests tended to be deep within the request tree. Nevertheless, Firefox blocked at least one request in only *81* out of *937* collected trees, making this metric statistically less reliable for Firefox.

Among all evaluated tools, Ghostery for Firefox achieved the lowest average block level, indicating that it blocked requests closest to the root of the request tree. It was closely followed by both versions of uBlock Origin. Blocking requests near the root of the tree is more effective, as it prevents entire chains of unwanted requests from being made. This strategy enhances privacy by transitively blocking more requests and also improves the tool's overhead by reducing the number of subsequent requests that require evaluation against blocking lists.

Tool	Blocked Requests with Follow-Up Requests	Average Request Block Level
Avast Secure Browser	750	3.548
Brave browser	1,872	3.454
Firefox browser	5	4.131
Chrome Adblock Plus	626	3.735
Firefox Adblock Plus	626	3.735
Chrome Ghostery	1,790	3.433
Firefox Ghostery	1,972	3.283
Chrome Privacy Badger	1,697	3.579
Firefox Privacy Badger	1,698	3.577
Chrome uBlock Origin Lite	1,843	3.298
Firefox uBlock Origin	1,853	3.301

Table 6.3: Results of the evaluation of blocked requests with follow-up requests and average block level.

6.1.5 Subtree Blocking

Table 6.4 shows how many subtrees were fully or partially blocked. In total, *60,837* subtrees were observed in the dataset. Both versions of uBlock Origin and Ghostery for Firefox fully blocked the most subtrees, indicating more aggressive blocking behavior than other tools. Brave and Avast Secure Browser's fully blocked subtrees results are distorted, as they blocked pages at the root level, which resulted in all subtrees on those pages being counted as fully blocked. Nevertheless, Brave partially blocked the most subtrees. Interestingly, Ghostery for Chrome partially blocked many more subtrees compared to its Firefox counterpart. Aggressive subtree blocking can enhance privacy and reduce page load times by preventing large sections of content from loading, as the subtrees are blocked at the root.

6.2 Anti-Tracking Performance

This section presents the results of the evaluation of blocked API calls often misused for fingerprinting. However, each individual call may not necessarily represent an actual attempt to compute a user's fingerprint. Blocking potential fingerprinting API calls is essential for preserving user privacy, as these techniques can uniquely identify and track users across

Tool	Subtrees Fully Blocked	Subtrees Partially Blocked	Subtrees Not Blocked
Avast Secure Browser	2,716	561	57,560
Brave browser	5,949	1,014	53,874
Firefox browser	42	52	60,743
Chrome Adblock Plus	1,814	528	58,495
Firefox Adblock Plus	1,814	528	58,495
Chrome Ghostery	5,851	944	54,042
Firefox Ghostery	6,447	636	53,754
Chrome Privacy Badger	6,727	1,027	53,083
Firefox Privacy Badger	6,733	1,028	53,076
Chrome uBlock Origin Lite	6,487	673	53,677
Firefox uBlock Origin	6,454	671	53,712

Table 6.4: Results of the evaluation of subtrees blocking.

sites. In practical terms, the more potential fingerprinting API calls a tool blocks, the more likely it is to prevent actual tracking, making these tools more reliable choices for privacy-oriented users.

As discussed in Subsection 5.1.3, JShelter occasionally fails to log fingerprinting API calls correctly. The issue was observed on *11.1 %* of the visited pages. While this could indicate that no fingerprinting API calls were present on those pages, manual testing confirmed that fingerprinting did occur on the tested pages, and JShelter failed to log it.

The metrics used to analyze fingerprinting API calls are described in Subsection 5.4.2. The same limitations outlined in Section 6.1 for evaluating blocking effectiveness also apply here. Privacy Badger is again excluded from the commentary due to its distorted results, although it is left in the results for completeness. Brave and Avast Secure Browsers's anti-tracking results are also slightly distorted due to their root node blocking, which caused indirect blocking of all fingerprinting API calls on those pages. The total fingerprinting API calls observed per category across the dataset are:

- *Browser Properties*: *307,226*
- *Algorithmic Methods*: *16,011*
- *Crawl Fp Inspector*: *7,460*

The results do not include API calls caused by unknown callers, as discussed in Subsection 5.2.2. Should such calls be included, the attempts in categories *Browser Properties* and *Crawl Fp Inspector* would remain the same, but the total attempts in *Algorithmic Methods* would increase 13.5×, while the number of blocked *Algorithmic Methods* by the tools would largely remain the same.

6.2.1 Directly Blocked Calls To APIs Misused For Fingerprinting

Table 6.5 shows the number of potential fingerprinting attempts each tool blocked directly. Ghostery for Firefox blocked the most API calls in the *Browser Properties* category, closely followed by both versions of uBlock Origin. Both versions of uBlock Origin blocked the most

Algorithmic Methods, slightly outperforming Ghostery for Firefox. For *Crawl Fp Inspector*, Brave blocked the most calls, surpassing all other tools in this category.

Tool	Browser Properties	Algorithmic Methods	Crawl Fp Inspector
Avast Secure Browser	29,556	877	635
Brave browser	55,453	1,183	1,175
Firefox browser	136	25	36
Chrome Adblock Plus	22,723	621	640
Firefox Adblock Plus	22,723	621	640
Chrome Ghostery	54,228	1,184	764
Firefox Ghostery	65,235	1,273	711
Chrome Privacy Badger	37,480	1,266	666
Firefox Privacy Badger	37,499	1,266	666
Chrome uBlock Origin Lite	64,717	1,276	708
Firefox uBlock Origin	64,972	1,276	707

Table 6.5: Results of the evaluation of directly blocked calls to APIs potentially usable for fingerprinting.

6.2.2 Transitively Blocked Calls To APIs Misused For Fingerprinting

Table 6.6 presents the results for transitively blocked fingerprinting API calls. Again, Ghostery for Firefox blocked the most calls in the *Browser Properties* category, with both versions of uBlock Origin following closely. For the *Algorithmic Methods*, Ghostery for Firefox again performed best, slightly ahead of both uBlock Origin versions. In the *Crawl Fp Inspector* category, Brave achieved the highest number of blocked calls.

Tool	Browser Properties	Algorithmic Methods	Crawl Fp Inspector
Avast Secure Browser	51,025	1,990	812
Brave browser	114,959	3,179	1,840
Firefox browser	7	0	0
Chrome Adblock Plus	39,832	1,597	468
Firefox Adblock Plus	39,832	1,597	468
Chrome Ghostery	113,912	3,058	1,476
Firefox Ghostery	139,537	5,010	1,490
Chrome Privacy Badger	88,486	4,811	1,844
Firefox Privacy Badger	88,487	4,811	1,844
Chrome uBlock Origin Lite	136,687	4,985	1,332
Firefox uBlock Origin	136,691	4,985	1,332

Table 6.6: Results of the evaluation of transitively blocked calls to APIs potentially usable for fingerprinting.

6.2.3 Total Blocked Calls To APIs Misused For Fingerprinting

Table 6.7 combines directly and transitively blocked fingerprinting API calls. Ghostery for Firefox blocked the most API calls in the *Browser Properties* category, outperforming its Chrome counterpart, with both uBlock Origin versions following closely. For the *Algorithmic Methods*, Ghostery for Firefox again led, but both versions of uBlock Origin achieved nearly identical results. In the *Crawl Fp Inspector* category, Brave blocked the most calls, followed by both versions of Ghostery, with the Chrome version slightly outperforming the Firefox version.

All evaluated extensions blocked an equal or greater number of *Browser Properties* and *Algorithmic Methods* calls in Firefox compared to Chrome. In contrast, for the *Crawl Fp Inspector* category, all extensions in Chrome blocked an equal or greater number of calls than their Firefox counterparts.

Interestingly, while Ghostery for Firefox blocked *24.5* % of all network requests, it reduced *Browser Properties* API calls by *66.6* %, and blocked *39.2* % of *Algorithmic Methods* calls. Its *29.5* % blocking rate for *Crawl Fp Inspector* was closest to its network request blocking rate. Similar patterns appeared for other tools, indicating that blocked requests were responsible for the majority of *Browser Properties* API calls and an above-average amount of *Algorithmic Methods* and *Crawl Fp Inspector* API calls.

Tool	Browser Properties	Algorithmic Methods	Crawl Fp Inspector
Avast Secure Browser	80,581	2,867	1,447
Brave browser	170,412	4,362	3,015
Firefox browser	143	25	36
Chrome Adblock Plus	62,555	2,218	1,108
Firefox Adblock Plus	62,555	2,218	1,108
Chrome Ghostery	168,140	4,242	2,240
Firefox Ghostery	204,772	6,283	2,201
Chrome Privacy Badger	125,966	6,077	2,510
Firefox Privacy Badger	125,986	6,077	2,510
Chrome uBlock Origin Lite	201,404	6,261	2,040
Firefox uBlock Origin	201,663	6,261	2,039

Table 6.7: Results of the evaluation of total blocked calls to APIs potentially usable for fingerprinting.

Chapter 7

Limitations and Future Work

While request trees provide a way to evaluate the transitive consequences of request blocking, the approach has limitations. Many of the issues are discussed in previous chapters in their respective sections. This chapter highlights additional limitations not previously addressed and discusses possible future work that could improve the evaluation process.

7.1 Page Behavior That Changes When Content-Blocking Tool Is Present

The method of replaying observed requests and checking which ones are blocked by a given content-blocking tool does not correctly capture how an actual page would behave. It assumes that blocking the requests does not impact the page's behavior. However, in a real scenario, the presence of the content-blocking tool during a page visit could influence how the page responds. For example, if a request is blocked, the page might detect the failure and load the resource from another source. This blocking could lead to a different request, possibly with the same fingerprinting API calls as the blocked original request. The current evaluation does not consider this and assumes nothing else follows if a request is blocked.

This dynamic behavior is impossible to capture for a deterministic evaluation, which requires a fixed dataset, as even if a content-blocking tool were present during the logging visit, another tool would behave differently. Since the dataset cannot account for every possible situation, this issue cannot be resolved.

7.2 Limitations of Tree-Based Request Modeling in the Presence of Cyclic Dependencies

Throughout the thesis, the request structure was referred to as a *request tree*. However, in practice, the structure of resources that load on a web page is not always a directed tree. Instead, it can form a regular directed graph that includes loops. For example, a page may load three scripts *A, B* and *C*. However, each script may re-request the other scripts. The situation may look as follows, with requests ordered chronologically:

- The initial page *Z* requests scripts *A* and *B*.
- Script *B* requests script *A*, even though it has already been loaded.
- Script *A* requests script *C*.

- Script A requests script B, creating a cycle.

Cycle such as $B \rightarrow A \rightarrow B$ may execute 1 to n times, depending on how the scripts invoke one another. But even if it executes only once, the resulting requests still constitute a loop within the graph. Figure 7.1 illustrates this situation. The proposed approach cannot correctly capture these cycles.

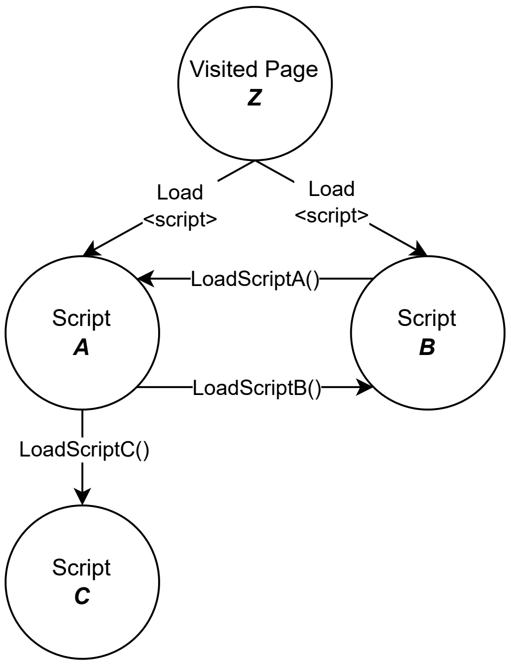

Figure 7.1: Example of a request structure containing a cycle, which makes it a directed graph that is not a tree.

7.2.1 Duplicating Nodes to Ensure Tree Structure

The proposed evaluation method transforms the structure into a tree by introducing duplicate nodes to handle the loop scenario depicted in Figure 7.1. This strategy, referred to as the *upper bound* approach, was used to evaluate and compare the selected tools as presented in Chapter 6. However, it overestimates the number of requests, fingerprinting API calls, and possible blocking results. Each valid event in the logs results in a new node, even if it is a duplicate of an already existing node[1].

For example, in the situation described in the list at the beginning of this section and captured in Figure 7.1, script A appears twice. Once as a direct child of the page and once as a child of script B. Similarly, script B also appears twice, as a direct child of the page and as a child of script A. Suppose the script B calls a function from script A that leads to requesting script C. Script C appears only once in the logs, but it is assigned two parents – both instances of script A, as both were already present in the tree and the current approach does not differentiate between them. Figure 7.2 shows the graph transformed to tree.

[1]As mentioned in Subsection 5.2.3, the exceptions are duplicate follow-up requests loaded by the same parent (i.e., when page Z loads the resource Y n times, only one copy of Y is preserved). These requests are not included in the *upper bound* approach as keeping them would further worsen the issues described in this Section.

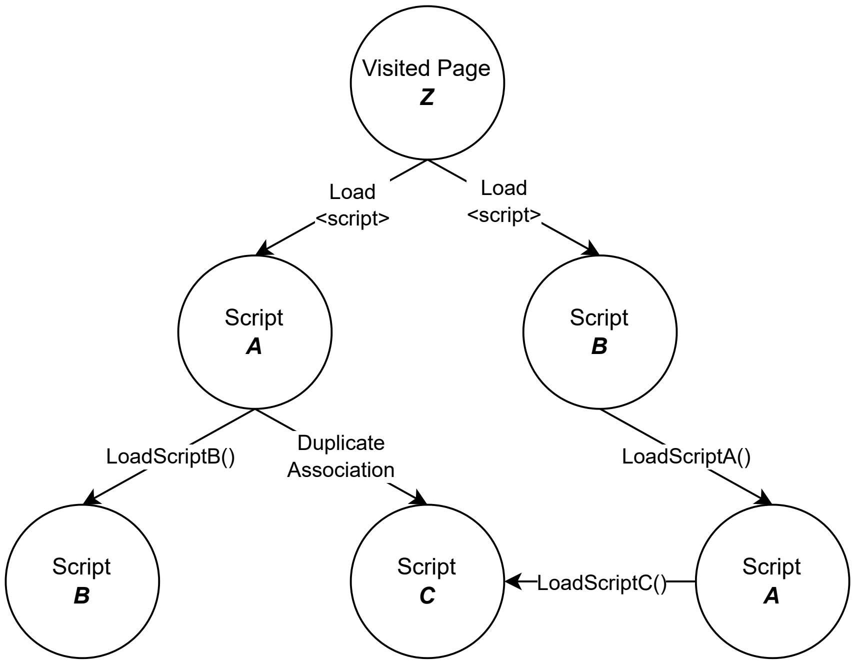

Figure 7.2: Transformed request structure from Figure 7.1 with resolved loop by introducing duplicate nodes, ensuring the structure is a tree. Script C is present only once, as only a single network request for it exists. However, as it has two parent scripts A, during the evaluation, it is counted as if it was present twice due to the tree traversal method.

7.2.2 Node Duplication Overestimates Requests

The duplication causes the overestimation of observed requests and possible blocking results. In Figure 7.2, script C, although requested only once, is effectively counted twice, as it is linked to two different parent nodes. Should a duplicated node be blocked, although it correctly represents two separate requests, the blocking results might get distorted due to the duplicated child requests. Also, if script A in Figure 7.2 is directly blocked, the analysis engine blocks both instances, and their child nodes are blocked transitively. As such, the analysis engine would mark one instance of script B as blocked, even though it would still be present on the page as a child of the root node representing the page itself.

However, the main issue lies within script C. Therefore, the analysis engine handles it more carefully. Since it is linked to both copies of script A, it is only transitively blocked if all of its parent nodes are blocked. All nodes with multiple parent nodes are only transitively blocked if all their parents are blocked. Despite this logic, if script C is blocked, it is still incorrectly counted as two blocked requests, which distorts the results.

7.2.3 Impact on Measurements of Access to Fingerprinting APIs

Another consequence of node duplication is the distortion of total fingerprinting API calls. The fingerprinting API calls are assigned to all corresponding nodes – for example, if script A is responsible for 100 fingerprinting API calls, both instances are assigned 100 API calls, resulting in 200 API calls in total.

The duplication is necessary as it is unclear which instance of script A was responsible for the fingerprinting API calls. For example, when script B requested script A, it may have caused script A to call 100 fingerprinting APIs, while the root node requesting script A did not cause it to use any. Suppose the traffic parser assigned the API calls to only one instance, and that particular node was blocked. In that case, it might be wrongly assumed that all fingerprinting API calls caused by script A were either entirely prevented or not prevented at all, depending on the assignment. The *upper bound* approach assigns them to all copies to avoid underestimating prevented fingerprinting API calls.

7.2.4 Avoiding Duplicate Requests

The request loops are not common. A *lower bound* approach can be used, which allows each resource to appear only once in the tree. If the same script is requested multiple times, the traffic parser includes only the first instance in the request tree. If an existing node appears multiple times in the logs, the traffic parser marks it as *repeated*. Should a *repeated* node be blocked, the evaluation would not transitively block any of its child nodes, as blocking them could lead to overestimating the effectiveness in terms of blocked potential fingerprinting API calls.

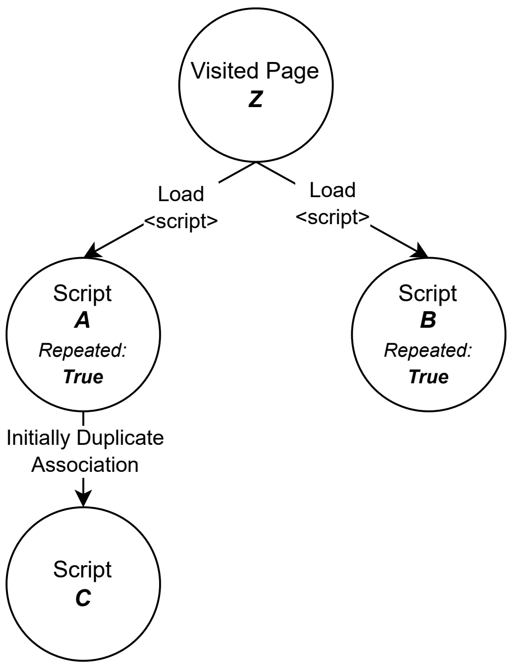

Figure 7.3: Request tree created from the same data as the tree in Figure 7.2, but using the *lower bound* approach. Script B requested by script A is missing, same as script A requested by script B. Both remaining instances of A and B are marked as repeated. Should script A be blocked, C would not be blocked transitively as its original connection to A is missing.

Figure 7.3 illustrates a request tree created from the same data as the request tree shown in Figure 7.2, but using the *lower bound* approach. It also explains why the repeated nodes are not blocked in this approach. Since the events are parsed chronologically and the *lower bound* approach does not allow duplicates, the copy of script B requested by script A is missing, same as the copy of script A that was initially requested by script B.

As a result, the missing instance of script A never gets the opportunity to request script C and make it call any fingerprinting APIs. Script C is still in the tree because it is connected to the other instance of script A, and that instance is marked as repeated. However, since their association was originally only a duplicate, it could not have been responsible for any fingerprinting API calls. However, the calls are still associated with the script C. But since script A is marked as repeated, if it were blocked, none of its children would be transitively blocked, keeping the number of blocked fingerprinting calls caused by script C as zero.

Additional limitation of the *lower bound* approach is, that it discards valid but repeated requests, as seen with scripts A and B in Figure 7.3. Therefore, their associations are lost since each script is only associated with its first parent. For example, if script A loads B but B is already in the tree as a child of the original page, the relation between A and B is not created. If script A is then blocked, its link to B is ignored, even if this instance of B made fingerprinting API calls caused by A. The reason why the request trees do not reflect the missing parentage is to avoid cycles. For instance, if script A loads script B, which then loads A again, it would introduce a cycle and break the logic of the trees.

7.2.5 Summary and Comparison

Table 7.1 compares the results of both the *lower* and the *upper bound* approaches in terms of observed requests and Fingerprinting API calls. Solving the described problems would require using a different internal request model that correctly captures all relations between requests without creating duplicates or endless cycles.

Total Metric	Lower Bound Approach	Upper Bound Approach
Observed Requests	98,244	100,753
Browser Properties Fingerprinting API Calls	293,498	307,226
Algorithmic Methods Fingerprinting API Calls	15,048	16,011
Crawl Fp Inspector Fingerprinting API Calls	6,706	7,460

Table 7.1: Comparison of total observed requests and fingerprinting API calls in the dataset between the *lower* and *upper bound* approaches. The number of observed requests and fingerprinting API calls is the same for all tested tools, as they were tested on the same data, depending on whether the *lower* or *upper bound* approach was used.

7.3 Simulation Limitations

The way the simulation is designed introduces several limitations. First limitation concerns request schemes and the other limitation concerns the functionality of the tested content-blocking tools. The issues are challenging to solve, as replicating the environment of the original pages from which the traffic logger collected the dataset would be required.

7.3.1 Request Schemes

Many web pages use *data*[2] and *blob*[3] URL schemes. These schemes behave uniquely and introduce complications during the evaluation.

[2] https://developer.mozilla.org/en-US/docs/Web/URI/Reference/Schemes/data
[3] https://www.w3.org/TR/FileAPI/#url

In the case of *blob* scheme, the issue is that such resources can only be accessed from the same domain on which they were created[4]. Thus, if a custom server requests a *blob* from another origin, it fails to load, returning an error. However, on the original page where the traffic logger observed the request, it may have either loaded successfully or been blocked by a content-blocking tool. Since the custom server does not replicate the exact environment of the original page, it cannot correctly reflect whether a given content-blocking would block this request, as the request fails before any blocking can occur. In Firefox, an additional issue stems from the fact that the failed *blob* requests do not trigger the `onSendHeaders` event from the *webRequest* API. As such, the custom logging extension described in Section 5.3.5 always considers the *blob* requests blocked, even though it is not necessarily true.

The *data* scheme presents another issue, as these requests do not generate actual network requests to the Internet. Therefore, they do not trigger the `onSendHeaders` event either, and the custom Firefox logging extension always considers them blocked, even though that may not be the case, same as the *blob* requests.

To solve the described issues, the current approach in Firefox skips the *blob* and *data* requests during evaluation, i.e., they are always considered successfully loaded. While this partially solves the problem, it does not capture the situation where a content-blocking tool would have blocked these requests. Firefox logging would need to be improved to handle these situations correctly. The *blob* scheme problem is unsolvable with the current approach since these requests always fail as they cannot be loaded from different domains.

7.3.2 Resource Fetching

The other limitation concerns the method the simulation loads the observed resources. During the simulation, resources are loaded using `fetch()` API. However, this approach may cause some requests to be blocked or allowed incorrectly because the simulation engine made them under conditions different from those on an actual page. For instance, a content-blocking tool may rely on internal heuristics that decide to block a request in the simulated environment, even though it would have been allowed during an actual page load. There is also the possibility that some tools detect they are being tested in a simulated setting and change their behavior.

Another issue stems from the fact that some content-blocking tools apply rules that block requests based on whether the requests originate from a first or third-party context[5]. Since the simulation makes requests from a custom server, these rules are not applicable.

The issues with resource fetching can distort the evaluation results. For example, the tested tool may falsely block a root node. As discussed in Subsection 6.1.1, the analysis engine counts all requests on a page as blocked if a requested root node were blocked, even though it might have loaded correctly in an actual visit.

7.4 Future Work

The issues presented in the thesis could be quantified based on how much they impact the results to improve the evaluation. Once better understood, the limitations could be addressed to ensure results are more closely aligned with the real-world behavior of the evaluated tools.

[4] `https://www.w3.org/TR/FileAPI/#url`
[5] `https://github.com/gorhill/ublock/wiki/Dynamic-filtering:-rule-syntax`

The dataset creation process could be optimized to utilize parallelization, as visiting each site one by one takes considerable time. Data collection and the construction of request models could also be improved to handle the issues described in Section 7.2 properly. Enhancing the JShelter Fingerprint Detector could also solve the logging issues described in Subsection 5.1.3, ensuring complete and accurate fingerprinting data collection across all sites. The simulation could be enhanced to replicate the environment of the original pages, which could solve the limitations of dynamic filters, heuristics and the *blob* scheme, described in Section 7.3.

Beyond dataset creation, other parts of the evaluation could also be optimized for parallel execution, which would be especially beneficial when working with large datasets or multiple tools. Only Firefox, Google Chrome, and Chromium-based custom browsers are currently supported. Support for other browsers could be added. New deterministic metrics could also be introduced to analyze different aspects of content-blocking tools. Additionally, the current metrics could be studied further to understand tool behavior and performance better. Finally, future evaluations could also examine the broader impact of the tools, for example, whether they disrupt website functionality or affect page load times.

Chapter 8

Conclusion

This thesis introduced a deterministic and repeatable method for evaluating privacy-enhancing tools that block network requests. The proposed method preserves the structure of network requests, enabling the assessment of transitive blocking effects. The method was implemented, validated through unit testing, and then applied to selected content-blocking tools.

The evaluation used data from 937 visited web pages, capturing network requests and DNS responses. Potential fingerprinting was also detected using the *JShelter Fingerprint Detector*. Request structures were reconstructed as directed trees, linking each resource to any associated fingerprinting activity. A *BIND 9* DNS server was employed to replay the original DNS responses using custom-generated zone files. The tools were tested via a simulation server replaying recorded requests. Blocked requests were logged, and blocking outcomes were propagated through the request trees to compute metrics.

The blocking performance of three browsers, *Avast Secure Browser*, *Brave*, and *Firefox*, was evaluated along with five content-blocking extensions: *Adblock Plus*, *Ghostery*, *uBlock Origin*, *uBlock Origin Lite* and *Privacy Badger*. These extensions were tested in their Firefox and Chrome versions when available; in cases where only one version existed, only that version was tested. Ghostery for Firefox delivered the best overall blocking effectiveness, achieving the highest scores in most evaluation metrics. uBlock Origin and uBlock Origin Lite also performed well, with results similar to but slightly below those of Ghostery for Firefox. Although the evaluation included Privacy Badger, its results are unreliable due to the limitations of the request simulation process. All tools were tested in their default state, meaning the results might significantly change if a user changes their configuration. Notably, many tested tools blocked over *20 %* of all network requests, underscoring the significance of using content blockers to reduce unnecessary or potentially invasive web traffic.

Several limitations of the proposed approach were identified. The most significant limitation is that the structure of real-world web requests does not always conform to a directed tree. In some cases, the requests form a general graph with loops, affecting the evaluation's accuracy as the proposed approach solves the loops by introducing artificial duplicates. Another limitation is that the simulation does not replicate the original environment, which limits the effectiveness of evaluating tools that rely on heuristics or dynamic blocking rules. Future work could address these limitations, refine the evaluation methodology, expand the set of metrics, and improve the evaluation system's overall performance.

Bibliography

[1] Acar, G.; Eubank, C.; Englehardt, S.; Juarez, M.; Narayanan, A. et al. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2014, p. 674–689. CCS '14. ISBN 9781450329576.

[2] Amazon Web Services, Inc.. *What is Virtualization? - Cloud Computing Virtualization Explained - AWS* online. 2024. Available at: https://aws.amazon.com/what-is/virtualization/. [cit. 2024-10-31].

[3] Barth, A. *HTTP State Management Mechanism* online. RFC Editor, april 2011. Available at: https://doi.org/10.17487/RFC6265. [cit. 2024-10-04].

[4] Brave Software. *Tracker Meaning and Definition* online. February 2023. Available at: https://brave.com/glossary/tracker/. [cit. 2024-10-04].

[5] Bujlow, T.; Carela Español, V.; Solé Pareta, J. and Barlet Ros, P. A Survey on Web Tracking: Mechanisms, Implications, and Defenses. *Proceedings of the IEEE*, 2017, vol. 105, no. 8, p. 1476–1510. ISSN 0018-9219.

[6] Cao, Y.; Li, S. and Wijmans, E. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In: *Network and Distributed System Security Symposium 2017*. 2017. DOI: 10.14722/ndss.2017.23152.

[7] Castell Uroz, I.; Sanz García, R.; Solé Pareta, J. and Barlet Ros, P. Demystifying Content-Blockers: Measuring Their Impact on Performance and Quality of Experience. *IEEE Transactions on Network and Service Management*, 2022, vol. 19, no. 3, p. 3562–3573. ISSN 1932-4537.

[8] Cyphers, B. and Gebhart, G. *Behind the One-Way Mirror: A Deep Dive Into the Technology of Corporate Surveillance* online. Electronic Frontier Foundation, 2019. Available at: https://www.eff.org/wp/behind-the-one-way-mirror. [cit. 2024-10-04].

[9] Dambra, S.; Sanchez Rola, I.; Bilge, L. and Balzarotti, D. When Sally Met Trackers: Web Tracking From the Users' Perspective. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, August 2022, p. 2189–2206. ISBN 978-1-939133-31-1.

[10] Dao, H.; Mazel, J. and Fukuda, K. CNAME Cloaking-Based Tracking on the Web: Characterization, Detection, and Protection. *IEEE Transactions on Network and Service Management*, 2021, vol. 18, no. 3, p. 3873–3888. ISSN 1932-4537.

[11] Datta, A.; Lu, J. and Tschantz, M. C. Evaluating Anti-Fingerprinting Privacy Enhancing Technologies. In: *The World Wide Web Conference*. New York, NY, USA: Association for Computing Machinery, 2019, p. 351–362. WWW '19. ISBN 9781450366748.

[12] Demir, N.; Theis, D.; Urban, T. and Pohlmann, N. Towards Understanding First-Party Cookie Tracking in the Field. In: *GI SICHERHEIT 2022*. Gesellschaft für Informatik, Bonn, 2022, p. 19–34. ISBN 978-3-88579-717-3.

[13] Eckersley, P. How unique is your web browser? In: *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*. Berlin, Heidelberg: Springer-Verlag, 2010, p. 1–18. PETS'10. ISBN 3642145264.

[14] Empey, Charlotte and Latto, Nica. *VPN Meaning: What Is a VPN & What Does It Do?* online. August 2023. Available at: https://www.avast.com/c-what-is-a-vpn. [cit. 2024-10-09].

[15] Englehardt, S. and Narayanan, A. Online Tracking: A 1-million-site Measurement and Analysis. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2016, p. 1388–1401. CCS '16. ISBN 9781450341394.

[16] Fifield, D. and Egelman, S. Fingerprinting Web Users Through Font Metrics. In: *In Proceedings of the 19th international conference on Financial Cryptography and Data Security*. January 2015, vol. 8975, p. 107–124. ISBN 978-3-662-47853-0.

[17] Frola, J. *AudioContext Browser Fingerprinting* online. Brno, CZ, 2022. Bachelor's Thesis. Masaryk University, Faculty of Informatics. Available at: https://is.muni.cz/th/ke5nb/.

[18] Iqbal, U.; Englehardt, S. and Shafiq, Z. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, p. 1143–1161. ISBN 978-1-7281-8934-5.

[19] Laperdrix, P.; Bielova, N.; Baudry, B. and Avoine, G. Browser Fingerprinting: A Survey. *ACM Trans. Web*. New York, NY, USA: Association for Computing Machinery, april 2020, vol. 14, no. 2. ISSN 1559-1131.

[20] Lerner, A.; Simpson, A. K.; Kohno, T. and Roesner, F. Internet jones and the raiders of the lost trackers: an archaeological study of web tracking from 1996 to 2016. In: *Proceedings of the 25th USENIX Conference on Security Symposium*. USA: USENIX Association, 2016, p. 997–1013. SEC'16. ISBN 9781931971324.

[21] Mazel, J.; Garnier, R. and Fukuda, K. A comparison of web privacy protection techniques. *Computer Communications*, 2019, vol. 144, p. 162–174. ISSN 0140-3664.

[22] Mowery, K. and Shacham, H. Pixel Perfect: Fingerprinting Canvas in HTML5. In: IEEE Computer Society. *Proceedings of W2SP 2012*. May 2012.

[23] Munir, S.; Siby, S.; Iqbal, U.; Englehardt, S.; Shafiq, Z. et al. CookieGraph: Understanding and Detecting First-Party Tracking Cookies. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. New

York, NY, USA: Association for Computing Machinery, 2023, p. 3490–3504. CCS ’23. ISBN 9798400700507.

[24] Petráňová, J. *Porovnání webových rozšíření zaměřených na bezpečnost a soukromí* online. 2021. Bachelor’s thesis. Brno University of Technology, Faculty of Information Technology. Available at: http://hdl.handle.net/11012/199325.

[25] Petráňová, J. *Běhová prostředí pro testování činnosti rozšíření pro webový prohlížeč* online. 2024. Master’s thesis. Brno University of Technology, Faculty of Information Technology. Available at: http://hdl.handle.net/11012/248340.

[26] Polčák, L.; Saloň, M.; Maone, G.; Hranický, R. and McMahon, M. JShelter: Give Me My Browser Back. In: *Proceedings of the 20th International Conference on Security and Cryptography*. SciTePress - Science and Technology Publications, 2023, p. 287–294. ISBN 978-989-758-666-8.

[27] Saloň, M. *Detekce metod zjišťujících otisk prohlížeče* online. 2021. Master’s thesis. Brno University of Technology, Faculty of Information Technology. Available at: http://hdl.handle.net/11012/200145.

[28] Sanchez Rola, I.; Dell’Amico, M.; Balzarotti, D.; Vervier, P.-A. and Bilge, L. Journey to the Center of the Cookie Ecosystem: Unraveling Actors’ Roles and Relationships. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, p. 1990–2004. ISBN 978-1-7281-8934-5.

[29] Sergey Mostsevenko. *How the Web Audio API is used for audio fingerprinting* online. March 2021. Available at: https://fingerprint.com/blog/audio-fingerprinting/. [cit. 2024-10-10].

[30] Smatana, A. *Evaluation of Little Lies in Browser Fingerprinting* online. Brno, CZ, 2024. Bachelor’s Thesis. Brno University of Technology, Faculty of Information Technology. Available at: http://hdl.handle.net/11012/247810.

[31] Snyde, A. and Livshits, B. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. *Proc. ACM Meas. Anal. Comput. Syst.* New York, NY, USA: Association for Computing Machinery, june 2020, vol. 4, no. 2. DOI: 10.1145/3392144.

[32] Starov, O. and Nikiforakis, N. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, p. 941–956. ISBN 978-1-5090-5533-3.

[33] Susnjara, S. and Smalley, I. *What are containers? | IBM* online. May 2024. Available at: https://www.ibm.com/topics/containers. [cit. 2024-10-31].

[34] Takei, N.; Saito, T.; Takasu, K. and Yamada, T. Web Browser Fingerprinting Using Only Cascading Style Sheets. In: *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*. 2015, p. 57–63. ISBN 978-1-4673-8315-8.

[35] Traverso, S.; Trevisan, M.; Giannantoni, L.; Mellia, M. and Metwalley, H. Benchmark and comparison of tracker-blockers: Should you trust them? In: *2017*

Network Traffic Measurement and Analysis Conference (TMA). 2017, p. 1–9. ISBN 978-3-901882-95-1.

[36] Zhang, D.; Zhang, J.; Bu, Y.; Chen, B.; Sun, C. et al. A Survey of Browser Fingerprint Research and Application. *Wireless Communications and Mobile Computing*. GBR: John Wiley and Sons Ltd., january 2022, vol. 2022. ISSN 1530-8669.

Appendix A

JSON Example of DNS Traffic

```
{
    "pubmatic.com": {
        "ads": {
            "A": ["23.75.64.240"],
            "CNAME": [
                "pubmatic.edgekey.net",
                "e6603.g.akamaiedge.net"
            ]
        }
    },
    "edgekey.net": {
        "pubmatic": {
            "A": [],
            "CNAME": ["e6603.g.akamaiedge.net"]
        }
    },
    "akamaiedge.net": {
        "e6603.g": {
            "A": ["23.75.64.240"],
            "CNAME": []
        }
    }
}
```

Listing A.1: Example of a logged DNS response for `ads.pubmatic.com` with both *A* and *CNAME* records. For each logged *CNAME* for `ads.pubmatic.com`, another separate record is made, containing only the record for the following *CNAME* in the original list. The record for the last *CNAME* in the original list has the same *A* response as the original request for `ads.pubmatic.com`.