Tracking Versus Security: Investigating the Two Facets of Browser Fingerprinting

Antoine Vastel

Tracking Versus Security: Investigating the Two Facets of Browser Fingerprinting.

Antoine Vastel

Supervisors: Prof. Romain Rouvoy and Prof. Walter Rudametkin

Université de Lille

This dissertation is submitted for the degree of
Doctor of Philosophy in Computer Science

Jury de thèse :

Président : Prof. Gilles Grimaud at *University of Lille*
Rapporteurs : Prof. Daniel Le Métayer at *Inria* and Prof. Christophe Rosenberger at *ENSICAEN*
Examinateurs : Dr. Nataliia Bielova at *Inria* and Dr. Clémentine Maurice at *CNRS*

October 24th, 2019

Acknowledgements

I would like to thank everyone who contributed to the realization of this thesis.

First, I would like to thanks my two supervisors, Romain Rouvoy and Walter Rudametkin. It has been a pleasure to work and exchange ideas with you during these 3 years of Ph.D. Thank you Walter Rudametkin, a second time, for having motivated me to do a Ph.D. when I was still a 4th year engineering student at Polytech Lille. Besides working together, I have also greatly enjoyed our different trips to conferences and to Mexico. Thank you, Lionel Seinturier, for your team management. Thanks to you, I have been able to fully focus on my research.

During these 3 years of Ph.D., it has always been a pleasure to come to the office. That is why I would like to thank all the members of the Spirals team. In particular, Vikas Mishra, Antonin Durey, Guillaume Fieni, and Thomas Durieux, both for their skills to design crawler logos, as well as for the beers at "La Capsule". I would also like to thank Pierre Laperdrix. Thanks a lot for your supervision during my first internship at Inria, as well as for the two papers we wrote together.

I have also been lucky to do an internship at Brave during my Ph.D. Thus, I would like to thank Peter Snyder and Ben Livshits for their supervision during this internship. Thank you also to everyone in the London office, in particular, Blake Loring, Ruba Abu-Salma, Leo Feng and Yezi Li.

Thank you, Marcia Marron, for welcoming us before the deadlines. Your cooking skills and the tequila really helped us to get our papers accepted.

Finally, I want to thank my friends and my family. In particular, my mother for having supported me to do a Ph.D., as well as Amélie, my girlfriend, for her support all along this adventure.

Abstract

Nowadays, a wide range of devices can browse the web, ranging from smartphones, desktop computers, to connected TVs. To increase their browsing experience, users also customize settings in their browser, such as displaying the bookmark bar or their preferred languages. Customization and the diversity of devices are at the root of browser fingerprinting. Indeed, to manage this diversity, websites can access attributes about the device using JavaScript APIs, without asking for user consent. The combination of such attributes is called a browser fingerprint and has been shown to be highly unique, making of fingerprinting a suitable tracking technique. Its stateless nature makes it also suitable for enhancing authentication or detecting bots. In this thesis, I report three contributions to the browser fingerprinting field:

1. I collect 122K fingerprints from 2,346 browsers and study their stability over more than 2 years. I show that, despite frequent changes in the fingerprints, a significant fraction of browsers can be tracked over a long period;

2. I design a test suite to evaluate fingerprinting countermeasures. I apply our test suite to 7 countermeasures, some of them claiming to generate consistent fingerprints, and show that all of them can be identified, which can make their users more identifiable;

3. I explore the use of browser fingerprinting for crawler detection. I measure its use in the wild, as well as the main detection techniques. Since fingerprints are collected on the client-side, I also evaluate its resilience against an adversarial crawler developer that tries to modify its crawler fingerprints to bypass security checks.

Résumé

De nos jours, une grande diversité d'appareils tels que des smartphones, des ordinateurs ou des télévisions connectées peuvent naviguer sur le web. Afin d'adapter leur expérience de navigation, les utilisateurs modifient également diverses options telles que l'affichage de la barre des favoris ou leurs langues préférées. Cette diversité d'appareils et de configurations sont à l'origine du suivi par empreintes de navigateurs. En effet, pour gérer cette diversité, les sites web peuvent accéder à des informations relatives à la configuration de l'appareil grâce aux interfaces du langage JavaScript, sans obtenir l'accord préalable de l'utilisateur. La combinaison de ces informations est appelée empreinte de navigateur, et est bien souvent unique, pouvant donc servir à des fins de suivi *marketing*. Néanmoins, le fait que les empreintes ne soient pas stockées sur la machine rend cette technique également intéressante pour des applications relatives à la sécurité sur le web. À travers cette thèse, je propose 3 contributions relatives aux domaines des empreintes de navigateurs :

1. Je collecte 122,000 empreintes de 2,346 navigateurs et analysons leur stabilité pendant plus de 2 ans. Je montre qu'en dépit de changements fréquents dans leur empreinte, une part significative des navigateurs peut être suivie pendant de longues périodes;

2. Je conçois une suite de tests afin d'évaluer la résistance des outils de protection contre le suivi par empreinte de navigateurs. Je l'applique à 7 outils de protection, et montre que tous peuvent être détectés, ce qui peut rendre leur utilisateurs plus facilement identifiables, et donc vulnérables au suivi;

3. Enfin, j'explore l'utilisation des empreintes de navigateurs pour la détection de *crawlers*. Après avoir mesuré l'usage de cette technique sur le web, je présente les différents attributs et tests permettant la détection. Comme les empreintes de navigateurs sont collectées côté client, j'évalue également la résilience de cette forme de détection contre un adversaire développant des *crawlers* dont les empreintes ont été modifiées.

Table of contents

List of figures

List of tables

Part I

Preface

Chapter 1

Introduction

1.1 Motivations

As users, we all have our own way to browse the web. Some users browse the web using a smartphone, while others prefer to use a laptop, sometimes with an external monitor. Some users decide to have the bookmark bar visible in their browser, while others prefer to increase the default font size because they usually sit far from their monitor. This diversity of devices, browsers, and operating systems, as well as their customization, is at the root of browser fingerprinting. In his thesis, Mayer [2] showed that browsers could be uniquely identified because of their configuration. Indeed, to adapt websites' behavior based on the user device, browsers enable scripts to access information about the user device and its configuration using JavaScript APIs. The combination of attributes collected from these APIs is called a browser fingerprint and can be collected by tracking scripts without obtaining the user consent.

In 2010, Eckersley [3] studied browser fingerprints uniqueness. He created the Panopticlick website and collected more 470K browser fingerprints, among which 83.6% were unique. He also showed that more than 94.2% of the fingerprints when Flash or Java plugins were activated. Because of this uniqueness, he argued browser fingerprints can be used for tracking. In particular, it can be used in addition to cookies, to respawn them when they have been deleted by the user. Indeed, while cookies are stored in the browser and can, therefore, be erased, browser fingerprints are collected in the browser but are then stored on a remote server the user has no control over.

After Eckersley's study, several studies [4–6] have measured the use of fingerprinting in the wild. These studies all showed that fingerprinting was used by a significant fraction of the most popular websites. They also showed that commercial fingerprinters adapt their behavior to leverage new APIs. Indeed, while Eckersley showed that both Flash and Java could be used to obtain the list of fonts, in their 2013 study, Nikiforakis *et al.* [4] showed that none of the commercial fingerprinters they studied were still using Java. They also noticed that since Flash was getting less popular due to its deprecation, one of the fingerprinters was using a new approach to obtain the list of fonts using JavaScript. More recently, Englehardt *et al.* [6] showed that fingerprinters had found new approaches to exploit APIs introduced by HTML5, such as the canvas, WebGL and audio APIs.

To protect against fingerprinting, several countermeasures have been proposed, ranging from simple browser extensions that lie about the device nature to forked browsers that lie about the list of fonts available [7–9]. Niforakis [4] and Acar [10] evaluated the effectiveness of fingerprinting countermeasures, such as simple user agent spoofers, or Fireglove, a browser extension that randomly lies about attributes constituting a fingerprint. Their evaluations showed that countermeasures could be detected because they generated inconsistent fingerprints. Thus, they argued that using these kinds of countermeasures could be counterproductive for a user since she could become identifiable.

Besides tracking, browser fingerprinting can also be used to improve web security. The main use-case studied in the literature is to enhance authentication [11–14] by using the fingerprint as a second factor. Burztein *et al.* [15] showed that browser fingerprinting can also be used to detect crawlers. They proposed a dynamic challenge-response protocol that leverages the unpredictability and yet stable nature of canvas rendering to detect devices that lie about their nature—*e.g.* emulated devices or devices that modify the browser and OS contained in their user agent.

In this thesis, I aim at improving the understanding of browser fingerprinting both concerning its impact on privacy, as well as its applications to improve web security. Concerning its impact on privacy, this thesis aims to answer the following research questions:

1. Are fingerprints stable enough to be used for tracking?
2. How long can browsers be tracked using only their fingerprint?
3. What is the overhead of using browser fingerprinting tracking algorithms at scale?

4. Are fingerprinting countermeasures effective and what are the privacy implications of using these countermeasures?

Regarding the adoption of browser fingerprinting in a security context, this thesis aims to answer the following research questions:

1. How widespread is the use of fingerprinting for crawler detection among popular websites?
2. What fingerprinting techniques are used specifically to detect crawlers?
3. How resilient is browser fingerprinting against an adversary that alters its fingerprint to escape detection?

1.2 Contributions

1.2.1 Tracking Browser Fingerprint Evolutions

While browser fingerprints need to be both unique and stable for tracking, studies tend to focus only on uniqueness at the expense stability. Nevertheless, browser fingerprints can change frequently for several reasons ranging from a browser or a driver update to a change in the browser settings. Thus, I argue that it is essential to accurately measure browser fingerprint stability, in particular, the stability of the different attributes constituting it, and whether or it varies across browsers. Moreover, to better understand how effective browser fingerprinting is as a tracking mechanism, there is a need to measure how long can browsers be tracked using only their fingerprints.

To address the study of the stability and the tracking duration, I analyze more than 122K browser fingerprints from 2,346 distinct browsers collected over a two year period using the AmIUnique browser extensions. My results confirm Eckersley findings that fingerprints change frequently. I show that half of the browser instances display at least one change in their fingerprint in less than five days. Nevertheless, we observe discrepancies across browsers, with some browsers having frequent changes in their fingerprints and others with more stable fingerprints. I also study the stability of fingerprinting techniques that were not available when Eckersley's study was conducted. In particular, I show that—in addition to having a high entropy—canvas fingerprint is one of the most stable attributes in a fingerprint. For half of the browsers, its value remains stable more than 300 days. Then, I study how long browsers can be tracked using only their fingerprints. I

propose two linking algorithms, one based on rules, and another hybrid one, that leverage both rules and machine learning to link fingerprint evolutions over time. I show that while a significant fraction of browsers is immune against fingerprinting, mostly because their fingerprints are not unique or are too close from other fingerprints, around 32% of browsers can be tracked for more than 100 days. Moreover, I show that these linking algorithms can be easily parallelized to run on cheap public cloud instances, making of fingerprinting a threat to privacy.

1.2.2 Studying The Privacy Implications of Browser Fingerprinting Countermeasures

Different defense strategies and countermeasures have been proposed to protect against browser fingerprinting. With new APIs being added frequently to browsers, it is difficult to always have up-to-date countermeasures that protect against new forms of fingerprinting. Moreover, studies [4, 10] revealed the risk of becoming more identifiable when using fingerprinting countermeasures. Thus, I plan to study the privacy implications of using fingerprinting countermeasures, and whether or not they are counterproductive. I propose Fp-Scanner, a test suite that detects inconsistent fingerprints created by fingerprinting countermeasures. I apply Fp-Scanner to 7 different countermeasures, ranging from simple browser extensions to peer-reviewed forked browsers, and I show that even when countermeasures claim to generate consistent fingerprints, their presence can be revealed. Beyond spotting fingerprinting countermeasures, I demonstrate that Fp-Scanner can also recover original values, such as the browser or the operating system. I leverage my findings to discuss different strategies for building more effective fingerprinting countermeasures that do not degrade user privacy.

1.2.3 Evaluating the Resilience of Browser Fingerprinting to Block Adversarial Crawlers

Although some studies showed browser fingerprinting can be used in a security context, for example, to enhance authentication or to detect emulated devices, fingerprinting is often associated with unwanted tracking. I propose to study the use of browser fingerprinting in a security context, as a mechanism to detect bots, in particular, crawlers on the web. I show that fingerprinting for crawler detection is popular among websites of the Top Alexa 10,000. I study the techniques used by commercial fingerprinting scripts. While these

scripts use techniques also used for tracking, such as canvas or font enumeration, they also developed specific techniques that aim at identifying if a fingerprint belongs to known headless browsers or if a browser is instrumented. I also evaluate the effectiveness and resilience of such detection techniques. Indeed, using fingerprinting in a security context is challenging due to the adversarial nature of an attacker. Since browser fingerprints are collected in the browser, it means a skilled attacker can modify its value to bypass security checks. Thus, I show that, while crawler detection using fingerprinting provides better results against simple crawlers with few modifications on their fingerprints compared to other existing approaches, it fails to detect crawlers with more modifications, as well as non-headless crawlers. Therefore, my results show that fingerprinting can quickly detect simple headless crawlers, while its integration in a layered approach, in addition to other existing detection approaches, can strongly increase its resilience.

1.3 List of Scientific Publications

During the course of this thesis, I published papers in the following conferences and workshops:

[16] Vastel, A., Laperdrix, P., Rudametkin, W., & Rouvoy, R. (2018, May). FP-STALKER: Tracking Browser Fingerprint Evolutions. In IEEE S&P 2018-39th IEEE Symposium on Security and Privacy (pp. 1-14). IEEE: https://hal.inria.fr/hal-01652021.[1]

[17] Vastel, A., Laperdrix, P., Rudametkin, W., & Rouvoy, R. (2018). FP-scanner: the privacy implications of browser fingerprint inconsistencies. In 27th USENIX Security Symposium (USENIX Security 18) (pp. 135-150): https://hal.inria.fr/hal-01820197. [2]

[18] Vastel, A., Rudametkin, W., & Rouvoy, R. (2018, April). FP-TESTER: Automated Testing of Browser Fingerprint Resilience. In 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (pp. 103-107). IEEE: https://hal.inria.fr/hal-01717158.[2]

[1] I am the main author of the paper. I wrote the majority of its contents. I proposed the contributions and the evaluation protocol and I wrote the experimental framework.

[2] I am the main author of the paper. I wrote the majority of its contents. I proposed most of the contributions and the evaluation protocol and I wrote the experimental framework.

Vastel, A., Blanc, X., Rudametkin, W., & Rouvoy, R. FP-Crawlers: Evaluating the Resilience of Browser Fingerprinting to Block Adversarial Crawlers (under submission). [1]

[19] Vastel, A., Snyder, P., & Livshits, B. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking (under submission): https://arxiv.org/abs/1810.09160. [3]

1.4 List of Tools and Prototypes

During the course of this thesis, I developed several algorithms, tools, prototypes, and libraries to gather data, test different research hypothesis or simply made or research more accessible. To encourage the reproducibility of my results, I published the entirety of the code:

- Implementations of Fp-Stalker, our two algorithms to link browser fingerprints over time [20],
- Implementation of Fp-Scanner, our test suite to detect inconsistencies introduced by fingerprinting countermeasures [21],
- Code of the crawlers and the labeling interface used in Chapter 5 to explore the use of browser fingerprinting for crawler detection [22],
- An open-source implementation of Picasso canvas as described in Burztein *et al.* [15] paper [23],
- Fp-Collect, a browser fingerprinting library oriented towards bot detection [24],
- Fp-Scanner (bis), a library that leverages Fp-Collect browser fingerprints to detect bots [25].

1.5 Outline

The thesis is organized as follows.

[3] I am the main author of the paper. I wrote a significant part of its contents. I proposed some of the contributions and some of the evaluation protocol. I was the main contributor of the experimental framework.

Chapter 2 starts by introducing the context of this thesis. I present how the diversity of devices and customization are at the root of browser fingerprinting. Then, I define what is browser fingerprinting, what are the main attributes constituting a fingerprint and how they are collected. I review the existing literature on browser fingerprinting. I analyze existing fingerprinting countermeasures along with their main shortcomings. I also explore existing approaches that use fingerprinting in a security context. Finally, I present other non-fingerprinting crawler detection approaches, such as time series analysis and CAPTCHAs, to explain how fingerprinting compare to them.

Chapter 3 presents my study on tracking using fingerprinting. Most large-scale studies focus on fingerprint uniqueness. In Chapter 3, I fill the gap by studying the stability of fingerprints over more than 2 years using data collected from the AmIUnique browser extensions. Moreover, I propose two linking algorithms that aim at linking evolutions of fingerprints of the same browser over time and show that, despite frequent changes, a significant fraction of browsers can be tracked for more than 100 days. This chapter is an extension of the FP-Stalker paper [16] published at S&P 18 and includes 25,000 new fingerprints than in the original paper.

Chapter 4 investigates the privacy impact of fingerprinting countermeasures. Because countermeasures may generate inconsistent fingerprints, they can be detected and harm their user privacy by making them more identifiable. We design a test suite that leverages inconsistencies to detect the presence of fingerprinting countermeasures and show that all of the 7 countermeasures I evaluate can be detected. This chapter was originally published as a conference paper entitled *Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies* [17] published at Usenix Security 18.

Chapter 5 explores the use of fingerprinting in a context of crawler detection. I explore its popularity among websites of the Top Alexa 10K and describe the main detection techniques used by commercial fingerprinters to distinguish humans from bots. Because fingerprints can be modified, I also measure the resilience of this approach against an adversarial crawler developer.

Finally, Chapter 6 concludes this thesis by summarizing my contributions, proposing future work and discussing a possible future for browser fingerprinting.

Part II

State of the Art

Chapter 2

State-of-the-art

2.1 Context

2.1.1 Browsers Evolution

The complexity of browsers has continuously increased over time. Before 1995 and the introduction of the JavaScript language, web pages were only constituted of static content structured using HTML tags to describe the semantics of the content. Thus, it was not possible for pages to perform any dynamic tasks on the client-side, such as reacting to clicks or mouse movements. In 1995, Brendan Eich developed the JavaScript language, while working at Netscape, the company behind the proprietary Netscape browser. The introduction of this new language in the browsers started a new era of a more dynamic web.

An increasing diversity of APIs. Since then, browser vendors have kept on adding new features to attract users. Applications that were once available only as heavy desktop clients are now available as web applications that can run in a browser. For example, advanced text and slides editors, such as Microsoft Word or Open Office Impress were only available as desktop clients. Nowadays, several online services propose similar tools running as web applications, such as Google Docs, Slides.com, and Prezi. From a developer point of view, web applications are supposed to make the development process more convenient, as it removes the burden of managing device compatibility issues. Indeed, web applications should be able to run in all browsers that stick to the web standards. Besides text and slides editors, other complex applications, such as

video games and real-time video chats, can now efficiently run in browsers. For these applications to work in browsers, it required browser vendors to add several APIs, like the canvas and the WebGL APIs, to efficiently generate 2D and 3D shapes or the WebRTC API that enables real-time communication.

An increasing diversity of devices. In addition to the increasing number of APIs and features available in browsers, the diversity of devices capable of browsing the web has also increased drastically. While a few years ago only desktop computers could browse the web, now, a wide range of devices ranging from mobile devices, desktop computers, to connected TV that embeds a browser can browse the web. To help websites to manage this diversity of devices, for example, to better display the content or adapt the website to the performance of the device, browser vendors provide several JavaScript APIs that enable websites to access information about the device, which as we show in this thesis, is at the root of browser fingerprinting.

Evolution of browsers market share and its consequences. While during the two browser wars,[1] there was a race between the different browser vendors to continuously add more features, often at the expense of a proper evaluation of their impact on privacy, nowadays, the situation has stabilized, with fewer browser vendors left. Google, with its Chrome browser, represents more than 62% of the browser market share,[2] followed by Safari with 15% and Firefox, with less than 5% of the market share. Browser vendors and the *World Wide Web Consortium* (W3C) tend to better take into account the privacy aspects before introducing new APIs, in particular, how the API could be used for to fingerprint a browser. For example, in the case of the new `navigator.deviceMemory` attribute introduced in December 2017,[3] the W3C recommended to round the value returned to reduce the fingerprinting risk.[4] Moreover, privacy and security have become strong commercial arguments.[5] Thus, major browser vendors, such as Mozilla and Apple, added more user-friendly mechanisms to manage privacy preferences and countermeasures in their browser, such as the anti browser fingerprinting protection in Firefox,[6] or the *Inteligent Tracking Prevention* (ITP) in Safari.[7] New privacy-friendly browsers, such

[1] https://en.wikipedia.org/wiki/Browser_wars
[2] http://gs.statcounter.com/browser-market-share#monthly-201812-201812-map
[3] https://developer.mozilla.org/en-US/docs/Web/API/Navigator/deviceMemory
[4] https://w3c.github.io/device-memory/#sec-security-considerations
[5] https://www.theverge.com/2019/3/14/18266276/apple-iphone-ad-privacy-facetime-bug
[6] https://blog.mozilla.org/futurereleases/2019/04/09/protections-against-fingerprinting-and-cryptocurrency-mining-available-in-firefox-nightly-and-beta/
[7] https://webkit.org/blog/7675/intelligent-tracking-prevention/

as Brave and Cliqz, have also emerged. Browser vendors are also more willing to take measure for fixing security issues, even though it can impact the user experience by adding significant performance overhead. For example, Google Chrome added site isolation[8] to enhance security, in particular against side-channel attacks, such as Spectre and Meltdown, even though this can lead to a memory increase of 10%. Similarly, browser vendors deprecated browser plugins because of the security issues they engendered, even though some of them—*e.g.* the Adobe Flash plugin—were used on popular websites, like YouTube.[9] Instead, they favored browser extensions that have fewer privileges than plugins and that use a system of permissions similar to the one used for mobile applications.

2.1.2 Monetizing Content on the Web: Advertising and Tracking

Evolution of online advertisting. Advertising is the most popular way to monetize content on the web [26]. Nevertheless, since the first online advertising banners in 1995, to the advanced ad-targeting platforms, advertising has gone through multiple stages. At the beginning of online advertising, websites charged advertisers an upfront cost to occupy some space with a banner on their website. Because of the popularity of these banners, advertisers started to help their customers choose the most adapted audience to display their banners depending on the demography of the users they were trying to target. To help companies to measure how their advertising campaigns were performing in real-time, Doubleclick introduced a service, called DART (*Dynamic Advertising Reporting and Targeting*) that aimed at helping companies to measure the number of times their ads had been viewed and clicked on the different websites their ads were present on. This new feature was game-changing and lead to the creation of a new pricing model. While advertisers used to pay websites to host their banner, no matter the amount of traffic, views, and clicks, after the introduction of DART, the price started to depend on the number of times ads were viewed (cost per impression). Around 2000, search engines became increasingly more important in the web ecosystem, providing users a convenient way to find relevant content on an ever-growing web. Search engines monetized their popularity by enabling advertisers to target users based on the keywords

[8] https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html

[9] https://youtube-eng.googleblog.com/2015/01/youtube-now-defaults-to-html5_27.html

they were searching for. This also created a new shift in the advertising pricing model, with the introduction of pay per click instead of pay per impression. Finally, around 2005, advertisers have started to gather data to make advertising more relevant to users and therefore maximize their incomes. This technique, called behavioral advertising or targeted advertising, consists in gathering data about the users, such as their IP address, the pages they have visited and the products they bought online, to build user profiles of interests that are later used to provide more relevant ads. Since users only see ads they are more interested in, there is more chance they click on it, which, therefore, increases the advertiser revenues.

The tracking industry. To build these user profiles, the advertising industry heavily relies on trackers. Trackers are scripts or images used to gather and transmit data to the tracking company servers. To increase the amount of data collected, trackers are placed on several websites, most of the time not owned by the tracking company, as third-party resources. To incentivize websites to use trackers on their pages, trackers tend to provide a useful service. For example, trackers may take the form social media widgets, such as the Facebook Like button or the Twitter retweet button that aim at increasing the website visibility by making it more easily shareable on social media. Trackers can also take the form of analytic services, *e.g.* Google Analytics, to help websites better understand their audience. To keep track of users over time and across different websites, trackers generate a *unique user identifier* (UUID) that they store in the browser using cookies or other storage APIs, such as local and session storage, as well as indexed database. Trackers also misuse the ETag cache header to store and retrieve user identifiers. The idea behind multiplying the number of storage mechanisms is that, if a user deletes only one of its stored identifiers, the other identifiers can still be regenerated using the other storage mechanisms.

Data protection laws. Because of the invasive nature of trackers, policymakers have proposed laws to protect users data. One of the most recent and important law is the European *General Data Protection Regulation* (GDPR) that requires websites and trackers to obtain user consent before they gather data. Moreover, websites are required to specify the purpose of the data collection, as well as the list of companies they will share the data with. While previous laws used to specifically targets cookies,[10] GDPR is more general. Thus, when they refer to the notion of user identifier, it does not refer only to explicit identifiers stored in cookies, but to any forms of data that could be used as an identifier, for example, a browser fingerprint.

[10] https://www.cookielaw.org/the-cookie-law/

Conclusion. To gather information about users, the online advertising industry heavily relies on trackers that take different forms, ranging from social media widgets to analytics services. To keep track of user identities along time and across different websites, trackers store a unique user identifier in the browser using cookies or other storages mechanisms. Nevertheless, by using a single storage mechanism, trackers run the risk that when a user deletes her cookies, they lose track of her valuable information. Thus, some trackers have come up with a more invasive tracking technique: browser fingerprinting. This technique consists in gathering attributes about the user device and configuration using APIs provided by the browsers. Due to the high diversity of devices and configurations, the combination of these attributes, called a browser fingerprint, is often unique, and can, therefore, be used for tracking. Contrary to cookies that can be erased by the user, fingerprints cannot be deleted since they are not stored on the user device, making it more difficult for users to protect themselves against it.

2.2 Browser Fingerprinting

2.2.1 Definition

A **Browser fingerprint** is a set of attributes that can be used to identify a browser. The analogy with a digital fingerprint arises from the fact that this combination of attributes is often unique [3, 27]. Browser fingerprints are used for tracking purposes, as well as for security purposes, such as bot detection or to enhance authentication. One of the main differences between browser fingerprinting and cookies lies in the stateless nature of browser fingerprints. While cookies used for tracking rely on storing an identifier in the browser, browser fingerprints are totally stateless, which means they are not stored on the user device, making its detection more difficult and its deletion impossible.

In this thesis, the words fingerprint and browser fingerprint, as well as the words fingerprinting and browser fingerprinting are used interchangeably. Moreover, we consider only permissionless browser fingerprinting—*i.e.*, attributes that can be accessed without requesting any permission to the user. Thus, it excludes several attributes, such as the precise geolocation using the `navigator.geolocation` API or advanced forms of `WebRTC` fingerprinting [6] that can obtain the name of multimedia peripherals connected to a device. While this definition of fingerprinting is widely accepted in the literature, the different analyses of fingerprinting scripts conducted during this thesis also show that commercial fingerprinters do not use attributes that require permissions. Nevertheless,

in the case where fingerprinting is used for more legitimate purposes, such as enhancing authentication, we consider these attributes could be part of the fingerprints as users would probably have more incentives to grant their authorization to the fingerprinting script.

Attributes constituting a browser fingerprint can be either collected in the browser using JavaScript or plugins, such as Flash, as well as attributes sent by the browser, such as HTTP headers. Typically, the IP address or the geolocation that can be derived from it are not considered as part of a browser fingerprint [3, 27, 28]. This definition also excludes other forms of fingerprinting techniques, such as TCP fingerprinting [29], a technique that leverages lower-level information from the TCP stack, such as the order of the TCP options. While our definition of browser fingerprinting allows fingerprints collected both on computers and mobiles, the only constraint is that it must bcollected using a browser. Thus, it excludes all forms of fingerprinting conducted using applications, whether or not they require permissions, such as presented by Kurtz *et al.* [30] and Wu *et al.* [31].

Collecting browser fingerprints. Figure 2.1 provides an overview of the process to collect a browser fingerprint. When a user visits a website with her browser, it sends a `GET` request to the server to retrieve a page. Upon receiving the request, the server sends a response containing the content of the page. Fingerprinting scripts are included as JavaScript files in the HTML returned. These scripts may be served as first-party scripts by the domain visited, or by third-party domains to track users across different websites. Once the script was loaded, the fingerprinting script can execute to collect the different attributes. In practice, most of the fingerprinting scripts wait for the *Document Object Model* (DOM) to be also loaded since the script may need to interact with it to collect some fingerprinting attributes, such as the list of fonts. After the JavaScript fingerprinting script completes to execute, it needs to transmit the fingerprint collected to a server. Some fingerprinting script etransmit the whole list of attributes, while others simply compute a hash that is transmitted. Different fingerprints can be used to transmit the fingerprint to a remote server. If only a hash is transmitted or if the fingerprint collected is small, the fingerprint can be sent using an image pixel where the value of the fingerprint is added as a GET parameter of the image URL. When fingerprints are too big to be sent as images, some can trigger a `POST` request using the `XMLHttpRequest` request API[11] or the `navigator.sendBeacon` API.[12] The `sendBeacon` function has the advantage of being asynchronous, which means that data can be transmitted when a

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

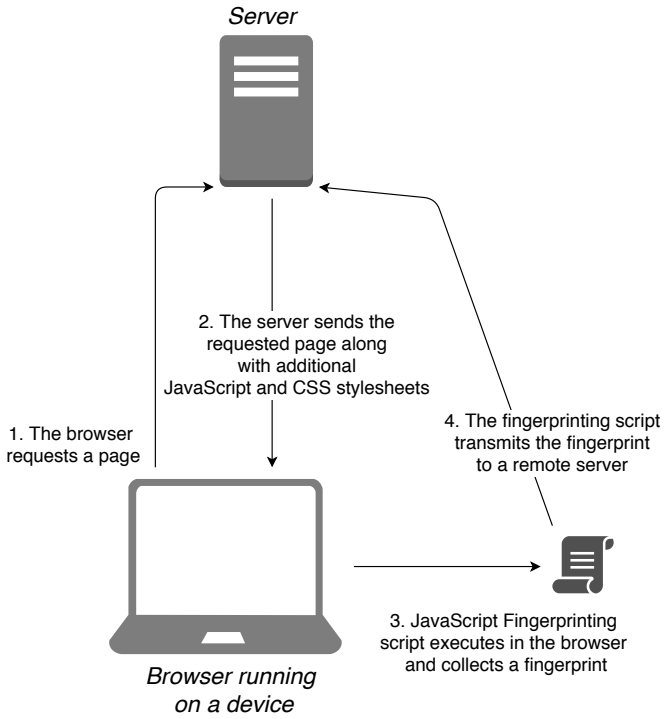


Figure 2.1 Schema representing the process to collect a browser fingerprint. To make the schema more comprehensive, we consider that all resources, including the fingerprinting script, are delivered by the first party.

user closes a tab without blocking it. It is particularly useful when fingerprinters also collect dynamic information, such as clicks and mouse movements in addition to the fingerprinting attributes. Thus, they besides its fingerprint, they can also monitor all her activity on the page. While this feature is also interesting for security purposes, one should be careful since the `beforeunload` event used to signal that a user is closing the page is often badly implemented in headless browsers.[13]

Upon reception of the fingerprint, the server can also collect the HTTP headers associated with the `GET` or the `POST` request used to send the fingerprint, add these attributes to fingerprint and then store the fingerprint in a database.

2.2.2 Building a Browser Fingerprint

In this subsection, we present the different attributes constituting a browser fingerprint. While fingerprinting can be used for security purposes, we focus on attributes used

[13] https://github.com/GoogleChrome/puppeteer/issues/2386

for tracking. We provide more details about fingerprinting attributes used for security at the end of this chapter, as well as in Chapter 5 where we explain how commercial fingerprinters detect crawlers based on their fingerprint. Fingerprint attributes require two properties when used for tracking:

1. **Uniqueness.** While not each attribute need to be unique individually, their combination—*i.e.*, the browser fingerprint—should be unique in order to distinguish between different browsers. Indeed, if different browsers have the same fingerprint, they cannot be tracked using browser fingerprinting.

2. **Stability.** Even in the case where a browser fingerprint is unique, tracking requires a certain stability of the fingerprinting. Indeed, if we consider an extension that randomizes the value of a canvas at each visit, then the browser fingerprint keeps on being unique solely because the canvas is unique. Nevertheless, since the canvas keeps on changing, it becomes challenging for a fingerprinter to keep track of the fingerprint over time.

We distinguish three main families of attributes constituting a fingerprint: HTTP headers, attributes collected using JavaScript and attributes collected using Flash. For each category, we present the different attributes of this category. We explain how these attributes are collected and we also provide examples, as well as information about the attribute such as its uniqueness.

2.2.2.1 HTTP Headers

When a browser sends an HTTP request to obtain a page or to transmit data using the `XMLHttpRequest` API for example, it attaches headers to its request that provide information to the server receiving this request. The role of these headers has been defined in different *Request For Comments* (RFC), in particular in the RFC 7231 [32] where they define the semantics and the contents of header. They also explain how some of the headers leak information about the user or the device, and the risk it can be used for fingerprinting (Section 9.7 of the RFC)[14].

We present four different HTTP headers, as well as a fifth attribute, the order of the headers, that leak information about the device and its user and that can therefore be used for fingerprinting.

[14]Fingerprinting risks related to HTTP headers: https://tools.ietf.org/html/rfc7231#section-9.7

User-Agent. This header provides information about the device and the software, a browser in our case, sending the request. The semantic and the content of this header are defined in the section 5.5.3 of the RFC 7231 [33]. It can be used by servers to gather analytics data or for compatibility purposes when an application is only available on certain kinds of devices. The `User-Agent` header provides several information useful for fingerprinting, such as the browser and its version, as well as the *Operating System* (OS). To protect against fingerprinting, the RFC advises developers not to include fine-grained details about the device. Nevertheless, it does not specify any format for the `User-Agent` header. Thus, as we show in the table presenting examples of user agents, some applications on mobile devices with an embedded browser may indicate sensitive information, such as the name of the carrier.

User-Agent	Description
`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36`	Chrome browser version 72 on MacOS
`Opera/9.30 (Nintendo Wii; U; ; 3642; en)`	Opera browser on a Wii
`Mozilla/5.0 (iPhone; CPU iPhone OS 12_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/16B92 [FBAN/MessengerForiOS;FBAV/ 192.0.0.46.101; FBBV/131204877; FBDV/iPhone8,4; FBMD/iPhone;FBSN/iOS; FBSV/12.1;FBSS/2;FBCR/Play; FBID/phone;FBLC/pl_PL;FBOP/5]`	Browser integrated in the Messenger app on iPhone

Accept-Language. This header is sent by the browser to indicate the languages the user prefers [34]. The user can declare multiple languages, each one associated with a preference value. This preference value, also called quality value, is specified using a `q` parameter. Thus, both the list of languages and their associated quality values chosen by the user can be collected to be part of a fingerprint. Contrary to the majority of the fingerprinting attributes that reflect the nature of the device or the browser, this attribute reflects the user preferences.

Accept-Encoding. This header is sent by the browser to indicate the accepted encodings for the response. Similarly to the `Accept-Language` header, the browser can indicate multiple encodings, each with a quality value. Nevertheless, quality values are not commonly used with this header in the main browsers.

Accept-Language	Comments
`ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7`	Russian in priority, then American english then any form of English.
`en,en-US; q=0.9,de-DE; q=0.8,de; q=0.7,fr; q=0.6,pl; q=0.5,uk; q=0.4,ru; q=0.3,sv; q=0.2,nb;q=0.1`	American English or any form or English in priority. Then German, French, Polish, Ukrainian, Russian, Swedish, Norwegian.
`zh-CN,zh;q=0.9,en;q=0.8`	Chinese then English.

Accept-Encoding	Comments
`br, gzip, deflate`	Encoding header sent by Safari
`gzip , deflate, br`	Encoding header sent Chrome and Firefox

Accept. The `Accept` header specifies the response media types accepted by the browser. Similarly to the `Accept-Language` header, the browser can indicate multiple types, each with a quality value to indicate its preferences.

Accept	Description
`text/html,application/xhtml+xml, application/xml; q=0.9,image/webp, image/apng,*/*;q=0.8`	Accept header when requesting a page on Chrome version 72.
`text/html,application/xhtml+xml, application/xml; q=0.9,image/webp, */*;q=0.8`	Accept header when requesting a page on Firefox version 65

Order of the HTTP headers. Besides the headers values, different studies [11, 3, 35] also showed that the order of the HTTP headers depend on the browser and can be used to identify a browser. While the type of browser is already specified in the `User-Agent` header, this can be used for verification.

2.2.2.2 JavaScript Attributes

Attributes collected using JavaScript are the main source of entropy for browser fingerprints. In order to help developers adapt their websites to their user device—for example, to change the style depending on the size of the screen—browsers expose different APIs that leak information about the device. We present how different JavaScript APIs accessible without any permission, such as the canvas or the audio API, are used by fingerprinters to gather highly unique fingerprinting attributes.

We first introduce several attributes that can be accessed using the `navigator` object,[15] a special object exposed by default in all main browsers, which provides information about the browser and the OS.

navigator.userAgent. The user agent value can also be accessed in JavaScript trough the `navigator.userAgent` property. In normal conditions—*i.e.*, in the absence of any user agent spoofers, this property returns the same value as the user agent contained in the HTTP headers.

navigator.plugins. This attribute returns the list of plugins installed in the browser. For each plugin, it provides information about its name, the associated filename, a description as well as a version of the plugin. Due to the deprecation of the *Netscape Plug-in API* (NPAPI),[16] mostly because of security reasons, the entropy of this entropy has decreased over time.

[15] Navigator object: https://developer.mozilla.org/en-US/docs/Web/API/Navigator
[16] https://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html

Plugins	Description
`Chromium PDF Plugin:: Portable Document Format::internal-pdf-viewer:: __application/x-google-chrome-pdf pdf Portable Document Format`	Plugins on a Chrome browser
`Shockwave Flash:: Shockwave Flash 31.0 r0:: NPSWF32_31_0_0_108.dll:: 31.0.0.108__application/x-shockwave-flash swf Adobe Flash movie,application/futuresplash spl FutureSplash movie`	Browser with the Flash plugin. The `.dll` file extension indicates that they browser is running on Windows.
`Edge PDF Viewer::Portable Document Format::::__application/pdf pdf Edge PDF Viewer`	Plugins on an Edge browser.

navigator.mimeTypes. The `mimeTypes` property returns an array containing the list of MIME types supported by the browser. Each MIME type object provides information about the type supported, a description and the filename:

- **Type:** 'Portable Document Format', **description:** 'application/x-google-chrome-pdf' and **filename:** 'pdf'
- **Type:** 'Widevine Content Decryption Module', **description:** 'application/x-ppapi-widevine-cdm'

navigator.platform. It returns the platform the browser is running on. While this information is redundant with the OS that contained in the `User-Agent` header, it can be used to verify if the OS claimed has been modified.

Platform	Comments
`Linux x86_64`, `Linux armv7l`, `Linux armv8l`, `Linux i686`, `Linux aarch64`	Possible values for browsers running on Linux.
`MacIntel`	Value for browsers running on MacOS.
`iPad`, `iPhone`	Possible values for browsers running on iOS.
`Win64`, `Win32`	Possible values for browsers running on Windows.

navigator.hardwareConcurrency. This property returns an integer representing the number of logical processors available to the browser.

navigator.oscpu. The `oscpu` property returns a string corresponding to the operating system of the device. Similarly to the `platform` attribute, it is also redundant with the OS contained in the `User-Agent` header. Contrary to `navigator.platform` that is available in all the main browsers, this attribute is only available in Firefox.

oscpu	Comments
`Linux x86_64`, `Linux armv7l`, `Linux armv8l`, `Linux i686`, `Linux aarch64`	Possible values for browsers running on Linux.
`Intel Mac OS X 10.12`, `Intel Mac OS X 10.9`, `Intel Mac OS X 10.11`	Value for browsers running on MacOS.
`Windows NT 6.1; Win64; x64`, `Windows NT 10.0; WOW64`, `Windows NT 5.2; WOW64`	Possible values for browsers running on Windows.

navigator.languages. It returns an array containing the user's preferred languages. The array is ordered by preference with the most preferred language first. The value returned is based on the same value as the `Accept-Language` header, the main difference is that it does not include the quality values represented by the letter `"q"` in the header.

Date.getTimezoneOffset. The `getTimezoneOffset` method of the `Date` class returns the difference in minutes between the user timezone and UTC timezone. As pointed out by Gomez *et al.* citegomez2018hiding, the entropy of this attribute mostly depends on the distribution of the location of the users visiting the website that collect the fingerprints.

navigator.enumerateDevices. The `enumerateDevices` function returns the list of input and output media devices, such as microphones or webcams. When no permission is granted, it can simply be used to count the distinct number of speakers, microphones, and webcams. Nevertheless, in the case a media permission has been granted to access a webcam, for example, then `enumerateDevices` can provide more fine-grained information about the peripherals, such as their name or whether or not it is built-in.

navigator.cookieEnabled. This property returns a boolean indicating whether or not cookies are enabled by the browser. Since it has only two possible values, `true` or `false`, this attribute has a low entropy [27].

navigator.doNotTrack. The `doNotTrack` property aims at indicating whether or not a user accepts to be tracked. Depending on the browser, it returns `"0"` if the users refuses to be tracked, `"1"` if she accepts to be tracked. Some browsers do not specify its value and decide to return `null` instead. Nevertheless, starting from version 12, Apple decided to remove the `doNotTrack` property from the `navigator` object because they consider it misleading.[17] Indeed, users tend to believe it protects them from tracking even though there are no proofs that advertisers and trackers in general respect its value.

navigator.getBattery. The function `getBattery` returns an object containing information about the device's battery that can be used for tracking [36]. The returned object contains the following information:

- **charging:** a property that represents whether or not the battery is charging,
- **chargingTime:** a property that represents the time before the battery is fully charged,
- **level:** a property that represents the charging level of the battery.

navigator.deviceMemory. The `deviceMemory` property returns the amount of memory of the device in gigabytes. It is only available on Chromium-based browsers, such as Chrome and Opera, since December 2017 (Chrome version 63).[18]

[17] https://developer.apple.com/safari/technology-preview/release-notes/
[18] https://developer.mozilla.org/en-US/docs/Web/API/Navigator/deviceMemory

Navigator prototype. Acar *et al.* [10] showed that the order of the properties of the `navigator` object, as well as the presence or absence of certain properties, can be used to fingerprint a browser and its version. For example, on Chrome 68 the navigator prototype has 58 properties, while the Samsung browser version 7 has only 56 properties, and Safari mobile 12 has between 33 and 39 properties. More generally, besides the special case of the `navigator` object, Mulazzani *et al.* [37] and Nikiforakis *et al.* [4] showed that the presence or absence of features could be used to accurately identify the version of a browser. While this feature does not bring any information not already contained in the `User-Agent` header, it can be used to verify if the browser claimed has been modified by a spoofer.

Canvas fingerprinting. Mowery *et al.* [38] showed that the HTML canvas API could be used to generate images whose rendering depends on the browser and the device. These canvas use different techniques that, when combined, generate an image whose rendering is highly unique. For example, Acar *et al.* [5] showed that commercial fingerprinters used strings that are pangrams—*i.e.*, strings constituted of all the letters of the alphabet—or use emojis since their rendering depends on the OS and the kinds of device. Figure 2.2 presents the canvas generated by Akamai and PerimeterX fingerprinting scripts.

(a) Canvas fingerprint generated by Akamai Bot Manager fingerprinting script.

(b) Canvas fingerprint generated by PerimeterX fingerprinting script.

Figure 2.2 Example of two canvas fingerprints used by commercial fingerprinting scripts.

Window and screen size. The browser exposes different properties, through the `screen` and the `window` objects, that reflect the size of the screen and the window.

Table 2.1 presents and defines these different attributes. Figure 2.3 presents a screenshot of a browser on MacOS that shows how these attributes relate to each other.

Table 2.1 Definition of different attributes that provide information about the size of the screen and the window. For each attribute we present a possible value for of the attribute. All the possible values shown in the table come from the same user.

Attribute	Possible value	Description
`screen.width`	1280	Width of the web-exposed screen area in pixels. In there case where there are multiple screen, it should return the value of the screen where the browser window is located. The value is not influenced by the size of the browser window.
`screen.height`	1024	Height of the web-exposed screen area in pixels. Similar definition as `screen.width`.
`screen.availWidth`	1280	Amount of horizontal space in pixels available to the browser window.
`screen.availHeight`	1024	Amount of vertical space in pixels available to the browser window.
`window.innerWidth`	1050	Width of the browser window in pixels, including the size of the scroll bar.
`window.innerHeight`	1050	height of the viewport, i.e. the part of the webpage a user can see, in pixels, including the size of the scroll bar.
`window.innerHeight`	932	height of the viewport in pixels, including the size of the scroll bar.
`window.outerWidth`	1050	Width in pixels of the whole browser window.
`window.outerHeight`	1004	Height in pixels of the whole browser window.
`screen.colorDepth`	24	Color depth of the screen.

Audio fingerprinting. Similarly to canvas fingerprinting that uses the HTML canvas API to generate highly unique images, audio fingerprinting leverages the Web Audio

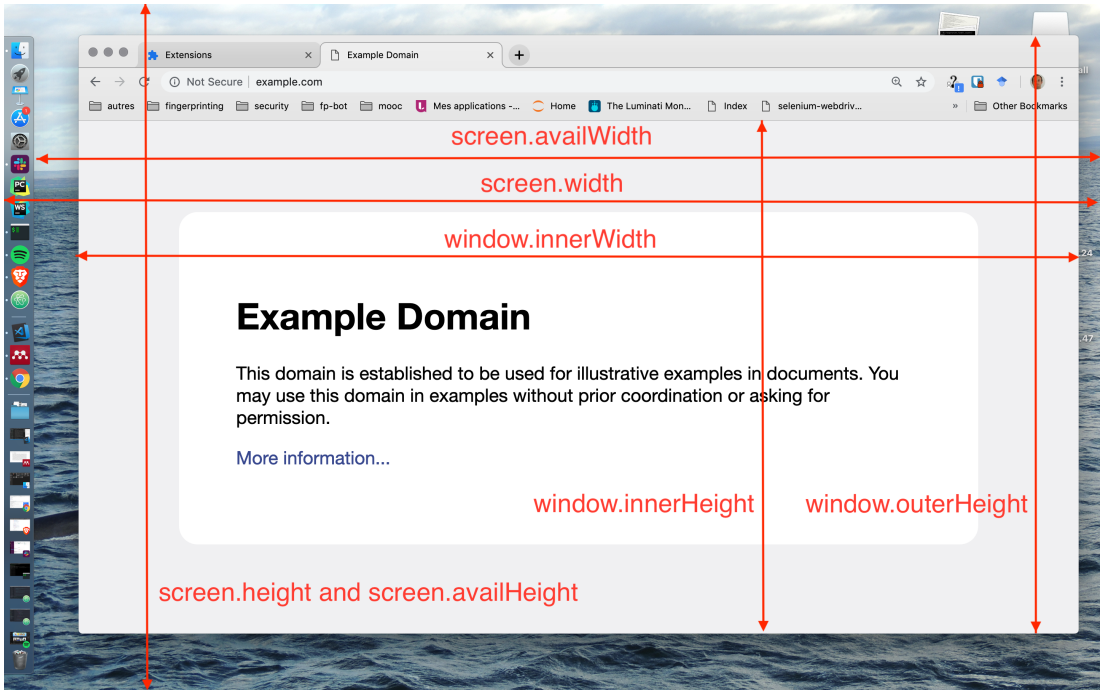


Figure 2.3 Presentation of the different attributes related to the size of the screen and the window.

API to generate sound signals with high entropy. Englehardt *et al.* [6] showed that one popular fingerprinting script relied on a `OscillatorNode` object to generate and process an audio signal. Due to hardware and software differences, the resulting signal is slightly different depending on the device.

WebGL.vendor/renderer. The WebGL API enables to draw 3D shapes in the browser. Although it works in the majority of the browsers and devices—even devices without a GPU thanks to technologies, such as SwiftSchader[19] that enables to have a compatible API on a CPU—the WebGL API keeps exposing information about the user device to help developers to tailor their code to the user device. In particular, two attributes exposed by the WebGL API can be used for fingerprinting. The first attribute is the WebGL vendor and returns the name of the GPU vendor:

- `Apple Inc.`
- `Intel Open Source Technology Center2`
- `Qualcomm`
- `ATI Technologies Inc`

The second attribute, WebGL renderer, returns the name of the GPU:

- `Adreno (TM) 405`
- `AMD PITCAIRN (DRM 2.50.0 / 4.15.0-43-generic, LLVM 6.0.0)`
- `ANGLE (AMD Radeon HD 7310 Graphics Direct3D9Ex vs_3_0 ps_3_0)`
- `NVIDIA Quadro K4000 OpenGL Engine`

WebGL canvas. Besides static attributes, the WebGL API can also be used to generate a 3D canvas fingerprint. Laperdrix *et al.* [27] used the WebGL API to generate 3D shapes. Nevertheless, they did not succeed in crafting a stable and unique WebGL canvas. More recently, Cao *et al.* [1] contradicted Laperdrix *et al.* findings and showed the WebGL API could be used to generate canvas that are both unique and stable, even across different browsers of the same machine. They carefully selected different parameters, such as the texture, the anti-aliasing or the light intensity to render more than 20 different tasks. To create unique 3D scenes, the tasks exploit different mechanisms, such as the fact that interpolation algorithms used by fragment shaders vary depending on the graphic card.

[19] https://developers.google.com/web/updates/2012/02/SwiftShader-brings-software-3D-rendering-to-Chrome

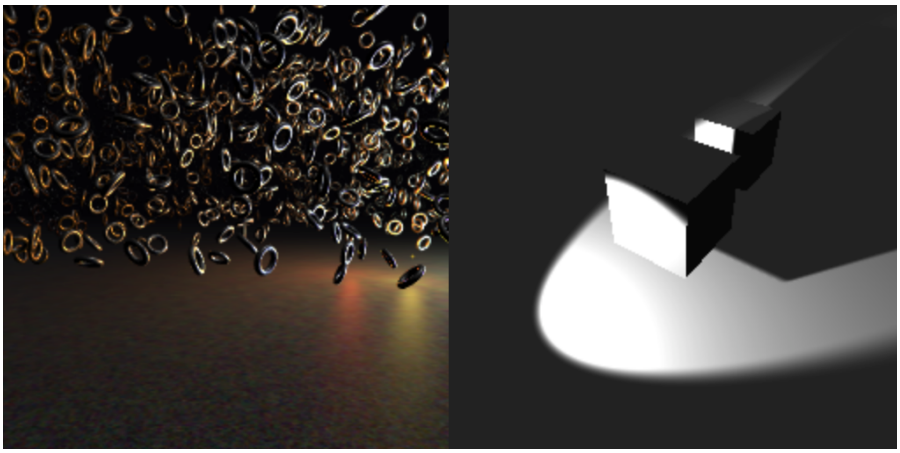


Figure 2.4 Examples of two 3D scenes generated with WebGL using Cao *et al.* 's [1] approach.

The tasks generate fingerprints that are also resilient when the screen or the window size changes, or when the zoom level is altered. Figure 2.4 presents two examples of 3D scenes they generate. They also showed that even when WebGL was not using GPU, *e.g.* when the device has no GPU or a blacklisted GPU, and uses the SwiftSchader library to run the computation on the CPU, the 3D scenes still have entropy.

Touch screen. The presence of a touch screen, as well as its characteristics, can be used for fingerprinting. In order to test the presence of touch support on the device, one can create a `TouchEvent` and observe if it succeeds or look at the presence of the `ontouchstart` property in the `window` object. In case the device has touch support, one can use the `navigator.maxTouchPoints` or the `navigator.msMaxTouchPoints` properties to obtain the number of simultaneous touch contact points supported by the device.

Audio and video codecs. Audio and video codecs support depends on the browser and the OS.[20] During the analyses conducted in this thesis, we observed some of the commercial fingerprinting scripts testing the presence of audio and video codecs using the function `HTMLMediaElement.canPlayType`. Given an audio or a video type, this function return three possible values:

1. `"probably"`, which means that the media type appears to be playable,
2. `"maybe"` indicates that it is not possible to tell if the type can be played without playing it,

[20] https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats

3. `""`, an empty string indicating that the type cannot be played.

Font enumeration. At the end of this section, we present how the whole list of fonts installed on the system can be obtained using Flash. Nevertheless, with the decrease of the popularity of the Flash plugin caused by its deprecation,[21] fingerprinters have come up with new approaches to obtain the list of fonts installed on the system [4]. The idea to test if a font is installed is to compare the size of two HTML elements, one that uses the system fallback font and the other element that uses the font whom the fingerprinter wants to test the presence. It can be done the following way:

1. The script creates a `div` element containing a `span` element,
2. The script sets a predefined text with a fixed size. Moreover, it sets a font-family that does not exist. Thus, the browser will use the fallback font of the system,
3. The script measure and save the size of the span element using its `offsetWidth` and `offsetHeight` properties,
4. For each font whom the script wants to test the presence on the user system, it creates a `span` element inside a `div`. Then, it sets the text of the `span` element using the same string and size as in step 2 and it specifies that the text should be rendered using the font that it wants to test. Finally, the script measures the size of the span element,
5. If the `span` element has the same dimensions as the `span` element that use the fallback font, then it means the font is not present on the device. Otherwise, it means the font is installed.

To be sure to decrease the chance of false negatives—*i.e.*, fonts that would not be detected—the font-size should be large enough, so that even small differences in the font rendering are amplified and can be detected by `offsetWidth` and `offsetHeight` properties. Gomez *et al.* [28] collected fonts on more than 2M users using this approach and showed that the list of fonts provided more than 6.9 bits of entropy.

Fifield *et al.* [39] showed that simply measuring how different Unicode glyphs are rendered can provide a stable and unique identifier. Indeed, the rendering of the font depends on different factors, such as the fonts or anti-aliasing. They measured the size of the glyph bounding boxes for different Unicode characters and found that across the 1,016 different devices in their experiment, 349 could be identified solely using the font metrics.

[21] https://www.bleepingcomputer.com/news/security/google-chrome-flash-usage-declines-from-80-percent-in-2014-to-under-8-percent-today/

Performance fingerprinting. Mowery *et al.* [40] used the SunSpider and the V8 benchmarks to build a fingerprint. In total, they run 39 performance tests, each five times and measured the time each test takes to execute. Using these timing information, they create different heuristics to predict the OS, the browser, as well as the CPU architecture. While the test sample is relatively small, less than 1,000 different configurations, they are still able to achieve a browser classification accuracy of more than 80%. In the case of CPU architecture, they achieve an accuracy of 45.3%, which is still interesting considering that a random choice would have resulted in an accuracy of 6.7%. While the CPU architecture can be used as an additional attribute in a fingerprint, being able to properly classify the OS and the browser enables to verify if the values displayed in the user agent have been spoofed. More recently, Sanchez *et al.* [41] proposed an approach that measures the time to execute sequences of cryptographic functions to generate fingerprints capable of distinguishing similar devices.

Extension probing. Similarly to the list of plugins, the list of extensions can be used as a fingerprinting attribute. Nevertheless, the main difference between plugins and extensions is that there is no API to retrieve the list of extensions installed by a user. Thus, the different techniques we present to obtain the list of extensions either rely on bugs or side effects caused by the usage of these extensions, Mowery *et al.* [40] showed that it was possible to infer the list of websites whitelisted by the NoScript extension by observing whether or not scripts from a certain domain could be executed or they were blocked. Since these whitelists are often unique, they argued it could be used as an additional fingerprinting technique. Starov *et al.* [42] showed that browser extensions could be identified because of the way they interact with the DOM. Among the 10,000 most popular extensions of the Chrome store, around 15% had a unique way to interact with the DOM, making their presence detectable. They also showed that among 854 users, 14.1% had a unique set of browser extensions. Sjosten *et al.* [43] proposed an approach that leverages Web Accessible Resources (WAR) to test the presence of browser extensions. Their approach is able to detect more than 50% of the top 1,000 Chrome extensions. Even though Firefox protected against this kind of attacks by randomizing each extension identifier,[22] Sjosten *et al.* [44] showed it was still possible to test the presence of extensions using a revelation attack. Their strategy is to convince the extension to inject content in the DOM using a WAR URL, making, therefore, the extension reveal its unique randomized identifier that can be used for tracking. Thus,

[22]Protecting against extension probing: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources

with this approach, they can reveal the presence of an extension and also obtain a unique and stable identifier.

2.2.2.3 List of Fonts Using Flash

Flash usage went from 80% in 2014 to 8% in 2018.[23] Flash, and plugins, in general, have been deprecated by browser vendors[24] mainly because of the security risk they represent and are now being replaced by browser extensions that have fewer rights as plugins used to have. In particular, plugins are able to access more information than JavaScript. We only present the Flash attribute with the most entropy, the list of fonts. As we show in the oldest contribution of this thesis presented in Chapter 3, FP-Stalker, where we use fingerprinting to track browser over time, the only Flash attribute still worth considering was the list of fonts. Nevertheless, we show that even on fingerprints collected around 2017, it does not bring significant information since most of the users had already Flash disabled. The `Font.enumerateFonts` enables to collect the complete list of fonts using Flash. Contrary to JavaScript font enumeration that needs to test the presence of each font, this method is straightforward and provides a simple mechanism to obtain the list of all the fonts installed on the system, even the most uncommon fonts. Thus, when Flash's use was still high, it was one of the attributes with the highest entropy [3, 27]. Eckersley [3] also showed that the order of the fonts depended on the system.

Even though the Flash plugin can be used to obtain other attributes, such as the platform, the preferred languages or the screen resolution, we decide not to present these attributes because of the decline of Flash usage, and the fact that these attributes do not provide significantly more entropy than their JavaScript counterpart. Moreover, other plugins, such as Java or Silverlight were also used by fingerprinters to obtain more fine-grained information as the one provided in JavaScript. Nevertheless, in a 2013 study conducted by Nikiforakis *et al.* [4], they showed that none of the three popular commercial fingerprinters they studied were still using Java.

2.2.3 Studying Browser Fingerprints Diversity

Mayer [2] brought to light the privacy problems that arise from browser diversity and customization. Since there are different OS, browsers, screen resolutions or plugins,

[23] https://www.bleepingcomputer.com/news/security/google-chrome-flash-usage-declines-from-80-percent-in-2014-to-under-8-percent-today/

[24] https://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html

this diversity could be exploited to uniquely identify browsers. At the time the thesis was written in 2009, the situation was even worse due to the widespread use of Java applets and Flash Action scripts that had access to even more attributes than JavaScript programs. Over two weeks, Mayer collected fingerprints from 1,328 different browsers, among which 1,278 (96.23%) were unique.

Mayer's work motivated the first large-scale study on browser fingerprinting uniqueness conducted by Eckersley [3], with the collaboration of the *Electronic Frontier Fondation* (EFF).[25] They created a website, Panopticlick,[26] on which they collected 470,161 fingerprints between 27th January and 15th January 2010. Their results confirm Mayer's initial findings: 83.6% of the browsers had a unique fingerprint. Uniqueness was even higher, 94.2%, for browsers with either Flash or Java activated. Indeed, among Flash and Java users, only 1% of the browsers had an anonymity set larger than two. They showed that that the list of plugins and the list of fonts were the two attributes with the most entropy. With this proportion of unique browser fingerprints, they argue that this technique can be used for tracking, in particular as a mechanism to regenerate supercookies or deleted cookies. To support this claim, they proposed a simple heuristic that aims at linking multiple fingerprints of the same browser. First, they studied the stability of browser fingerprints over time and showed that among the 8,833 users that had accepted a cookie and that had visited the websites multiple times, more than 37% displayed at least one change (besides activating or deactivating JavaScript) in their fingerprint. Nevertheless, they are aware this number may be overestimated because of the nature of their website that tends to make people change their fingerprint on purpose, *e.g.* by changing the list languages they prefer or by deactivating a plugin. Nevertheless, they showed that despite these frequent changes in the fingerprint, browser fingerprinting could still be used for tracking. Their heuristic was able to make correct predictions 65% of the time, incorrect predictions 0.56% of the times. Otherwise, 35% of the time, it made no prediction.

Laperdrix *et al.* [27] also created a website, AmIUnique, to study the diversity of fingerprints. Between 2014 and 2015, they collected more than 118,000 fingerprints fingerprinting. In addition to the attributes collected in the study conducted by Eckerlsey [3], they also collect new attributes, such as canvas [38] and WebGL fingerprinting. They use the normalized Shannon's entropy to compare their dataset with Panopticlick dataset. They found similar results, except for the list of plugins and the list of fonts where they obtained a lower entropy. This difference can be explained by the decrease of Flash usage,

[25] https://www.eff.org/
[26] https://panopticlick.eff.org/

which means that the list of fonts was not collected for all the fingerprints, therefore decreasing its entropy. The difference can also be explained by the rise of mobile usage, on which there are no plugins, which also includes Flash. Besides attributes also collected on Panopticlick, they analyzed the entropy of seven new attributes, such as canvas and WebGL fingerprinting or the presence of an ad-blocker. They found that canvas was among the five most discriminating attributes, with a normalized entropy close to the entropy of the list of plugins. Among the 118,934 fingerprints they collected, there obtained 8,375 distinct canvas values, among which 5,533 were unique. They also studied the differences between computer, either desktop or laptops, and mobile fingerprints. While 90% of desktop fingerprints were unique, only 81% of mobile fingerprints were unique. This difference was mostly explained by the low entropy of the list of fonts and the list of plugins on mobile. Nevertheless, mobile fingerprints still achieve a high uniqueness because of attributes such as the user agent or the canvas that are more unique on mobile. Indeed, in the case of the user agent, they noticed that some phone manufacturers were adding sensitive information to this header, such as the precise version of the model or the version of the Android firmware. In the case of the canvas, they noticed that the emoji included in it was also a great source of entropy since its rendering depends on the phone OS version as well as the phone manufacturer.

More recently, between 2016 and 2017, Gomez *et al.* [28] collected more than 2 million fingerprints on a popular french website from the Top 15 Alexa. Since it is a popular website visited by a wide range of users, it avoids the bias of data collected by Eckersley and Laperdrix studies. Indeed, as acknowledged by Eckersley, Panopticlick was mostly visited by users aware of privacy issues on the web. Thus, these users may have a more customized browser and device configurations than random users. Gomez *et al.* compared the diversity of browser fingerprints in their dataset with the ones from Eckersley and Laperdrix studies. They collected the same set of attributes, at the exception of the canvas that was modified to obtain a higher uniqueness. They also collected the list of fonts using JavaScript instead of Flash since Flash usage had already hugely decreased at the time their study was conducted. They found significantly different results compare to the two previous studies. While previous studies claimed that more than 80% of the fingerprints were unique, only 33.6% are unique in their dataset. The difference is even more important for mobile devices. While 81% of the mobile fingerprints collected on AmIUnique were unique, only 18.5% of the fingerprints in their dataset are unique. Despite this uniqueness difference, the attributes with the most entropy are still the same:

1. The list of plugins (9.49 bits of entropy),
2. The canvas fingerprint (8.55 bits of entropy),
3. The user agent (7.15 bits of entropy),
4. The list of fonts (6.90 bits of entropy).

Similarly to the AmIUnique study, they also observed a decrease of uniqueness on mobile for the list of plugins (10.3 bits on computer against 0.2 bits on mobile) and the lists of fonts (7.0 bits on computer against 2.2 bits on mobile), even when fonts are obtained using JavaScript. Their study also confirmed that mobile user agents provide more information compare to computer user agents (6.3 bits on computer against 8.7 bits on mobile).

While their study shows that browser fingerprint uniqueness has probably been overestimated by previous studies, either because of small or biased datasets, it is unclear in which proportions. It is difficult to measure the variation caused by their use of a more representative dataset and the variation caused by the fact that attributes that used to have a high entropy, such as the list of plugins, become less unique over time due to the deprecation of plugins[27] that started being replaced by browser extensions. Indeed, to compare fingerprint uniqueness and attributes entropy with previous studies, they have restricted themselves to 17 attributes already collected by Eckersley [3] and Laperdrix *et al.* [27]. While they improved the canvas and modified the way fonts are collected, they did not take into account several attributes available at the time their study was conducted. Thus, the main critic of this study is that it while it properly evaluates the entropy of the attributes studied, it underestimates the fingerprint uniqueness by excluding attributes that were available at the time their study was conducted between 2016 and 2017. In particular, they did not consider the following attributes:

1. **navigator.enumerateDevices.** This function has been available since Chrome version 45[28] (September 2015)[29] and provides information about the number of microphones, speakers and webcams;

2. **Audio fingerprinting.** This technique relies on the HTML Audio API available since Chrome version 35 and Firefox version 25. In a crawl conducted in January

[27] https://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html
[28] https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/enumerateDevices
[29] Google Chrome version history: https://en.wikipedia.org/wiki/Google_Chrome_version_history

2016, Englehardt *et al.* [6] already mentioned the use of audio fingerprinting by popular websites and estimated it had an entropy of 5.4 bits;

3. **Screen and window properties.** While they collect information about the screen width and height, as well as the color depth, they did not collect more advanced information `window.innerHeight/Width` or `window.outerHeight/Width` that enables to infer the presence of a desktop toolbar and its size, or whether or not a bookmark bar is displayed in the browser;

4. **Audio and video codecs.** The presence of audio and video codecs can be tested using the `HTMLMediaElement.canPlayType` function that was already available at the time their study was conducted [30];

5. **Touch screen details.** In the case of mobile devices, they did not collect any information about the maximum simultaneous touch points supported by the screen using the `navigator.maxTouchPoints` property available since Chrome 35;[31]

6. **Number of cores.** The `navigator.hardwareConcurrency` returns the number of logical processors available to the browser and has been available since Chrome 37 and Firefox 48.[32]

Thus, it is unclear how different the fingerprint uniqueness would have been, had they considered these attributes. In particular, when we consider their second research question that studied the proportion of almost unique fingerprints—*i.e.*, fingerprints that would become unique if a slight modification was applied to attributes whom the user can naturally modify through the browser user interface, such as the value of do not track or the list of preferred languages—they showed that, for computer fingerprints, applying small changes on random fingerprints would lead to a uniqueness rate of 80%. Therefore, we should take care of the real fingerprint uniqueness. Moreover, adding to fingerprints new attributes presented in Section 2.2, such as `navigator.deviceMemory` or extensions probing, would probably also rise fingerprint uniqueness. We consider evaluating the entropy of these attributes as part of future work.

[30] https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/canPlayType
[31] https://developer.mozilla.org/en-US/docs/Web/API/Navigator/maxTouchPoints
[32] https://developer.mozilla.org/en-US/docs/Web/API/NavigatorConcurrentHardware/hardwareConcurrency

2.2.4 Use of Browser Fingerprinting on the Web

We present multiple large-scale studies that analyzed the use of browser fingerprinting on the web. We present these studies in a chronological order to better convey the evolution of fingerprinting use and techniques over time.

The first large-scale studies on browser fingerprinting started in 2013, three years after Mayer [2] and Eckersley [3] brought to light the privacy risk arising from browser customization. Nikiforakis *et al.* [4] analyzed the code of three popular fingerprinters. They noticed that commercial fingerprinters used more aggressive techniques than those presented by Eckerlsey [3]. For example, commercial fingerprinters heavily relied on Flash and ActiveX plugins to obtain information not available in JavaScript, such as whether or not the browser is behind a proxy. They noticed that even for simple attributes, such as the platform that can be accessed using `navigator.platform` or the user agent, the Flash platform attribute provides more detailed information, such as the exact version of the Linux kernel, which can be used both for tracking, as well as to exploit vulnerabilities. They detected that fingerprinters adapted their behavior based on the nature of the browser and the plugins available. For example, when the script detected Internet Explorer, it tried to exploit specific APIs available only on Internet Explorer, such as `navigator.systemLanguage`. When specific plugins were detected, two of the fingerprinters even tried to invoke them to obtain sensitive information, such as the hard disk identifier, the computer's name, the installation date of Windows as well as the list of installed system drivers. They also detected a shift in the way fonts were obtained because of the decline of Flash. Thus, while two of the fingerprinters used Flash to obtain the list of available fonts, one of the fingerprinters was using JavaScript [39].

They also crawled the Top Alexa 10K to study the adoption of these three fingerprinting scripts among websites of the Top Alexa 10K. They detected 40 sites (0.4%) of sites using scripts provided by one of the three commercial fingerprinters. They also used Wepawet,[33] an online platform for the detection of web-based threats, to detect if these scripts were used by less popular websites and found out that 3,804 domains analyzed by Wepawet used one of these scripts.

Also in 2013, Acar *et al.* [10] proposed FPDetective, a crawling framework to detect and analyze fingerprinting on the web. They applied FPDetective to the Top Alexa 1M websites and were able to detect 16 new fingerprinting scripts, as well as new fingerprinting techniques that had not been documented by previous studies. Instead of relying on lists

[33] Wepawet: https://wepawet.cs.ucsb.edu

of URLs to detect fingerprinting scripts, their crawler logs access to properties commonly used for fingerprinting, such as the properties of the `navigator` and `screen` objects, as well as properties used for JavaScript font enumeration such as `offsetWidth/Height`. The crawler also intercepts calls to the `getFontData` functions used in Flash to obtain the list of fonts. They consider a script is doing fingerprinting if it loads more than 30 fonts, enumerates plugins or mimeTypes and accesses the screen and navigator properties. With this methodology, they detected 13 distinct fingerprinting scripts present on 404 websites doing JavaScript font enumeration.

In 2014, Acar *et al.* [5] conducted a large scale study about stateful and stateless tracking mechanisms used in the wild. In particular, they were the firsts to measure the use of canvas fingerprinting at scale. To detect scripts that collect canvas fingerprints, they log values returned by the `toDataURL` function used to obtain the value of a canvas. They also monitor the arguments of the `fillText` and `strokeText` functions used to draw text on a canvas. To decrease false positives, they consider a script is using canvas for fingerprinting if both the `toDataURL` and `fillText` or `strokeText` functions are called. Moreover, they also define a constraint on the size of the canvas that should be at least 16x16 pixels. Finally, the image should not be requested in a lossy compression formation, such as JPEG. They observed that 5.5% of the websites in the Top Alexa 100K were using canvas fingerprinting on their home page. While there were 20 different companies providing canvas fingerprinting scripts, one of the companies, AddThis, represented more than 95% of the scripts. Moreover, they noticed that fingerprinters had considerably improved the canvas fingerprinting techniques since the original study conducted by Mowery *et al.* [38]. For example, new canvas fingerprinting scripts draw the same text twice with different color and trigger the default fallback font. These scripts also use pangrams—*i.e.*, strings that include all the letters in the alphabet—as well as different emojis. While Acar *et al.* argued that emojis were used to check if the browser supported emojis, Laperdrix *et al.* [27] later showed that beyond testing emoji support, emojis were also rich source of entropy since their representation depends on the OS and the device.

More recently, in 2016, Englehardt *et al.* [6] crawled the top Alexa 1M to study the use of cookies and multiple fingerprinting techniques. They proposed OpenWPM, an extensible crawler framework that aims at making privacy studies at scale easier. They detected more than 81,000 third parties present on at least two first-parties. Moreover, they showed that four companies, Google, Facebook, Twitter and Adnexus, were present each on more than 10% of the websites crawled.

To measure the use of fingerprinting, they monitored access to properties commonly used for fingerprinting, similarly to the approach proposed by Acar *et al.* [10]. They detected that among the Top Alexa 1M websites, canvas fingerprinting was only used by 1.6% of the websites. Nevertheless, canvas fingerprint was used by 5.1% of the websites in the Top Alexa 1K. Thus, they showed a decrease of canvas fingerprinting use compare to the previous study conducted by Acar [5] in 2014. In particular, the popular fingerprinting script delivered by AddThis was no longer in use in 2016. They also measured the use of canvas-based font enumeration and showed that it was used by 2.5% of the websites in the Top Alexa 1M. Finally, they measured the use of audio fingerprinting at scale and detected 518 websites that compute an audio fingerprint, among which 512 delivered scripts from the same company.

2.3 Countermeasures Against Fingerprinting

In this section, we present the three main strategies to protect browser against browser fingerprinting:

1. **Blocking the execution of fingerprinting scripts.** This strategy can be achieved by disabling JavaScript or by intercepting requests that load fingerprinting scripts;

2. **Breaking the stability of browser fingerprints.** Fingerprint tracking requires both uniqueness and stability to be effective. This strategy aims at frequently modifying the attributes constituting a fingerprint in order to break the fingerprint stability, and thus make tracking impossible or less effective;

3. **Breaking the uniqueness of browser fingerprints.** This strategy acts on the uniqueness required for tracking. It aims at increasing the anonymity set of each fingerprint so that multiple browsers from different users share the same fingerprint or fingerprints with high similarity.

Countermeasures can achieve these strategies by implementing different mechanisms. For example, in order to unify the value of fingerprints, one can either lie about the values returned by fingerprint attributes so that all browsers return the same value or one can block access to attributes with a high entropy so that browsers converge towards a similar fingerprint. In the following subsections, we go through the three different defense strategies. For each of the strategies, we present the different countermeasures

that use this strategy and the different mechanisms they implement to achieve it. Note that some countermeasures may implement multiple strategies or hybrid strategies. For example, FaizKhademi *et al.* [45] proposed a modified Chromium with two modes, a first mode that aims at making all fingerprints look the same and a second mode that aims at breaking the stability of fingerprints by randomizing their values.

We first present countermeasures that protect by blocking the execution of fingerprinting scripts. Then, we present the countermeasures that aim at breaking the stability of fingerprints and the countermeasures that unify browser fingerprints to make each browser less unique. Finally, we present the main weaknesses of the countermeasures presented in this section.

2.3.1 Blocking Fingerprinting Script Execution

The first strategy we present relies on blocking the execution of fingerprinting scripts. Blocking the execution of the script makes the collection of the fingerprint impossible. While the server can still collect a reduced version of the fingerprint using HTTP headers, not collecting JavaScript attributes hugely decreases the entropy of the fingerprint. Countermeasures that aim at blocking script execution are not specifically designed to counter browser fingerprinting. Nevertheless, they may include rules that block some fingerprinting scripts. These countermeasures are among the most popular privacy-enhancing technologies [46]. For example, in March 2019, four out of the ten most popular browser extensions for Firefox where ad-blockers and tracker-blockers [47]. In particular, the two most popular extensions, AdblockPlus [48] and uBlock Origin [49] represent more than 11% of the total browser extensions used on Firefox. The majority of these script blocking countermeasures relies on crowdsourced filter lists that specify if a resource should be blocked. There exist different lists that serve different purposes. One of the most popular, EasyList [50], focuses on blocking advertising content while EasyPrivacy [51] focuses on blocking trackers, which can include fingerprinting-based trackers. These lists are used in popular browser extensions, such as AdblockPlus [48], uBlock origin [49] or Adblock [52], as well as browsers, such as Brave [53] that integrates a native ad-blocker. Other browser extensions, such as Ghostery [54], rely on proprietary filter lists to block content. One of the main problems of these lists, whether they are proprietary or not, is that they need to be manually updated and require a significant amount of work to be maintained [19]. Thus, other more dynamic approaches have been proposed to get rid of these lists. For example, Privacy Badger [55], a browser extension

developed by the EFF, uses heuristics to determine if a request should be blocked. It keeps track of third-party resources included in the pages visited and observe if their behavior is similar to the ones of trackers based on their use of cookies, local storage or even browser fingerprinting techniques. When it observes a suspicious third-party on more than three domains, Privacy Badger automatically blocks the content. Umar *et al.* [56] apply a machine learning-based approach that considers features extracted from HTML elements, HTTP requests, and JavaScript to determine if a request should be blocked. Merzdovnik *et al.* [57] quantified the effectiveness of ad-blockers and trackers blockers at scale. They show that rule-based extensions, such as uBlock Origin or Ghostery outperform learning-based extensions, such as Privacy Badger, even though they took care of training Privacy Badger's heuristic on 1,000 websites before applying it during their evaluation. They show that while the majority of these blocking tools are effective against stateful third-party trackers, they all failed to block well-known stateless trackers that use browser fingerprinting. Englehardt *et al.* [6] also showed that popular filter lists tend to detect only a fraction of fingerprinting scripts.

Finally, a more radical approach is to block the execution of JavaScript code. The most popular tool for blocking JavaScript is the NoScript [58] browser extension. Other browser extensions, such as uBlock Origin [49] and uMatrix [59], as well as other browsers, such as Brave [53] or Tor browser [60] also propose convenient mechanisms to disable JavaScript execution. While this approach guarantees to block JavaScript-based browser fingerprinting, it may also render many websites unusable since the majority of websites relies on JavaScript to make their site dynamic. Moreover, as shown by Yu *et al.* [61], breaking websites and thus decreasing the usability can also lead to a decrease of privacy as users are more tempted to disable their countermeasures, without understanding the privacy implications of doing it.

2.3.2 Breaking Fingerprint Stability

Another defense strategy consists in modifying frequently the values of different attributes of a fingerprint to break the stability property required for tracking fingerprints over time. The user agent is a key attribute for fingerprinting as its value reflects the browser and the OS used by the user. For this reason, a wide range of user agent spoofer extensions enables to lie on the user agent sent by the browser. For example, Ultimate User Agent [62], a Chrome extension enables to change the user agent enclosed in the HTTP requests as the original purpose of this extension is to access websites that demand a specific

browser. The main drawbacks of user agent spoofers to protect against fingerprinting lies in the fact that they create inconsistent browser fingerprints [4]—*i.e.*, combinations of attributes that cannot be found in the wild.

More advanced extensions, such as Random Agent Spoofer [63] aim to address this inconsistency problem. Random Agent Spoofer (RAS) is an extension that was available until Firefox 57—in version 57, Firefox changed the APIs for browser extensions so that they become compatible with the Chrome browser—that protects against fingerprinting by providing a mechanism that enables to switch between different device profiles, composed of several attributes, such as the user agent, the platform, and the screen resolution. Even though RAS is not available on modern versions of Firefox, it has been forked and recently ported to web extensions that are supported by the most recent versions of Firefox [64]. Since the device profiles used to spoof fingerprints are extracted from real browser configurations, all of the attributes contained in a profile are consistent with respect to each other. Besides spoofing attributes, RAS also enables to block advanced fingerprinting techniques, such as canvas, WebGL or WebRTC fingerprinting.

Nikiforakis *et al.* [7] proposed PriVaricator, a modified Chromium browser that randomizes the list of plugins and the list of fonts. Besides the high entropy of these two attributes [3, 27, 28], the main reason PriVaricator focuses on these attributes is to avoid inconsistencies in the fingerprints generated. Indeed, their strategy does not lie about the browser or its version nor about the platform the browser is running on, making it more difficult for an adversarial fingerprinter to detect the use of a countermeasure. To randomize the list of plugins, they define a probability of hiding each individual entry in the list of plugins. Concerning the list of fonts, they focus on font enumeration using JavaScript. They override values returned by two properties, `offsetHeight` and `offsetWidth` as well as the `getBoundingClientRect` function, the three of them being used for font enumeration [65, 4, 45]. They proposed three font randomization policies that become active whenever a script accessed one of these properties more than a defined threshold. They implemented their changes directly into Chromium C++ code for performance purposes and also because the `offsetWidth` and `offsetHeight` properties are not properties of the `HTMLElement` prototype, making it more difficult to override these properties directly in JavaScript in an efficient way. They evaluated their approach based on three criteria:

1. **Performance.** They measured the performance overhead using three JavaScript benchmarks and noticed no statistically significant overhead;

2. **Privacy protection.** They also evaluated the privacy gain against four fingerprinters: BlueCava and Coinbase, two commercial fingerprinting scripts, FingerprintJS, an open-source fingerprinting script and PetPortal, an academic research platform. Overall, their approach was able to generate unique and different fingerprints against the four fingerprinters, which means it was effective to protect against browser fingerprinting;

3. **Visual breakage.** Finally, they evaluated the visual breakage caused by their tool. To do so, they instrumented PriVaricator with different randomization policies that were considered successful during the evaluation step; They measured a negligible visual breakage of 0.6% on average with their third randomization policy.

Torres *et al.* [66] proposed FP-Block, a Firefox browser extension that ensures that any embedded party will see a different fingerprint for each site it is embedded in. Thus, a browser fingerprint can no longer be linked between different websites, even when the same third-parties are included on these websites. Whenever a user visits a new site, FP-Block generates a new fingerprint so that different websites observe different fingerprints for the same browser. Their approach focuses on properties of the `navigator` and the `screen` objects. It also adds random noise to canvas fingerprints. FP-Block's authors are aware of what they call the "fingerprinting countermeasure paradox"—*i.e.*, the fact that using a fingerprinting countermeasure can make a user more identifiable if the countermeasure is detected since she becomes more unique and identifiable. Thus, contrary to naive countermeasures that randomize the value of attributes without any constraints, FP-Block tries to ensure fingerprint consistency. To generate consistent fingerprints, they model how different attributes of a fingerprint relate to each other using a Markov chain model.

FaizKhademi *et al.* [45] proposed FPGuard, a combination of a modified Chromium and a browser extension. The browser extension aims at detecting fingerprinting scripts to blacklist them, while the modified Chromium aims at spoofing attributes of a fingerprint. The browser extension monitor accesses to attributes and functions used for fingerprinting. To detect if a script uses fingerprinting, they rely on 9 different metrics, such as the number of `navigator` and `screen` properties accessed or whether a canvas element has been programmatically accessed. Then, using these metrics, they assign a score that represents a level of suspicion that the script is doing fingerprinting. Whenever a script is considered suspicious, it is added to a blacklist and is automatically blocked the next times a website tries to load it. Thus, FPGuard can also be partly classified in

the set of countermeasures that block fingerprinting script execution. Their modified Chromium spoofs the values of attributes and functions used in fingerprinting whenever a user visits a website. They argue that naively randomizing the values of fingerprints can degrade the user experience since some websites rely on the screen resolution or the type of device to properly display their content. Thus, they aim at generating fingerprints that "represent the properties of the browser almost correctly". To do so, they developed different randomization strategies for the `screen` and `navigator` objects, for the list of plugins, for the canvas as well as for the list of fonts. For example, they may alter the browser sub-version or randomize canvas by adding minor noise to its content. FPGuard's authors are aware that their countermeasure can be detected because of their extension module that overrides getters in JavaScript. Nevertheless, they argue that since a fingerprinter cannot recover the original values, it is still a privacy improvement.

Baumann *et al.* [8] proposed DCB, a modified Chromium, that protects against browser fingerprinting and in particular, Flash fingerprinting. To do so, they proposed two opposite strategies that leverage a dataset of real browser fingerprints:

1. **1:N.** one browser, many configurations. The goal of this strategy is to have unique fingerprints that keep on changing to break the stability required for tracking;

2. **N:1.** many browsers, one configuration. The goal of this strategy is to create collisions between the fingerprint of different browsers. To do so they apply modifications to the fingerprints of different browsers so that they converge towards a unique fingerprint. Thus, browsers with this fingerprint become more difficult to track since they are not unique anymore.

The *N:1* strategy means that DCB can also be partly classified in the set of countermeasures that aim at breaking the uniqueness of browser fingerprints. They proposed the notion of configuration groups to generate consistent fingerprints. While this notion is not well defined in the paper, they provide an example saying that a possible configuration group could consist of users with the same browsers, OS and language. Thus, no actual users would have to adopt a configuration that contradicts their real system and browser configurations. In addition to modifying attributes related to the size of the screen or the browser language, they also proposed a particular strategy to randomize canvas fingerprint. Instead of applying random noise to canvas, DCB modifies the `drawTextInternal` function used to render canvas in a deterministic way. Thus, between two sessions, canvas is different, but for a given session the modifications applied to canvas are constant. Thus, DCB is resilient to replay attacks—*i.e.*, attacks

where a fingerprinter asks to generate the same canvas multiple time to observe if the values returned are the same. Similarly to Nikiforakis *et al.* [7], they tested DCB against BlueCava and Coinbase, two commercial fingerprinting scripts, and FingerprintJS, an open-source fingerprinting library. While their *1:N* strategy was able to generate more than 99% of unique fingerprints, making it effective against long-term fingerprinting tracking, their *N:1* strategy that aimed at breaking the uniqueness of browser fingerprints performed worse since some of the fingerprinters were still able to distinguish different browsers that should have had the same fingerprint.

FPRandom [9] is a modified version of Firefox that adds randomness in the computation of the canvas fingerprint, as well as the audio fingerprint. FPRandom also randomizes the order of the `navigator` properties. They decided to focus on canvas fingerprinting since it is a strong source of entropy [27, 28]. Moreover, both audio and canvas fingerprints rely on multimedia functions that can be slightly altered without significantly degrading the user experience. Concerning the order of the `navigator` properties, it is not defined by the ECMAScript specification [67], which means that it is up to the browser vendor to decide the order. FPRandom includes two modes, one in which the noise added to the canvas and the audio fingerprint is different at every call and a second mode where the noise remains constant during a session. The goal of the second mode is to protect against replay attacks. Contrary to canvas poisoning extensions that apply random noise independently for each pixel, FPRandom adds a more consistent noise. Indeed, they modify the `parseColor` function of the `canvasRenderingContext2D` class so that whenever a color is added to the canvas, it is slightly modified. Thus, with their approach, all the shapes of the canvas that use a given color will have the same color.

Canvas Defender [68] is a browser extension available on Chrome and Firefox that adds a uniform noise to a canvas. The first time Canvas Defender is installed, it generates four random numbers corresponding to the noise that will be applied to the red, green, blue and alpha components of each pixel. Thus, since the four random noise numbers are constant, the extension is not vulnerable to replay attacks.

Finally, Blink [69] exploits reconfiguration through virtual machines or containers to clone real fingerprints—*i.e.*, contrary to countermeasures that lie on their identity by simply altering the values of the attributes collected—Blink generates virtual environments containing different fonts, plugins, browsers in order to break the stability of fingerprints, without introducing inconsistencies. Thus, the main strength of Blink is that contrary to other countermeasures that lie on the values of the attributes in order to

change the fingerprint, Blink does not lie on the fingerprint. Indeed, all the fingerprint it generates are genuine, the modifications are done directly at the virtual machine level, which means that they could exist in the wild. Nevertheless, this approach has a cost: it requires to use a virtual machine of multiple gigabits in order to browse the web, which makes this approach less user-friendly than a browser extension or a forked browser. Moreover, since Blink is running in a virtual machine, users can be detected using red pill techniques [70].

2.3.3 Breaking the Uniqueness of Browser Fingerprints

The last defense strategy we present aims at breaking the uniqueness of browser fingerprints from different browsers so that users become less unique. Contrary to the previous strategies that aimed at breaking the stability of the fingerprints by randomizing different attributes, this strategy aims at increasing the anonymity set of each user. To achieve this goal, one can either block access to attributes with high entropy or spoof their values so that multiple users return the same value.

Started from version 41 [34], Firefox started to implement anti-browser fingerprinting features similar to those available in the Tor Browser. Firefox standardizes values of attributes used for fingerprinting, such as the user agent, the timezone or the number of cores in the CPU. For the user agent, it spoofs the browser version by replacing it with the version of the latest *extension support release* (ESR) of Firefox. To spoof the timezone, Firefox pretends the user is located UTC timezone—*i.e.*, it returns a timezone offset of 0 when `new Date().getTimezoneOffset()` is called. Concerning the number of cores, it modifies the value of `navigator.hardwareConcurrency` so that all browsers pretend to have two cores. Besides standardizing the values of attributes, Firefox also blocks access to critical functions used for canvas fingerprinting. Whenever a script tries to access a canvas value using `toDataURL` or `getImageData`, Firefox asks the permission to the user,[35] similarly to what is done with the geolocation API for example. Firefox also blocks the access to several APIs, such as `WEBGL_debug_renderer_info`,[36] the geolocation or the device sensors. Moreover, whenever the fingerprinting protection is activated, `navigator.plugins` and `navigator.mimeTypes` also return empty arrays.

[34] https://bugzilla.mozilla.org/show_bug.cgi?id=418986
[35] https://bugzilla.mozilla.org/show_bug.cgi?id=967895
[36] https://bugzilla.mozilla.org/show_bug.cgi?id=1337157

Brave browser [53] is a Chromium-based browser oriented towards privacy that proposes specific countermeasures against browser fingerprinting. In particular, they enable to block attributes identified as having a high entropy, such as audio, canvas, and WebGL fingerprinting. They also block local IP address leakage through WebRTC and they disable the battery API since it can be used for fingerprinting purposes [36]. Moreover, Brave browser also integrates natively an ad-blocker and a tracker blocker that rely on crowdsourced filter lists, as well as a simple mechanism to disable the execution of JavaScript on a page.

Canvas Blocker [71] is a Firefox extension that blocks access to the HTML 5 canvas API. Besides blocking, it also provides another mode, similar to Canvas Defender [68], that randomizes the value of a canvas every time it is retrieved. Thus, it can also be classified in the category of countermeasures that act by breaking the stability of fingerprints.

2.3.4 Summary of Existing Countermeasures

Table 2.2 provides an overview of the different countermeasures we presented in this section associated with their strategies to protect against fingerprinting.

2.3.5 Limits of Fingerprinting Countermeasures

Eckersley discussed the impact of privacy-enhancing technologies against fingerprinting. He considers some of the countermeasures can be productive in the case they countermeasures can be detected because of their side effects. For example, they noticed users in their dataset that were using Privoxy,[37] a privacy-enhancing proxy that aims at blocking advertising and trackers. Privoxy was altering the user agent sent by adding the "Privoxy" string in it. Thus, while the original goal of the extension was to preserve users against trackers, this feature could be used by fingerprinters to specifically target privacy-aware users. More generally, they argue that whenever a countermeasure has observable side-effect, its effect can be counterproductive if it is used by a few users.

Nikiforakis *et al.* [4] also discussed the privacy impact of using simple countermeasures, such as user agent spoofers. They showed that, while the user agent provides information about the user's browser and OS, changing its value can be counterproductive. Indeed,

[37] https://www.privoxy.org/

Table 2.2 Overview of the different countermeasures and their strategies to protect against browser fingerprinting.

Countermeasure	Blocking script	Breaking stability	Unifying
uBlock Origin [49]	✓		
Adblock [52]	✓		
Ghostery [54]	✓		
Privacy Badger [55]	✓		
Adblock Plus [48]	✓		
NoScript [58]	✓		
uMatrix [59]	✓		
Tor browser/Firefox [60]	✓		✓
Brave browser [53]	✓		✓
FPGuard [45]	✓	✓	
PriVaricator [7]		✓	
FP-Block [66]		✓	
DCB [8]		✓	✓
Canvas Defender [68]		✓	
Random Agent Spoofer [63]		✓	✓
Canvas Blocker [71]		✓	✓
Blink [69]		✓	
FP-Random [9]		✓	
Ultimate User Agent [62]		✓	

naive spoofers tend to generate inconsistent fingerprints—*i.e.*, combination of attributes that cannot be found in the wild. For example, in the user agent the browser claims to be on MacOS while the `navigator.platform` attribute returns Linux. Thus, they argue that using such countermeasures for privacy purposes can be counterproductive since fingerprinters can more easily target users with this kind of extensions. They also showed that it is unclear how effective user agent spoofers are at breaking the stability of fingerprints. Similarly to Mulazanni *et al.* [37], they showed that using different techniques such as the presence or absence of different browser features, or the behavior (mutability and order) of special browser built-in objects such as `navigator` or `screen`, it was possible to infer the real browser of the user.

Acar *et al.* [10] studied the Tor browser protection against fingerprinting. They showed the difficulty to properly protect against all fingerprinting channels. Indeed, while the Tor browser aimed at protecting against font enumeration by limiting the number of fonts that can be queried by a page, it was still possible to test the presence of fonts without any limits using the CSS `font-face` directive. They also studied Fireglove, a browser extension created for research purposes. Fireglove aimed also at protecting against font enumeration by limiting the number of fonts that can be loaded by a tab by reporting wrong `offsetHeight/Width` values. Nevertheless, they could bypass the protection using the `getBoundingClientRect` function that provides similar information as `offsetHeight/Width`. Fireglove also randomized the value of different attributes such as the screen resolution, or the user agent. Nevertheless, Acar *et al.* showed that Fireglove failed at properly lying about the real nature of the browser. For example, when they pretended to be a Chrome browser, they did not remove APIs only available on Firefox such as `navigator.mozCameras`.

More recently, in 2019, Schwarz *et al.* [72] proposed an approach to automatically learn the browser fingerprint differences between browsers. While their approach can be used for targeting exploits that work only on specific OS, architecture or browser, it can also be used to detect the presence of privacy-enhancing extensions. They applied their automated approach to 6 privacy extensions and were able to detect all of them. In particular, similarly to the evaluation we conduct in Chapter 4, they were able to detect the presence of the Canvas Defender countermeasures because of its side-effects.

2.4 Security Applications

In this section, we first present approaches that leverage browser fingerprinting in a security context. We show how it can be used to enhance authentication and to detect crawlers. Then, we present other non-fingerprinting crawler detection techniques to better understand how fingerprinting based detection compare to them.

2.4.1 Enhancing Web Security Using Browser Fingerprinting

2.4.1.1 Enhancing Authentication Using Fingerprinting

Different studies focused on the use of browser fingerprinting to enhance the security of HTTP sessions and authentication, either a second factor or in addition to other traditional second factors, such as SMS or emails. The idea is, for each user, to collect the browser fingerprints of her trusted devices. Then, whenever she tries to connect to a website, the server can verify both the correctness of her password and whether or not the device the user is trying to connect from belongs to her list of trusted devices.

Unger *et al.* [11] proposed to use fingerprinting to protect against HTTP(s) session hijacking. Their framework, SHPF, collects the browser fingerprint when the user logs-in. The fingerprint collected is constituted of HTTP headers, as well as features corresponding to the presence or the absence of several CSS and HTML5 features, whom presence was not widespread at the time the paper was written in 2013. Then, continuously during a session, their framework collects and monitors the browser fingerprint to detect changes that could indicate that the session has been hijacked by an attacker, for example, because the session cookie has been stolen using an XSS vulnerability [73]. Thus, if the fingerprint changes, going from a Windows 10 on Chrome 65, to an Ubuntu on Firefox 55 for example, or if the IP address suddenly changes, SHPF interrupts the session and redirects the user to a stronger authentication mechanism.

Preuveneers *et al.* [14] developed an authentication framework, SmartAuth, that uses dynamic context fingerprinting to enhance authentication. Contrary to Unger's [11] approach that considers only static fingerprints, whose values did not depend on any kind of context, SmartAuth takes different context information into account, such as the geolocation or the time. For example, when a user is connecting at work she may use a monitor with a certain resolution. Nevertheless, when she comes home, she may use another monitor that has a different resolution. Thus, their approach aims at taking this

context into account to provide better security guarantees, as well as less false positives during the authentication process. The fingerprints they collect are constituted of different attributes accessible using JavaScript, such as the user language, the screen resolution or the list of plugins. They also collect the IP address range and information provided by HTTP headers. They acknowledged one of the problems with using fingerprinting in a security context: fingerprint attributes spoofing. Since fingerprints are collected in the browser—*i.e.*, on the client side—every attribute can be modified by an attacker, either when it is transmitted to the server or at runtime by overriding JavaScript getters and functions used in the fingerprinting process. Thus, they added a checksum mechanism to ensure the fingerprint collected has not been modified during its transmission to the server. They also acknowledged the possibility of an attacker stealing a browser fingerprint in addition to the password, for example using a phishing website, and then this attacker could try to replay the fingerprint stolen in order to bypass the security mechanisms. To protect against replay attacks, they added a counter whose value is incremented during the user session, as well as a timestamp whom they verify the value on the server-side whenever a fingerprint is collected. Thus, if a fingerprint corresponds to a non-trusted device, or if they detect that it has been tampered, they redirect the user a stronger fallback authentication mechanism. They evaluated their approach using more than 2,000 different system and network configurations in 10 different contexts and were able to achieve a 99% accuracy using only 10 fingerprinting attributes.

Alaca *et al.* [12] analyzed the strengths and weaknesses of several fingerprinting attributes when used in an authentication context. Contrary to tracking, when fingerprinting is used in a security context, it is important to consider the fact that an attacker may try to spoof fingerprint attributes or replay fingerprints. They analyzed 29 attributes, ranging from simple attributes provided by the browser using JavaScript, to more complex attributes, such as the TCP fingerprinting stack. For each of these attributes, they analyzed five important properties to consider when fingerprinting is used in a security context:

1. **Entropy.** Attributes with high entropy enables to distinguish between different users and thus reduce the chance of fingerprint collisions that could be used by an attacker to connect from an untrusted device;

2. **Repeatability.** Given the same browser, hardware and network configuration, has an attribute the same value?

3. **Resource consumption.** The CPU and memory overhead required to obtain the value of an attribute; While they consider a high overhead is not necessarily a

problem for authentication since each attribute is collected only once, this property is important when fingerprinting is used continuously to protect against session hijacking [11];

4. **Spoofing resistance.** While some attributes, such as the IP address are difficult to spoof, it is not the case of simple attributes, such as the user agent that can be overridden using few lines of JavaScript or a browser extension;

5. **Stability.** The attribute should be stable over time in order to link different fingerprints of the same browser.

Van Goethem *et al.* [13] focused on authentication on mobile devices. They proposed an approach that does not require any permission and that works in a browser. To build a fingerprint, they use data provided by the accelerometer sensor, all while making the phone vibrate using the `navigator.vibrate` API. They consider that accelerometer sensor is a good candidate for authentication since it enables to create an approach more resilient against replay and spoofing attacks. When a user registers, their fingerprinting script collects several traces of accelerometer data. Each trace is constituted of multiple chunks that correspond to a varying period of time during which the phone is vibrating. For each chunk, they collect the information returned by the accelerometer and extract features such as the minimum, maximum and mean acceleration along the different axis. Then, when a user tries to connect to her account, they send her a challenge that consists in generating randomly selected chunks. If enough chunks are consistent, according to a determined threshold, the user is allowed to connect. They evaluated the uniqueness and stability of the different features they proposed using three different mobile devices. They showed that accelerometer data were both unique and stable, in particular, short chunks contained more entropy than long chunks. They evaluated their approach using 15 different devices and only one fingerprint was classified as accepted even though it should have been rejected.

In his thesis, Laperdrix [74] proposed to use canvas fingerprinting in a challenge-response protocol as a second-factor authentication mechanism. Their approach relies on canvas rendering because of its unpredictable but yet stable nature. Contrary to other static fingerprinting attributes studied in the literature, it is less vulnerable to spoofing and replay attacks when used properly. The first time a user connects to her account, the server sends her one challenge that consists in drawing a canvas on which several random operations are applied, such as writing text or drawing geometric shapes. Once rendered, the browser sends the result of the canvas rendering to the server that stores it. The next

time the users tries to connect, the server sends her two different challenges. The first challenge is the same as the one used for the previous connection. After the first canvas is drawn, the browser sends the value back to the server that verifies if it is equal to the canvas they stored at the previous connection. If it is the case, the user can connect to her account, otherwise, the server may ask the user to verify her identity using another mechanism, such as an SMS or an email. The browser also solves the second challenge and sends the value back to the server that stores it. Thus, the value of the second canvas will be used for the next connection of the user. Since canvas challenges are not reused, even if an attacker was able to steal a canvas value from her victim to replay it, she will not be able to solve the canvas rendering challenge because it will have changed.

For this approach to work, they need, given a certain random seed, to generate unique and stable canvas. They run three collect phases where they collected different kinds of canvas fingerprints that use different canvas primitives, such as drawing text or curves. Their final canvas algorithm takes as input a seed and randomizes different parameters such as the number of strings to draw, their size and their rotation, as well as curves, color gradient and the shadow applied to the canvas. They verified the uniqueness of their canvas challenges on more than 1 million browsers using different seeds and did not observe any collisions. They also used the AmIUnique extensions[38] to study the stability of their canvas and showed that, on 27 browsers monitored for a year, half of them had less than three canvas changes. Concerning performance, they showed that from end to end, when considering the generation of the challenge, the time for the browser to solve it by drawing the canvas and then hash the result, it takes less than 250 ms on average.

2.4.1.2 Detecting Bots and Crawlers Using Fingerprinting

Another security application of browser fingerprinting is bot and crawler detection. While Acar *et al.* [10] identified scripts from companies that claimed to use it for bot and crawler detection, few studies focused on this topic.

Bursztein *et al.* [15] proposed Picasso, an approach that relies on canvas fingerprinting [38] to create dynamic challenges that aim at detecting emulated devices or devices that lie in their user agent. Their approach has several applications, such as distinguishing between real Android devices used by humans and emulated devices used to post fake reviews on App Store or to artificially increase the number of videos viewed. Similarly to the

[38] https://amiunique.org/timeline

approach proposed in Laperdrix's thesis [74], Picasso is a challenge-response protocol that relies on the unpredictable but yet stable nature of canvas rendering. Their approach aims at being resilient to replay attacks and to skilled adversaries with perfect knowledge of the code used for the detection. Picasso relies on drawing random canvas primitives, such as Bezier curves, polynomial curves or text, with random parameters. Contrary to situations where canvas is used for tracking, the canvas generated should be the same among devices and browsers of the same nature but different otherwise. Their approach works as follows:

1. A server sends the canvas algorithm code, along with a random seed used to initialize the pseudo-random generator (PRNG), as well as a number of rounds, *i.e.*, the number of canvas primitives to draw,
2. The client initializes the PRNG and creates an empty canvas,
3. The client executes the challenge. At each round, it randomly selects a canvas primitive with random parameters and applies shadow and color gradient to the canvas,
4. At the end of each round, the value of the canvas is hashed with the output of the previous round,
5. The final result is sent back to the server for verification.

Their challenge can be seen as a proof of work since it requires the device to spend CPU or GPU resources, as well as RAM resources to solve it. Moreover, the amount of resources needed to solve the challenge can easily be increased, either by increasing the size of the canvas or the number of rounds. A challenging part of their approach is the bootstrap phase. During this phase, they need to collect values of canvas for different seeds and different classes of devices. To address this problem, they proposed several solutions, such as sending challenges to trusted devices, *e.g.* devices belonging to users logged in to the website service and with a good reputation, or to buy missing devices, which can be expensive.

To protect against replay attacks or against attackers that would pre-compute canvas values for several challenges using other devices, they continuously generate new challenges. Their idea is similar to what was done for the first versions of Google reCAPTCHA that displayed two words in the CAPTCHA: a word known and a word unknown. In their case, they send a challenge whom they know the response along with multiple challenges whom they ignore the response. If the known challenge is solved correctly,

they consider the responses to all the other unknown challenges to be correct. Thus, they can continuously get values associated with new challenges. To protect against a pollution attack—*i.e.*, an attacker that would submit the right answer to the known challenge but wrong answers to unknown challenges—they apply similar techniques as the one used to protect against cheating in user-generated labels [75].

They evaluated their approach on more than 52 million devices and were able to distinguish all device classes with 100% accuracy. In total, they obtained around 130K unique responses from 52M challenges. They showed that the time to generate a canvas is linear with respect to the number of rounds. When 50 rounds were used, it required at most 400 ms. Anecdotally, during their experiment they were able to detect PhantomJS browsers that spoofed their browser and OS, running attacks from AWS ec2 instances.

2.4.2 Detecting Bots and Crawlers Without Fingerprinting

HTTP headers and trafic shape analysis.

Jacob *et al.* [76] proposed a system to manage unwanted crawlers. Their system builds a knowledge base of IP addresses and indicates if they belong to a human or to a crawler. When a request is done, they consult the knowledge. If the IP address belongs to a human or to an allowed crawler, *e.g.* Google bot, the request is allowed. Otherwise, if the request belongs to an unwanted crawler, the request is blocked. When the IP address is not in the knowledge base, they need to determine whether or not it belongs to a human or a crawler. To do so, they proposed three approaches:

1. **Heuristic based detection.** Their first approach leverages a set of heuristics that rely on features extracted from the HTTP headers, as well as the URLs requested by the user. Some of these features had already been proposed in the state-of-the-art [77–82], such as the error rate, the proportion of pages revisited or whether or not the user ignores cookies. They also proposed new features, such as low URL parameters usage or whether or not URLs are accessed in alphabetic order. If the majority of the heuristics consider the user is a human, then the IP address is whitelisted in the knowledge base. Otherwise, they consider the IP address is used by a crawler and the IP address is blacklisted;

2. **Traffic shape detection.** Their second approach models the traffic of a user as a time series. It aims at being more effective than other state-of-the-art approaches that divide chunks of requests into session on which they extract features, such

as the mean arrival time or the number of requests. In their case, the goal of this second approach is to extract features from the user time series to classify the traffic. First they compute the *sample auto-correlation* function (SAC). This statistical function aims at estimating the stability of the traffic for an IP address. They extract features from this function, such as the speed of decay, that captures the short term stability of the traffic. For example, a small decay is often linked to a non-human activity since it indicates the traffic is stable over a long period of time, contrary to a fast decay that indicates more instability, which is often linked to human activity. They also extract the number of local spikes in the function, as well as whether or not these spikes are observed at defined lags such as half days or days, which are often signs of human activity. In a second time, they use time-series decomposition [83]—*i.e.*, a process that consists in decomposing a time-series into a trend, a season, and a noise component. Since naive crawlers tend to have a stable activity, the trend component should be almost stable—*i.e.*, the derivative of the trend component should be almost 0, contrary to users which have more erratic patterns. Concerning seasonality, humans tend to have patterns at the day and weeks frequency, contrary to crawlers which do not follow human cycles. To classify the traffic as human or crawler, they proposed three machine-learning classifiers, each using features extracted from the SAC function and from the time-series decomposition. The final result is the majority of the three classifiers.

3. **Distributed crawlers.** Their third approach enables to detect crawling campaign distributed over multiple hosts. To do detect such campaigns, they apply incremented clustering techniques on time series. The idea is to cluster time-series with high similarity, which could represent crawlers with the same code, launched from different IP address. Since crawlers are not launched at the exact same time, their clustering approach aims at being resilient to translation in the time series.

All of their three approaches require a significant amount of data before they can accurately classify the traffic. Thus, if an IP address is not yet in the knowledge base, they need to obtain data on it. To address this problem, they allow a number of requests for each IP address not in the knowledge base. Once the IP address has reached the limit, then their system sends crawler traps, such as CAPTCHAs or hidden links.

They implemented their approach and evaluated it on a large social network. They used 10 days of real-world traffic taken from the server logs. They exclude users who had less than 1,000 requests per day since they consider it is not enough for their second and third approaches based on time-series analysis. Nevertheless, they consider these sources

will be handled by their active containment policy. For their evaluation they needed to obtain the ground truth—*i.e.*, whether or not an IP address belongs to a human or a crawler. To obtain these labels, they used a semi-automatic approach:

- They applied their approach based on HTTP headers and asked for feedback from the social network engineers. Based on this feedback, they adjusted the labels.
- They performed manual analysis by looking at the time series, user agent strings to adjust the labels.

In order to choose the different parameters of their heuristics and train their machine learning models, they used a train set of more than 70M requests from 813 IP addresses. Then, they evaluated their three heuristics on a test set of 62M requests from 763 IP addresses. Their first approach that relies on a set of heuristics based on HTTP headers and URLs achieved a detection rate of 71.6%. Their second approach based on traffic shape analysis achieved an accuracy of 94.89%. Finally, their third approach that aims at detecting distributed crawling campaign was able to achieve 91.89% of accuracy. Combining the different the first two approaches did not improve the accuracy since crawlers detected by HTTP headers and URLs based heuristics were already detected by the traffic shape detection approach.

CAPTCHAs. CAPTCHAs [84] rely on Turing tests, such as image recognition, to determine if a user is human. Figure 2.5 shows an example of a Google reCAPTCHA, a popular service proposed by Google. While the use of CAPTCHAs is widespread, their main drawback is that they require user interaction. Moreover, recent progress in image and audio recognition, as well as crowdsourcing services [85, 86] have made it easier to break popular CAPTCHA services, such as Google's reCAPTCHA [87, 88].

Behavior biometrics. Chu *et al.* [89] leverage behavioral biometrics to detect bots that post spam in the comment section of blogs. Their hypothesis is that human users need their mouse to navigate on a blog and also their keyboard to type comments. They run customized bots in a controlled environment to collect events such as keystrokes or clicks and train a decision tree that relies on the features collected to predict if a user trying to post a comment is a human or a bot.

Video games & social networks. The problem of bot detection has also been studied in the context of video games [90, 91] and social networks, such as Twitter [92, 93]. Wang *et al.* [94] addressed the problem of detecting fake accounts used to send spam or spread malware via social media. They trained an SVM classifier using features

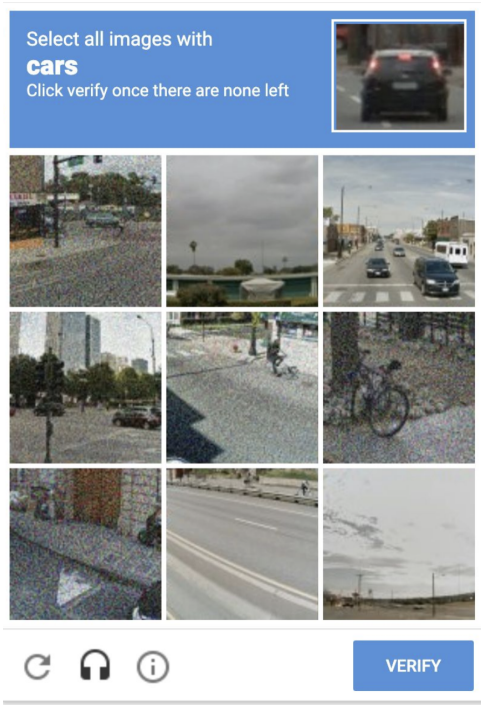


Figure 2.5 Example of a Google reCAPTCHA.

extracted from clickstreams—*i.e.*, sequences of HTTP requests made by a user. They also proposed an unsupervised approach that requires less labeled data than the state-of-the-art. Nevertheless, this problem is different from crawler detection since social media users are logged in. Moreover, contrary to crawlers that are not human by definition, fake accounts may be operated by humans.

Detecting virtual machines. Another strategy to detect crawlers is to look for features correlated with crawling. For example, the use of an IP address blacklist [95] to detect if a user is connected from a public proxy, or if the requests are coming from a cloud provider, such as AWS or Azure. Ho *et al.* [70] proposed several red pills that rely on the time to execute different operations such as spawning web workers, to detect browsers running in virtual machines, which may also indicate the presence of a non-human user.

2.5 Conclusion

This thesis provides three main contributions to better understand fingerprinting in a context of tracking and security. My first contribution aims at providing a better

measurement of the stability of browser fingerprints, as well as evaluating how long users can be tracked using only their browser fingerprints.

Through my second contribution, I evaluate the privacy implications of using fingerprinting countermeasures. I show that even countermeasures that claim to generate consistent fingerprints can be detected, which can harm user privacy. Based on these findings, I provide recommendations to build more effective fingerprinting countermeasures.

While several studies measured the use of fingerprinting on the web as a tracking mechanism, none of them focused on its use for bot detection. Through my third and last contribution, I aim to fill this gap by measuring the fraction of websites that use fingerprinting for crawler detection, describing the techniques they use and evaluating their resilience against adversarial attackers.

2.5.1 FP-Stalker: Tracking Browser Fingerprint Evolutions

Through my first contribution, I aim to provide better measurements of browser fingerprints stability and how long can browsers be tracked using only their fingerprint. In 2010, Eckersley [3] measured fingerprint stability and how it could be used for tracking using 470,161 fingerprints collected on the Panopticlick website. Eckersley acknowledged two main biases in the dataset collected:

- First, users coming on the Panopticlick are aware their fingerprint is collected and may be challenged to change it purposefully, *e.g.* by removing a plugin or adding a language, to see how it impacts their uniqueness;
- Secondly, the Panoticlick website is likely to be visited by users with privacy countermeasures, in particular fingerprinting countermeasures, such as user agent spoofers, that artificially modify fingerprints.

These biases can impact the accuracy of the fingerprint stability and the tracking measurements. Moreover, since their study was conducted, new fingerprinting attributes that rely on HTML5 features have been proposed. Thus, I conduct a large scale study that aims to minimize these measurement biases by leveraging browser extensions over long period of times, and that analyze the most recent fingerprinting attributes. In particular, I study the stability of new techniques with high entropy such as canvas fingerprinting that did not exist when Eckersley's study was conducted. Moreover, no study measured how the uniqueness and the stability of browser fingerprints translate

in terms of tracking duration. Thus, I address this issue by measuring how long can browsers be tracked using only their fingerprints.

FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. My second contribution aims at better understanding the privacy implications of using browser fingerprinting countermeasures. Nikiforakis *et al.* [4] and Acar *et al.* [10] showed that countermeasures, such as user agent spoofers can be detected because they generate inconsistent fingerprints, which can harm user privacy. Nevertheless, since these two studies have been published, several countermeasures that claim to generate consistent fingerprints have been developed [9, 45, 66] but the consistency of the fingerprints they generate has not been properly evaluated. Moreover, during the crawls conducted in this thesis, I encountered fingerprinting scripts from a commercial tracking company, Augur [96], and discovered the presence of multiple tests that aim at detecting the presence of fingerprinting countermeasures. Thus, this finding shows the need to properly evaluate fingerprinting countermeasures since commercial fingerprinters may try to detect their presence.

To better evaluate these countermeasures, I propose to extend the notion of fingerprint inconsistency introduced by Nikiforakis *et al.* to more advanced countermeasures, such as canvas poisoners. I design a test suite that leverages these inconsistencies and evaluate how existing countermeasures can be detected, and what is the impact on user privacy. I also evaluate how existing fingerprinting solutions, either open-source or commercial, perform to detect these countermeasures. Finally, I address a lack of recommendations for countermeasures developers by providing good practices when developing fingerprinting countermeasures so that they do not end up being counterproductive.

FP-Crawlers: Evaluating the Resilience of Browser Fingerprinting to Block Adversarial Crawlers. In 2013, Acar *et al.* [10] crawled 100,000 websites and showed that a significant fraction of fingerprinting scripts was used for security purposes, in particular, crawler and bot detection. While several studies [10, 4, 6] focus on the techniques used by commercial fingerprinters for tracking, none investigates how fingerprinting can be used to detect bots.

Different non-fingerprinting based techniques have been proposed for crawler detection. One of the most popular, CAPTCHA [84], relies on Turing tests, such as image or audio recognition, to detect if a user is human or not. However, with recent progress in automatic image and audio recognition, as well as different services that propose to solve CAPTCHAs for money [87, 85, 86], CAPTCHAs can easily be bypassed [88].

Other techniques rely on the analysis of the sequence of requests sent by the web client. Traditional rate-limiting techniques [97–99, 94] analyze features, such as the number of requests or the number of pages loaded, to classify the web client as a human or a malicious crawler. More advanced techniques [76] extract features from time series representing the data sent to a website in order to identify if the traffic originates from a human user or a crawler.

However, I argue that browser fingerprinting can be used to address some of the weaknesses of state-of-the-art crawler detection techniques, such as:

- Contrary to CAPTCHAs, browser fingerprinting does not require any user interaction;
- Contrary to methods based on HTTP requests or time series analysis [76], fingerprinting requires a single request to decide whether or not a client is a crawler.

In this thesis, I plan to improve the understanding concerning the use of browser fingerprinting as an additional layer for crawler detection. First, I measure its use among popular websites of the top Alexa 10K and analyze the different techniques used by commercial fingerprinters to detect crawlers. I explain how fingerprinting techniques for crawler detection differ from techniques used for tracking. Finally, one of the main challenges in using fingerprinting in a security context lies in the fact that fingerprints are collected in the client-side, and therefore, can be modified by an attacker. Thus, I evaluate the resilience of current fingerprinting techniques against an adversarial crawler developer.

Part III

Contributions

Chapter 3

Fp-Stalker: Tracking Browser Fingerprint Evolutions

The majority of the browser fingerprinting literature has focused on fingerprint uniqueness, a condition required for tracking. However, fingerprint uniqueness, by itself, is insufficient for tracking if fingerprints change frequently. Indeed, one needs to keep track of these evolutions to link them to previous fingerprints belonging to the same browser. In this chapter, I aim at measuring how vulnerable are browsers against fingerprinting-based tracking by measuring how long they be tracked using solely their fingerprint. This chapter extends my Fp-Stalker paper published at S&P 18 [16]. I conduct an analysis on a dataset that contains more than 25,000 new fingerprints, which results in a dataset of 122,350 fingerprints from 2,346 browsers. First, in Section 3.1 I describe our dataset and highlight the limits of browser fingerprint uniqueness for tracking purposes by showing that fingerprints change frequently (around 50% of browser instances changed their fingerprints in less than 5 days, 70% in less than 10 days). Then, in Section 3.2 I propose Fp-Stalker, a novel algorithm that detects if two fingerprints originate from the same browser instance, which refers to an installation of a browser on a device. However, browser instances change overtime, *e.g.* they are updated or configured differently, causing their fingerprints to evolve. Therefore, I introduce two variants of Fp-Stalker: a rule-based and a hybrid variant, which leverages rules and a random forest. In Section 3.3, I evaluate my approach using 122,350 browser fingerprints originating from 2,346 browser instances, which we collected over two years. The fingerprints were collected using two browser extensions advertised on the AmIUnique website, one for Firefox and the other for Chrome. I compare both variants of Fp-Stalker and an

implementation of the algorithm proposed by Eckersley. In my experiments, I evaluate Fp-Stalker's ability to correctly link browser fingerprints originating from the same browser instance, as well as its ability to detect fingerprints that originate from unknown browser instances. I show that Fp-Stalker can link, on average, fingerprints from a given browser instance for more than 49.3 days, which represents an improvement of 34 days compared to the closest algorithm from the literature. I also discuss the impact of my findings, in particular when browser fingerprinting is used in addition to other stateful tracking techniques such as cookies or E-tag. Finally, I conclude this chapter in Section 3.4.

3.1 Browser Fingerprint Evolutions

This paper focuses on the *linkability of browser fingerprint evolutions over time*. Using fingerprinting as a long-term tracking technique requires not only obtaining unique browser fingerprints, but also linking fingerprints that originate from the same browser instance. Most of the literature has focused on studying or increasing fingerprint uniqueness [1, 3, 27]. While uniqueness is a critical property of fingerprints, it is also critical to understand fingerprint evolution to build an effective tracking technique. Our study provides more insights into browser fingerprint evolution in order to demonstrate the effectiveness of such a tracking technique.

Input dataset The raw input dataset we collected contains 199,909 fingerprints obtained from 8,898 different browser instances. All browser fingerprints were obtained from AmIUnique extensions for Chrome and Firefox installed from July 2015 to early October 2017 by 8,898 participants in this study. The extensions load a page in the background that fingerprints the browser. Compared to a fingerprinting website, the only additional information we collect is a unique identifier we generate per browser instance when the extension is installed. This serves to establish the ground truth. Moreover, we pre-process the raw dataset by applying the following rules:

1. We remove browser instances with less than 7 browser fingerprints. This is because to study the ability to track browsers, we need browser instances that have been fingerprinted multiple times;

2. We discard browser instances with inconsistent fingerprints due to the use of countermeasures that artificially alter the fingerprints. To know if a user installed

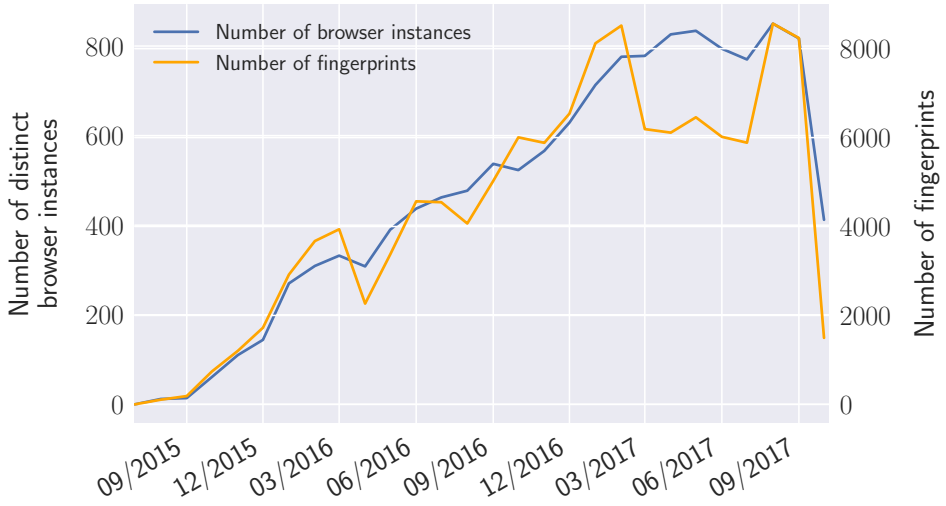


Figure 3.1 Number of fingerprints and distinct browser instances per month

such a countermeasure, we check if the browser or OS changes and we check that the attributes are consistent among themselves. Although countermeasures exist in the wild, they are used by a minority of users and, we argue, should be treated by a separate specialized anti-spoofing algorithm. We leave this task for future work.

After applying these rules, we obtain a final dataset of 122,350 fingerprints from 2,346 browser instances. All following graphs and statistics are based on this final dataset. Figure 3.1 presents the number of fingerprints and distinct browser instances per month over the two year period. The decrease in October 2017 is caused by the fact that we collected fingerprints until October 6th.

Most users heard of our extensions through posts published on popular tech websites, such as Reddit, Hackernews or Slashdot. Users install the extension to visualize the evolution of their browser fingerprints over a long period of time, and also to help researchers understand browser fingerprinting in order to design better countermeasures. We explicitly state the purpose of the extension and the fact it collects their browser fingerprints. Moreover, we received an approval from the *Institutional Review Board* (IRB) of our research center for the collection as well as the storage of these browser fingerprints. As a ground truth, the extension generates a unique identifier per browser instance. The identifier is attached to all fingerprints, which are automatically sent every 4 hours. In this study, the browser fingerprints we consider are composed of the standard attributes described in Table 3.1.

Table 3.1 An example of a browser fingerprint collect by the AmIUnique extension.

Attribute	Source	Value Examples
`Accept`	HTTP header	text/html,application/xhtml+xml,application /xml;q=0.9,image/webp,*/*;q=0.8
`Connection`	HTTP header	close
`Encoding`	HTTP header	gzip, deflate, sdch, br
`Headers`	HTTP header	Connection Accept X-Real-IP DNT Cookie Accept-Language Accept-Encoding User-Agent Host
`Languages`	HTTP header	en-US,en;q=0.8,es;q=0.6
`User-agent`	HTTP header	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36
`Canvas`	JavaScript	Cwm fjordbank glyphs vext quiz, 😃
`Cookies`	JavaScript	yes
`Do not track`	JavaScript	yes
`Local storage`	JavaScript	no
`Platform`	JavaScript	MacIntel
`Plugins`	JavaScript	Plugin 0: Chrome PDF Viewer; ; mhiehjai. Plugin 1: Chrome PDF Viewer; Portable Document Format; internal-pdf-viewer. Plugin 2: Native Client; ; internal-nacl-plugin.
`Resolution`	JavaScript	2560x1440x24
`Timezone`	JavaScript	-180
`WebGL`	Javascript	NVIDIA GeForce GTX 750 Series; Microsoft .
`Fonts`	Flash	List of fonts installed on the device

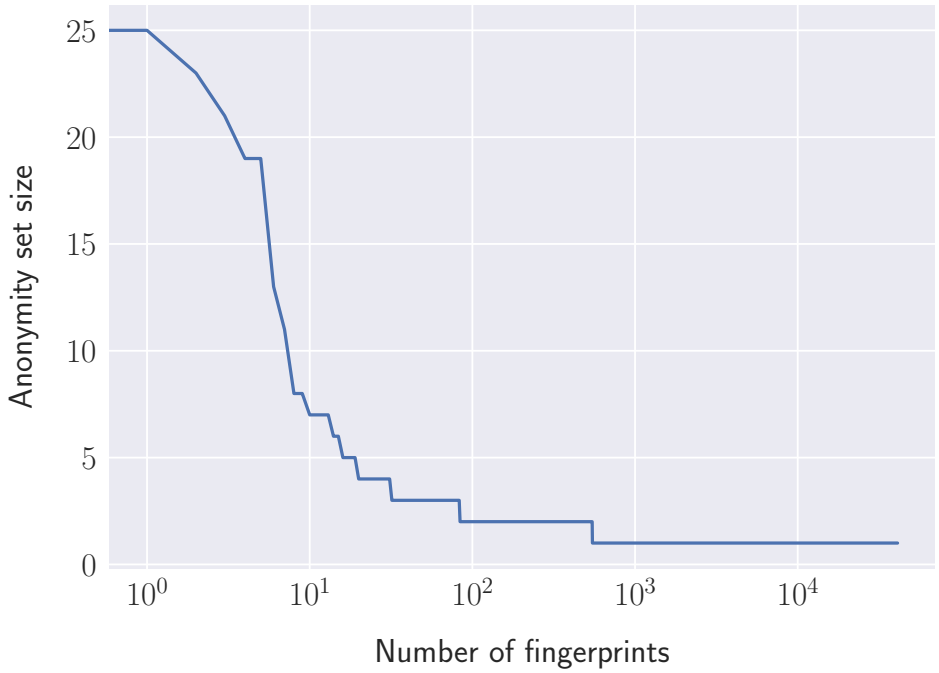


Figure 3.2 Browser fingerprint anonymity set sizes

Figure 3.2 illustrates the anonymity set sizes against the number of participants involved in this study. The long tail reflects that 95 % of the browser fingerprints are unique among all the participants and belong to a single browser instance, while only 20 browser fingerprints are shared by more than 5 browser instances.

Evolution triggers Browser fingerprints naturally evolve for several reasons. We identified the following categories of changes:

Automatic evolutions happen automatically and without direct user intervention. This is mostly due to automatic software upgrades, such as the upgrade of a browser or a plugin that may impact the `user agent` or the list of `plugins`;

Context-dependent evolutions being caused by changes in the user's context. Some attributes, such as `resolution` or `timezone`, are indirectly impacted by a contextual change, such as connecting a computer to an external screen or traveling to a different timezone; and

User-triggered evolutions that require an action from the user. They concern configuration-specific attributes, such as `cookies`, `do not track` or `local storage`.

To know how long attributes remain constant and if their stability depends on the browser instance, we compute the average time, per browser instance, that each attribute does

Table 3.2 Durations the attributes remained constant for the median, the 90^{th} and the 95^{th} percentiles.

		Percentile (days)		
Attribute	**Trigger**	**50th**	**90th**	**95th**
`Resolution`	Context	Never	3.0	1.8
`User agent`	Automatic	31.9	11.7	7.4
`Plugins`	Automatic/User	42.6	12.9	8.8
`Fonts`	Automatic	Never	15.8	6.4
`Headers`	Automatic	371.2	33.0	13.9
`Canvas`	Automatic	306.9	36.5	17.8
`Major browser version`	Automatic	49.8	30.8	20.0
`Timezone`	Context	291.8	58.6	30.1
`Renderer`	Automatic	Never	85.2	33.7
`Vendor`	Automatic	Never	146.1	56.5
`Language`	User	Never	155.6	55.6
`Dnt`	User	Never	203.7	58.6
`Encoding`	Automatic	Never	124.1	70.4
`Accept`	Automatic	Never	194.6	128.5
`Local storage`	User	Never	Never	Never
`Platform`	Automatic	Never	Never	Never
`Cookies`	User	Never	Never	Never

not change. Table 3.2 presents the median, the 90^{th} and 95^{th} percentiles of the duration each attribute remains constant, on average, in browser instances. In particular, we observe that the `User agent` is rather unstable in most browser instances as its value is systematically impacted by software updates. In comparison, attributes such as `cookies`, `local storage` and `do not track` rarely change if ever. Moreover, we observe that attributes evolve at different rates depending on the browser instance. For example, `canvas` remains stable for 306.9 days in 50% of the browser instances, whereas it changes every 36.5 days for 10% of them. The same phenomena can be observed for the `screen resolution` where more than 50% of the browser instances never see a change, while 10% change every 3 days on average. This is likely explained by laptops that are connected regularly to external monitors. More generally this points to some browser instances being quite stable, and thus, more trackable, while others are not.

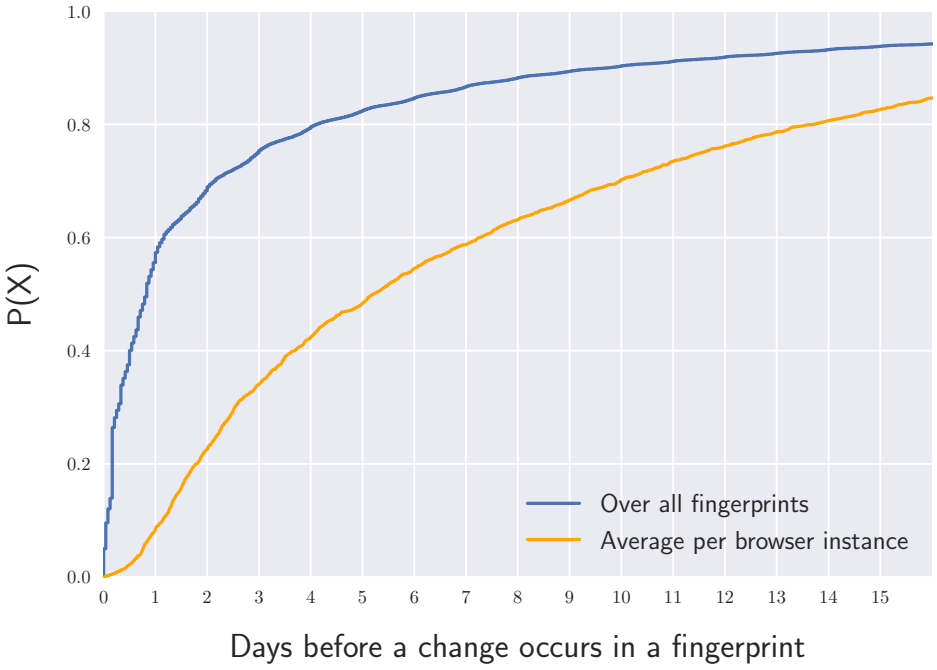


Figure 3.3 CDF of the elapsed time before a fingerprint evolution for all the fingerprints, and averaged per browser instance.

Evolution frequency Another key indicator to observe is the elapsed time (E_t) before a change occurs in a browser fingerprint. Figure 3.3 depicts the cumulative distribution function of E_t for all fingerprints (blue), or averaged per browser instance (orange). After one day, at least one transition occurs in 57.35 % of the observed fingerprints. The 90^{th} percentile is observed after 9.8 days and the 95^{th} percentile after 18.08 days. This means the probability that at least one transition occurs in 9.8 days is 0.9 (blue). It is important to point out that changes occur more or less frequently depending on the browser instance (orange). While some browser instances change often (22.6% change in less than two days) others, on the contrary, are much more stable (29.8% have no changes after 10 days). In this context, keeping pace with the frequency of change is likely a challenge for browser fingerprint linking algorithms and, to the best of our knowledge, has not been explored in the state of the art.

Evolution rules While it is difficult to anticipate browser fingerprint evolutions, we can observe how individual attributes evolve. In particular, evolutions of the `User agent` attribute are often tied to browser upgrades, while evolutions of the `Plugins` attribute refers to the addition, deletion or upgrade of a plugin (upgrades change its version). Nevertheless, not all attribute changes can be explained in this manner, some values are

difficult to anticipate. For example, the value of the `canvas` attribute is the result of an image rendered by the browser instance and depends on many different software and hardware layers. The same applies, although to a lesser extent, to screen `resolution`, which can take unexpected values depending on the connected screen. Based on these observations, the accuracy of linking browser fingerprint evolutions depends on the inference of such evolution rules. The following section introduces the evolution rules we first identified empirically, and then learned automatically, to achieve an efficient algorithm to track browser fingerprints over time.

3.2 Linking Browser Fingerprints

Fp-Stalker's goal is to determine if a browser fingerprint comes from a known browser instance—*i.e.*, it is an evolution—or if it should be considered as from a new browser instance. Because fingerprints change frequently, and for different reasons (see section 3.1), a simple direct equality comparison is not enough to track browsers over long periods of time.

In Fp-Stalker, we have implemented two variant algorithms with the purpose of linking browser fingerprints, as depicted in Figure 3.4. The first variant is a rule-based algorithm that uses a static ruleset, and the second variant is an hybrid algorithm that combines both rules and machine learning. We explain the details and the tradeoffs of both algorithms in this section. Our results show that the rule-based algorithm is faster but the hybrid algorithm is more precise while still maintaining acceptable execution times. We have also implemented a fully random forest-based algorithm, but the small increase in precision did not outweigh the large execution penalty, so we do not present it further in this paper.

3.2.1 Browser fingerprint linking

When collecting browser fingerprints, it is possible that a fingerprint comes from a previous visitor—*i.e.*, a known browser instance—or from a new visitor—*i.e.*, an unknown browser instance. The objective of fingerprint linking is to match fingerprints to their browser instance and follow the browser instance as long as possible by linking all of its fingerprint evolutions. In the case of a match, linked browser fingerprints are given the same identifier, which means the linking algorithm considers they originate from the same

browser instance. If the browser fingerprint cannot be linked, the algorithm assigns a new identifier to the fingerprint.

More formally, given a set of known browser fingerprints F, each $f \in F$ has an identifier $f.id$ that links to the browser instance it belongs to. Given an unknown fingerprint $f_u \notin F$ for whom we ignore the real id, a linking algorithm returns the browser instance identifier $f_k.id$ of the fingerprint f_k that maximizes the probability that f_k and f_u belong to the same browser instance. This computation can be done either by applying rules, or by training an algorithm to predict this probability. If no known fingerprint can be found, it assigns a new id to f_u. For optimization purposes, we only hold and compare the last ν fingerprints of each browser instance b_i in F. The reason is because if we linked, for example, 3 browser fingerprints f_A, f_B and f_C to a browser instance b_i then, when trying to link an unknown fingerprint f_u, it is rarely useful to compare f_u to the oldest browser fingerprints of b_i. That is, newer fingerprints are more likely to produce a match, hence we avoid comparing old fingerprints in order to improve execution times. In our case we set the value of ν to 2.

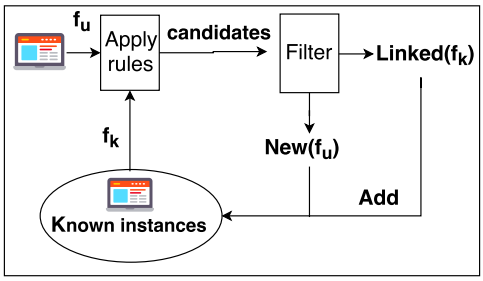
3.2.2 Rule-based Linking Algorithm

The first variant of Fp-Stalker is a rule-based algorithm that uses static rules obtained from statistical analyses performed in Section 3.1. The algorithm relies on rules designed from attribute stability presented in Table 3.2 to determine if an unknown fingerprint f_u belongs to the same browser instance as a known fingerprint f_k. We also define rules based on constraints that we would not expect to be violated, such as, a browser's family should be constant (*e.g.*, the same browser instance cannot be Firefox one moment and Chrome at a later time), the Operating System is constant, and the browser version is either constant or increases over time. The full list of rules are as follow:

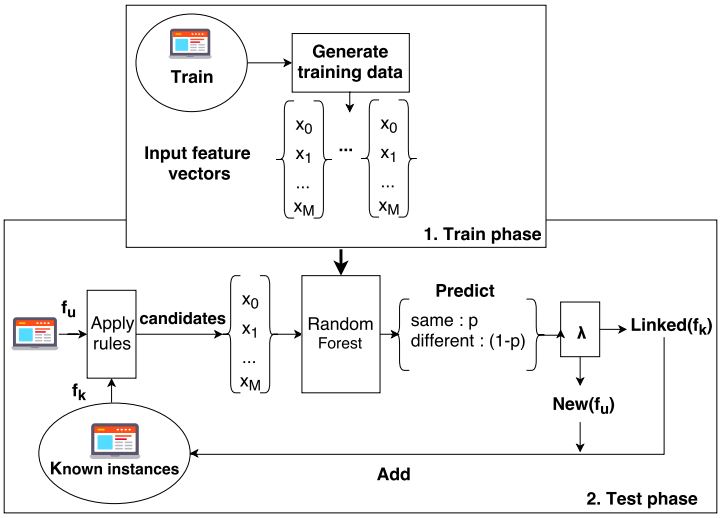
1. The `OS`, `platform` and `browser family` must be identical for any given browser instance. Even if this may not always be true (*e.g.* when a user updates from Windows 8 to 10), we consider it reasonable for our algorithm to lose track of a browser when such a large change occurs since it is not frequent;

2. The `browser version` remains constant or increases over time. This would not be true in the case of a downgrade, but this is also, not a common event;

3. Due to the results from our statistical analyses, we have defined a set of attributes that must not differ between two fingerprints from the same browser instance. We



(a) Rule-based variant of Fp-Stalker. Uses a set of static rules to determine if fingerprints should be linked to the same browser instance or not.



(b) Hybrid variant of Fp-Stalker. The training phase is used to learn the probability that two fingerprints belong to the same browser instance, and the testing phase uses the random forest-based algorithm to link fingerprints.

Figure 3.4 Fp-Stalker: Overview of both algorithm variants. The rule-based algorithm is simpler and faster but the hybrid algorithm leads to better fingerprint linking.

consider that `local storage`, `Dnt`, `cookies` and `canvas` should be constant for any given browser instance. As observed in Table 3.2, these attributes do not change often, if at all, for a given browser instance. In the case of `canvas`, even if it seldomly changes for most users (see Table 3.2, the changes are unpredictable making them hard to model. Since `canvas` are quite unique among browser instances [27], and do not change too frequently, it is still interesting to consider that it must remain identical between two fingerprints of the same browser instance;

4. We impose a constraint on fonts: if both fingerprints have Flash activated—*i.e.* we have a list of fonts available—then the fonts of f_u must either be a subset or a superset of the fonts of f_k, but not a disjoint set. That means that between two fingerprints of a browser instance, it will allow deletions or additions of fonts, but not both;

5. We define a set of attributes that are allowed to change, but only within a certain similarity. That means that their values must have a similarity ratio > 0.75, as defined in the Python library function `difflib.SequenceMatcher().ratio`. These attributes are `user agent`, `vendor`, `renderer`, `plugins`, `language`, `accept`, `headers`. We allow at most two changes of this kind;

6. We also define a set of attributes that are allowed to change, no matter their value. This set is composed of `resolution`, `timezone` and `encoding`. However, we only allow one change at the same time among these three attributes;

7. Finally, the total number of changes from rules 5 and 6 must be less than 2.

The order in which rules are applied is important for performance purposes: we ordered them from the most to least discriminating. The first rules discard many candidates, reducing the total number of comparisons. In order to link f_u to a fingerprint f_k, we apply the rules to each known fingerprint taken from F. As soon as a rule is not matched, the known fingerprint is discarded and we move onto the next. If a fingerprint matches all the rules, then it is added to a list of potential candidates, *candidates*. Moreover, in case fingerprints f_k and f_u are identical, we add it to the list of exact matching candidates, *exact*. Once the rule verification process is completed, we look at the two lists of candidates. If *exact* is not empty, we check if there is only one candidate or if all the candidates come from the same browser instance. If it is the case, then we link f_u with this browser instance, otherwise we assign a new id to f_u. In case no exact candidate is found, we look at *candidates* and apply the same technique as for *exact*. We summarize the rule-based approach in Algorithm 1.

Algorithm 1 Rule-based matching algorithm

function FINGERPRINTMATCHING(F, f_u)
  $rules = \{rule_1, ..., rule_6\}$
  $candidates \leftarrow \emptyset$
  $exact \leftarrow \emptyset$
  for $f_k \in F$ **do**
    if VERIFYRULES($f_k, f_u, rules$) **then**
      if $nbDiff = 0$ **then**
        $exact \leftarrow exact \cup \langle f_k \rangle$
      else
        $candidates \leftarrow candidates \cup \langle f_k \rangle$
      end if
    end if
  end for
  if $|exact| > 0$ **and** SAMEIDS($exact$) **then**
    return $exact[0].id$
  else if $|candidates| > 0$ **and** SAMEIDS($candidates$) **then**
    return $candidates[0].id$
  else
    return GENERATENEWID()
  end if
end function

SAMEIDS is a function that, given a list of candidates, returns true if all of them share the same id, else false.

On a side note, we established the rules using a simple univariate statistical analysis to study attribute stability (see Table 3.2), as well as some objective (*e.g.,* rule 1) and other subjective (*e.g.,* rule 4) decisions. Due to the difficulty in making complex yet effective rules, the next subsection presents the use of machine learning to craft a more effective algorithm.

3.2.3 Hybrid Linking Algorithm

The second variant of Fp-Stalker mixes the rule-based algorithm with machine learning to produce a hybrid algorithm. It reuses the first three rules of the previous algorithm, since we consider them as constraints that should not be violated between two fingerprints of a same browser instance. However, for the last four rules, the situation is more fuzzy. Indeed, it is not as clear when to allow attributes to be different, how many of them can be different, and with what dissimilarity. Instead of manually crafting rules for each of these attributes, we propose to use machine learning to discover them. The interest of combining both rules and machine learning approaches is that rules are faster than machine learning, but machine learning tends to be more precise. Thus, by applying the rules first, it helps keep only a subset of fingerprints on which to apply the machine learning algorithm.

3.2.3.1 Approach Description

The first step of this algorithm is to apply rules 1, 2 and 3 on f_u and all $f_k \in F$. We keep the subset of browser fingerprints f_{ksub} that verify these rules. If, during this process, we found any browser fingerprints that exactly match f_u, then we add them to *exact*. In case *exact* is not empty and all of its candidates are from the same browser instance, we stop here and link f_u with the browser instance in *exact*. Otherwise, if there are multiple exact candidates but from different browser instances, then we assign a new browser id to f_u. In the case where the set of exact candidates is empty, we continue with a second step that leverages machine learning. In this step, for each fingerprint $f_k \in f_{ksub}$, we compute the probability that f_k and f_u come from the same browser instance using a random forest model. We keep a set of fingerprint candidates whose probability is greater than a λ threshold parameter. If the set of candidates is empty, we assign a new id to f_u. Otherwise, we keep the set of candidates with the highest and second highest probabilities, c_{h1} and c_{h2}. Then, we check if c_{h1} contains only one candidate or if all of the candidates come from the same browser instance. If it is not the case, we

check that either the probability p_{h1} associated with candidates of c_{h1} is greater than the probability p_{h2} associated with candidates of $c_{h2} + diff$, or that c_{h2} and c_{h1} contains only candidates from the same browser instance. Algorithm 2 summarizes the hybrid approach.

3.2.3.2 Machine Learning

Computing the probability that two fingerprints f_u and f_k originate from the same browser instance can be modeled as a binary classification problem where the two classes to predict are `same browser instance` and `different browser instance`. We use the random forest algorithm [100] to solve this binary classification problem. A random forest is an ensemble learning method for classification that operates by constructing a multitude of decision trees at training time and outputting the class of the individual trees. In the case of Fp-Stalker, each decision tree makes a prediction and votes if the two browser fingerprints come from the same browser instance. The result of the majority vote is chosen. Our main motivation to adopt a random forest instead of other classifiers is because it provides a good tradeoff between precision and the interpretation of the model. In particular, the notion of feature importance in random forests allows Fp-Stalker to interpret the importance of each attribute in the decision process.

In summary, given two fingerprints, $f_u \notin F$ and $f_k \in F$, whose representation is reduced to a single feature vector of M features $X = \langle x_1, x_2, ..., x_M \rangle$, where the feature x_n is the comparison of the attribute n for both fingerprints (the process of transforming two fingerprints into a feature vector is presented after). Our random forest model computes the probability $P(f_u.id = f_k.id \mid (x_1, x_2, ..., x_M))$ that f_u and f_k belong to the same browser instance.

Input Feature Vector To solve the binary classification problem, we provide an input vector $X = \langle x_1, x_2, ..., x_M \rangle$ of M features to the random forest classifier. The features are mostly pairwise comparisons between the values of the attributes of both fingerprints (*e.g.*, `Canvas`, `User agent`). Most of these features are binary values (`0` or `1`) corresponding to the equality or inequality of an attribute, or similarity ratios between these attributes. We also include a `number of changes` feature that corresponds to the total number of different attributes between f_u and f_k, as well as the time difference between the two fingerprints.

Algorithm 2 Hybrid matching algorithm

function FingerprintMatching(F, f_u, λ)
  $rules = \{rule_1, rule_2, rule_3\}$
  $exact \leftarrow \emptyset$
  $F_{ksub} \leftarrow \emptyset$
  for $f_k \in F$ **do**
    if VerifyRules($f_k, f_u, rules$) **then**
      if $nbDiff = 0$ **then**
        $exact \leftarrow exact \cup \langle f_k \rangle$
      else
        $F_{ksub} \leftarrow F_{ksub} \cup \langle f_k \rangle$
      end if
    end if
  end for
  if $|exact| > 0$ **then**
    if SameIds($exact$) **then**
      return $exact[0].id$
    else
      return GenerateNewId()
    end if
  end if
  $candidates \leftarrow \emptyset$
  for $f_k \in F_{ksub}$ **do**
    $\langle x_1, x_2, ..., x_M \rangle =$ FeatureVector(f_u, f_k)
    $p \leftarrow P(f_u.id = f_k.id \mid \langle x_1, x_2, ..., x_M \rangle)$
    if $p \geq \lambda$ **then**
      $candidates \leftarrow candidates \cup \langle f_k, p \rangle$
    end if
  end for
  if $|candidates| > 0$ **then**
    $c_{h1}, p_{h1} \leftarrow$ getCandidatesRank($candidates, 1$)
    $c_{h2}, p_{h2} \leftarrow$ getCandidatesRank($candidates, 2$)
    if SameIds(c_{h1}) **and** $p_{h1} > p_{h2} + diff$ **then**
      return $candidates[0].id$
    end if
    if SameIds($c_{h1} \cup c_{h2}$) **then**
      return $candidates[0].id$
    end if
  end if
  return GenerateNewId()
end function

getCandidatesRank is a function that given a list of candidates and an rank i, returns a list of candidates with the ith greatest probability, and this probability.

In order to choose which attributes constitute the feature vector we made a feature selection. Indeed, having too many features does not necessarily ensure better results. It may lead to *overfitting*—*i.e.*, our algorithm correctly fits our training data, but does not correctly predict on the test set. Moreover, having too many features also has a negative impact on performance. For the feature selection, we started with a model using all of the attributes in a fingerprint. Then, we looked at feature importance, as defined by [101], to determine the most discriminating features. In our case, feature importance is a combination of uniqueness, stability, and predictability (the possibility to anticipate how an attribute might evolve over time). We removed all the components of our feature vector that had a negligible impact (feature importance < 0.001). Finally, we obtained a feature vector composed of the attributes presented in Table 3.3. We see that the most important feature is the number of differences between two fingerprints, and the second most discriminating attribute is the time difference between the two fingerprints compared. Even attributes with low entropy, such as the list of languages, are among the most important features. Although this may seem surprising, it can be explained by the stability of the list of languages, as shown in Table 3.2, which means that if two fingerprints have different languages, this often means that they do not belong to the same browser instance. In comparison, screen resolution also has low entropy but it changes more often than the list of languages, leading to low feature importance. This is mostly caused by the fact that since screen resolution changes frequently, having two fingerprints with a different resolution does not add a lot of information to determine whether or not they are from the same browser instance. Finally, we see a high drop in feature importance after rank 7 (from 0.018 to 0.004), which means that most of the information required for the classification is contained in the first seven features.

Training Random Forests This phase trains the random forest classifier to estimate the probability that two fingerprints belong to the same browser instance. To do so, we split the input dataset introduced in Section 3.1 chronologically into two sets: a training set and a test set. The training set is composed of the first 40 % of fingerprints in our input dataset, and the test set of the last 60 %. The random forest detects fingerprint evolutions by computing the evolutions between fingerprints as feature vectors. During the training phase, it needs to learn about correct evolutions by computing relevant feature vectors from the training set. Algorithm 3 describes this training phase, which is split into two steps.

Table 3.3 Feature importances of the random forest model calculated from the fingerprint train set.

Rank	Feature	Importance
1	`Number of changes`	0.324
2	`Time difference`	0.225
3	`User agent HTTP`	0.194
4	`Languages HTTP`	0.112
5	`Plugins`	0.094
6	`Canvas`	0.020
7	`Renderer`	0.018
8	`Resolution`	0.004
9	`Timezone`	0.002
10	`Fonts`	0.001

Algorithm 3 Compute input feature vectors for training

function BUILDTRAININGVECTORS(ID,F,δ,ν)
  $T \leftarrow \emptyset$
  for $id \in ID$ **do** ▷ Step 1
    $F_{id} \leftarrow$ BROWSERFINGERPRINTS(id, F)
    for $f_t \in F_{id}$ **do**
      $T \leftarrow T \cup$ FEATUREVECTOR(f_t, f_{t-1})
    end for
  end for
  for $f \in F$ **do** ▷ Step 2
    $f_r \leftarrow$ RANDOM(F)
    if $f.id \neq f_r.id$ **then**
      $T \leftarrow T \cup$ FEATUREVECTOR(f, f_r)
    end if
  end for
  return T
end function

In `Step 1`, for every browser instance (id) of the training set, we compare each of its fingerprints ($f_t \in$ BrowserFingerprints(id, F)) present in the training set (F) with the previous one (f_{t-1}). By doing so, Fp-Stalker captures the atomic evolutions that occur between two consecutive fingerprints from the same browser instance. We apply BuildTrainingVectors() for different collect frequencies (time difference between t and $t-1$) to teach our model to link fingerprints even when they are not equally spaced in time.

While `Step 1` teaches the random forest to identify fingerprints that belong to the same browser instance, it is also necessary to identify when they do not. `Step 2` compares fingerprints from different browser instances. Since the number of fingerprints from different browser instances is much larger than the number of fingerprints from the same browser instance, we limit the number of comparisons to one for each fingerprint. This technique is called *undersampling* [102] and it reduces overfitting by adjusting the ratio of input data labeled as *true*—*i.e.*, 2 fingerprints belong to the same browser instance—against the number of data labeled as *false*—*i.e.*, 2 fingerprints are from different browser instances. Otherwise, the algorithm would tend to simply predict *false*.

Random forest hyperparameters. Concerning the *number of trees* of the random forest, there is a tradeoff between precision and execution time. Adding trees does obtain better results but follows the law of diminishing returns and increases training and prediction times. Our goal is to balance precision and execution time. The *number of features* plays a role during the tree induction process. At each split, N_f features are randomly selected, among which the best split is chosen [103]. Usually, its default value is set to the square root of the length of the feature vector. The *diff* parameter enables the classifier to avoid selecting browser instances with very similar probabilities as the origin of the fingerprint; we would rather create a new browser instance than choose the wrong one. It is not directly related to random forest hyperparameters but rather to the specificities of our approach. In order to optimize the hyperparameters *number of trees* and *number of features*, as well as the *diff* parameter, we define several possible values for each and run a grid search to optimize the accuracy. This results in setting the hyperparameters to 10 trees and 3 features, and the *diff* value to 0.20.

After training our random forest classifier, we obtain a forest of decision trees that predict the probability that two fingerprints belong to the same browser instance. Figure 3.5 illustrates the first three levels of one of the decision trees. These levels rely on the `languages`, the `number of changes` and the `user agent` to take a decision. If an

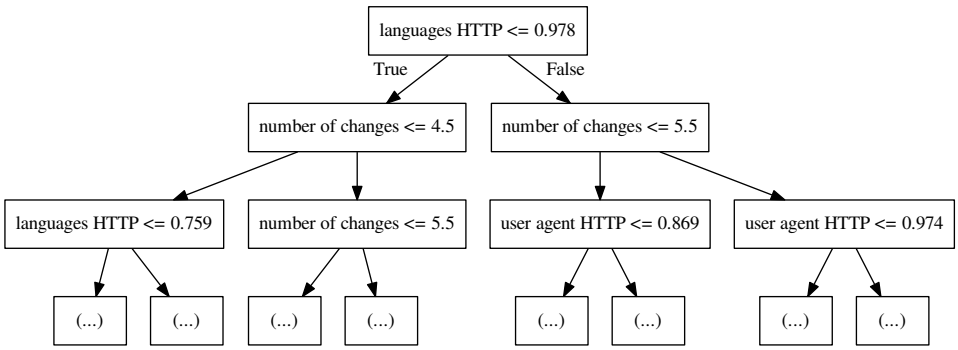


Figure 3.5 First 3 levels of a single tree classifier from our forest.

attribute has a value below its threshold, the decision path goes to the left child node, otherwise it goes to the right child node. The process is repeated until we reach a leaf of the tree. The prediction corresponds to the class (same/different browser instance) that has the most instances over all the leaf nodes.

Lambda threshold parameter For each browser fingerprint in the test set, we compare it with its previous browser fingerprint and with another random fingerprint from a different browser, and compute the probability that it belongs to the same browser instance using our random forest classifier with the parameters determined previously. Using these probabilities and the true labels, we choose the λ value that minimizes the false positive rate, while maximizing the true positive rate. However, this configuration parameter depends on the targeted application of browser fingerprinting. For instance, if browser fingerprinting is used as a second-tier security mechanism (*e.g.*, to verify the user is connecting from a known browser instance), we set λ to a high value. This makes the algorithm more conservative, reducing the risk of linking a fingerprint to an incorrect browser instance, but it also increases false negatives and results in a reduction of the duration the algorithm can effectively track a browser. On the opposite end, a low λ value will increase the false positive rate, in this case meaning it tends to link browser fingerprints together even though they present differences. Such a use case might be acceptable for constructing ad profiles, because larger profiles are arguably more useful even if sometimes contaminated with someone else's information. By applying this approach, we obtained a λ threshold equal to 0.994.

3.3 Empirical Evaluation of Fp-Stalker

This section assesses Fp-Stalker's capacity to *i)* correctly link fingerprints from the same browser instance, and to *ii)* correctly predict when a fingerprint belongs to a browser instance that has never been seen before. We show that both variants of Fp-Stalker are effective in linking fingerprints and in distinguishing fingerprints from new browser instances. However, the rule-based variant is faster while the hybrid variant is more precise. Finally, we discuss the impact of the collect frequency on fingerprinting effectiveness, and we evaluate the execution times of both variants of Fp-Stalker.

Figure 3.6 illustrates the linking and evaluation process. Our database contains perfect tracking chains because of the unique identifiers our extensions use to identify browser instances. From there, we sample the database using different collection frequencies and generate a test set that removes the identifiers, resulting in a mix of fingerprints from different browsers. The resulting test set is then run through Fp-Stalker to reconstruct the best browser instance chains as possible.

3.3.1 Key Performance Metrics

To evaluate the performance of our algorithms and measure how vulnerable users are to browser fingerprint tracking, we consider several metrics that represent the capacity to keep track of browser instances over time and to detect new browser instances. This section presents these evaluation metrics, as well as the related vocabulary. Figure 3.6 illustrates the different metrics with a scenario.

A *tracking chain* is a list of fingerprints that have been linked—*i.e.*, fingerprints for which the linking algorithm assigned the same identifier. A chain may be composed of one or more fingerprints. In case of a perfect linking algorithm, each browser instance would have a unique tracking chain—*i.e.*, all of its fingerprints are grouped together and are not mixed with fingerprints from any other browser instances. However, in reality, fingerprinting is a statistical attack and mistakes may occur during the linking process, which means that:

1. Fingerprints from different browser instances may be included in the same tracking chain,
2. Fingerprints from a given browser instance may be split into different tracking chains.

The lower part of Figure 3.6 shows examples of these mistakes. *Chain 1* has an incorrect fingerprint `fpB1` from `Browser B`, and `chain 3` and `chain 4` contain fingerprints from `browser C` that have not correctly been linked—*i.e.*, `fpC3` and `fpC4` were not linked leading to a split).

We present the *tracking duration* metric to evaluate the capacity of an algorithm to track browser instances over time. We define *tracking duration* as the period of time a linking algorithm matches the fingerprints of a browser instance within a single tracking chain. More specifically, the tracking duration for a browser $\texttt{b}_i$ in a chain $\texttt{chain}_k$ is defined as $CollectFrequency \times (\#b_i \in chain_k - 1)$. We subtract one because we consider a browser instance to have been tracked, by definition, from the second linked fingerprint onwards.

The *average tracking duration* for a browser instance b_i is the arithmetic mean of its tracking duration across all the tracking chains the instance is present in. For example, in Figure 3.6, the tracking duration of `browser B` in `chain 1` is $0 \times CollectFrequency$, and the tracking duration in `chain 2` is $1 \times CollectFrequency$, thus the average tracking duration is $0.5 \times CollectFrequency$. In the same manner, the *average tracking duration* of *browser C* is $1.5 \times CollectFrequency$.

The *maximum tracking duration* for a browser instance b_i is defined as the maximum tracking duration across all of the tracking chains the browser instance is present in. In the case of `browser C`, the maximum tracking duration occurred in `chain 3` and is equal to $2 \times CollectFrequency$.

The *Number of assigned ids* represents the number of different identifiers that have been assigned to a browser instance by the linking algorithm. It can be seen as the number of tracking chains in which a browser instance is present. For each browser instance, a perfect linking algorithm would group all of the browser's fingerprints into a single chain. Hence, each browser instance would have a *number of assigned ids* of 1. Figure 3.6 shows an imperfect case where `browser C` has been assigned 2 different ids (`chain 3` and `chain 4`).

The *ownership ratio* reflects the capacity of an algorithm to not link fingerprints from different browser instances. The `owner` of a tracking chain $\texttt{chain}_k$ is defined as the browser instance $\texttt{b}_i$ that has the most fingerprints in the chain. Thus, we define *ownership ratio* as the number of fingerprints that belong to the `owner` of the chain divided by the length of the chain. For example, in `chain 1`, `browser A` owns the chain with an `ownership ratio` of $\frac{4}{5}$ because it has 4 out of 5 of the fingerprints. In practice, an *ownership ratio*

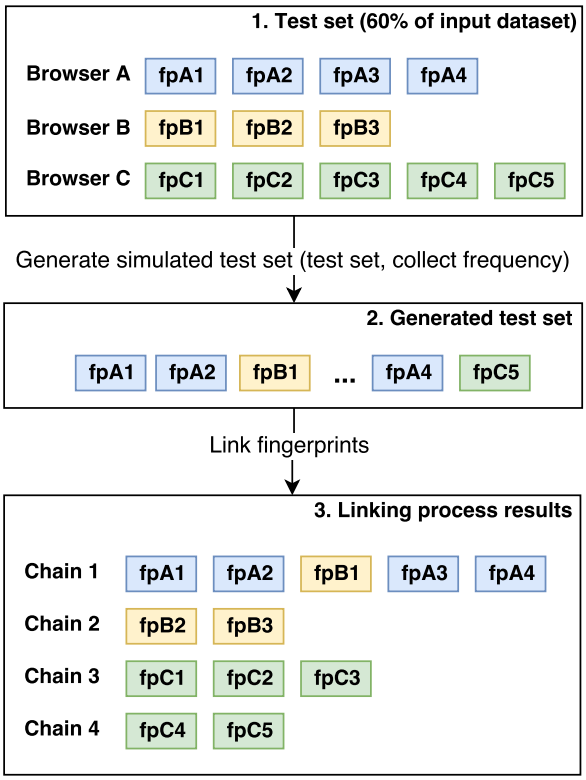


Figure 3.6 Overview of our evaluation process that allows testing the algorithms using different simulated collection frequencies.

close to 1 means that a tracking profile is not polluted with information from different browser instances.

3.3.2 Comparison With Panopticlick's Linking Algorithm

We compare Fp-Stalker to the algorithm proposed by Eckersley [3] in the context of the Panopticlick project. To the best of our knowledge, there are no other algorithms to compare to. Although Eckersley's algorithm has been characterized as "naive" by its author, we use it as a baseline to compare our approach. The Panopticlick algorithm is summarized in Algorithm 4. It uses the following 8 attributes: `User agent`, `accept`, `cookies enabled`, `screen resolution`, `timezone`, `plugins`, `fonts` and `local storage`. Given an unknown fingerprint f_u, Panopticlick tries to match it to a previous fingerprint of the same browser instance if a sufficiently similar one exists—*i.e.*, no more than one attribute changed. Otherwise, if it found no similar fingerprints, or too many similar fingerprints that belong to different browser instances, it assigns a new

Algorithm 4 Eckersley fingerprint matching algorithm [3]

$ALLOWED = \{cookies, resolution, timezone, local\}$
function FingerprintMatching(F, f_u)
  $candidates \leftarrow \emptyset$
  for $f_k \in F$ **do**
    $changes \leftarrow$ Diff(f_u, f_k)
    if $|changes| = 1$ **then**
      $candidates \leftarrow candidates \cup \langle f_k, changes \rangle$
    end if
  end for
  if $|candidates| = 1$ **then**
    $\langle f_k, a \rangle \leftarrow candidates[0]$
    if $a \in ALLOWED$ **then**
      return f_k
    else if MatchRatio($f_u(a), f_k(a)$) > 0.85 **then**
      return f_k
    else
      return $NULL$
    end if
  end if
end function

MatchRatio refers to the Python standard library function `difflib.SequenceMatcher().ratio()` for estimating the similarity of strings.

id. Moreover, although at most one change is allowed, this change can only occur among the following attributes: `cookies`, `resolution`, `timezone` and `local storage`.

3.3.3 Dataset Generation Using Fingerprint Collect Frequency

To evaluate the effectiveness of Fp-Stalker we start from our test set of 59,159 fingerprints collected from 1,395 browser instances(60% of our input dataset, see Section 3.2.3.2). However, we do not directly use this set. Instead, by sampling the test set, we generate new datasets using a configurable collect frequency. Because our input dataset is fine-grained, it allows us to simulate the impact fingerprinting frequency has on tracking. The intuition being that if a browser is fingerprinted less often, it becomes harder to track.

To generate a dataset for a given collect frequency, we start from the test set of 59,159 fingerprints, and, for each browser instance, we look at the collection date of its first fingerprint. Then, we iterate in time with a step of $collect\_frequency$ days and recover the browser instance's fingerprint at time $t + collect\_frequency$. It may be the same fingerprint as the previous collect or a new one. We do this until we reach the last fingerprint collected for that browser id. This allows us to record a sequence of fingerprints that correspond to the sequence a fingerprinter would obtain if the browser instance was fingerprinted at a frequency of $collect\_frequency$ days. The interest of sampling is that it is more realistic than using all of the fingerprints from our database since they are very fine-grained. Indeed, the extension is capable of catching even short-lived changes in the fingerprint (*e.g.*, connecting an external monitor), which is not always possible in the wild. Finally, it allows us to investigate how fingerprint collection frequency impacts browser tracking. Figure 3.7 provides an example of the process to generate a dataset with a $collect\_frequency$ of two days. Table 3.4 presents, for each simulated collect frequency, the number of fingerprints in the generated test sets.

The browser fingerprints in a generated test set are ordered chronologically. At the beginning of our experiment, the set of known fingerprints (F) is empty. At each iteration, Fp-Stalker tries to link an unknown fingerprint f_u with one of the fingerprints in F. If it can be linked to a fingerprint f_k, then Fp-Stalker assigns the id $f_k.id$ to f_u, otherwise it assigns a new id. In both cases, f_u is added to F. The chronological order of the fingerprints implies that at time t, a browser fingerprint can only be linked with a former fingerprint collected at a time $t' < t$. This approach ensures a more realistic

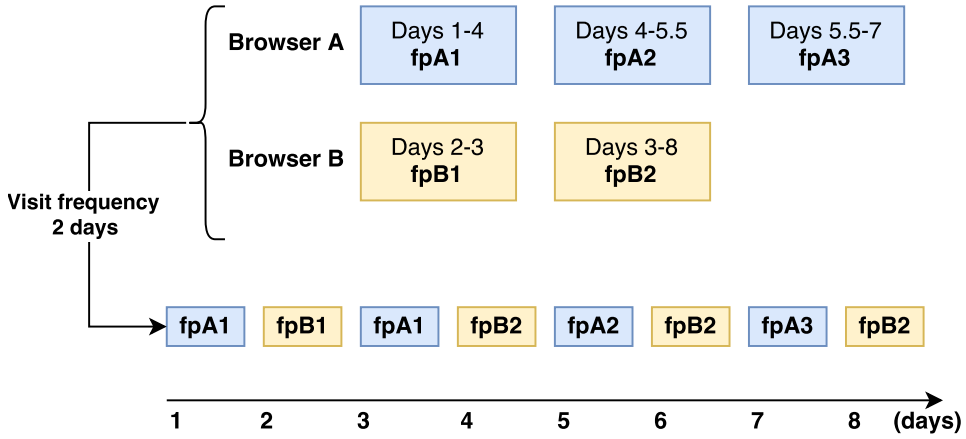


Figure 3.7 Example of the process to generate a simulated test set. The dataset contains fingerprints collected from browser's `A` and `B`, which we sample at a *collect_frequency* of 2 days to obtain a dataset that allows us to test the impact of *collect_frequency* on fingerprint tracking.

scenario, similar to online fingerprint tracking approaches, than if we allowed fingerprints from the past to be linked with fingerprints collected in the future.

3.3.4 Tracking Duration

Figure 3.8 plots the average *tracking duration* against the collect frequency for the three algorithms. On average, browser instances from the test set were present for 129.4 days, which corresponds to the maximum value our linking algorithm could potentially achieve. We see that the hybrid variant of Fp-Stalker is able to keep track of browser instances for a longer period of time than the two other algorithms. In the case where a browser gets fingerprinted every three days, Fp-Stalker can track it for 50.8 days, on average. More generally, the hybrid variant of Fp-Stalker has an average tracking duration of about 9 days more than the rule-based variant and 34 days more than the Panopticlick algorithm.

Figure 3.9 presents the average *maximum tracking duration* against the collect frequency for the three algorithms. We see that the hybrid algorithm still outperforms the two other algorithms because the it constructs longer tracking chains with less mistakes. On average, the maximum average tracking duration for Fp-Stalker's hybrid version is in the order of 82 days, meaning that at most users were generally tracked for this duration.

Figure 3.10 shows the *number of ids* they assigned, on average, for each browser instance. We see that Panopticlick's algorithm often assigns new browser ids, which is caused

Table 3.4 Number of fingerprints per generated test set after simulating different collect frequencies.

Collect frequency (days)	Number of fingerprints
1	227,706
2	114,288
3	76,480
4	57,569
5	46,247
6	38,681
7	33,285
8	29,219
10	23,560
15	16,003
20	12,231

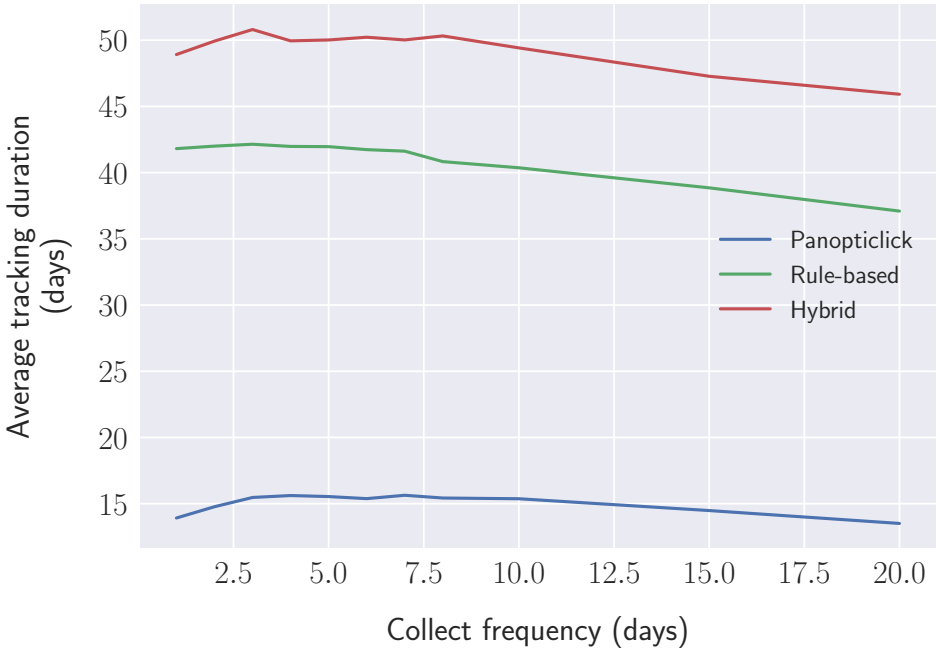


Figure 3.8 Average *tracking duration* against simulated collect frequency for the three algorithms

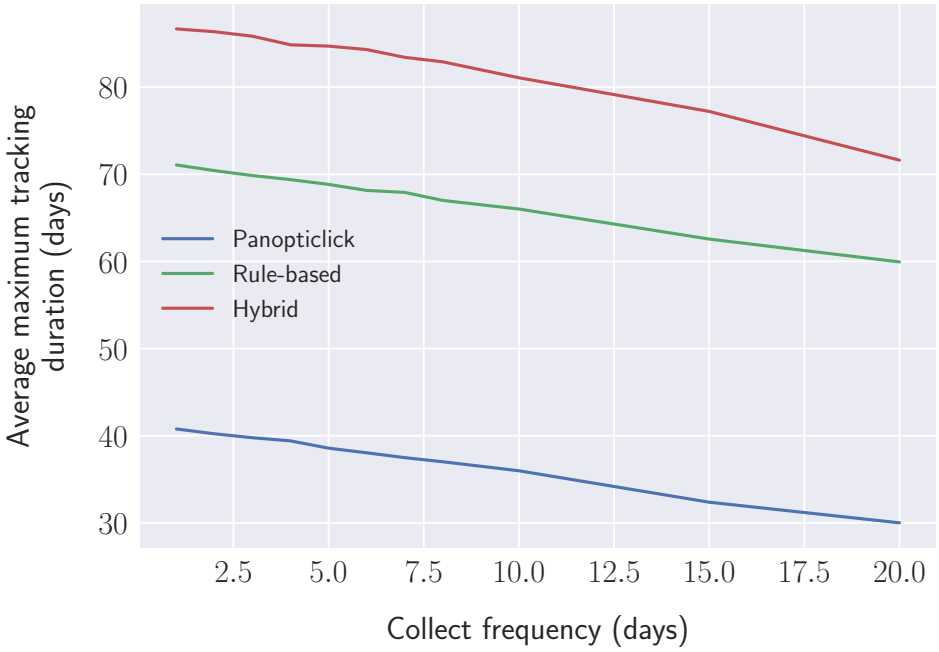


Figure 3.9 Average maximum tracking duration against simulated collect frequency for the three algorithms. This shows averages of the longest tracking durations that were constructed.

by its conservative nature. Indeed, as soon as there is more than one change, or multiple candidates for linking, Panopticlick's algorithm assigns a new id to the unknown browser instance. However, we can observe that both Fp-Stalker's hybrid and rule-based variants perform similarly.

Finally, Figure 3.11 presents the average *ownership* of tracking chains against the collect frequency for the three algorithms. We see that, despite its conservative nature, Panopticlick's ownership is 0.94, which means that, on average, 6% of a tracking chain is constituted of fingerprints that do not belong to the browser instance that owns the chain—*i.e.*, it is contaminated with other fingerprints. We also see that Fp-Stalker has an average *ownership* of 0.924, against 0.977 for the rule-based. Thus, while the hybrid version can tracker browser instances for longer period of times than the rule-based version, this can be partly explained by the fact that the hybrid version links more frequently fingerprints from different browser instances.

When it comes to linking browser fingerprints, Fp-Stalker's hybrid variant is better, or as good as, the rule-based variant. The next paragraphs focus on a few more results we obtain with the hybrid algorithm. Figure 3.12 presents the cumulative distribution of the average and maximum tracking duration when $collect\_frequency$ equals 7 days

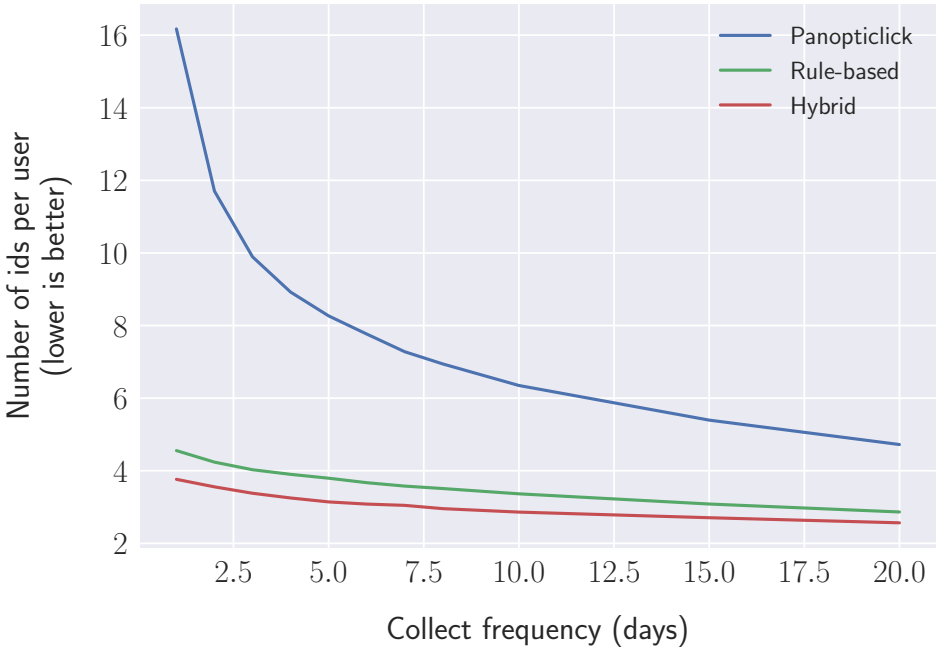


Figure 3.10 Average number of assigned ids per browser instance against simulated collect frequency for the three algorithms (lower is better).

for the hybrid variant. We observe that, on average, 12,4% of the browser instances are tracked more than 100 days. When it comes to the the longest tracking chains, we observe that more than 32.4% of the browser instances have been tracked at least once for more than 100 days during the experiment. These numbers show how tracking may depend on the browser and its configuration. Indeed, while some browsers are never tracked for a long period of time, others may be tracked for multiple months. This is also due to the duration of presence of browser instances in our experiments. Few browser instances were present for the whole experiment, most for a few weeks, and at best we can track a browser instance only as long as it was present. The graph also shows the results of the perfect linking algorithm (grey line), which can also be interpreted as the distribution of duration of presence of browser instances in our test set.

The boxplot in Figure 3.13 depicts the *number of ids* generated by the hybrid algorithm for a collect frequency of 7 days. It shows that half of the browser instances have been assigned 2 identifiers, which means they have one mistake, and more than 90% have less than 9 identifiers.

Finally, we also look at the distribution of the chains to see how often fingerprints from different browser instances are mixed together. For the Fp-Stalker hybrid variant, more than 95% of the chains have an ownership superior to 0.8, and more than 90%

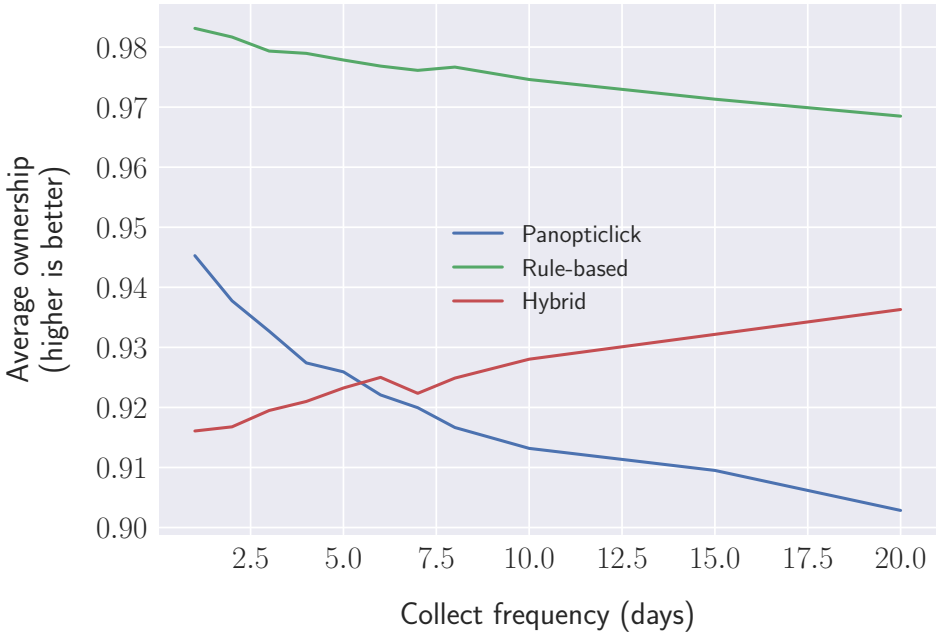


Figure 3.11 Average ownership of tracking chains against simulated collect frequency for the three algorithms. A value of 1 means the tracking chain contains only fingerprints of the same browser instance.

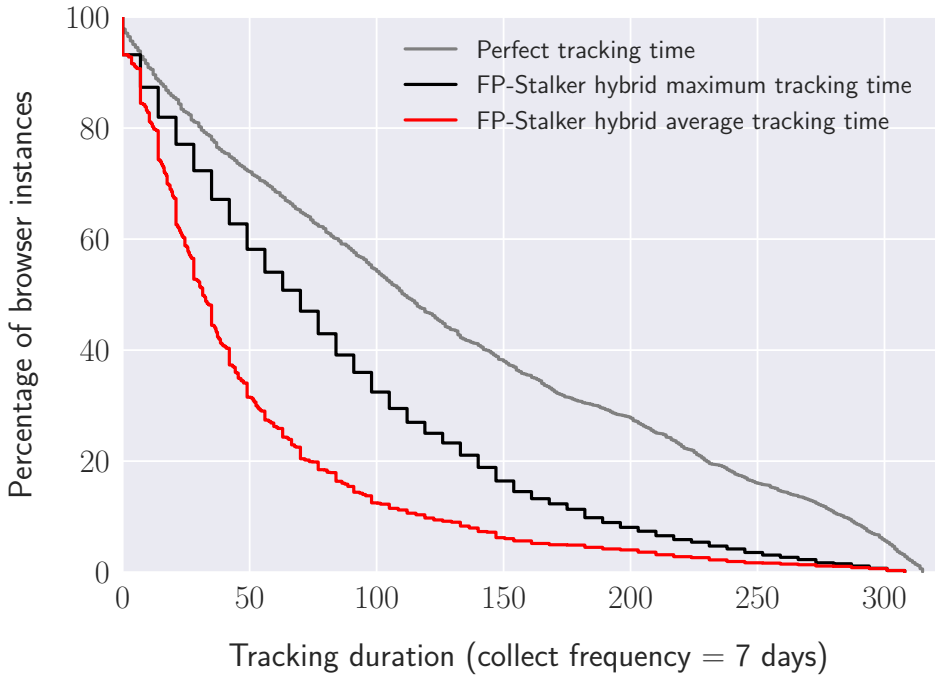


Figure 3.12 CDF of average and maximum tracking duration for a collect frequency of 7 days (Fp-Stalker hybrid variant only).

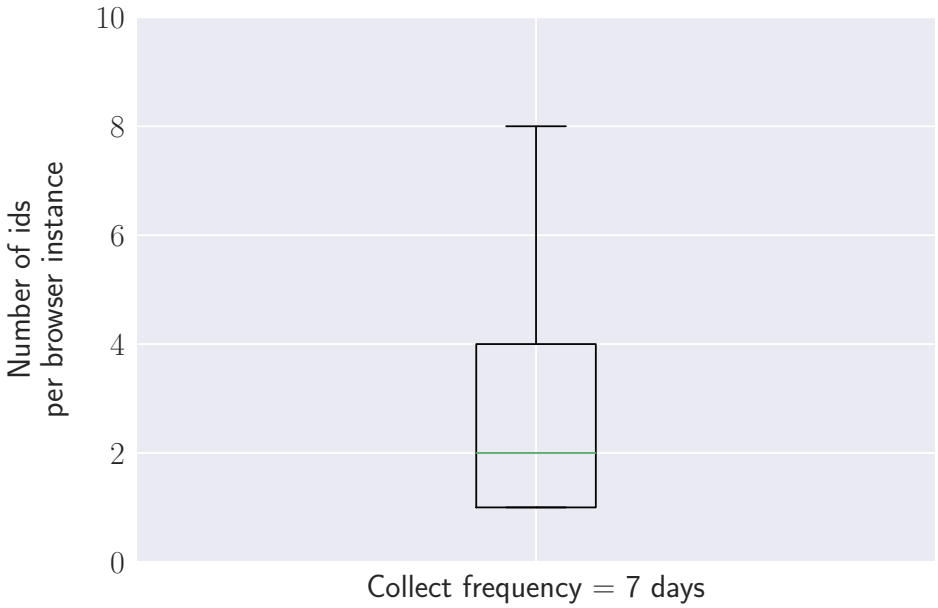


Figure 3.13 Distribution of `number of ids` per browser for a collect frequency of 7 days (Fp-Stalker hybrid variant only).

have perfect ownership—*i.e.*, 1. This shows that a small percentage of browser instances become highly mixed in the chains, while the majority of browser instances are properly linked into clean and relatively long tracking chains.

Comparison with the original dataset. Compare to the original version of the paper [16] published in 2018 with fewer fingerprints, we observe that the hybrid algorithm has a similar average tracking duration. The average maximum tracking duration increases, which can be explained by the fact that our evaluation dataset spans over a longer period. However, this difference does not necessarily imply there is a performance improvement. It is likely caused by the fact that as browser instances remain longer in the dataset, this increases the chance that some of them can be tracked longer. Concerning the number of identifiers per user, we also observe similar results with the original dataset: the hybrid and rule-based algorithms have similar performances, all while significantly outperforming the Panopticlick algorithm. Nevertheless, we observe a significant difference concerning the average ownership, *i.e.*, the quality of the tracking chains. With the original dataset, the hybrid algorithm and the rule-based algorithms performed similarly, with an average ownership > 0.975, no matter the collect frequency. However, in this chapter, we observe a significant drop for the hybrid algorithm, going from an average ownership of 0.985 in the original paper to 0.924 with the new dataset. This difference illustrates the difficulty of having machine learning models whose accuracy

remains stable over time. Indeed, even though the model was trained on 40% of the new dataset, its performance decreased. This drop can be partly explained by the fact that the random forest model is trained on old fingerprints while applied on recent fingerprints for evaluation. Nevertheless, browser fingerprinting is a constantly evolving field, which means that when we learn to predict fingerprint evolutions on the first 40% of the dataset, it does not fully transfer to the way the latest 60% of the fingerprints collected evolved. Thus, for better performance, it would have required to train the model on more recent fingerprints. While this is not possible in our case because of the number of fingerprints available and our need to have an evaluation dataset of significant size, it is not the case of commercial fingerprinters that have more fingerprints available to train their models.

A second factor that can explain the average ownership drop lies in the increase of the dataset size. Since there are more fingerprints, there is more chance the dataset contains similar or close fingerprints. This problem has already been discussed by Eckersley [3] and shows the limits of using relatively small datasets to study browser fingerprinting. We discuss this problem of representativity of the dataset as well as other limits in Section 3.3.6. Thus, it is possible that our machine learning model lacked discrimination power to distinguish between fingerprints originating from the same browser instances and fingerprinting originating from different browser instances. Nevertheless, there exist solutions to address this problem:

1. Doing more advanced feature engineering;
2. Adding new attributes to the fingerprints collected and use them in the machine learning model.

Concerning the second solution, I argue that adding new attributes, such as audio fingerprinting or more detailed attributes related to the size of the screen and the window, would help to improve the accuracy of the tracking algorithm.

3.3.5 Benchmark/Overhead

This section presents a benchmark that evaluates the performance of Fp-Stalker's hybrid and rule-based variants. We start by providing more details about our implementation, then we explain the protocol used for this benchmark, demonstrate that our approach can scale, and we show how our two variants behave when the number of browser instances increases.

The implementations of Fp-Stalker used for this benchmark are developed in Python, and the implementation of the random forest comes from the Scikit-Learn library. In order to study the scalability of our approach, we parallelized the linking algorithm to run on multiple nodes. A master node is responsible for receiving linkability requests, then it sends the unknown fingerprint to match f_u to slave nodes that compare f_u with all of the f_k present on their process. Then, each slave node sends its set of candidates associated either with a probability in case of the hybrid algorithm, or the number of changes in case of the rule-based version. Finally, the master node takes the final decision according to the policy defined either by the rule-based or hybrid algorithm. After the decision is made, it sends a message to each node to announce whether or not they should keep f_u in their local memory. In the case of the benchmark, we do not implement an optimization for exact matching. Indeed, normally the master nodes should hold a list of the exact matches associated with their ids.

The experimental protocol aims to study scalability. We evaluate our approach on a standard Azure cloud instance. We generate fake browser fingerprints to increase the test set size. Thus, this part does not evaluate the previous metrics, such as `tracking duration`, but only the execution times required to link synthetic browser fingerprints, as well as how well the approach scales across multiple processes.

The first step of the benchmark is to generate fake fingerprints from real ones. The generation process consists in taking a real fingerprint from our database and applying random changes to the canvas and the timezone attributes. We apply only two random changes so that generated fingerprints are unique, but they do not have too many differences which would reduce the number of comparisons. This point is important because our algorithms include heuristics related to the number of differences. Thus, by applying a small number of random changes, we do not discard all f_k fingerprints, making it the worst case scenario for testing execution times. Regarding the browser ids, we assign two generated fingerprints to each browser instance. It would not have been useful to generate more fingerprints per browser instance since we compare an unknown fingerprint only with the last 2 fingerprints of each browser instance. Then, the master node creates n slave processes and sends the generated fingerprints to them. The fingerprints are spread evenly over the processes.

Once the fingerprints are stored in the slave processes memory, we start our benchmark. We get 100 real fingerprints and try to link them with our generated fingerprints. For each

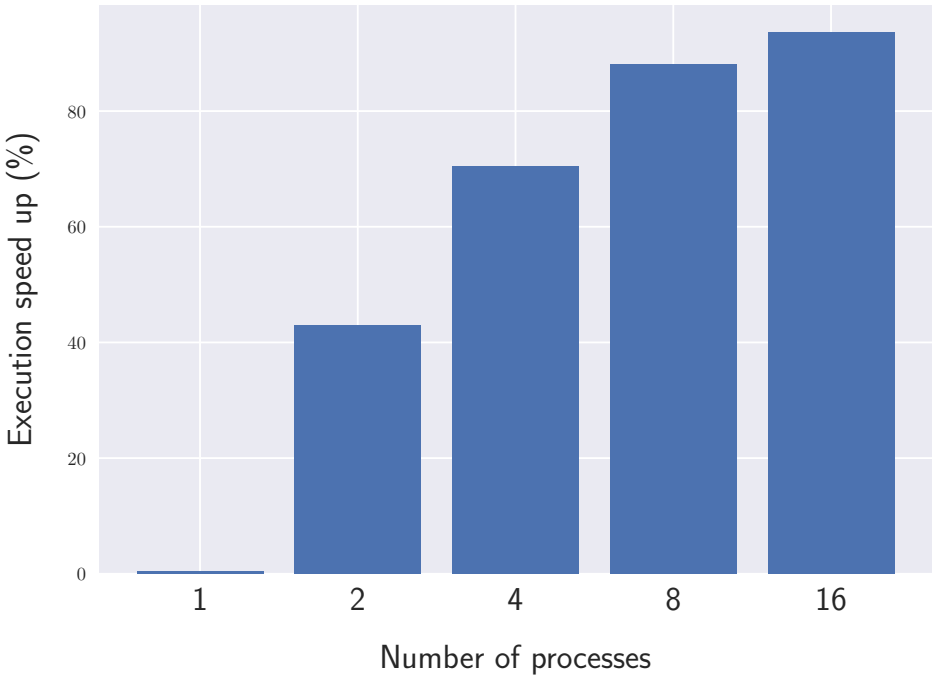


Figure 3.14 Speedup of average execution time against number of processes for Fp-Stalker's hybrid variant

fingerprint, we measure the execution time of the linking process. In this measurement, we measure:

1. The number of fingerprints and browser instances.
2. The number of processes spawned.

We execute our benchmark on a `Standard D16 v3` Azure instance with 16 virtual processors and 64 Gb of RAM, which has an associated cost of $576 USD per month. Figure 3.14 shows the execution time speedup in percentage against the number of processes for the hybrid approach. We see that that as the number of processes increases, we obtain a speedup in execution time. Going from 1 to 8 processes enables a speed up of more than 80%. Figure 3.15 shows the execution time to link a fingerprint against the number of browser fingerprints for Fp-Stalker's hybrid and rule-based variants, using 16 processes. Better `tracking duration` from the hybrid variant (see 3.3.4) is obtained at the cost of execution speed. Indeed, for any given number of processes and browser instances, the rule-based variant links fingerprints about 5 times faster. That said, the results show that the hybrid variant links fingerprints relatively quickly.

However, the raw execution times should not be used directly. The algorithm was implemented in Python, whose primary focus is not performance. Moreover, although we

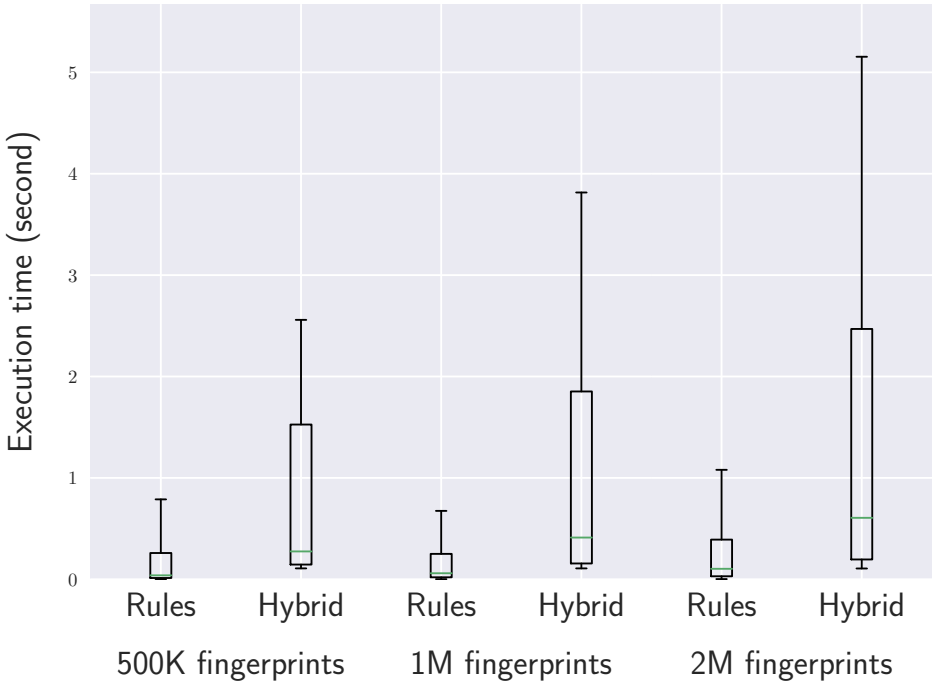


Figure 3.15 Execution times for Fp-Stalker hybrid and rule-based to link a fingerprint using 16 processes. Time is dependent on the size of the test set. The increased effectiveness of the hybrid variant comes at the cost slower of execution times.

scaled by adding processes, it is possible to scale further by splitting the linking process (*e.g.*, depending on the combination of OS and browser, send the fingerprint to more specialized nodes). In our current implementation, if an unknown fingerprint from a Chrome browser on Linux is trying to be matched, it will be compared to fingerprints from Firefox on Windows, causing us to wait even though they have no chance of being linked. By adopting a hierarchical structure where nodes or processes are split depending on their OS and browser, it is possible to increase the throughput of our approach.

Furthermore, the importance of the raw execution speeds depend highly on the use case. In the case where fingerprinting is used as a way to regenerate cookies (*e.g.*, for advertising), a fingerprint only needs to be linked when the cookie is missing or has been erased, a much less frequent event. Another use case is using browser fingerprinting as a way to enhance authentication [12]. In this case, one only needs to match the fingerprint of the browser attempting to sign-in with the previous fingerprints from the same user, drastically reducing the number of comparisons.

3.3.6 Threats to Validity

First, the results we report in this work depend on the representativity of our browser fingerprint dataset. We developed extensions for Chrome and Firefox, the two most popular web browsers, and distributed them through standard channels. This does provide long term data, and mitigates a possible bias if we had chosen a user population ourselves, but it is possible that the people interested in our extension are not a good representation of the average Web surfer.

Second, there is a reliability threat due to the difficulty in replicating the experiments. Unfortunately, this is inherent to scientific endeavors in the area of privacy: these works must analyze personal data (browser fingerprints in our case) and the data cannot be publicly shared. Yet, the code to split the data, generate input data, train the algorithm, as well as evaluate it, is publicly available online on GitHub.[1]

Finally, a possible internal threat lies in our experimental framework. We did extensive testing of our machine learning algorithms, and checked classification results as thoroughly as possible. We paid attention to split the data and generate a scenario close to what would happen in a web application. However, as for any large scale experimental

[1] https://github.com/Spirals-Team/FPStalker

infrastructure, there are surely bugs in this software. We hope that they only change marginal quantitative things, and not the qualitative essence of our findings.

3.3.7 Discussion

This chapter studies browser fingerprint linking in isolation, which is its worst-case scenario. In practice, browser fingerprinting is often combined with stateful tracking techniques (*e.g.*, cookies, Etags) to respawn stateful identifiers [5]. In such cases, fingerprint linking is performed much less frequently since most of the time a cookie is sufficient and inexpensive to track users. Our work shows that browser fingerprinting can provide an efficient solution to extend the lifespan of cookies, which are increasingly being deleted by privacy-aware users.

Browser vendors and users would do well to minimize the differences that are so easily exploited by fingerprinters. Our results show that some browser instances have highly trackable fingerprints, to the point that very infrequent fingerprinting is quite effective. In contrast, other browser instances appear to be untrackable using the attributes we collect. Vendors should work to minimize the attack surfaces exploited by fingerprinters, and users should avoid customizing their browsers in ways that make them expose unique and linkable fingerprints.

Depending on the objectives, browser fingerprint linking can be tuned to be more conservative and avoid false positives (e.g., for second-tier security purposes), or more permissive (e.g., ad tracking). Tuning could also be influenced by how effective other tracking techniques are. For example, it could be tuned very conservatively and simply serve to extend cookie tracking in cases where privacy-aware users, which are in our opinion more likely to have customized (*i.e.*, unique and linkable) browser configurations, delete their cookies.

3.4 Conclusion

In this chapter, we investigated the stability of browser fingerprints and show that besides having a high entropy, new techniques, such as canvas fingerprinting, remain stable for long periods of time. Then, we measured how long can browsers be tracked using solely their fingerprint and proposed Fp-Stalker, an approach to link fingerprint changes over time. We address the problem with two variants of Fp-Stalker. The first one

builds on a ruleset identified from an analysis of grounded programmer knowledge. The second variant combines the most discriminating rules by leveraging machine learning to sort out the more subtle ones.

We trained the Fp-Stalker hybrid variant with a training set of fingerprints that we collected for 2 years through browser extensions installed by 2,346 volunteers. By analyzing the feature importance of our random forest, we identified the `number of changes`, the `time difference`, as well as the `user agent`, as the three most important features.

We ran Fp-Stalker on our test set to assess its capacity to link fingerprints, as well as to detect new browser instances. Our experiments demonstrate that the hybrid variant can correctly link fingerprint evolutions from a given browser instance for 49.3 consecutive days on average, against 40.9 days for the rule-based variant. When it comes to the maximum tracking duration, with the hybrid variant, more than 32.4% of the browsers can be tracked for more than 100 days.

Concerning the differences with the paper published at S&P 18 [16], while the performances of the hybrid algorithm in term of tracking duration and number of ids per users are quite similar, we observe a significant difference concerning the average ownership, *i.e.*, the quality of the tracking chains. We observe a significant drop for the hybrid algorithm, going from an average of 0.985 in the original paper to 0.924 with the new dataset. This can be partly explained by the fact that the model is trained on old fingerprints while applied on recent fingerprints during the evaluation. Nevertheless, browser fingerprinting is a constantly evolving field as new features are added and deprecated. Thus, when our random forest model learns how fingerprints evolve on the first 40% fingerprints, it does not fully transfer to the most recent 60% fingerprints. Thus, to obtain better performance, it would have required to train the linking model on more recent fingerprints. While this was not possible in our cause because of the number of fingerprints we have and our need to have an evaluation dataset of significant size, it is not a problem for commercial fingerprinters that have more fingerprints available to train their models. A second reason to explain this drop comes from the fact that as the size of the dataset increase, so does the probability that two fingerprints are the same or close, even though they belong to different browser instance. Thus, to address this problem, one either needs to improve the feature engineering of the machine learning model or add new fingerprinting attributes such as the audio fingerprinting or attributes related to the size of the screen and the window.

Regarding the usability of Fp-Stalker, we measure the average execution time to link an unknown fingerprint when the number of known fingerprints is growing. We show that both our rule-based and hybrid variants scale horizontally. However, even though our hybrid variant is better in term of tracking duration, we show that it introduces a non-negligible overhead compared to the pure rule-based approach.

As we showed in this chapter, browser fingerprinting is a threat to privacy and can be used in addition to other stateful mechanisms, such as cookies, to track users for longer period of times. To protect against browser fingerprinting, several countermeasures have been proposed. Nevertheless, these countermeasures may generate inconsistent fingerprints that can make their users more identifiable and therefore trackable. Thus, in the next chapter, we evaluate the effectiveness of fingerprinting countermeasures and how using them impact the privacy of their users.

Chapter 4

Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies

A wide range of countermeasures has been developed to protect against browser fingerprinting. Some of them lie on the nature of the device in order to fool the trackers, while others add random noise to pixels of canvas fingerprints in order to break to the stability required for tracking. Nevertheless, the way these countermeasures have been evaluated does not properly asses their impact on user privacy, in particular regarding the quantity of information they may indirectly leak by revealing their presence.

My work was motivated by Nikiforakis *et al.* [4] that first demonstrated how inconsistencies introduced by user agent spoofers could be used to reveal their presence, which could help fingerprinters track browsers with such extensions. This chapter goes beyond the specific case of user agent spoofers and studies a wider range of state-of-the-art fingerprinting countermeasures. Moreover, I also challenge the claim that being more distinguishable necessarily makes tracking more accurate.

My motivation to conduct a better evaluation of fingerprinting countermeasures was strengthened by recent findings when inspecting the code of a commercial fingerprinting script used by Augur [96]. I discovered that this script computes an attribute called `spoofed`, which is the result of multiple tests that evaluate the consistency between the `user agent`, the `platform`, `navigator.oscpu`, `navigator.productSub`, as well as the value returned by `eval.toString.length` used to detect a browser. Moreover, the code also tests for the presence of touch support on devices that claim to be mobiles. Similar

tests are also present in the widely used open source library FingerprintJS2 [65]. While we cannot know the motivations of fingerprinters when it comes to detecting browsers with countermeasures—*i.e.*, this could be used to identify bots, to block fraudulent activities, or to apply additional tracking heuristics—I argue that countermeasures should avoid revealing their presence as this can be used to better target the browser. Thus, I consider it necessary to evaluate the privacy implications of using fingerprinting countermeasures.

In this chapter, I show how most of the fingerprinting countermeasures presented in the state-of-the-art can negatively impact user privacy. First, in Section 4.1 I propose FP-Scanner, a test suite that leverages fingerprint inconsistencies to detect if a user has a fingerprinting countermeasure installed in her browser. In Section 4.2, I detail my implementation of a fingerprinting script and an inconsistency scanner capable of detecting altered fingerprints. I evaluate it against state-of-the-art countermeasures and show that none of them can hide their presence to our scanner. Then, I go further and show that even when countermeasures modify attributes such as the user agent or the platform, our scanner can still recover the original OS and browser values. In Section 4.3 I discuss how fingerprinters can leverage this information to improve their tracking algorithms. Finally, I conclude in Section 4.4. This chapter was originally published as a conference paper entitled *Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies* [17] published at Usenix Security 18.

4.1 Investigating Fingerprint Inconsistencies

Based on our study of existing browser fingerprinting countermeasures published in the literature, we organized our test suite to detect fingerprint inconsistencies along 4 distinct components. The sequence of tests is ordered by the increasing complexity required to detect an inconsistency. In particular, the first two tests aim at detecting inconsistencies at the OS and browser levels, respectively. The third one focuses on detecting inconsistencies at the device level. Finally, the fourth test aims at revealing canvas poisoning techniques. Each test focuses on detecting specific inconsistencies that could be introduced by a countermeasure. While some of the tests we integrate, such as checking the values of both user agents or browser features, have already been proposed by Nikiforakis *et al.* [4], we also propose new tests to strengthen our capacity to detect inconsistencies. Figure 4.1 depicts the 4 components of our inconsistency test suite.

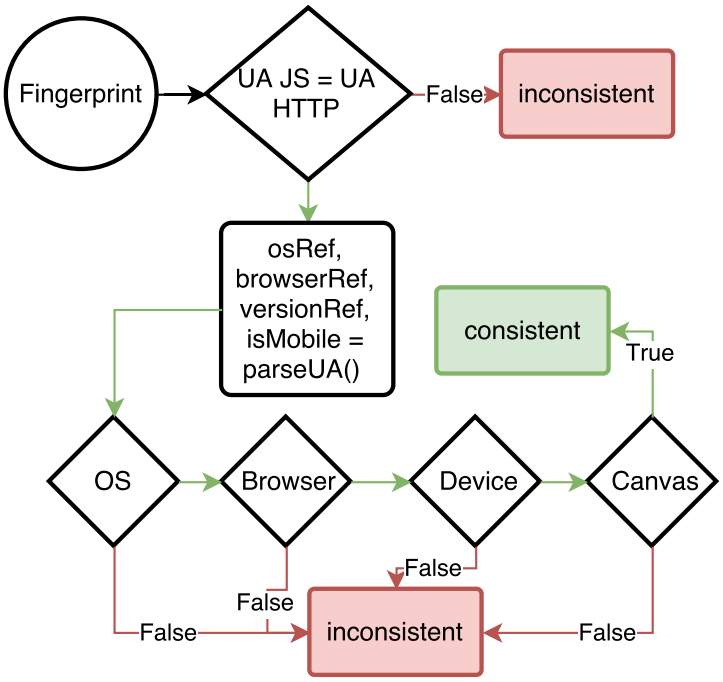


Figure 4.1 Overview of the inconsistency test suite.

4.1.1 Uncovering OS Inconsistencies

Although checking the browser's identity is straightforward for a browser fingerprinting algorithm, verifying the host OS is more challenging because of the sandbox mechanisms used by the script engines. In this section, we present the heuristics applied to check a fingerprinted OS attribute.

User Agent. We start by checking the user agent consistency [4], as it is a key attribute to retrieve the OS and browser of a user. The user agent is available both from the client side, through the navigator object (`navigator.userAgent`), and from the server side, as an HTTP header (`User-Agent`). The first heuristics we apply checks the equality of these two values, as naive browser fingerprinting countermeasures, such as basic user agent spoofers, tend to only alter the HTTP header. The difference between the two user agent attributes reflects a coarse-grained inconsistency that can be due to the OS and/or the browser. While extracting the OS and the browser substrings can help to reveal the source of the inconsistency, the similarity of each substring does not necessarily guarantee the OS and the browser values are true, as both might be spoofed. Therefore, we extract and store the OS, browser and version substrings as internal variables `OSref`, `browserRef`, `browserVersionRef` for further investigation.

Table 4.1 Mapping between common `OS` and `platform` values.

OS	Platforms
`Linux`	Linux i686, Linux x86_64, Linux, Linux armv8l
`Windows`	Win32, Win64
`iOS`	iPhone, iPad
`Android`	Linux armv71, Linux i686, Linux armv8l
`macOS`	MacIntel
`FreeBSD`	FreeBSD amd64, FreeBSD i386

Navigator platform. The value of `navigator.platform` reflects the platform on which the browser is running. This attribute is expected to be consistent with the variable `OSref`, extracted in the first step [4]. Nevertheless, consistent does not mean equal as, for example, the user agent of a 32-bits Windows will contain the substring `WOW64`, which stands for *Windows on Windows 64-bits*, while the attribute `navigator.platform` will report the value `Win32`. Table 4.1 therefore maps `OSref` and possible values of `navigator.platform` for the most commonly used OSes.

WebGL. `WebGL` is a JavaScript API that extends the HTML 5 canvas API to render 3D objects from the browser. In particular, we propose a new test that focuses on two WebGL attributes related to the OS: `renderer` and `vendor`. The first attribute reports the name of the GPU, for example `ANGLE (VMware SVGA 3D Direct3D11 vs_4_0 ps_4_0)`. Interestingly, the substring `VMware` indicates that the browser is executed in a virtual machine. Also, the `ANGLE` substring stands for *Almost Native Graphics Layer Engine*, which has been designed to bring OpenGL compatibility to Windows devices. The second WebGL attribute (`vendor`) is expected to provide the name of the GPU vendor, whose value actually depends on the OS. On a mobile device, the attribute `vendor` can report the string `Qualcomm`, which corresponds to the vendor of the mobile chip, while values, like `Microsoft`, are returned for Internet Explorer on Windows, or `Google Inc` for a Chrome browser running on a Windows machine. We summarize the mapping for the attributes `renderer` and `vendor` in Table 4.2.

Browser plugins. Plugins are external components that add new features to the browser. When querying for the list of plugins via the `navigator.plugins` object, the browser returns an array of plugins containing detailed information, such as their filename and the associated extension, which reveals some indication of the OS. On Windows, plugin

Table 4.2 Mapping between `OS` and substrings in WebGL `renderer`/`vendor` attributes for common OSes.

OS	Renderer	Vendor
`Linux`	Mesa, Gallium	Intel, VMWare, X.Org
`Windows`	ANGLE	Microsoft, Google Inc
`iOS`	Apple, PowerVR	Apple, Imagination
`Android`	Adreno, Mali, PowerVR	Qualcomm, ARM, Imagination
`macOS`	OpenGL, Iris	Intel, ATI
`Windows Phone`	Qualcomm, Adreno	Microsoft

file extensions are `.dll`, on macOS they are `.plugin` or `.bundle` and for Linux based OS extensions are `.so`. Thus, we propose a test that ensures that `OSref` is consistent with its associated plugin filename extensions. Moreover, we also consider constraints imposed by some systems, such as mobile browsers that do not support plugins. Thus, reporting plugins on mobile devices is also considered as an inconsistency.

Media queries. Media query is a feature included in CSS 3 that applies different style properties depending on specific conditions. The most common use case is the implementation of responsive web design, which adjusts the stylesheet depending on the size of the device, so that users have a different interface depending on whether they are using a smartphone or a computer. In this step, we consider a set of media queries provided by the Firefox browser to adapt the content depending on the value of desktop themes or Windows OS versions. Indeed, it is possible to detect the Mac graphite theme using `-moz-mac-graphite-theme` media query [104]. It is also possible to test specific themes present on Windows by using `-moz-windows-theme`. However, in the case of Windows, there is a more precise way to detect its presence, and even its version. It is also possible to use the `-moz-os-version` media query to detect if a browser runs on Windows XP, Vista, 7, 8 or 10. Thus, it is possible to detect some Mac users, as well as Windows users, when they are using Firefox. Moreover, since these media queries are only available from Firefox, if one of the previous media queries is matched, then it likely means that the real browser is Firefox.

Fonts. Saito *et al.* [105] demonstrated that fonts may be dependent on the OS. Thus, if a user claims to be on a given OS `A`, but does not list any font linked to this OS `A` and,

at the same time, displays many fonts from another OS `B`, then we may assume that OS `A` is not its real OS.

This first step in Fp-Scanner aims to check if the OS declared in the user agent is the device's real OS. In the next step, we extend our verification process by checking if the browser and the associated version declared by the user agent have been altered.

4.1.2 Uncovering Browser Inconsistencies

This step requires the extraction of the variables `browserRef` and `browserVersion Ref` from the user agent to further investigate their consistency.

Error. In JavaScript, `Error` objects are thrown when a runtime error occurs. There exist 7 different types of errors for client-side exceptions, which depend on the problem that occurred. However, for a given error, such as a stack overflow, not all the browsers will throw the same type of error. In the case of a stack overflow, Firefox throws an `InternalError` and Chrome throws a `RangeError`. Besides the type of errors, depending on the browser, error instances may also contain different properties. While two of them—`message` and `name`—are standards, others such as `description`, `lineNumber` or `toSource` are not supported by all browsers. Even for properties, such as `message` and `name`, which are implemented in all major browsers, their values may differ for a given error.

For example, executing *null*[0] on Chrome will generate the following error message *"Cannot read property '0' of null"*, while Firefox generates *"null has no properties"*, and Safari *"null is not an object (evaluating 'null[0]')"*.

Function's internal representation. It is possible to obtain a string representation of any object or function in JavaScript by using the `toString` method. However, such representations—*e.g.*, `eval.toString()`—may differ depending on the browser, with a length that characterizes it. Firefox and Safari return the same string, with a length of 37 characters, while on Chrome it has a length of 33 characters, and 39 on Internet Explorer. Thus, we are able to distinguish most major desktop browsers, except for Firefox and Safari. Then, we consider the property `navigator.productSub`, which returns the build number of the current browser. On Safari, Chrome and Opera, it always returns the string `20030107` and, combined with `eval.toString().length`, it can therefore be used to distinguish Firefox from Safari.

Navigator object. `Navigator` is a built-in object that represents the state and the identity of the browser. Since it characterizes the browser, its prototype differs depending not only on the browser's family, but also the browser's version. These differences come from the availability of some browser-specific features, but also from two other reasons:

1. The order of `navigator` is not specified and differs across browsers [4];
2. For a given feature, different browsers may name it differently. For example, if we consider the feature `getUserMedia`, it is available as `mozGetUserMedia` on FIREFOX and `webkitGetUserMedia` on a Webkit-based browser.

Moreover, as `navigator` properties play an important role in browser fingerprinting, our test suite detects if they have been overridden by looking at their internal string representation. In the case of a genuine fingerprint whose attributes have not been overridden in JavaScript, it should contain the substring `native code`. However, if a property has been overridden, it will return the code of the overridden function.

Browser features. Browsers are complex software that evolve at a fast pace by adding new features, some being specific to a browser. By observing the availability of specific features, it is possible to detect if a browser is the one it claims to be [4, 37]. Since for a given browser, features evolve depending on the version, we can also check if the features available are consistent with `browserVersionRef`. Otherwise, this may indicate that the browser version displayed in the `user agent` has been manipulated.

4.1.3 Uncovering Device Inconsistencies

This section aims at detecting if the device belongs to the class of devices it claims to be—*i.e.*, mobile or computer.

Browser events. Some events are unlikely to happen, such as touch-related events (`touchstart`, `touchmove`) on a desktop computer. On the opposite, mouse-related events (`onclick`, `onmousemove`) may not happen on a smartphone. Therefore, the availability of an event may reveal the real nature of a device.

Browser sensors. Like events, some sensors may have different outputs depending on the nature of devices. For example, the accelerometer, which is generally assumed to only be available on mobile devices, can be retrieved from a browser without requesting any authorization. The value of the acceleration will always slightly deviate from 0 for a real mobile device, even when lying on a table.

(a) Canvas fingerprint with no countermeasure.

(b) Canvas fingerprint with a countermeasure.

Figure 4.2 Two examples of canvas fingerprints (a) a genuine canvas fingerprint without any countermeasures installed in the browser and (b) a canvas fingerprint altered by the Canvas Defender countermeasure that applies a uniform noise to all the pixels in the canvas.

4.1.4 Uncovering Canvas Inconsistencies

Canvas fingerprinting uses the HTML 5 canvas API to draw 2D shapes using JavaScript. This technique, discovered by Mowery *et al.* [38], is used to fingerprint browsers. To do so, one scripts a sequence of instructions to be rendered, such as writing text, drawing shapes or coloring part of the image, and collects the rendered output. Since the rendering of this canvas relies on the combination of different hardware and software layers, it produces small differences from device to device. An example of the rendering obtained on a Chrome browser running on Linux is presented in Figure 4.2a.

As we mentioned, the rendering of the canvas depends on characteristics of the device, and if an instruction has been added to the script, you can expect to observe its effects in the rendered image. Thus, we consider these scripted instructions as constraints that must be checked in the rendered image. For example, the canvas in Figure 4.2b has been obtained with the Canvas Defender extension installed. We observe that contrary to the vanilla canvas that does not use any countermeasure (Figure 4.2a), the canvas with the countermeasure has a background that is not transparent, which can be seen as a constraint violation. We did not develop a new canvas test, we reused the one adopted by state-of-the-art canvas fingerprinting [27]. From the rendered image, our test suite checks the following properties:

1. *Number of transparent pixels* as the background of our canvas must be transparent, we expect to find a majority of these pixels;

2. *Number of isolated pixels*, which are pixels whose `rgba` value is different than $(0,0,0,0)$ and are only surrounded by transparent pixels. In the rendered image, we should not find this kind of pixel because shapes or texts drawn are closed;
3. *Number of pixels per color* should be checked against the input canvas rendering script, even if it is not possible to know in advance the exact number of pixels with a given color, it is expected to find colors defined in the canvas script.

We also check if canvas-related functions, such as `toDataUrl`, have been overridden.

4.2 Empirical Evaluation

This section compares the accuracy of FP-SCANNER with FINGERPRINTJS2 [65], an open source fingerprinting script and AUGUR [96], a commercial fingerprinting script, to classify genuine and altered browser fingerprints modified by state-of-the-art fingerprinting countermeasures.

4.2.1 Implementing FP-Scanner

Instead of directly implementing and executing our test suite within the browser, thus being exposed to countermeasures, we split FP-SCANNER into two parts. The first part is a client-side fingerprinter, which uploads raw browser fingerprints on a remote storage server. For the purpose of our evaluation, this fingerprinter extends state-of-the-art fingerprinters, like FINGERPRINTJS2, with the list of attributes covered by FP-SCANNER (*e.g.*, WebGL fingerprint). Table 4.3 reports on the list of attributes collected by this fingerprinter. The resulting dataset of labeled browser fingerprints is made available to leverage the reproducibility of our results. [1]

The second part of FP-SCANNER is the server-side implementation, in Python, of the test suite we propose (cf. Section 4.1). This section reports on the relevant technical issues related to the implementation of the 4 components of our test suite.

4.2.1.1 Checking OS Inconsistencies

`OSRef` is defined as the OS claimed by the user agent attribute sent by the browser and is extracted using a UA PARSER library [106]. We used the browser fingerprint

[1] FP-Scanner dataset: https://github.com/Spirals-Team/FP-Scanner

Table 4.3 List of attributes collected by our fingerprinting script.

Attribute	Description
`HTTP headers`	List of HTTP headers sent by the browser and their associated value
`User agent navigator`	Value of `navigator.userAgent`
`Platform`	Value of `navigator.platform`
`Plugins`	List of plugins (description, filename, name) obtained by `navigator.plugins`
`ProductSub`	Value of `navigator.productSub`
`Navigator prototype`	String representation of each property and function of the `navigator` object prototype
`Canvas`	Base 64 representation of the image generated by the canvas fingerprint test
`WebGL renderer`	`WebGLRenderingContext.getParameter("renderer")`
`WebGL vendor`	`WebGLRenderingContext.getParameter("vendor")`
`Browser features`	Presence or absence of certain browser features
`Media queries`	Collect if media queries related to the presence of certain OS match or not using `window.matchMedia`
`Errors type 1`	Generate a `TypeError` and store its properties and their values
`Errors type 2`	Generate an error by creating a socket not pointing to an URL and store its string representation
`Stack overflow`	Generate a stack overflow and store the error `name` and `message`
`Eval toString length`	Length of `eval.toString().length`
`Media devices`	Value of `navigator.mediaDevices.enumerateDevices`
`TouchSupport`	Collect the value of `navigator.maxTouchPoints`, store if we can create a `TouchEvent` and if `window` object has the `ontouchstart` property
`Accelerometer`	`true` if the value returned by the accelerometer sensor is different of 0, else `false`
`Screen resolution`	Values of `screen.width/ height`, and `screen.availWidth/ Height`
`Fonts`	Font enumeration using JavaScript [39]
`Overwritten properties`	Collect string representation of `screen.width/height` getters, as well as `toDataURL` and `getTimezoneOffset` functions

dataset from AmIUnique [27] to analyze if some of the fonts they collected were only available on a given OS. We considered that if a font appeared at least 100 times for a given OS family, then it could be associated to this OS. We chose this relatively conservative value because the AmIUnique database contains many fingerprints that are spoofed, but of which we are unaware of. Thus, by setting a threshold of 100, we may miss some fonts linked to a certain OS, but we limit the number of false positives—*i.e.*, fonts that we would classify as linked to an OS but which should not be linked to it. Fp-Scanner checks if the fonts are consistent with `OSRef` by counting the number of fonts associated to each OS present in the user font list. If more than $N_f = 1$ fonts are associated to another OS than `OSRef`, or if no font is associated to `OSRef`, then Fp-Scanner reports an OS inconsistency. It also tests if `moz-mac-graphite-theme` and `@media(-moz-os-version: $win-version)` with `$win-version` equals to Windows XP, Vista, 7, 8 or 10, are consistent with `OSRef`.

4.2.1.2 Checking Browser Inconsistencies

We extract `BrowserRef` using the same user agent parsing library as for `OSRef`. With regards to JavaScript errors, we check if the fingerprint has a prototype, an error message, as well as a type consistent with `browserRef`. Moreover, for each attribute and function of the `navigator` object, Fp-Scanner also checks if the string representation reveals that it has been overridden. Testing if the features of the browser are consistent with `browserRef` is achieved by comparing the features collected using Modernizr [107] with the open source data file provided by the website Caniuse [108]. The file is freely available on Github[2] and represents most of the features present in Modernizr as a JSON file. For each of them, it details if they are available on the main browsers, and for which versions. We consider that a feature can be present either if it is present by default or it can be activated. Then, for each Modernizr feature we collected in the browser fingerprint, we check if it should be present according to the Caniuse dataset. If there are more than $N_e = 1$ errors, either features that should be available but are not, or features that should not be available but are, then we consider the browser as inconsistent.

[2]List of available features per browser: https://github.com/Fyrd/caniuse/blob/master/data.json

4.2.1.3 Checking Device Inconsistencies

We verify that, if the device claims to be a mobile, then the accelerometer value is set to `true`. We apply the same technique for touch-related events. However, we do not check the opposite—*i.e.*, that computers have no touch related events—as some new generations of computers include touch support. Concerning the screen resolution, we first check if the screen `height` and `width` have been overridden.

4.2.1.4 Checking Canvas Poisoning

To detect if a canvas has been altered, we extract the 3 metrics proposed in Section 4.1. We first count the number of pixels whose `rgba` value is $(0,0,0,0)$. If the image contains less than $N_{tp} = 4{,}000$ transparent pixels, or if it is full of transparent pixels, then we consider that the canvas has been poisoned or blocked. Secondly, we count the number of isolated pixels. If the canvas contains more than 10 of them, then we consider it as poisoned. We did not set a lower threshold as we observed that some canvas on macOS and Safari included a small number of isolated pixels that are not generated by a countermeasure. Finally, the third metric tests the presence of the orange color $(255,102,0,100)$ by counting the number of pixels having this exact value, and also the number of pixels whose color is slightly different—*i.e.*, pixels whose color vector v_c satisfies the following equation $\|(255,102,0,100) - v_c\| < 4$. Our intuition is that canvas poisoners inject a slight noise, thus we should find no or few pixels with the exact value, and many pixels with a slightly different color.

For each test of our suite, Fp-Scanner stores the details of each test so that it is possible to know if it is consistent, and which steps of the analysis failed.

Estimating the parameters Different parameters of Fp-Scanner, such as the number of transparent pixels, may influence the accuracy of Fp-Scanner, resulting in different values of true and false positives. The strategy we use to optimize the value of a given parameter is to run the scanner test that relies on this parameter, and to tune the value of the parameter to minimize the *false positive rate* (FPR)—*i.e.*, the ratio of fingerprints that would be wrongly marked as altered by a countermeasure, but that are genuine. The reason why we do not run all the tests of the scanner to optimize a given parameter is because there may be some redundancy between different tests. Thus, changing a parameter value may not necessarily results in a modification of the detection

Table 4.4 List of relevant tests per countermeasure.

Test (scope)	***RAS***	***UA spoofers***	***Canvas extensions***	***FPRandom***	***Brave***	***Firefox***
`User Agents` (global)	✓	✓				✓
`Platform` (OS)	✓	✓				✓
`WebGL` (OS)	✓	✓				✓
`Plugins` (OS)	✓	✓				✓
`Media Queries` (OS, browser)	✓	✓				✓
`Fonts` (OS)	✓	✓				✓
`Error` (browser)	✓	✓				✓
`Function representation` (browser)	✓		✓			
`Product` (browser)	✓	✓				
`Navigator` (browser)	✓	✓			✓	
`Enumerate devices` (browser)					✓	
`Features` (browser)	✓	✓			✓	✓
`Events` (device)	✓	✓				
`Sensors` (device)	✓	✓				
`ToDataURL` (canvas)	✓		✓			
`Pixels` (canvas)	✓		✓	✓	✓	✓

as a given countermeasure may be detected by multiple tests. Moreover, we ensure that countermeasures are detected for the appropriate symptoms. Indeed, while it is normal for a canvas countermeasure to be detected because some pixels have been modified, we consider it to be a false positive when detected because of a wrong browser feature threshold, as the countermeasure does not act on the browser claimed in the user agent. Table 4.4 describes, for each countermeasure, the tests that can be used to reveal its presence. If a countermeasure is detected by a test not allowed, then it is considered as a false positive.

Figures 4.3, 4.4 and 4.5 shows the detection accuracy and the *false positive rate* (FPR) for different tests and different values of the parameters to optimize. We define the accuracy as $\frac{\#TP+\#TN}{\#Fingerprints}$ where *true positives* (TP) are the browser fingerprints correctly classified as inconsistent, and *true negatives* (TN) are fingerprints correctly classified as genuine. Table 4.5 shows, for each parameter, the optimal value we considered for the

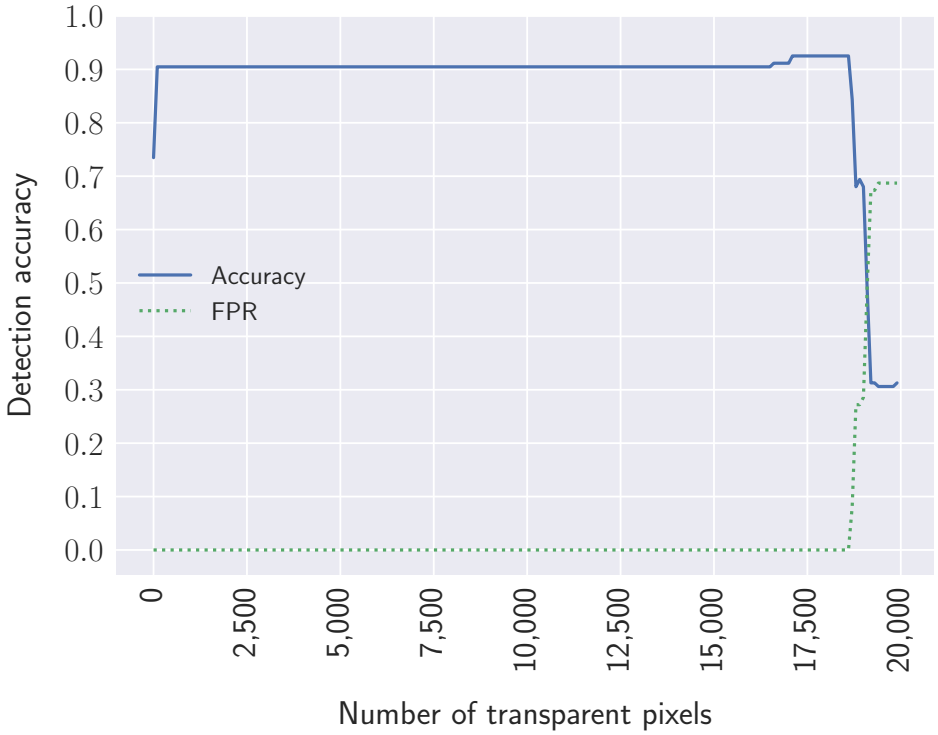


Figure 4.3 Detection accuracy and false positive rate using the transparent pixels test for different values of N_{tp} (number of transparent pixels).

evaluation. The last column of Table 4.5 reports on the false positive rate, as well as the accuracy obtained by running only the test that makes use of the parameter to optimize.

In the case of the number of transparent pixels N_{tp} we observe no difference between 100 and 16,500 pixels. Between 16,600 and 18,600 there is a slight improvement in terms of accuracy caused by a change in the true positive rate. Thus, we chose a value of 17,200 transparent pixels since it provides both a false positive rate of 0 while maximizing the accuracy.

Table 4.5 Optimal values of the different parameters to optimize, as well as the FPR and the accuracy obtained by executing the test with the optimal value.

Attribute	**Optimal value**	**FPR (accuracy)**
Pixels: N_{tp}	17,200	0 **(0.93)**
Fonts: N_f	2	0 **(0.42)**
Features: N_e	1	0 **(0.51)**

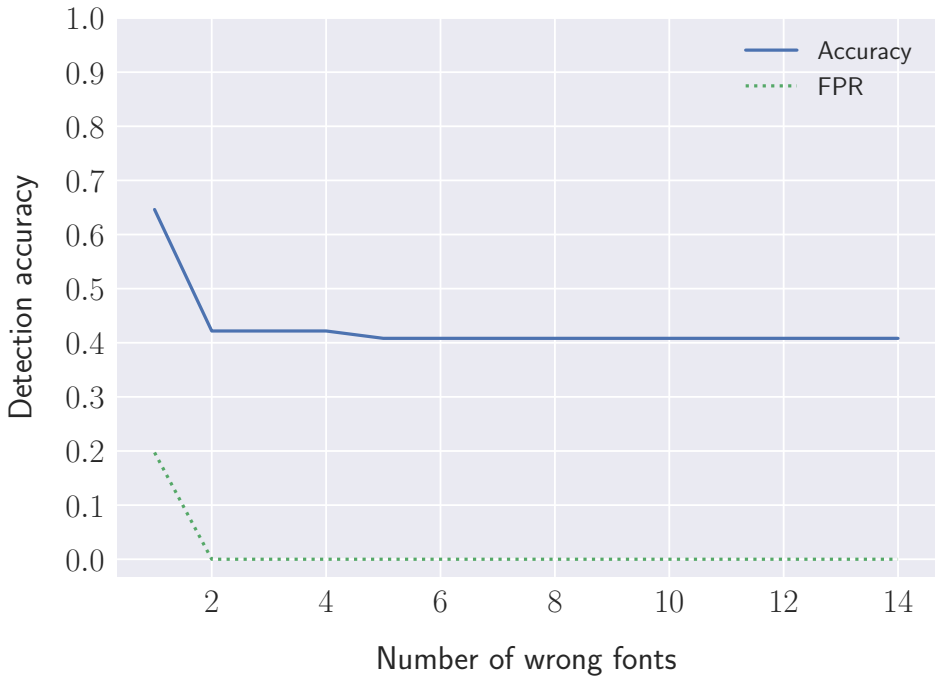


Figure 4.4 Detection accuracy and false positive rate using the fonts test for different values of N_f (number of fonts associated with the wrong OS).

Figure 4.5 Detection accuracy and false positive rate of the browser feature test for different values of N_e (number of wrong features).

Concerning the number of wrong fonts N_f, we obtained an accuracy of 0.646 with a threshold of one font, but this resulted in a false positive rate of 0.197. Thus, we chose a value of $N_f = 2$ fonts, which makes the accuracy of the test decrease to 0.42 but provides a false positive rate of 0.

Finally, concerning the number of browser features N_e, increasing the threshold resulted in a decrease of the accuracy, and an increase of the false negative rate. Nevertheless, only the false negative and true positve rates are impacted, not the false positive rate that remains constant for the different values of N_e. Thus, we chose a value of $N_e = 1$

Even if the detection accuracy of the tests may seem low—0.42 for the fonts and 0.51 for the browser features—these are only two tests among multiple tests, such as the `media queries`, `WebGL` or `toDataURL` that can also be used to verify the authenticity of the information provided in the user agent or in the canvas.

4.2.2 Evaluating FP-Scanner

4.2.2.1 Building a Browser Fingerprints Dataset

To collect a relevant dataset of browser fingerprints, we created a web page that includes the browser fingerprinting script we designed. Besides collecting fingerprints, we also collect the system ground truth—*i.e.*, the real os, browser family and version, as well as the list of countermeasures installed. In the scope of our experiment, we consider countermeasures listed in Table 4.6, as they are representative of the diversity of strategies we reported in the state-of-the-art (Section 2.3). Although other academic countermeasures have been published [45, 69, 7, 66], it was not possible to consider them due to the unavailability of their code or because they could not be run anymore. Moreover, we still consider Random Agent Spoofer even though it is not available as a web extension—*i.e.*, for Firefox versions > 57—since it modifies many attributes commonly considered by browser fingerprinting countermeasures.

We built this browser fingerprints dataset by accessing this web page from different browsers, virtual machines and smartphones, with and without any countermeasure installed. The resulting dataset is composed of 147 browser fingerprints, randomly challenged by 7 different countermeasures. Table 4.6 reports on the number of browser fingerprints per countermeasure. The number of browser fingerprints per countermeasure is different since some countermeasures are deterministic in the way they operate. For example, Canvas Defender always adds a uniform noise on all the pixels of a canvas.

Table 4.6 Comparison of accuracies per countermeasures

Countermeasure	Number of fingerprints	Accuracy FP Scanner	Accuracy FP-JS2 / Augur
Random Agent Spoofer (RAS)	69	**1.0**	0.55
User agent spoofers (UAs)	22	**1.0**	0.86
Canvas Defender	26	**1.0**	0.0
Firefox protection	6	**1.0**	0.0
Canvas FP Block	3	**1.0**	0.0
FPRandom	7	**1.0**	0.0
Brave	4	**1.0**	0.0
No countermeasure	10	**1.0**	1.0

On the opposite, some countermeasures, such as Random Agent Spoofer, add more randomness due to the usage of real profiles, which requires more tests.

4.2.2.2 Measuring the Accuracy of FP-Scanner

We evaluate the effectiveness of Fp-Scanner, FingerprintJS2 and Augur to correctly classify a browser fingerprint as *genuine* or *altered*. Our evaluation metric is the accuracy, as defined in Section 4.2.1. Overall, Fp-Scanner reaches an accuracy 1.0 against 0.45 for FingerprintJS2 and Augur, which perform equally on this dataset. When inspecting the Augur and FingerprintJS2 scripts, and despite Augur's obfuscation, we observe that they seem to perform the same tests to detect inconsistencies. As the number of fingerprints per countermeasure is unbalanced, Table 4.6 compares the accuracy achieved per countermeasure.

We observe that Fp-Scanner outperforms FingerprintJS2 to classify a browser fingerprint as *genuine* or *altered*. In particular, Fp-Scanner detects the presence of canvas countermeasures while FingerprintJS2 and `Augur` spotted none of them.

4.2.2.3 Analyzing the Detected Countermeasures

For each browser fingerprint, Fp-Scanner outputs the result of each test and the value that made the test fail. Thus, it enables us to extract some kinds of signatures for different countermeasures. In this section, we execute Fp-Scanner in `depth` mode—*i.e.*, for each fingerprint, Fp-Scanner executes all of the steps, even if an inconsistency is

detected. For each countermeasure considered in the experiment, we report on the steps that revealed their presence.

User Agent Spoofers are easily detected as they only operate on the user agent. Even when both values of user agent are changed, they are detected by simple consistency checks, such as platform for the OS, or function's internal representation test for the browser.

Brave is detected because of the side effects it introduces, such as blocking canvas fingerprinting. Fp-Scanner distinguishes Brave from a vanilla Chromium browser by detecting it overrides `navigator.plugins` and `navigator.mimeTypes` getters. Thus, when Fp-Scanner analyzes Brave's navigator prototype to check if any properties have been overridden, it observes the following output for plugins and mimeTypes getters string representation: `() => { return handler }`. Moreover, Brave also overrides `navigator.mediaDevices.enumerateDevices` to block devices enumeration, which can also be detected by Fp-Scanner as it returns a `Proxy` object instead of an object representing the devices.

Random Agent Spoofer (RAS) By using a system of profiles, RAS aims at introducing fewer inconsistencies than purely random values. Indeed, RAS passes simple checks, such as having identical user agents or having a user agent consistent with `navigator.platform`. Nevertheless, Fp-Scanner still detects inconsistencies as RAS only ensures consistency between the attributes contained in the profile. First, since RAS is a Firefox extension, it is vulnerable to the media query technique. Indeed, if the user is on a Windows device, or if the profile selected claims to be on Windows, then the OS inconsistency is directly detected. In the case where it is not enough to detect its presence, plugins or fonts linked to the OS enables us to detect it. Browser inconsistencies are also easily detected, either using function's internal representation test or errors attributes. When only the browser version was altered, Fp-Scanner detects it by using the combination of Modernizr and Caniuse features.

RAS overrides most of the navigator attributes from the Firefox configuration file. However, the `navigator.vendor` attribute is overridden in JavaScript, which makes it detectable. Fp-Scanner also detects devices which claimed to be mobile devices, but whose accelerometer value was undefined.

Firefox fingerprinting protection standardizes the user agent when the protection is activated and replaces it with `Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:52.0) Gecko/20100101 Firefox/52.0`, thus lying about the browser version and

the operating system for users not on Windows 7 (Windows NT 6.1). While OS-related attributes, such as `navigator.platform` are updated, other attributes, such as `webgl vendor` and `renderer` are not consistent with the OS. For privacy reasons, Firefox disabled OS-related media queries presented earlier in this chapter for its versions > 57, whether or not the fingerprinting protection is activated. Nevertheless, when the fingerprinting protection is activated, Firefox pretends to be version 52 running on Windows 7. Thus, it should match the media query `-moz-os-version` for Windows 7, which is not the case. Additionally, when the browser was not running on Windows, the list of installed fonts was not consistent with the OS claimed.

Canvas poisoners including Canvas Defender, Canvas FP Block and FPRandom were all detected by Fp-Scanner. For the first two, as they are browser extensions that override canvas-related functions using JavaScript, we always detect that the function `toDataURL` has been altered. For all of them, we detect that the canvas pixel constraints were not enforced from our canvas definition. Indeed, we did not find enough occurrences of the color $(255, 102, 0, 100)$, but we found pixels with a slightly different color. Moreover, in case of the browser extensions, we also detected an inconsistent number of transparent pixels as they apply noise to all the canvas pixels.

Table 4.7 summarizes, for each countermeasure, the steps of our test suite that detected inconsistencies. In particular, one can observe that Fp-Scanner leverages the work of Nikiforakis *et al.* [4] by succeeding to detect a wider spectrum of fingerprinting countermeasures that were previously escaped by their test suite (*e.g.*, canvas extensions, FPRandom [9] and Brave [109]). We also observe that the tests to reveal the presence of countermeasures are consistent with the tests presented in Table 4.4.

4.2.2.4 Recovering the Ground Values

Beyond uncovering inconsistencies, we enhanced Fp-Scanner with the capability to restore the ground value of key attributes, like `OS`, `browser family` and `browser version`. To recover these attributes, we rely on the hypothesis that some attributes are harder to spoof, and hence more likely to reflect the true nature of the device. When Fp-Scanner does not detect any inconsistency in the browser fingerprint, then the algorithm simply returns the values obtained from the user agent. Otherwise, it uses the same tests used to spot inconsistencies, but to restore the ground values:

- **OS value** To recover the real OS, we combine multiple sources of information, including plugins extensions, WebGL renderer, media queries, and fonts linked to

Table 4.7 Fp-Scanner steps failed by countermeasures

Test (scope)	***RAS***	***UA spoofers***	***Canvas extensions***	***FPRandom***	***Brave***	***Firefox***
`User Agents` (global)						
`Platform` (OS)		✓				
`WebGL` (OS)	✓	✓				✓
`Plugins` (OS)	✓	✓				
`Media Queries (OS, browser)`	✓	✓				✓
`Fonts` (OS)	✓					✓
`Error` (browser)	✓	✓				
`Function representation (browser)`	✓					
`Product` (browser)	✓	✓				
`Navigator` (browser)	✓				✓	
`Enumerate devices (browser)`					✓	
`Features` (browser)	✓	✓				
`Events` (device)	✓	✓				
`Sensors` (device)	✓	✓				
`ToDataURL` (canvas)	✓		✓			
`Pixels` (canvas)			✓	✓	✓	✓

OS. For each step, we obtain a possible OS. Finally, we select the OS that has been predicted by the majority of the steps;

- **Browser family** Concerning the browser family, we rely on function's internal representation (`eval.toString().length`) that we combine with the value of `productSub`. Since these two attributes are discriminatory enough to distinguish most of the major browsers, we do not make more tests;
- **Browser version** To infer the browser version, we test the presence or absence of each Modernizr feature for the recovered browser family. Then, for each browser version, we count the number of detected features. Finally, we keep a list of versions with the maximum number of features in common.

Evaluation. We applied this recovering algorithm to fingerprints altered only by countermeasures that change the OS or the browser—*i.e.*, RAS, *User agent spoofers* and Firefox fingerprinting protection. Fp-Scanner was able to correctly recover the browser ground value for 100 % of the devices. Regarding the OS, Fp-Scanner was always capable of predicting the OS family—*i.e.*, Linux, MacOS, Windows—but often failed to recover the correct version of Windows, as the technique we use to detect the version of Windows relies on Mozilla media queries, which stopped working after version 58, as already mentioned. Finally, Fp-Scanner failed to faithfully recover the browser version. Given the lack of discriminatory features in Modernizr, Fp-Scanner can only recover a range of candidate versions. Nevertheless, this could be addressed by applying natural language processing on browser release notes in order to learn the discriminatory features introduced for each version.

4.2.3 Benchmarking FP-Scanner

This part evaluates the overhead introduced by Fp-Scanner to scan a browser fingerprint. The benchmark we report has been executed on a laptop having an Intel Core i7 and 8 GB of RAM.

Performance of FP-Scanner. We compare the performance of FP-Scanner with FingerprintJS2 in term of processing time to detect inconsistencies. First, we automate Chrome headless version 64 using Pupeteer and we run 100 executions of FingerprintJS2. In case of FingerprintJS2, the reported time is the sum of the execution time of each function used to detect inconsistencies—*i.e.*, `getHasLiedLanguages`, `getHasLiedResolution`, `getHasLiedOs` and `getHasLiedBrowser`. Then, we execute

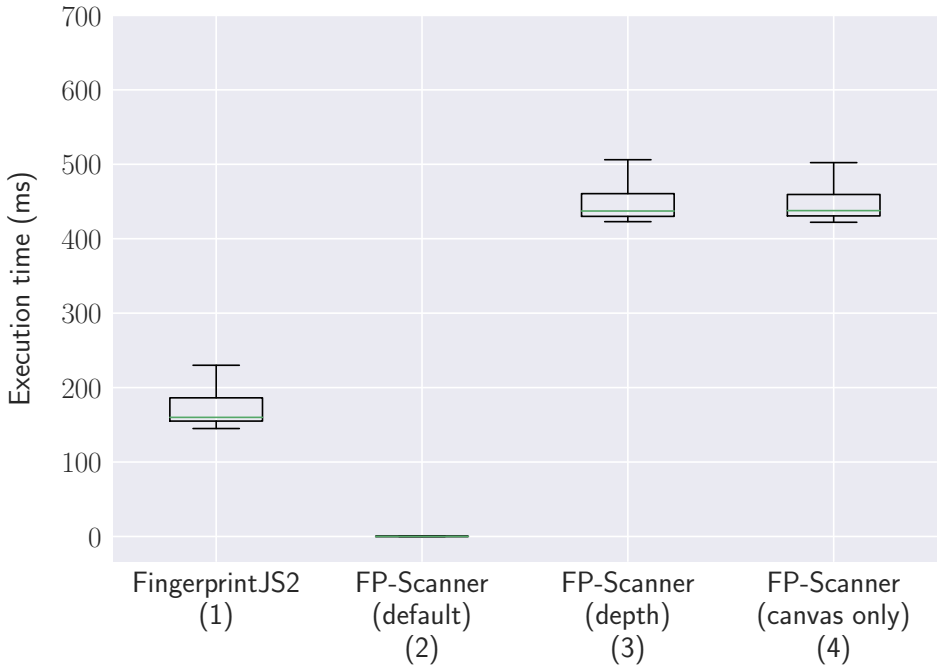


Figure 4.6 Execution time of FingerprintJS2 inconsistency tests and Fp-Scanner with different settings.

different versions of FP-Scanner on our dataset. Input datasets, such as the Caniuse features file, are only loaded once, when Fp-Scanner is initialized. We start measuring the execution time after this initialization step as it is only done once. Depending on the tested countermeasure, Fp-Scanner may execute more or less tests to scan a browser fingerprint. Indeed, against a simple user agent spoofer, the inconsistency might be quickly detected by checking the two user agents, while it may require to analyze the canvas pixels for more advanced countermeasures, like FPRandom. Thus, in Figure 4.6, we report on 4 boxplots representing the processing time for the following situations:

1. FingerprintJS2 inconsistency tests,
2. The scanner stops upon detecting one inconsistency (Fp-Scanner (default) mode),
3. All inconsistency tests are executed (Fp-Scanner (depth) mode),
4. Only the test that manipulates the canvas (`pixels`) is executed (Fp-Scanner (canvas only) mode).

One can observe that, when all the tests are executed (3)—which corresponds to genuine fingerprints—90% of the fingerprints are processed in less than 513ms. However, we observe a huge speedup when stopping the processing upon the first occurrence of an

inconsistency (2). Indeed, while 83 % of the fingerprints are processed in less than 0.21 ms, the remaining 17 % need more than 440 ms. This is caused by the fact that most of the fingerprints we tested had installed countermeasures that could be detected using straightforward tests, such as media queries or testing for overridden functions, whereas the other fingerprints having either no countermeasures or FPRandom (17 fingerprints), require to run all the tests. This observation is confirmed by the fourth boxplot, which reports on the performance of the pixel analysis step and imposes additional processing time to analyze all the canvas pixels. We recall that the pixel analysis step is required only to detect FPRandom since even other canvas countermeasures can be detected by looking at the string representation of `toDataURL`. Thus, when disabling the pixel analysis test, Fp-Scanner outperforms FingerprintJS2 with a better accuracy (> 0.92) and a faster execution (90th percentile of 220 ms).

Based on this evaluation, we can conclude that adopting an inconsistency test suite like Fp-Scanner in production is a viable solution to detect users with countermeasures.

4.3 Discussion

In this chapter, we demonstrated that state-of-the-art fingerprinting countermeasures could be detected by scanning for inconsistencies they introduce in browser fingerprints. We first discuss the privacy implications of such a detection mechanism and then explain how these techniques could be used to detect browser extensions in general.

4.3.1 Privacy Implications

We identify and present the two main privacy implications that can arise from the use of browser fingerprinting countermeasures: discrimination and tracking.

4.3.1.1 Discrimination

Being detected with a countermeasure could lead to discrimination. For example, Hannak *et al.* [110] demonstrated that some websites adjust prices depending on the user agent. Moreover, many websites refuse to serve browsers with ad blockers or users of the Tor browser and network. We can imagine users being delivered altered content or being denied access if they do not share their true browser fingerprint. Similarly to ad

blocker extensions, discrimination may also happen with a countermeasure intended to block fingerprinting scripts.

4.3.1.2 Trackability

Detecting countermeasures can, in some cases, be used to improve tracking. Nikiforakis *et al.* [4] talk about the counterproductiveness of using user agent spoofers because they make browsers more identifiable. We extend this line of thought to more generally argue that being detected with a fingerprinting countermeasure can make browsers more trackable, albeit this is not always the case. We assert that the ease of tracking depends on different factors, such as being able to identify the countermeasure, the number of users of the countermeasure, the ability to recover the real fingerprint values, and the volume of information leaked by the countermeasure. To support this claim, we analyze how it could impact the countermeasures we studied in this chapter.

Anonymity Set. In the case of countermeasures with large user bases, like Firefox with fingerprinting protection or Brave, although their presence can be detected, these countermeasures tend to increase the anonymity set of their users by blocking different attributes, and, in the case of Firefox, by sharing the same user agent, platform, and timezone. Since they are used by millions of users at the time we wrote this chapter, the information obtained by knowing that someone uses them does not compensate the loss in entropy from the removal of fingerprinting attributes. Nevertheless, for countermeasures with small user bases, such as Canvas Defender ($21k$ downloads on Chrome, $5k$ on Firefox) or RAS ($160k$ downloads on Firefox), it is unlikely that the anonymity gained by the countermeasures compensate the information obtained by knowing that someone uses them.

Increasing Targetability. In the case of RAS, we show that it is possible to detect its presence and recover the original browser and OS family. Also, since the canvas attribute has been shown to have high entropy, and that RAS does not randomize it nor block it by default, the combination of few attributes of a fingerprint may be enough to identify a RAS user. Thus, under the hypothesis that no, or few, RAS users have the same canvas, many of them could be identified by looking at the following subset of attributes: `being a RAS user`, `predicted browser`, `predicted OS`, and `canvas`.

Blurring Noise. In the case of Canvas Defender, we show that even though they claim to have a safer solution than other canvas countermeasure extensions, the way they

operate makes it easier for a fingerprinter to track their users. Indeed, Canvas Defender applies a uniform noise vector on all pixels of a canvas. This vector is composed of 4 random numbers between -10 and 30 corresponding to the red, green, blue and alpha (rgba) components of a color. With a small user base, it is unlikely that two or more users share both the same noise and the same original canvas. In particular, the formula hereafter represents the probability that two or more users of Canvas Defender among k share the same noise vector, which is similar to the birthday paradox: $1-\prod_{i=1}^{k}(1-\frac{1}{40^4-i})$. Thus, if we consider that the $21k$ Chrome users are still active, there is a probability of 0.0082 that at least two users share the same noise vector. Moreover, by default Canvas Defender does not change the noise vector. It requires the user to trigger it, which means that if a user does not change the default settings or does not click on the button to update the noise, she may keep the same noise vector for a long period. Thus, when detecting that a browser has Canvas Defender installed, which can be easily detected as the string representation of the `toDataURL` function leaks its code, if the fingerprinting algorithm encounters different fingerprints with the same canvas value, it can conclude that they originate from the same browser with high confidence. In particular, we discovered that Canvas Defender injects a script element in the DOM (cf. Listing 4.1). This script contains a function to override canvas-related functions and takes the noise vector as a parameter, which is not updated by default and has a high probability to be unique among Canvas Defender users. By using the JavaScript Mutation observer API [111] and a regular expression (cf. Listing 4.2), it is possible to extract the noise vector associated to the browser, which can then be used as an additional fingerprinting attribute.

```
function overrideMethods(docId, data) {
  const s = document.createElement('script')
  s.id = getRandomString();
  s.type = "text/javascript";
  const code = document.createTextNode('try{('+overrideDefaultMethods+
     )(' +data.r+ ','+data.g + ',' + data.b + ','+data.a+',"'+s.id +'
     ","'+storedObjectPrefix+'");}catch(e){console.error(e);}');
  s.appendChild(code);
  var node = document.documentElement;
  node.insertBefore(s, node.firstChild);
  node[docId] = getRandomString();
}
```

Listing 4.1 Script injected by Canvas Defender to override canvas-related function

```
var o = new MutationObserver((ms) => {
  ms.forEach((m) => {
```

```
    var script = "overrideDefaultMethods";                                    3
    if (m.addedNodes[0].text.indexOf(script) > -1) {                          4
      var noise = m.addedNodes[0].text.match(/\d{1,2},\d{1,2},\d{1,2},\       5
          d{1,2}/)[0].split(",");
    }                                                                         6
  });                                                                         7
});                                                                           8
o.observe(document.documentElement, {childList:true, subtree:true});          9
```

Listing 4.2 Script to extract the noise vector injected by CANVAS DEFENDER

Protection Level.. While it may seem more tempting to install an aggressive fingerprinting countermeasure—*i.e.*, a countermeasure, like RAS, that blocks or modifies a wide range of attributes used in fingerprinting—we believe it may be wiser to use a countermeasure with a large user base even though it does not modify many fingerprinting attributes. Moreover, in the case of widely-used open source projects, this may lead to a code base being audited more regularly than less adopted proprietary extensions. We also argue that all the users of a given countermeasure should adopt the same defense strategy. Indeed, if a countermeasure can be configured, it may be possible to infer the settings chosen by a user by detecting side effects, which may be used to target a subset of users that have a less common combination of settings. Finally, we recommend a defense strategy that either consists in blocking the access to an attribute or unifying the value returned for all the users, rather than a strategy that randomizes the value returned based on the original value. Concretely, if the value results from a randomization process based the original value, as does CANVAS DEFENDER, it may be possible to infer information on the original value.

4.3.2 Perspectives

In this chapter, we focused on evaluating the effectiveness of browser fingerprinting countermeasures. We showed that these countermeasures can be detected because of their side-effects, which may then be used to target some of their users more easily. We think that the same techniques could be applied, in general, to any browser extension. Starov *et al.* [42] showed that browser extensions could be detected because of the way they interact with the DOM. Similar techniques that we used to detect and characterize fingerprinting countermeasures could also be used for browser extension detection. Moreover, if an extension has different settings resulting in different fingerprintable side effects, we argue

that these side effects could be used to characterize the combination of settings used by a user, which may make the user more trackable.

4.3.3 Threats to Validity

A possible threat lies in our experimental framework. We did extensive testing of Fp-Scanner to ensure that browser fingerprints were appropriately detected as altered. Table 4.7 shows that no countermeasure failed the steps unrelated to its defense strategy. However, as for any experimental infrastructure, there might be bugs. We hope that they only change marginal quantitative results and not the quality of our findings. However, we make the dataset, as well as the algorithm, publicly available online[11], making it possible to replicate the experiment.

We use a ruleset to detect inconsistencies even though it may be time-consuming to maintain an up-to-date set of rules that minimize the number of false positives while ensuring it keeps detecting new countermeasures. Moreover, in this chapter, we focused on browser fingerprinting to detect inconsistencies. Nonetheless, we are aware of other techniques, such as TCP fingerprinting [29], that are complementary to our approach.

Fp-Scanner aims to be general in its approach to detect countermeasures. Nevertheless, it is possible to develop code to target specific countermeasures as we showed in the case of Canvas Defender. Thus, we consider our study as a lower bound on the vulnerability of current browser fingerprinting countermeasures.

4.4 Conclusion

In this chapter, we identified a set of attributes explored by Fp-Scanner to detect inconsistencies and to classify browser fingerprints into 2 categories: *genuine* fingerprints and *altered* fingerprints by a countermeasure. Thus, instead of taking the value of a fingerprint for granted, fingerprinters could check whether attributes of a fingerprint have been modified to escape tracking algorithms, and apply different heuristics accordingly.

To support this study, we collected 147 browser fingerprints extracted from browsers using state-of-the-art fingerprinting countermeasures and we showed that Fp-Scanner was capable of accurately distinguishing genuine from altered fingerprints. We measured the overhead imposed by Fp-Scanner and we observed that both the fingerprinter and the test suite impose a marginal overhead on a standard laptop, making our approach

feasible for use by fingerprinters in production. Finally, we discussed how the possibility of detecting fingerprinting countermeasures, as well as being capable of predicting the ground value of the browser and the OS family, may impact user privacy. We argued that being detected with a fingerprinting countermeasure does not necessarily imply being tracked more easily. We took as an example the different countermeasures analyzed in this chapter to explain that tracking vulnerability depends on the capability of identifying the countermeasure used, the number of users having the countermeasure, the capacity to recover the original fingerprint values, and the information leaked by the countermeasure.

Although Fp-Scanner is general in its approach to detect the presence of countermeasures, using Canvas Defender as an example, we show it is possible to develop countermeasure-specific code to extract more detailed information.

In the first two contribution chapters of this thesis, I showed that browser fingerprinting is a threat to privacy. Nevertheless, in the next chapter, I show that browser fingerprinting can also be applied to increase security on the web. In particular, I focus on how fingerprinting can be used to detect crawlers in addition to other approaches, such as CAPTCHAs. After I measure the use of fingerprinting for crawler detection among popular websites, I study the detection techniques they use and show some similarities with the approaches proposed in this chapter for detecting fingerprinting countermeasures.

Chapter 5

FP-Crawlers: Evaluating the Resilience of Browser Fingerprinting to Block Adversarial Crawlers

In 2017, bot traffic represented more than 40% of the traffic on the web [112]. Bots are used for various purposes ranging from ad-fraud, to automatically creating social media fake accounts to spread malware. Bots are also used to automatically gather data on the web, such as competitor prices, or to steal a website content. In this chapter, I focus on crawlers—*i.e.*, bots specialized in the collection of data available on the web. While some of them collect data in agreement of the websites they crawl, it is not the case of the majority that often infringe the terms of service.

To protect websites against unwanted crawlers, different techniques have been proposed, such as CAPTCHAs or techniques that rely on features extracted from a series of HTTP requests. In this chapter, I study how browser fingerprinting can be used to complement existing crawler detection techniques. Indeed, browser fingerprinting addresses some of the weaknesses of state-of-the-art crawler detection techniques:

- Contrary to CAPTCHAs, browser fingerprinting does not require any user interaction;
- Contrary to methods based on HTTP requests or time series analysis [76], fingerprinting requires a single request to decide whether or not a client is a crawler.

In Section 5.1, I first study the adoption of browser fingerprinting for crawler detection. I crawl the Alexa Top 10K and identify 291 websites that block crawlers and show

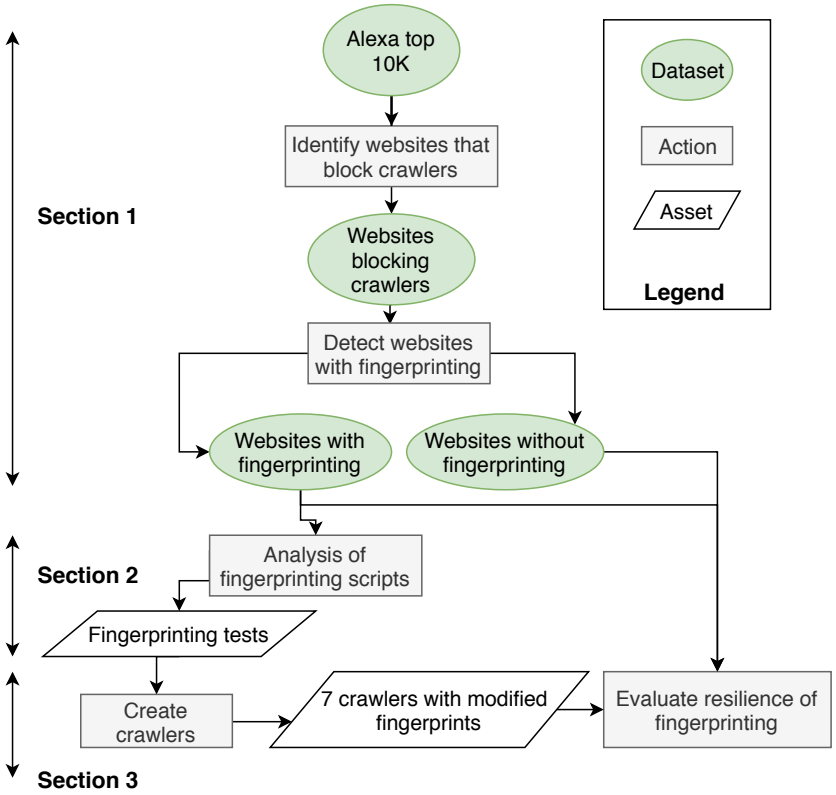


Figure 5.1 Overview of FP-Crawlers: In Section 5.1 I crawl the Alexa's Top 10K to measure the ratio of websites using fingerprinting for crawler detection. In Section 5.2, I explain the key fingerprinting techniques they use. Finally, in Section 5.3 I evaluate the resilience of fingerprinting against adversary crawlers.

that browser fingerprinting is used by 31.96 % of these websites (93 websites). Then, in Section 5.2, I report on the key crawler detection techniques implemented by the major fingerprinters and show that they use similar techniques to the ones presented in Chapter 4 used to reveal the presence of fingerprinting countermeasures. In Section 5.3, I evaluate the resilience of these fingerprinting scripts against adversarial crawlers that try to hide their identity to bypass security checks. I show that, while browser fingerprinting is a good candidate for crawler detection, it can be bypassed by an adversary having some knowledge on the fingerprints collected. In Section 5.4, I discuss the limits and the challenges of browser fingerprinting for crawler detection. Finally, I provide a conclusion to this chapter in Section 5.5.

Figure 5.1 provides an overview of this chapter and the different contributions along sections.

5.1 Detecting Crawler Blocking and Fingerprinting Websites

In this section, we describe our experimental protocol to classify websites that adopt browser fingerprinting to detect and block crawlers.

This protocol is composed of two main steps:

1. **Detecting websites that block crawlers.** From Alexa's Top 10K, we identify the websites that detect and block crawlers. We consider that these websites at least use the user agent in the HTTP headers to detect and block the crawlers. The subset of detected websites then provides us an oracle that we use to evaluate the resilience of browser fingerprinting (cf. Section 5.3);
2. **Detecting websites that use fingerprinting.** Among the websites that block crawlers (step 1), we detect the ones that use fingerprinting for crawler detection and the ones that either do not use fingerprinting, or use fingerprinting but not to detect crawlers. We then use the set of websites that use fingerprinting to make our analysis (cf. the Section 5.2).

5.1.1 Detecting Websites Blocking Crawlers

We first identify websites from Alexa's Top 10K that block crawlers based on their user agent.

Crawler 1: ***easy to detect.*** For each website of the Alexa Top 10K, a first crawler (crawler 1) visits the homepage of the website and then browses up to 4 random links of the same domain accessible from the home page. We crawl only 4 links since the crawler can be easily identified by its user agent. We consider that a page is loaded when there is no more than two active network connections for at least $500\,ms$ (`networkidle2` event of the Puppeteer library). If the page is not loaded after $30\,seconds$, we consider it failed to load and we add the link of the page to a queue of links that are retried at the end of the crawl. If a page is loaded, the crawler waits for $3\,seconds$, dumps the downloaded HTML, and takes a screenshot of the rendered page. Crawler 1 is based on the Chromium headless version bundled with the Puppeteer library [113] and is instrumented using Puppeteer. Using a Chrome headless based crawler enables to crawl the majority of the websites as it supports modern features found in common non-headless browser. We do not modify the crawler user agent, allowing it to

be detected solely because this: `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome /72.0.3582.0 Safari /537.36`. Thus, we can identify websites that block crawlers when these crawlers do not try to hide their identity. Although the blocking decision might not be based on the crawler's entire fingerprint, but mostly on its user agent, it still provides an indication concerning the way the website reacts to crawlers. Since Chrome headless is a popular solution for crawling and has replaced older headless browsers, such as PhantomJS, we consider it unlikely that websites that aim at blocking crawlers do not detect Chrome headless. Moreover, its user agent has been added to popular user agent lists used for crawler detection in 2017 [114]. We also make the hypothesis that websites that try to block bots using advanced approaches, such as traffic shape analysis, are also likely to use lists of crawlers. Indeed, these lists have no false positives and enable to block crawlers before they can even load a single page. Thus, it protects websites against large distributed campaigns where each crawler visit only a few pages.

Crawler 2: *verify errors.* In parallel, on a second machine that uses another IP address, a second crawler (crawler 2) visits the homepage of each website of the Alexa Top 10K. Contrary to crawler 1, this crawler modifies its user agent to pretend to be a vanilla Chrome browser. We use this second crawler to verify, in case of errors encountered by crawler 1, that the website is lying about the error in order to stop the crawler, or if it really exhibits an error. On each home page, crawler 2 waits for the page to be loaded (using the same criteria as the first crawler), waits 3 seconds, dumps the HTML and takes a screenshot.

Labeling screenshots. Then, we label websites as blocking crawlers or not. To do so, we developed a web interface that displays the screenshots taken by crawler 1 and crawler 2, side by side, for each visited URL. Each URL is assigned 1 of 3 possible labels:

1. `"not blocked"`: we consider that the crawler has not been blocked if the page does not show any sign of blocking, such as an explicit message or a CAPTCHA;
2. `"blocked"`: we consider that the crawler has been blocked if the page reports some obvious symptom of blocking, such as a CAPTCHA not linked to any form or a message indicating that we are blocked.Moreover, in the case an error reports that the page is not available or that the website is down in the screenshot taken by crawler 1, we verify if it is actually the case using the screenshot taken by crawler 2. If only crawler 1 has an error, then we consider it has been blocked;
3. `"unknown"`: corresponds to cases where we cannot assess with certainty the crawler has been blocked. This situation can occur because the page timed-out. Moreover,

sometimes both crawler 1 and crawler 2 report a `403` error. In this case, we verify that the website always returns a `403` error for human users. To do so, we manually visit the website using a computer with a residential IP address, which has not been used for the experiment and we verify if the website keeps returning the `403` error. If it still returns a `403` error, then we classify the URL as `"unknown"` as it blocks all users, and not only crawlers. Otherwise, the URL is classified as `"blocked"`.

We consider that a website blocks crawlers if a crawler has been blocked on at least one of its pages.

5.1.2 Detecting Websites that Use Fingerprinting

In the second phase, we focus on the websites we labeled as `"blocked"` because they block crawlers. We crawl these websites to classify them as either using fingerprinting for crawler detection or not.

Crawler modifications. We apply two modifications to the crawler's fingerprint to escape detection based on HTTP headers. Indeed, in order to detect if a website uses fingerprinting, the crawler needs to load and execute the JavaScript included in the pages. First, we modify the user agent to look like a user agent from a vanilla Chrome. Second, we add an `accept-language` header field as it is not included by default in Chrome headless [115].

The crawler visits each home page of the websites identified as blocking crawlers and, for each website, visits up to 3 randomly selected links on the same domain. We visit only 3 links as the goal of this crawl is to detect websites using fingerprinting on popular pages that can easily be reached from their home page. Thus, it does not aim at detecting if the website use fingerprinting on specific sensitive pages, such as login or payment pages. We consider the same heuristics as in the previous step to check if a page is loaded. If the page fails to load, it is added to a queue of URLs that are retried at the end of the crawl. For each visited URL, the crawler records the attributes commonly accessed in browser fingerprinting [10, 28, 6]. To record the access, we inject a JavaScript snippet executed in each page to override the default behavior of getters and functions, and that stores, for each script of the page, when it accesses an attribute or calls a function. We override attributes commonly associated to browser fingerprinting in the literature, such as the properties of the `navigator` and the `screen` objects. We also override functions related to canvas, audio, and WebGL fingerprinting. Finally, we monitor access

to attributes considered by security fingerprinting scripts, such as `window._phantom` or `navigator.webdriver`, which are known to belong to crawlers. We explain the role of these attributes in more details in the next subsection. We monitor these attributes to detect if a fingerprinting script tries to detect crawlers or if it uses fingerprinting for tracking purposes. A complete list of the attributes and functions monitored is available in Appendix A.

Finally, we consider that a website uses fingerprinting for crawler detection if:

1. There is at least one script on one of its pages that called one or more functions related to canvas, WebGL, audio or WebRTC fingerprinting;
2. This script also accesses one crawler-related attribute, such as `window._phantom` or `navigator.webdriver`;
3. This script also retrieves at least 12 fingerprinting attributes.

We adopt this definition since there is no clear agreement on how to characterize fingerprinting, in particular when used in the context of crawler detection. For example, Acar *et al.* [10] consider font enumeration as a good indicator of fingerprinting. However, as we show in the next section, it is not the most discriminant feature for crawler detection. Englehardt *et al.* [6] do not define fingerprinting in general, but detect if functions used for complex fingerprinting attributes, such as canvas, canvas font enumeration or WebRTC fingerprinting are used. Thus, our definition of fingerprinting ensures that a script accesses a sufficient number of fingerprinting attributes, in particular attributes considered as strong indicators of fingerprinting, such as canvas. As we study fingerprinting for crawler detection, we also add a constraint to check that the script must access at least one crawler-related attribute, given that these are widely known and show intent to block crawlers [116].

5.2 Analyzing Fingerprinting Scripts

In this section, we first study the ratio of websites that adopt browser fingerprinting for crawler detection. Then, we present the detection techniques implemented by the main fingerprinters present on the Alexa Top 10K.

5.2.1 Describing our Experimental Dataset

The different crawls to detect websites that block crawlers and websites using fingerprinting were conducted in December 2018.

Sites blocking crawlers. Among the 10,000 crawled websites, we identified 291 websites that block crawlers (2.91%). The median Alexa's rank of websites blocking crawlers is 4,946, against 5,001 for websites that do not block crawlers. If we compare the two distributions of the Alexa's rank among websites that block crawlers and those that do not, using a Kolmogorov Smirnov test [117], we obtain a p-value of 0.324. This statistical test indicates that there is no significant difference in the distribution of the rank of websites that block crawlers and websites that do not block crawlers.

Fingerprinting attributes. For each website that blocks crawlers, we study the number of fingerprinting attributes they access. For a given website, we look at the script that accesses the maximum number of distinct fingerprinting attributes. The median number of distinct fingerprinting attributes accessed is 12. 10 % of the websites access more than 33 distinct fingerprinting attributes. Concerning crawler-related attributes, such as `navigator.webdriver` or `window._phantom`, 51.38 % of the websites do not access even one of such attributes, while 10% of them access 10 crawler attributes. Based on our definition of browser fingerprinting, we classified 93 websites as using fingerprinting for crawler detection, which represents 31.96% of the websites that block crawlers.

Diversity of fingerprinting scripts. We group fingerprinting scripts by the combination of attributes they access. In total, we observe 20 distinct groups among websites blocking crawlers. While each group is constituted of scripts from the same company, we also observe that some companies are present in different clusters, as they have multiple versions of their script. In order to analyze the main fingerprinting techniques used for crawler detection, we focus on the scripts of 4 fingerprinting companies as they represent more than 90 % of the scripts among the websites that block crawlers. Since these companies have multiple versions of their script, we chose the script that accesses the greatest distinct number of fingerprinting attributes. We decided not to disclose the names of these companies since it does not contribute to the understanding of fingerprinting and our findings could be used by crawler developers to specifically target some websites. Moreover, for copyright reasons, we cannot distribute the original scripts nor their deobfuscated version. We discuss this point and other ethical issues in Section 5.4.3.

In the remainder of this Section, we present the techniques used by the 4 main fingerprinting scripts to detect crawlers. These scripts collect fingerprinting attributes and then either perform a detection test directly in the browser or transmit the fingerprints to a server that applies heuristics to perform the detection test. Table 5.1 provides an overview of the different attributes and tests. In particular for each given attribute and script, there are three possible values:

1. ✓ indicates that the script collects the attribute and test it in the browser—*i.e.* its value is explicitly verified or we know the attribute is used for the detection because of the evaluation conducted in Section 5.3;
2. ∼ indicates that the script collects the attribute, but no test is run directly in the script. This means that the fingerprinter may use the value collected to run the test on the server side. The empirical evaluation we conduct in Section 5.3 help us to understand if some of the attributes are used on the server side to detect inconsistencies;
3. The absence of symbol indicates that the attribute is not collected by the script.

It should be noted that the 4 scripts we analyze are obfuscated. We then cannot use the names of the variables or functions to infer their purpose. Thus, we use different techniques, such as open source fingerprinting libraries [65], the state-of-the-art literature, as well as an empirical evaluation conducted in Section 5.3 to explain how different combinations of collected attributes are used to detect crawlers.

5.2.2 Detecting Crawler-Specific Attributes

The first and most widely used detection technique in the 4 fingerprinting scripts relies on the presence of specific attributes injected into the JavaScript execution context or in the HTML DOM by headless browsers or instrumenting frameworks, such as Selenium. For example, in the case of Google Chrome or Firefox, it is possible to detect if a browser instance is automated by checking if the attribute `webdriver` is included in the `navigator` object and if its value is equal to `true`. The scripts test for the presence of the following properties, which used to be added to the `document` object by older versions of Selenium: 1. `__fxdriver _unwrapped`, 2. `__selenium _unwrapped`, and 3. `__webdriver_script_fn`.

Older versions of Selenium and its associated plugins also used to inject variables into the `window` object, such as `_Selenium_IDE_Recorder`, `callSelenium`, or `_selenium`. While all the scripts test for the presence of these variables, they are not added anymore

Table 5.1 Different fingerprinting tests associated with the scripts that use them. The symbol ✓ indicates the attribute is collected and that a verification test is run directly in the script. The ∼ symbol indicates that the attribute is collected but there is no verification test in the script.

		Scripts			
	Name of the test	1	2	3	4
	Crawler-related attributes	✓	✓	✓	✓
Browser	productSub	∼	∼	∼	✓
	eval.toString()	✓			✓
	Error properties	✓			✓
	Browser-specific/prefixed APIs	✓	✓	✓	✓
	Basic features	✓		✓	✓
	Different feature behaviour				✓
	CSS features	✓			
	Codecs supported		✓		
	HTTP headers	∼	✓	∼	✓
OS	Touch screen support	∼	∼	∼	✓
	Oscpu and platform	∼	∼	∼	✓
	WebGL vendor and renderer	∼	∼		∼
	List of plugins	∼	∼	∼	✓
	List of fonts		∼	∼	
	Screen dimensions	✓	∼	∼	✓
	Overriden attributes/functions	✓	✓	✓	
Other	Events	∼			∼
	Crawler trap	✓			
	Red pill				✓
	Audio fingerprint	∼			
	Canvas fingerprint	∼	∼	∼	
	WebGL fingerprint	∼	∼		
	WebRTC				∼

by the most recent versions of Selenium [118] or its associated plugins, such as Selenium IDE [119]. Besides Selenium, the scripts also detect headless browsers and automation libraries, such PhantomJS and NightmareJS, by checking the presence of specific attributes in the `window` object: `_phantom`, `callPhantom`, `phantom` or `__nightmare`.

While the presence of any of these attributes provides a straightforward heuristic to detect crawlers with certainty, these attributes can be easily removed, or renamed to escape detection. Thus, we describe more robust detection techniques, based on browser fingerprint inconsistencies, to overcome this limitation. We structure the presentation of the inconsistencies searched by fingerprinters into four categories and we present a fifth family of common non-fingerprinting tests found in the fingerprinting scripts: 1. browser and version inconsistencies, 2. OS inconsistencies, 3. screen inconsistencies, 4. overridden functions inconsistencies, 5. other tests.

5.2.3 Checking Browser Inconsistencies

Commercial fingerprinting scripts also leverage the notion of fingerprint inconsistency presented in Chapter 4 to detect automated browsers. The first set of inconsistency verifications found across the 4 fingerprinting scripts aim at verifying if the browser claimed in the user agent has been altered. Before we present the different tests used to verify the nature of a browser, we provide a brief overview of what inconsistencies are and how they can help reveal crawlers.

Fingerprint inconsistencies. Similarly to the detection test suite I proposed in the previous chapter, fingerprinters also use inconsistencies to detect combinations of attributes that cannot be found in the wild for non-headless or non-automated browsers. In the case of crawlers, inconsistencies can occur in different situations:

- When crawler developers alter the user agent of their crawlers to not be detected. By doing this, it may introduce inconsistencies between the browser and the OS claimed in the user agent, and the different attributes composing the fingerprint;
- When a crawler is based on a headless browser. Because headless browsers do not always implement all the features of modern browsers, or they implement them differently, they can introduce inconsistencies between the browser claimed in the user agent and the features it should expose.

5.2.3.1 Explicit browser consistency tests

One of the scripts implements tests similar to the function `getHasLiedBrowser` proposed by the open source FingerprintJS2 library [65]:

productSub. First, it extracts the browser from the user agent and verifies if it has a consistent `navigator.productSub` value. While originally this attribute returned the build number of the browser, it always returns `20030107` on Chromium-based browsers or Safari and it returns `20100101` on Firefox;

eval.toString. Then, it runs the following snippet of code: `eval.toString().length`, which returns the length of the string representation of the native `eval` function. While on Safari and Firefox it is equal to 37, on Internet Explorer it is equal to 39 and it is equal to 33 on Chromium-based browsers;

Error properties. It throws an exception and catches it to analyze the properties of the error. While some of the properties of the `Error` objects, such as `message` and `name` are standard accross different browsers, some of them such as `toSource` exist only in Firefox. Thus, the script verifies that if the `toSource` property is present in the error, then the browser is Firefox.

5.2.3.2 Feature Detection

The 4 scripts test the presence of different features. The previous set of tests was the same as the `getHasLiedBrowser` function of the FingerprintJS2 library. Thus, we did not need to infer its purpose. Here, we present how different features tested across the 4 fingerprinting scripts can be used to reveal inconsistencies in the nature of the browser and its versions, even when the tests are not executed in the fingerprinting scripts.

Browser-specific and browser-prefixed APIs. Instead of relying on the user agent string to infer the nature of a browser, it is possible to test for the presence of specific features linked to certain browsers and certain versions [37]. In particular, all the scripts test for the presence of the `window.chrome` object, a utility for extension developers available on Chromium-based browsers. This feature can also help to reveal Chrome headless since it does not have this object, even though it is based on Chrome.

To verify if a browser is Safari, one of the scripts tests for the presence of the `pushNotification` function in `window.safari`. For Opera, the script verifies the presence of `window.opera`. Finally, for Firefox, it verifies if the `InstallTrigger` variable is defined, and for Internet Explorer, it verifies the value returned by `eval("/*@cc_on!@*/false")`. The latter test

relies on conditional compilation, a feature available in old versions of Internet Explorer but not in recent browsers. Thus, the code returns true on Internet Explorer and false on modern browsers.

One of the scripts tests for the presence of features whose name depends on the browser vendor. For example, it verifies that the function `requestAnimationFrame` is present together with `msRequestAnimationFrame` or `webkitRequestAnimationFrame`.

Basic features. Two of the studied scripts verify the presence of the `bind` function. While this test does not help in detecting recent headless browsers, it used to detect PhantomJS [120] as it did not have this function. Another script collects the first 100 properties of the `window` object, returned by `Object.keys` to verify if they are consistent with the browser claimed. Finally, one of the scripts tests a set of 18 basic features, such as creating or removing event listeners using `addEventListener` and `removeEventListener`. It also tests other APIs that have been available in mainstream browsers for a long time, such as `Int8Array` [121], which have been included since Internet Explorer 10, or the `MutationObserver` [122] API, available since Internet Explorer 11. Since, the majority of these features are present in all the recent versions of mainstream browsers such as Chrome, Safari or Firefox, they can be used to detect non-standard or headless browsers that do not implement these features.

Different feature behavior. Even when a feature is present, its behavior may depend on the browser. For example, in a blog post [123], I showed that Chrome Headless fails to handle permissions [124] in a consistent way. When requesting permissions using two different techniques, as showed in listing 5.1—`Notification.permission` and `navigator.permissions.query`—Chrome Headless returns conflicting values, which differs from a vanilla Chrome behavior. One of the scripts exploits this inconsistency to detect crawlers based on Chrome headless.

```
navigator.permissions.query({name:'notifications'})
    .then(function(permissionStatus) {
        if(Notification.permission === 'denied' &&
            permissionStatus.state === 'prompt') {
            console.log('This is Chrome headless')
        } else {
            console.log('This is not Chrome headless')
        }
    });
```

Listing 5.1 Checking if permissions are consistent.

Another feature whose behavior depends on the browser and whether or not it is in headless mode is the image error placeholder. When an image cannot be loaded, the browser replaces it with a placeholder. Nevertheless, in the early versions of Chrome headless, there was no placeholder [123]. Thus, these versions of Chrome headless can be detected because the width and the height of the placeholder are equal to 0 pixels. One of the scripts detects this by creating an image whose `src` attribute points to a random URL that does not exist and then measures the size of the placeholder. Since the size also depends on the browser, it can be used to verify its nature:

- On Chromium-based browsers it measures `16x16` pixels and its size does not depend on the zoom level,
- On Safari it measures `20x20` pixels and its size depends on the zoom level,
- On Firefox it measures `24x24` pixels and its size does not depend on the zoom level.

CSS features. One of the scripts we studied also collects the CSS properties applied to the body element using the `getComputedStyle` function. Similarly to JavaScript features that are browser-dependent, such as `webkit/ms RequestAnimationFrame`, it is possible to test if some of the browser-dependent CSS features are consistent with the browser claimed in the user agent.

5.2.3.3 Audio & Video Codecs

One of the scripts tests for the presence of different audio and video codecs. To do so, it creates an audio and a video element on which it applies the `canPlayType` method to test the availability of audio and video codecs presented in Tables 5.2 and 5.3. The `canPlayType` function returns 3 possible values:

1. `"probably"`, which means that the media type appears to be playable,
2. `"maybe"` indicates that it is not possible to tell if the type can be played without playing it,
3. `""`, an empty string indicating that the type cannot be played.

Tables 5.2 and 5.3 report on the audio and video codecs supported by vanilla browsers. Tables are based on the dataset from Caniuse [125–131], as well as data collected on the personal website of the thesis author. We can observe that some codecs are not supported by all browsers, which means that they can be used to check the browser claimed in the user agent.

Table 5.2 Support of audio codecs for the main browsers.

Audio codec	Chrome	Firefox	Safari
ogg vorbis	probably	probably	""
mp3	probably	maybe	maybe
wav	probably	maybe	maybe
m4a	maybe	maybe	maybe
aac	probably	maybe	maybe

Table 5.3 Support of video codecs for the main browsers.

Video codec	Chrome	Firefox	Safari
ogg theora	probably	probably	""
h264	probably	probably	probably
webm vp8	probably	probably	""

5.2.3.4 HTTP headers

Contrary to JavaScript and CSS features, which are collected in the browser, HTTP headers are collected on the server side. Thus, we cannot directly observe if fingerprinters collect these headers. Nevertheless, because of side-effects, such as being blocked, we observe that all fingerprinters collect at least the user agent header. Moreover, we also detect that 2 of the fingerprinters test for the presence of the `accept-language` header. Indeed, by default, Chrome headless does not send this header. In the evaluation, we show that its absence enables some of the fingerprinters to block crawlers based on Chrome headless.

5.2.4 Checking OS Inconsistencies

Only one script among the four performs an explicit OS verification. Nevertheless, it does not mean that others do not conduct such tests on the server side using attributes collected by the fingerprinting script or using other techniques, such as TCP fingerprinting [29]. However, while users with fingerprinting countermeasures, such as spoofers, may be tempted to lie about their OS in order to increase their privacy [17], it is not required by crawlers, which only need to hide their nature to escape detection. Thus, even though verifying OS consistency can catch crawlers that modified the OS displayed in the user agent, it does not help against crawlers that only modify the nature of the browser.

5.2.4.1 Explicit OS consistent tests

The set of tests conducted by the only fingerprinter that verifies the OS in its script is similar to the `getHasLiedOs` function of the library FingerprintJS2 [65]. It extracts the OS claimed in the user agent to use it as a reference and then runs the following set of tests:

1. **Touch screen verification.** It tests if the device supports touch screen by verifying the following properties: the presence of the `ontouchstart` property in the object `window` and `navigator.maxTouchPoints` or `navigator.msMaxTouchPoints` are greater than `0`. If the device claims to have touch support, then it should be running one of the following operating systems: Windows Phone, Android or iOS;
2. **Oscpu and platform.** `Oscpu` is an attribute, only available on Firefox, that returns a string representing the platform on which the browser is executing. The script verifies that the OS claimed in the user agent is consistent with the `navigator.oscpu` attribute. For example, if a platform attribute indicates that the device is running on `arm`, then the OS should be Android or Linux. Similarly, if the platform is `iPad` or `iPhone`, then the OS should be `iOS`. They also conduct similar tests with the `navigator.platform` attribute.

Only one fingerprinter runs the above set of tests directly in its script. Nevertheless, the other three fingerprinting scripts also collect information about the presence of a touch screen, `navigator.platform` and `navigator.oscpu`. Thus, they may run similar verifications on the server side.

WebGL information. Three of the scripts use the WebGL API to collect information about the vendor and the renderer of the graphic drivers. As we explained in Chapter 4, these values are linked to the OS and can be used to verify OS consistency [17]. For example, a renderer containing `"Adreno"` indicates the presence of an Android device, while a renderer containing `"Iris OpenGL"` reveals the presence of MacOS. One of the scripts also verifies if the renderer is equal to `"Mesa OffScreen"`, which is one of the values returned by the first versions of Chrome headless [132, 123].

List of plugins. The four scripts collect the list of plugins using the `navigator.plugins` property. While some of the plugins are browser dependent and can be used to verify the claimed browser, they can also be used to verify the OS [17]. Indeed, the extension of the filename of a plugin provides indications about the current OS. Plugin file extensions should be `.so` on Linux, `.plugin` on Mac and `.dll` on Windows.

List of fonts. Two of the fingerprinting scripts collect a list of fonts using JavaScript font enumeration [4]. While it can be used to increase the uniqueness of the fingerprint [39], it can also be used to reveal the underlying OS [105, 17] since some fonts are only found on specific OSes by default.

5.2.5 Checking Screen Inconsistencies

The four scripts collect information related to the screen and window sizes. In particular, they all collect the following attributes:

- `screen.width/height`
- `screen.availWidth/Height`
- `screen.colorDepth`
- `window.innerWidth/Height`
- `window.devicePixelRatio`

For example, the `screen.width` and `screen.height` represent the width and the height of the web-exposed screen, respectively. The `screen.availWidth` and `screen.availHeight` attributes represent the horizontal and vertical space in pixels available to the window, respectively. Thus, one of the scripts verifies that the available height and width are always less than (in case there is a desktop toolbar) or equal to the height and the width. Another property used to detect some headless browsers is the fact that, by definition, `window.outerHeight/Width` should be greater than `window.innerHeight/Width`. Nevertheless, one should be careful when using this test since it does not hold on iOS devices [133] where the `outerHeight` is always equal to `0`.

5.2.5.1 Overriden Inconsistencies

Crawler developers may be aware of the detection techniques presented in this section and try to hide such inconsistencies by forging the expected responses—*i.e.*, providing a fingerprint that could come from a vanilla browser, and thus not be detected as a crawler. To do so, one solution is to intercept the outgoing requests containing the fingerprint to modify them on the fly, however, this cannot always be easily done when scripts are carefully obfuscated and randomized. Another solution is to use JavaScript to override the functions and getters used to collect the fingerprint attributes. However, when doing

this, the developer should be careful to hide the fact she is overriding native functions and attributes. If not, checking the string representation of the functions will reveal that a native function has been intentionally overridden. While a standard execution of `functionName.toString()` returns a string containing `native code` in the case of a native function, it returns the code of the new function if it has been overridden.

We observe that all the scripts check fingerprinting functions, such as `getImageData` used to obtain a canvas value or the `WebRTC` class constructor, have been overridden. In particular, they verify functions, such as `setTimeout`, `requestAnimationFrame` and `bind`. Beyond native functions, one of the scripts also checks if native objects, such as `navigator.geolocation`, have been overridden. By default, the `toString` representation of the geolocation object is `"[object Geolocation]"`. Nevertheless, if a developer overrides the function that returns a geolocation object, its string representation will be similar to any other object: `"[object Object]"`.

Detection using side effects. Besides looking at the string representation of native functions and objects, one script goes further by verifying the value returned by a native function. It verifies that the `getImageData` function used to collect the value of a canvas has been overridden by looking at the value of specific pixels.

5.2.6 Other Non-fingerprinting Attributes

Events. Crawlers may programmatically generate fake mouse movements and fake clicks to simulate human behavior and fool behavioral analysis detection systems. To detect such events, two of the fingerprinting scripts check that events originate from human actions. If an event has been generated programmatically, the browser sets its `isTrusted` property to `false`. Nevertheless, this approach does not help in detecting crawlers automated using Selenium or the Chrome DevTools protocol, since the events they generate are considered trusted.

Crawler trap. One script creates a crawler trap using an invisible link with the `"nofollow"` property and appends a unique random identifier to the URL pointed to by the link. Thus, if a user selects the link or loads the URL, it can be identified as a crawler that does not respect the `nofollow` policy.

Red pill. One script collects a red pill similar to the one presented by Ho *et al.* [70] to test if a browser is running in a virtual machine or an emulated device. The red pill exploits performance differences caused by caching and virtual hardware.

WebRTC. WebRTC is an API that enables *Real Time Communication* in the browser. One of the scripts uses the WebRTC API to collect the IP address of the user. Indeed, it can be used to obtain the public IP address of a user, even when she is behind a proxy or a VPN [6].

Other fingerprinting techniques. Beyond the features reported in this section, all the scripts collect common attributes identified in the browser fingerprinting literature, such as WebGL, canvas and audio fingerprinting. Fingerprinters only collect a hash of these values to save the cost of storing raw values. Besides one of the scripts that tests the value of a given pixel in a canvas to verify if it has been altered, none of the other scripts extract any features that could be used to assess some sort of consistency. In the next section, we show that these attributes do not influence the detection of crawlers. Our main hypothesis is that these attributes may be used for short-term tracking purposes in order to identify crawlers that may change their IP address [134].

In this section, we showed that 291 sites from the Alexa Top 10K block crawlers using the user agent. Among these, 93 websites (31.96%) that use fingerprinting for crawler detection. They use different techniques that leverage attributes added by automated browsers or fingerprint inconsistencies to detect crawlers.

5.3 Detecting Crawler Fingerprints

In this section, we first evaluate the effectiveness of browser fingerprinting to detect crawlers. Then, we study the resilience of browser fingerprinting against an adversary who alters its fingerprint to escape detection.

5.3.1 Experimental Protocol

Ground truth challenge. The main challenge to evaluate crawler detection approaches is to obtain ground truth labels to assess the evaluation. The typical approach to obtain labels is to request experts from the field who check raw data, such as fingerprints and HTTP logs, and use their knowledge to label these samples. The main problem of this approach is that labels assigned by the experts are as good as the current knowledge of the experts labeling the data. Similarly to machine learning models that struggle to generalize to new data, these experts may be good at labeling old crawlers they have already encountered, but not at labeling new kinds of crawlers they are unaware of,

which may artificially increase or decrease the performance of the approach evaluated. To address this issue, we decide to use ourselves different kinds of crawlers on websites that have been identified as blocking crawlers. Thus, no matter how the crawler tries to alter its fingerprint, we can always assert that it is a crawler because it is under our control. Then, in order to measure the effectiveness of fingerprinting for crawler detection, we rely on the fact that the crawled websites have been identified as websites that block crawlers. We consider that, whenever they detect a crawler, they will block it. We use this blocking information as an oracle for the evaluation. We discuss the limits of this hypothesis in Section 5.4.3. A solution to obtain the ground truth would have been to subscribe to the different bot detection services. Nevertheless, besides the significant cost, bot detection companies verify the identity of their customers to ensure it is not used by competitors trying to reverse engineer their solution or by bot creators trying to obtain an oracle to maximize their ad-fraud incomes for example.

5.3.1.1 Crawler Family

In order to evaluate the resilience of fingerprinting, we send 7 different crawlers that incrementally modify their fingerprints to become increasingly more difficult to detect. Table 5.4 presents the crawlers and the attributes they modify. The first six crawlers are based on Chrome headless for the following reasons:

1. It has become a popular headless browser for crawling. Since its first release, the once popular PhantomJS stopped being maintained [**?**];
2. It implements the majority of features present in popular non-headless browsers, making therefore its detection more challenging compare to older headless browsers;
3. Had we use older headless browsers such as PhantomJS (not maintained since March 2018) and SlimerJS (works only with Firefox version < 59 released in 2017), crawlers would have been easily detected because of the lack of modern web features [120].

The last crawler is based on a vanilla Chrome browser. We use this crawler to better understand why blocking occurs, and to assess that crawlers are blocked because of their fingerprint. Indeed, since this crawler is based on a vanilla Chrome, the only difference in its fingerprint is the `navigator.webdriver` attribute. Once this attribute is removed, it can no longer be detected through fingerprinting.

We restrict the evaluation to 7 different crawlers. Ideally, a perfect protocol would randomly mutate fingerprint attributes to provide a fine-grained understanding. However, this was not feasible in practice, as our evaluation requires residential IP addresses of

Table 5.4 List of crawlers and altered attributes.

Crawler	Attributes modified
Chrome headless based	
Crawler 1	User agent
Crawler 2	Crawler 1 + `webdriver`
Crawler 3	Crawler 2 + `accept-language`
Crawler 4	Crawler 3 + `window.chrome`
Crawler 5	Crawler 4 + `permissions`
Crawler 6	Crawler 5 + screen resolution + codecs + touch screen
Vanilla Chrome based	
Crawler 7	`webdriver`

which we have a limited supply, as well as the exponential complexity resulting from testing all attribute permutations on the set of evaluated websites. While we could have used residential proxy services to acquire more residential IP addresses, this approach still has several drawbacks:

1. **Ethical issues.** Mi *et al.* [135] showed that a majority of the devices proposed by residential proxy services, such as Luminati, did not give their consent, which raises ethical issue since we may pollute the reputation of their IP address;
2. **Inconsistencies.** Since these residential proxy services do not provide mechanisms to ensure the nature of the device that will act as a proxy, there can be inconsistencies between the browser fingerprint of our crawlers and the TCP, TLS, and HTTP fingerprints of the proxy, making it more difficult to understand why a crawler was blocked.

Details of the modified attributes. Crawlers 2 to 6 build on the previous one, adding new modifications each time to increase the difficulty of detection. For example, crawler 4 implements the changes made by crawlers 1, 2 and 3.

1. Crawler 1 is based on Chrome headless with a modified user agent to look like a vanilla Chrome user agent;
2. In the case of Crawler 2, we delete the `navigator.webdriver` property;
3. By default, Chrome headless does not add an `accept-language` header to its requests. Thus, for Crawler 3, we add an `accept-language` header whose value is set to `"en-US"` for all the requests sent by the crawler;
4. Crawler 4 injects a `chrome` property to the `window` object;

5. For Crawler 5, we override the management of the permissions for the notifications to hide the inconsistency exposed by Chrome headless [123]. Since we override the behavior of native functions, we also override their `toString` method, as well as `Function.prototype.toString`—*i.e.*, the `toString` of the `Function` type in order to hide our changes;
6. For Crawler 6, we apply modifications related to the size of the screen, the availability of touch support and the codecs supported by the browser. First, we override the following properties of the `window` object: `innerWidth/Height`, `outerWidth/Height` and `window.screenX/Y`. We also modify properties of the `screen` object: `availWidth/Height` and `width/height`. By default, Chrome headless simulates touch screen support even when Chrome headless is running on a device that does not support it. To emulate a desktop computer without touch support, we override the `document.createEvent` function so that it throws an exception when trying to create a `TouchEvent`. We also override `navigator.maxTouchPoints` to return `0` and we delete the `ontouchstart` property of the `window` object. We also lie about the codecs supported to return the same value as a vanilla Chrome, by overriding the `canPlayType` function for both the `HTMLAudioElement` and `HTMLVideoElement`. In order to hide changes made to native functions, we override their `toString`;
7. Contrary to the first six crawlers, Crawler 7 is based on a vanilla Chrome—*i.e.*, non-headless. Thus, we only remove the `webdriver` attribute from the `navigator` object.

5.3.1.2 Evaluation Dataset

We present how we select websites used for the evaluation.

Cross-domain detection. Since we want to evaluate fingerprinting for crawler detection, we try to eliminate other detection factors that could interfere with our evaluation. One such factor is cross-domain detection. This occurs when a company provides a crawler detection service that is present on multiple domains being crawled. In this situation, the company can leverage metrics collected on different domains, such as the number of requests, to classify traffic no matter the website. In order to minimize the risk that cross-domain detection interferes with our evaluation, we need to decrease the number of websites that belong to the same company in the evaluation dataset. Thus, there is a tradeoff between the number of websites in the evaluation dataset and the capacity to eliminate other factors, such as cross-domain detection. While to our knowledge, no

research has been published on cross-domain detection, we encountered this phenomenon during the different crawls we conducted. Moreover, during informal discussions with a crawler detection company, engineers also mentioned this practice.

Selection of websites. We group websites identified as blocking crawlers and using fingerprinting (as defined in Section 5.1) based on the combination of fingerprinting attributes they access. We obtain 20 groups of fingerprinting scripts and, for each of the groups, we randomly select one website. We argue this selection is a good tradeoff. Even though it does not totally eliminate cross-domain detection since, as shown in Section 5.2, fingerprinters can have different scripts, it still enables to evaluate all the different fingerprinting scripts present in the dataset. Then, we randomly select 20 websites that block crawlers without using fingerprinting to compare fingerprinting-based detection against other approaches.

Crawling protocol. For each of the 7 crawlers, we run 5 crawls on the previously selected websites. Each crawl is run from a machine with a residential IP address that has not been used for crawling for at least 2 days in order to limit the risk of being blocked because of a bad IP address reputation. Studying how long crawler detection systems block crawlers and how/if the previous history of the IP address has an impact on the duration of the blocking is out of the scope of this study and is left as future work. A crawl consists of the following steps:

a) We randomly shuffle the order of the websites in the evaluation dataset. It enables to minimize and measure the side effects that can occur because of cross-domain detection;
b) For each website, the crawler visits the home page and then visits up to 10 randomly-selected pages from the same domain. As explained later in this section, we crawl only 10 links to ensure we evaluate the effectiveness of browser fingerprinting detection and not the effectiveness of other state-of-the-art detection approaches;
c) Once a page is loaded, the crawler takes a screenshot and stores the HTML of the page for further analysis;
d) Between two consecutive crawled pages, the crawler waits for 15 seconds plus a random time between 1 and 5 seconds.

5.3.1.3 Crawler behaviors

In the previous subsection, we explain how we select websites in a way that minimizes cross-domain detection. Here, we present how we adapt the behavior of the 7 crawlers

so that other approaches, such as rate limiting techniques or behavioral analysis do not interfere with our evaluation. Thus, the crawlers should not be detected by state-of-the-art techniques presented in Section 2.4.2 that rely on the following features:

1. Number of HTTP requests,
2. Number of bytes requested from the server,
3. Number and percentage of HTML requests,
4. Percentage of PDF requests,
5. Percentage of image requests,
6. Duration of a session,
7. Percentage of `4xx` error requests,
8. ROBOTS.TXT file request,
9. Page popularity index,
10. Hidden links.

To address points (1) to (6), crawlers request few pages so that it looks like the requests originate from a human. Moreover, we do not block any resources, such as images or PDFs, nor do we ask for these resources in particular. The crawlers visit only up to 10 pages for a given website. Since the attributes used in fingerprinting are constant on short time periods, such as a crawling session, a fingerprinter does not need multiple pages to detect if a fingerprint belongs to a crawler, which means that this criterion should not affect our evaluation. Moreover, the navigation delay between 2 pages is 15 seconds plus a random delay between 1 and 5 seconds. We chose a mean value of 15 seconds since it has been observed that a majority of users do not stay more than 15 seconds on a page on average [136]. We add some randomness so that if a website measures the time between two requested pages, it does not look deterministic. Points (7) and (9) are addressed by only following internal links exposed from the home page or pages directly linked by the home page, which is more likely to point to both popular and existing pages. To address point (8), the crawlers never request the `Robots.txt` file, which means that we do not take into account the policy of the website concerning crawlers. Nevertheless, since we crawl only a few pages, it should have little impact. We discuss this point in more detail in Section 5.4.3 about ethical considerations.

5.3.2 Experimental Results

5.3.2.1 Presentation of the dataset

In total, we crawl 40 different websites, randomly selected from the list of websites blocking crawlers between December 2018 and January 2019. 22 of them use browser fingerprinting and 18 do not use fingerprinting. Initially, we selected two equal sets of 20 websites *using* and *not using* fingerprinting. Nevertheless, we noticed that 2 of the websites had been misclassified. We did not detect fingerprinting on these websites but we observed the side-effects of cross-domains fingerprinters. Since the crawler used for fingerprinting detection had been detected on some other websites, its IP address was blacklisted. Thus, when the crawler visited other websites with the fingerprinter that blocked it earlier, it was blocked at the first request because of its IP address, without having the possibility to load and execute the JavaScript present on the page. In total, we run 35 crawls—*i.e.*, 5 per crawler—each with a residential IP address that has not been used for at least two days for crawling.

5.3.2.2 Blocking results

Figure 5.2 reports on the results of the crawls for the 7 crawlers. The results have been obtained by labeling the data using the same web interface we used in Section 5.1. For each crawler, we present the average number of times per crawl it is blocked by websites that use fingerprinting and websites that do not use fingerprinting.

Influence of fingerprint modifications. We see that the more changes are applied to the crawler's fingerprint, the less it gets blocked. While Crawler 1 gets blocked 11.8 times on average, the detection falls to 1.0 time for Crawler 6 that applies more extensive modifications to its fingerprint. We also observe an important decrease in the number of times crawlers are blocked between crawlers 1 and 2. It goes from 11.8 for Crawler 1 to 3.6 for Crawler 2. The only difference being the removal of the `webdriver` attribute from the `navigator` object, which means that fingerprinters heavily rely on this attribute to detect crawlers.

Blocking speed. We also analyze the speed at which crawlers get blocked—*i.e.*, after how many pages crawled on a given website a crawler is blocked. Except for Crawler 5 that gets blocked after 3.1 pages crawled on average, crawlers are blocked after they have crawled less than 3 pages of a website on average.

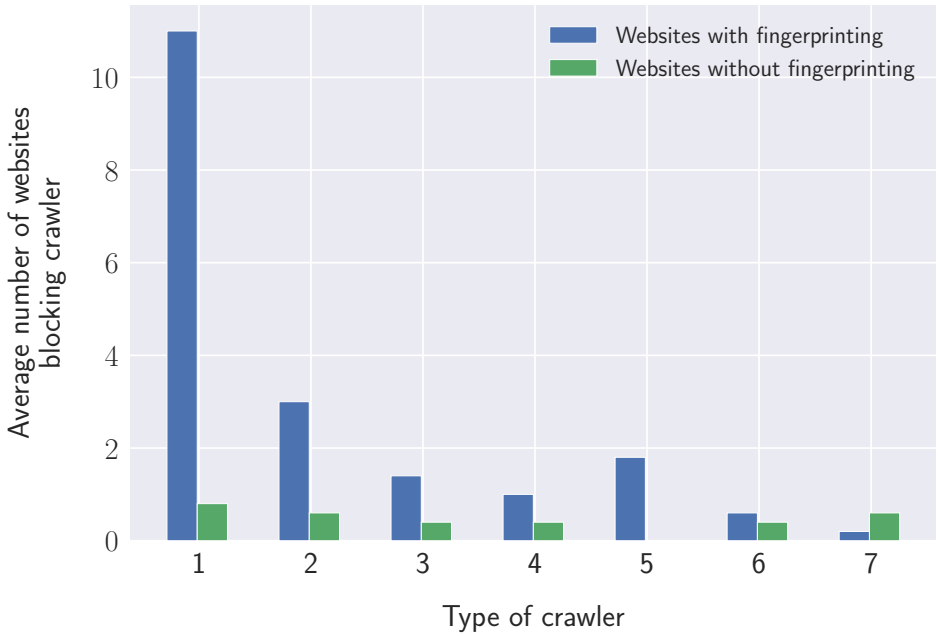


Figure 5.2 For each kind of crawler, we report on the average number of times per crawl it is blocked by websites that use and that do not use fingerprinting.

Fingerprinters detect more crawlers. We also observe that, on average, websites using fingerprinting block more crawlers than websites without fingerprinting. For example, on average, 93.2 % (11.0) of websites blocking Crawler 1 use fingerprinting. The only exception is Crawler 7, where 75 % of the time it gets blocked, it is by a website not using fingerprinting. This is the expected result since Crawler 7 is based on a vanilla Chrome, which means that its fingerprint is not different from the one of a standard browser.

Analysis of other detection factors. The fact that Crawler 7 still gets blocked despite the fact it has a normal fingerprint raises the question of other detection factors used in addition to fingerprinting. Even though we take care to adapt the behavior of the crawlers to minimize the chance they get detected by other techniques, we cannot exclude that it occurs. Thus, we verify if crawlers are detected because of their fingerprint or because of other state-of-the-art detection techniques.

First, we investigate if some of the crawlers have been blocked because of cross-domain detection. To do so, we manually label, for each fingerprinting script in the evaluation dataset, the company it belongs to. Whenever we cannot identify the company, we assign a random identifier. We identify 4 fingerprinters present on more than 2 websites in the evaluation dataset and that could use their presence on multiple domains to do

cross-domain detection. We focus only on websites that blocked Crawlers 4, 5 and 6. Indeed, only one fingerprinting company succeeds to detect Crawlers 4, 5 and 6. Thus, we argue that Crawlers 1, 2 and 3 detected by websites using fingerprinting, are indeed detected because of their fingerprint. If their detection had relied on other techniques, then some of the Crawlers 4, 5, 6 and 7 would have also been blocked by these websites. Moreover, the analysis of the fingerprinting scripts we conduct in Section 5.2 shows that some of these fingerprinters have the information needed to detect Crawlers 1, 2 and 3, but not to detect more advanced crawlers using fingerprinting.

We analyze in more details the only fingerprinter that detected Crawlers 4, 5 and 6. At each crawl, the order of the websites is randomized. Thus, for each crawler and each crawl, we extract the rank of each of the websites that have a fingerprinting script from this company. Then, we test if the order in which websites from this fingerprinter are crawled impact the chance of a crawler to be detected. We observe that crawlers get blocked on websites independently of their rank. For example, Crawler 4 is blocked on the first website crawled where the fingerprinter is present and Crawler 6 on the second one.

Non-stable blocking behavior. We also notice that websites that use the fingerprinting scripts provided by the only fingerprinter that blocked crawlers 4, 5 and 6 do not all behave the same way. Indeed, depending on the website, some of the advanced crawlers have never been blocked. It can occur for several reasons: 1. The websites have different versions of the scripts that collect different attributes; 2. On its website, the fingerprinter proposes different service plans. While some of them are oriented towards blocking crawlers, others only aim at detecting crawlers to improve the quality of the analytics data.

Even on the same website, the blocking behavior is not always stable over time. Indeed, some of the websites do not always block a given crawler. Moreover, some of the websites able to block advanced crawlers do not block crawlers easier to detect. For example, the only website that is able to block both crawlers 5 and 6, only blocked 13 times over the 35 crawls made by all the crawlers. It means that 37.1% of the time, this website did not block crawlers, even though it could have done so. In particular, this website never blocked Crawlers 1 and 2 even though they are easier to detect than Crawlers 5 and 6.

Undetected crawlers. We also observe that some websites could have detected Crawlers 3 and 4 using the information they collected. Indeed, these websites verify the consistency of the notification permission, which as we show in Section **??**, enables

to detect crawlers based on Chrome headless. A possible explanation to why the fingerprinter present on these websites was blocking Crawlers 1, 2, but not Crawlers 3 and 4 is because the first two crawlers can be detected solely using information contained in the HTTP headers (lack of `accept-language` header). However, Crawlers 3 and 4 require information collected in the browser, which may be handled differently by the fingerprinter.

In this section, we showed that fingerprinting helps to detect more crawlers than non-fingerprinting techniques. For example, 93.2% (11 websites) of the websites that have detected crawler 1 use fingerprinting. Nevertheless, the important decrease in the average number of times crawlers are blocked between crawlers 1 and 2, from 11.8 websites to 3.6, indicates that websites rely on simple features such as the presence of the `webdriver` attribute to block crawlers. Finally, we show that only 2.5% of the websites detect crawler 6 that applied heavier modifications to its fingerprint to escape the detection, which shows one of the main flaws of fingerprinting for crawler detection: its lack of resilience against adversarial crawlers.

5.4 Discussion

5.4.1 Limits of Browser Fingerprinting

The analysis of the major fingerprinting scripts shows that it is heavily used to detect older generations of headless browsers or automation frameworks, such as PhantomJS. These browsers and frameworks used to be easily identifiable because of the attributes they injected in the `window` or `document` objects. In addition to these attributes, older headless browsers lacked basic features that were present by default in mainstream browsers, making them easily detectable using feature detection. Since 2017, Chrome headless has proposed a more realistic headless browser that implements most of the features available in a vanilla Chrome. Even though we show that fingerprinters use differences between vanilla Chrome and headless Chrome for detection, it is much harder to find such differences compared to older headless browsers. One of the implications is that, since there are fewer differences, it makes it easier for an adversarial crawler developer to escape detection by altering the fingerprint of her crawlers. Indeed, these changes require few lines of code (less than 300 lines in the case of Crawler 6) and can be done directly using JavaScript without the need to modify and compile a whole Chromium browser.

Unsurprisingly, we also show that fingerprinting-based approaches are totally ineffective against non-headless automated browsers (like Crawler 7), since the fingerprint of such crawlers is the same as the one of a vanilla browser (modulo the `navigator.webdriver` attribute which is easy to remove).

5.4.2 Threats to Validity

While the goal of our study is to evaluate the effectiveness of browser fingerprinting for crawler detection, a threat lies in the possibility that we may have missed external techniques, other than browser fingerprinting and the techniques presented in Section 2.4.2, that could have contributed to the detection of crawlers. A second threat lies in the choice of our oracle—*i.e.*, being blocked by a website when a crawler is detected. While we ensured that all the websites used in the evaluation block crawlers upon detection, it may have been caused by some user agent blacklisting. Thus, we make the hypothesis that, if fingerprinting was also used for crawler detection, then the website would be consistent in its strategy against the crawlers. However, it is possible that a website adopts fingerprinting not against crawlers, but against credit card fraudsters or to label crawlers in its analytics reports, and thus does not focus on blocking crawlers. Finally, a possible threat lies in our experimental framework. We did extensive testing of our code, and we manually verified the data from our experiments. However, as for any experimental infrastructure, there may be bugs. We hope that they only change marginal quantitative results and not the quality of our findings.

5.4.3 Ethical Considerations

Concerning the crawled websites, we asked for permission from our IRB to conduct our study, and we used the data collected only for research purposes—*i.e.*, verifying if we were blocked. Even though we may not have respected the policy of the *robots.txt* file, since we did not read it to avoid triggering crawler detection techniques that could have interfered with our research, we visited only a few pages of each website ranked in the Alexa Top 10K. Thus, we consider it had a negligible impact on their resources. Moreover, we decided not to disclose the name of the websites, as well as the name of the fingerprinters, since we think the benefits to the reader do not outweigh the fact that our findings could be used specifically against these domains in order to bypass their crawler detection.

5.5 Conclusion

Crawler detection has become widespread among popular websites to protect their data. While existing approaches, such as CAPTCHAs or trafic shape analysis, have been shown to be effective, they either require the user to solve a difficult problem, or they require enough data to accurately classify the traffic.

In this chapter, we show that, beyond its adoption for tracking, browser fingerprinting is also used as a crawler detection mechanism. Browser fingerprinting exploits the lack of basic features in some headless browsers or the inconsistencies introduced by crawler developers to detect if a fingerprint belongs to a crawler. We analyze the scripts from the main fingerprinters present in the Alexa Top 10K and show that they exploit the lack of browser features, errors or overridden native functions to detect crawlers. Then, using 7 crawlers that apply different modifications to their fingerprint, we show that websites with fingerprinting are better and faster at detecting crawlers compared to websites that use other state-of-the-art detection techniques. Nevertheless, while 29.5 % of the evaluated websites are able to detect our most naive crawler that applies only one change to its fingerprint, this rate decreases to 2.5% for the most advanced crawler that applies more extensive modifications to its fingerprint. We also show that fingerprinting does not help detecting crawlers based on standard browsers since they do not expose inconsistent fingerprints.

These findings demonstrate the strengths and the limits of fingerprinting for crawler detection. While fingerprinting can help to quickly detect crawlers based on headless or non-standard browsers, it remains unable to detect standard automated browsers. Moreover, we also showed it requires few efforts to develop crawlers with modified fingerprints capable to escape detection. Thus, we argue that fingerprinting provides clear benefits for crawler detection but it should be used in a layered approach, in addition to other crawler detection techniques, such as crawler traps or rate limiting techniques.

Part IV

Final Remarks

Chapter 6

Conclusion

Browsers have become increasingly more complex. Applications that were once possible only as desktop clients can now be accessed from a browser. Moreover, the diversity of devices capable of browsing the web has also steadily increased. While browsers used to run only on computers, they can now run on a wide range of devices ranging from smartphones to connected TVs. This diversity of devices associated with the possibility for browsers to obtain information about it using JavaScript APIs made it possible to identify devices based on their characteristics.

Building a browser fingerprint consists in gathering a set of attributes that can be accessed by any website when a user visits it. Thus, fixing browser fingerprinting tracking is challenging since contrary to cookies that can be simply blocked or deleted, fingerprints rely on APIs that are also used genuinely by websites. Browser fingerprinting is also a constantly evolving field. As new APIs are added to the browser, there is a risk they get used by commercial fingerprinters to create more robust, unique and stable fingerprints. On the other hand, when browser vendors take actions, such as decreasing the precision of an API or blocking its access, this can render certain forms of fingerprinting unusable.

Because of its stateless nature, detecting fingerprinting is also challenging. While detecting stateful tracking mechanisms that rely on cookies or other storage APIs is more straightforward since the tracking identifier is stored in the browser, detecting if a website fingerprints users is more complicated:

- First, there is no formal definition of browser fingerprinting. Does simple analytics become fingerprinting after more than 5, 6, 7 or more attributes have been accessed? While some techniques, such as font enumeration or canvas fingerprinting, are clear

markers of fingerprinting, there is a blurry line where it is unclear if a script collects data such as the user agent, the screen resolution and the number of cores of the device for analytical purposes, or if it is for fingerprinting;

- Secondly, while monitoring the use of known fingerprinting techniques at scale has become easier with modern crawling tools, detecting new and unknown techniques is more challenging as any API or any data obtained from a sensor can be used to create a fingerprint.

Fortunately, browser vendors have started to take privacy issues more into account when creating new features. Saying that it has been only caused thanks to the research on web privacy may be overemphasized. It is also likely the results of other factors, such as the increasing number of data breaches[1] and important privacy scandals like Facebook and Cambridge analytica. These scandals have made of privacy a strong commercial argument for browser vendors and tech companies in general [2]. Nevertheless, studies on privacy and browser fingerprint also played a role in the way browsers have evolved. For example, two days after Acar *et al.* [5] published that a popular social widget was using canvas fingerprinting for tracking, the company behind it decided to stop using canvas fingerprinting.[3] When browser vendors design new APIs, their privacy impact, in particular, the way they could be used for fingerprinting, is also more taken into account. Even though browser vendors take more into account the privacy impact of new APIs, this does not mean new fingerprintable APIs are not added anymore to browsers. For example, in 2017, Chrome and other Chromium-based browser added a new property, `deviceMemory`, to the `navigator` object that provides the memory of the device. Nevertheless, evaluating how a new API could be used for fingerprinting is challenging. Indeed, even APIs that do not explicitly return information about the device can be misused to extract information because of side-effects, as it is the case with the canvas, the audio, and the WebGL APIs. Even increasing the accuracy of timing measurement or adding thread support can be used for fingerprinting:

1. Timers can be used to conduct timing attacks that extract information about the device.,

2. Threads can be used to use run costly operations, such as 3D scenes rendering, without blocking the main thread and, therefore, extract more unique fingerprints.

[1] https://breachlevelindex.com/

[2] https://www.theverge.com/2019/3/14/18266276/apple-iphone-ad-privacy-facetime-bug

[3] https://www.addthis.com/blog/2014/07/23/the-facts-about-our-use-of-a-canvas-element-in-our-recent-rd-test/#.XLcfbpPRZQI

The state of browser fingerprinting has evolved these past years, from techniques heavily relying on plugins such as Flash and Java to techniques that leverage HTML5 APIs and exploit their side effects, such as canvas and audio fingerprinting. In this thesis, I tried to provide an up-to-date picture of browser fingerprinting, both in terms of its impact on privacy, as well as its application to detect crawlers.

6.1 Contributions

6.1.1 FP-Stalker: Tracking Browser Fingerprint Evolutions

I collected a dataset of more than 120K browser fingerprints from 2,346 distinct browsers over two years using the AmIUnique browser extensions. First, I analyzed the stability of browser fingerprints and the different attributes constituting it. My results confirm Eckersley's findings, the majority of fingerprints change frequently. More than half of the browsers in our dataset displayed at least one change in their fingerprint after five days. Nevertheless, I also showed that not all browser fingerprints change at the same pace. Through a two-year study, I also measured the stability of attributes that did not exist at the time Eckersley's study was conducted. In particular, I showed that besides being highly unique, canvas fingerprints are also highly stable. For half of the browsers in our dataset, the canvas remained stable for more than 300 days.

I also evaluated how long could browsers be tracked using only their fingerprint. I proposed two linking algorithms and showed that while a significant fraction of browsers are immune to fingerprinting, either because their fingerprint is not unique or is too similar with fingerprints of other browsers, more than 32% of the browsers can be tracked more than 100 days.

6.1.2 FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies

Several fingerprinting countermeasures claim to generate consistent browser fingerprints, *i.e.*, fingerprints that could be found in the wild. The reason countermeasures aim at generating consistent fingerprints is to avoid being detected, which would make their users more unique and trackable. In Chapter 4, I proposed Fp-Scanner, a test suite that aims at detecting the presence of fingerprinting countermeasures because of the

inconsistencies they introduce. I used it to evaluate 7 countermeasures, ranging from browser extensions that lie about the user device, to more complex peer-reviewed forked browsers that modify the value of canvas and audio fingerprints. Fp-Scanner was able to spot inconsistencies for all the countermeasures, even those claiming to generate consistent fingerprints, making, therefore, their presence detectable. Moreover, since different APIs or techniques can be used to obtain certain attributes, I showed we can correlate them to infer the real OS and browser, even though it has been modified by a countermeasure. Thus, my findings show the difficulty of designing effective countermeasures that do not end up being counterproductive. I argue that while it may not be possible to design undetectable countermeasures, countermeasures' defense strategies should avoid leaking unnecessary information. In particular, the defense strategy should not be tuned since then, the defense strategy could be used as a fingerprinting feature. Moreover, I argue that for countermeasures to be effective, they should be used by enough users, so that knowing that a user has a countermeasure installed does is not discriminatory in itself. Thus, a possible way to make a countermeasure available to many users is to integrate it natively into a browser, the software used to access websites. Since my paper on fingerprinting countermeasures has been published in 2018, several browser vendors, such as Brave, Firefox and Safari, have either added fingerprinting countermeasures natively in their browser or made them more easily available to their users.

6.1.3 FP-Crawlers: Evaluating the Resilience of Browser Fingerprinting to Block Adversarial Crawlers

The use of browser fingerprinting in a security context has often been overlooked. I focus on crawler detection and show that among the 3% of websites of the Alexa Top 10K that block crawlers, around 30% of them use browser fingerprinting. I analyze the detection techniques of the most popular scripts encountered during my crawls and show that while some of the techniques, such as canvas or font enumeration, are similar to the ones used for tracking, these scripts rely on techniques specifically developed for crawler detection. In particular, crawler detection scripts look for traces left by instrumentation frameworks such as Selenium, as well as for inconsistent fingerprints that could reflect the use of a headless browser. To evaluate the effectiveness of fingerprinting for crawler detection, I developed multiple crawlers, each gradually more difficult to detect and show that browser fingerprinting can quickly detect crawlers.

An important challenge of using browser fingerprinting for security lies in the fact that fingerprints are collected on the client-side and that they can, therefore, be modified by an attacker. Thus, I evaluate the resilience of fingerprinting against an adversarial attacker that tries to hide its presence by lying about its fingerprint. My results show that while fingerprinting can detect crawlers with few modifications applied to their fingerprints, it cannot detect crawlers that apply more changes to their fingerprint or crawlers based on non-headless browsers, hence the need of using fingerprinting in addition to other crawler detection approaches.

6.2 Future work

6.2.1 Automating Crawler Detection Rules Learning

Detecting crawlers and bots can benefit several applications ranging from website content protection to ad-fraud detection. In particular, improving fingerprinting-based detection could help to protect against certain forms of attacks that cannot be fully addressed using only other existing detection techniques:

- **Traffic-shape approaches.** In a context of ad-fraud, one cannot wait for bots to visualize many ads from the same IP address before they get blacklisted. Indeed, bots may use residential proxies to frequently change their IP addresses [135];
- **CAPTCHAs.** In a context of ad-fraud, it is also unrealistic to require users to solve CAPTCHAs for them to see ads.

While fingerprinting can help to detect bots, there is a continuous arms race between bot developers trying to make their bots look more human and the fingerprinters that try to detect them. I argue that automating the learning of detection rules could help fingerprinters win the arms race. Indeed, whenever new features are added to a browser, there may a short time window during which the behavior differs between the normal browser and its headless counterpart. During this time window, bot developers may not be aware of these new detection rules and are less likely to have modified the fingerprints of their bots to lie consistently. Thus, I propose to extend the approach proposed by Schwarz *et al.* [72] to automatically learn rules capable of distinguishing non-headless browsers from headless browsers. In particular, I plan to address one of the main challenges they identified: exploring properties hidden behind function calls. To call JavaScript functions with correct parameters, I propose the following two strategies:

1. Use fuzzing techniques to generate valid parameters [137],
2. Crawl specialized programming websites, such as GitHub and StackOverflow, to gather code snippets that use specific APIs, or crawl any websites to monitor the execution of several functions and obtain the value of their arguments at runtime.

6.2.2 Investigate New Fingerprinting Attributes

Due to the difficulty of collecting unbiased fingerprints over a long period, it is challenging to accurately measure the entropy and the stability of browser fingerprints. The latest large scale study was conducted in 2016 [28] on more than 2 million browser fingerprints collected on a French popular website, and focuses only on fingerprint uniqueness. As I explained in the state-of-the-art, the study did not examine several fingerprinting techniques, such as audio fingerprinting and `navigator.enumerateDevices`, that were available at the time the study was conducted. More recently, new APIs that can be used for fingerprinting, such as `navigator.deviceMemory`, have also been added to browsers. Moreover, approaches that did not apply to real-world traffic because of performance issues may have become applicable. For example, Cao *et al.* [1] proposed to generate complex 3D scenes to create stable fingerprints, even across different browsers of the same device. While a few years ago, the rendering of the 3D scenes was blocking the main JavaScript execution threads for more than 10 seconds, now it could leverage the `offscreenCanvas` API [4] to generate the scenes in web workers. [5] Therefore, evaluating the entropy and the stability of these different fingerprinting techniques on real-world traffic would provide valuable information on the current state of browser fingerprinting.

6.2.3 Studying Fingerprinting for Authentication

With an ever-increasing number of data breaches[6], there is a need for additional mechanisms to protect against credential stuffing.[7] Different studies proposed to use browser fingerprinting as a semi-transparent second authentication factor. While the guarantees it provides are not as strong as the one provided by traditional second factors such as

[4] OffscreenCanvas API: https://developer.mozilla.org/en-US/docs/Web/API/OffscreenCanvas
[5] Web Workers API: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API
[6] https://breachlevelindex.com/
[7] https://www.owasp.org/index.php/Credential_stuffing

U2F,[8] it can still be used to enhance security in a relatively transparent manner for the user. Moreover, fingerprinting is also convenient for companies since it does not require to install authentication applications on the employees' mobiles, or to buy external devices, such as Yubikeys that can be costly at the scale of a company. Nevertheless, studies on browser fingerprinting for authentication are limited. While they propose several approaches ranging from the use of the accelerometer sensor to the generation of dynamic canvas, none of them has been evaluated against real-world authentication traffic. Besides providing a more accurate estimation of the security gain provided by browser fingerprinting, evaluating it in a more realistic context would also enable to better understand its usability for end-users, which as shown by several studies [138], often impacts the security.

6.2.4 Developing Web Red Pills

Crawler detection approaches can act at different levels:

1. **Behavioral.** Approaches such as traffic shape analysis and CAPTCHAs analyze a user behavior to determine if it is a human or a bot;

2. **Execution environment.** Detection using device and browser fingerprinting targets the browser or the system the bot is running on to distinguish between humans and bots.

I propose to investigate red pills, sequences of instructions that aim at detecting if a browser is running in a virtualized environment, for bot detection. Indeed, since crawling at scale requires a large infrastructure, a significant fraction of crawlers likely runs on virtual machines from public cloud providers. Ho *et al.* [70] proposed several red pills capable of detecting if the host system is a virtual machine from within the browser. Nevertheless, the paper has been published in 2014 and has not been evaluated on popular public cloud providers. Moreover, the underlying implementations of some of the APIs used in the red pills may have evolved, which can impact the accuracy of the red pills. Thus, I argue there is a need for evaluating these red pills on the main public cloud providers and developing new red pills techniques.

[8] https://www.yubico.com/solutions/fido-u2f/

6.3 Future of Browser Fingerprinting

Recent privacy scandals have made of privacy a strong commercial argument. Firefox and Safari, two popular browsers now natively integrate fingerprinting protections. New privacy-friendly browsers, such as Brave and Cliqz, have also emerged and propose native fingerprinting countermeasures. As I explained in Chapter 4, having enough users is one of the conditions required to have more effective fingerprinting countermeasures that do not make their users more vulnerable to tracking. Thus, this shift from fingerprinting countermeasures running as browser extensions, to countermeasures natively integrated into the browser is a positive change from a privacy point of view. Moreover, new mechanisms, such as the `Feature-Policy` header,[9] also help websites control the APIs that can be accessed in different frames. However, these features do not help to protect against websites that willingly allow third-party to track their users. Browser vendors also tend to more take into account fingerprinting when designing new APIs. Thus, before releasing new APIs, browser vendors thoroughly evaluate how it could be used for fingerprinting. As showed by Gomez *et al.* , these changes helped to decrease the entropy of fingerprinting techniques used in the past.

Nevertheless, fingerprintable APIs are still added to browsers, *e.g.* the `navigator.deviceMemory` that provides information about the device memory was added to Chrome in December 2017.[10] Moreover, even benign APIs, such as canvas and WebGL, can be misused to create highly unique and stable fingerprints. I believe this is likely to occur again as browser vendors keep on adding new features to their browsers.

From static to dynamic fingerprints. For a long time, fingerprinters have built fingerprints constituted of static attributes such as the user agent, the list of plugins or canvas whose value is the same between consecutive executions. With recent progress in machine learning, I argue there is a risk fingerprinters could shift towards more dynamic and less stable attributes, such as the approach proposed by Sanchez *et al.* [41] that measures the performance to execute different cryptographic operations. Due to their instability, these attributes cannot simply be hashed and transformed into a fingerprint. Nevertheless, one can extract features that are later integrated into machine learning models used to link fingerprints of the same browser. This shift also raises challenges concerning the measure of fingerprint uniqueness. Indeed, while entropy and anonymity

[9] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Feature-Policy
[10] deviceMemory: https://developer.mozilla.org/en-US/docs/Web/API/Navigator/deviceMemory

set are meaningful metrics for stable attributes, it is not necessarily the case for attributes whose value constantly changes.

Fingerprinting for security. Current headless browsers used for crawling will probably become more and more similar to their non-headless counterparts. For example, until September 2018, there was no mechanism to manage permissions in Chrome headless using the Chrome DevTools protocol. It was due to the novelty of Chrome headless rather than to a deliberate choice of the developers [139]. Thus, these kinds of inconsistencies will likely be fixed in the future. Nevertheless, I argue that even if headless browsers become more realistic, there is some space for fingerprinting to be used in addition to other crawler detection techniques. Indeed, since browser vendors keep on adding new features, during a certain lapse of time, these new features may not be implemented in the headless browser or they may be implemented in a different way, which could be used for detection.

References

[1] Yinzhi Cao, Song Li, Erik Wijmans, et al. (cross-) browser fingerprinting via os and hardware level features. In *NDSS*, 2017.

[2] Jonathan R Mayer. Any person... a pamphleteer": Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, page 85, 2009.

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

[4] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and privacy (SP), 2013 IEEE symposium on*, pages 541–555. IEEE, 2013.

[5] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 674–689. ACM, 2014.

[6] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1388–1401. ACM, 2016.

[7] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, pages 820–830. International World Wide Web Conferences Steering Committee, 2015.

[8] Peter Baumann, Stefan Katzenbeisser, Martin Stopczynski, and Erik Tews. Disguised chromium browser: Robust browser, flash and canvas fingerprinting protection. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, pages 37–46. ACM, 2016.

[9] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems*, pages 97–114. Springer, 2017.

[10] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In

Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 1129–1140. ACM, 2013.

[11] Thomas Unger, Martin Mulazzani, Dominik Fruhwirt, Markus Huber, Sebastian Schrittwieser, and Edgar Weippl. Shpf: Enhancing http (s) session security with browser fingerprinting. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 255–261. IEEE, 2013.

[12] Furkan Alaca and Paul C van Oorschot. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 289–301. ACM, 2016.

[13] Tom Van Goethem, Wout Scheepers, Davy Preuveneers, and Wouter Joosen. Accelerometer-based device fingerprinting for multi-factor mobile authentication. In *International Symposium on Engineering Secure Software and Systems*, pages 106–121. Springer, 2016.

[14] Davy Preuveneers and Wouter Joosen. Smartauth: dynamic context fingerprinting for continuous user authentication. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 2185–2191. ACM, 2015.

[15] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. Picasso: Lightweight device class fingerprinting for web clients. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 93–102. ACM, 2016.

[16] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. In *IEEE S&P 2018-39th IEEE Symposium on Security and Privacy*, pages 1–14. IEEE, 2018.

[17] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-scanner: The privacy implications of browser fingerprint inconsistencies. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[18] Antoine Vastel, Walter Rudametkin, and Romain Rouvoy. Fp-tester: Automated testing of browser fingerprint resilience. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 103–107. IEEE, 2018.

[19] Antoine Vastel, Peter Snyder, and Benjamin Livshits. Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking. *arXiv preprint arXiv:1810.09160*, 2018.

[20] Antoine Vastel. Repository of the code used in fp-stalker., 2017. URL https://github.com/Spirals-Team/FPStalker.

[21] Antoine Vastel. Repository of the code used in fp-scanner., 2018. URL https://github.com/Spirals-Team/FP-Scanner.

[22] Antoine Vastel. Repository of the code used in fp-crawler., 2018. URL https://github.com/antoinevastel/exp_fp_bot.

[23] Antoine Vastel. Open source implementation of a picasso-like canvas fingerprinting algorithm., 2019. URL https://github.com/antoinevastel/picasso-like-canvas-fingerprinting.

[24] Antoine Vastel. Fp-collect: Fingerprinting script of the fp-scanner library., 2018. URL https://github.com/antoinevastel/fp-collect.

[25] Antoine Vastel. Fp-scanner: a browser fingerprinting-based bot detection library., 2018. URL https://github.com/antoinevastel/fpscanner.

[26] Jean-Samuel Beuscart and Kevin Mellet. Business models of the web 2.0: Advertising or the tale of two stories. *Communications & Strategies, Special Issue*, 2008.

[27] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 878–894. IEEE, 2016.

[28] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *WWW 2018: The 2018 Web Conference*, 2018.

[29] Michal Zalewski. p0f v3, 2019. URL http://lcamtuf.coredump.cx/p0f3/.

[30] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix Freiling. Fingerprinting mobile devices using personalized configurations. *Proceedings on Privacy Enhancing Technologies*, 2016(1):4–19, 2016.

[31] Wenjia Wu, Jianan Wu, Yanhao Wang, Zhen Ling, and Ming Yang. Efficient fingerprinting-based android device identification with zero-permission identifiers. *IEEE Access*, 4:8073–8083, 2016.

[32] Internet Engineering Task Force. Rfc 7231: Hypertext transfer protocol (http/1.1): Semantics and content, 2014. URL https://tools.ietf.org/html/rfc7231.

[33] Internet Engineering Task Force. Rfc 7231: semantics and content of the user-agent header, 2014. URL https://tools.ietf.org/html/rfc7231#section-5.5.3.

[34] Internet Engineering Task Force. Rfc 7231: semantics and content of the accept-language header, 2014. URL https://tools.ietf.org/html/rfc7231#section-5.3.5.

[35] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martin Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *NDSS*, volume 62, page 66, 2012.

[36] Lukasz Olejnik, Steven Englehardt, and Arvind Narayanan. Battery status not included: Assessing privacy in web standards. In *3rd International Workshop on Privacy Engineering (IWPE'17)*, 2017.

[37] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FC Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, volume 5. Citeseer, 2013.

[38] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, pages 1–12, 2012.

[39] David Fifield and Serge Egelman. Fingerprinting web users through font metrics. In *International Conference on Financial Cryptography and Data Security*, pages 107–124. Springer, 2015.

[40] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. Fingerprinting information in JavaScript implementations. In Helen Wang, editor, *Proceedings of W2SP 2011*. IEEE Computer Society, May 2011.

[41] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Clock around the clock: Time-based device fingerprinting. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1502–1514. ACM, 2018.

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

[43] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 329–336. ACM, 2017.

[44] Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld. Latex gloves: Protecting browser extensions from probing and revelation attacks. *Power*, page 57, 2018.

[45] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. Fpguard: Detection and prevention of browser fingerprinting. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 293–308. Springer, 2015.

[46] Enric Pujol, Oliver Hohlfeld, and Anja Feldmann. Annoyed users: Ads and ad-block usage in the wild. In *Proceedings of the 2015 Internet Measurement Conference*, pages 93–106. ACM, 2015.

[47] Firefox. Firefox public data report, 2019. URL https://data.firefox.com/dashboard/usage-behavior.

[48] Eyeo GmbH. Adblock plus, 2018. URL https://adblockplus.org/.

[49] Raymond Hill. ublock origin - an efficient blocker for chromium and firefox. fast and lean., 2018. URL https://github.com/gorhill/uBlock.

[50] EasyList. About easylist, 2018. URL https://easylist.to/pages/about.html.

[51] EasyPrivacy. Easyprivacy, 2018. URL https://easylist.to/easylist/easyprivacy.txt.

[52] AdBlock. Adblock, 2018. URL https://getadblock.com/.

[53] Brave Software Inc. Brave browser, 2018. URL https://brave.com/.

[54] Cliqz International GmbH. Ghostery, 2018. URL https://www.ghostery.com.

[55] Electronic Frontier Foundation. Privacy badger, 2019. URL https://www.eff.org/fr/node/99095.

[56] Umar Iqbal, Zubair Shafiq, Peter Snyder, Shitong Zhu, Zhiyun Qian, and Benjamin Livshits. Adgraph: A machine learning approach to automatic and effective adblocking. *arXiv preprint arXiv:1805.09155*, 2018.

[57] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. Block Me if You Can: A Large-Scale Study of Tracker-Blocking Tools. *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017*, pages 319–333, 2017. doi: 10.1109/EuroSP.2017.26.

[58] Giorgio Maone. Noscript firefox extension, 2018. URL https://noscript.net/.

[59] Raymond Hill. umatrix: Point and click matrix to filter net requests according to source, destination and type, 2019. URL https://github.com/gorhill/uMatrix.

[60] The Tor Project. Tor browser, 2018. URL https://www.torproject.org/projects/torbrowser.html.en.

[61] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M Pujol. Tracking the trackers. In *Proceedings of the 25th International Conference on World Wide Web*, pages 121–132. International World Wide Web Conferences Steering Committee, 2016.

[62] Smart Software. Ultimate user agent switcher, url sniffer, 2019. URL http://iblogbox.com/chrome/useragent/alert.php.

[63] dillbyrne. Random agent spoofer, 2019. URL https://github.com/dillbyrne/random-agent-spoofer.

[64] sereneblue. Chameleon, a webextension port of random agent spoofer, 2019. URL https://github.com/sereneblue/chameleon.

[65] Valentin Vasilyev. Modern and flexible browser fingerprinting library, 2019. URL https://github.com/Valve/fingerprintjs2.

[66] Christof Ferreira Torres, Hugo Jonker, and Sjouke Mauw. Fp-block: usable web privacy by controlling browser fingerprinting. In *European Symposium on Research in Computer Security*, pages 3–19. Springer, 2015.

[67] ECMA international. Ecmascript® 2016 language specification, 2016. URL http://www.ecma-international.org/ecma-262/7.0/index.html.

[68] Multilogin. Canvas Defender browser extension (canvas fingerprint blocker). URL https://multiloginapp.com/canvasdefender-browser-extension/.

[69] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 98–108. IEEE Press, 2015.

[70] Grant Ho, Dan Boneh, Lucas Ballard, and Niels Provos. Tick tock: Building browser red pills from timing side channels. In *WOOT*, 2014.

[71] kkapsner. CanvasBlocker: A Firefox Plugin to block the <canvas>-API, July 2017. URL https://github.com/kkapsner/CanvasBlocker.

[72] Michael Schwarz, Florian Lackner, and Daniel Gruss. Javascript template attacks: Automatically inferring host information for targeted exploits. In *NDSS*, 2019.

[73] OWASP. Cross-site scripting (xss), 2018. URL https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[74] Pierre Laperdrix. *Browser Fingerprinting : Exploring Device Diversity to Augment Authentification and Build Client-Side Countermeasures*. Theses, INSA de Rennes, October 2017. URL https://tel.archives-ouvertes.fr/tel-01729126.

[75] Luis Von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 319–326. ACM, 2004.

[76] Gregoire Jacob and Christopher Kruegel. PUB CRAWL : Protecting Users and Businesses from CRAWLers. *Protecting Users and Businesses from CRAWLers Gregoire*, 2009.

[77] Derek Doran and Swapna S Gokhale. Web robot detection techniques: overview and limitations. *Data Mining and Knowledge Discovery*, 22(1-2):183–210, 2011.

[78] Weigang Guo, Shiguang Ju, and Yi Gu. Web robot detection techniques based on statistics of their requested url resources. In *Computer Supported Cooperative Work in Design, 2005. Proceedings of the Ninth International Conference on*, volume 1, pages 302–306. IEEE, 2005.

[79] Xiaozhu Lin, Lin Quan, and Haiyan Wu. An automatic scheme to categorize user sessions in modern http traffic. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–6. IEEE, 2008.

[80] Anália Lourenço and Orlando Belo. Applying clickstream data mining to real-time web crawler detection and containment using clicktips platform. In *Advances in Data Analysis*, pages 351–358. Springer, 2007.

[81] Athena Stassopoulou and Marios D Dikaiakos. Crawler detection: A bayesian approach. In *Internet Surveillance and Protection, 2006. ICISP'06. International Conference on*, pages 16–16. IEEE, 2006.

[82] Pang-Ning Tan and Vipin Kumar. Discovery of web robot sessions based on their navigational patterns. In *Intelligent Technologies for Information Analysis*, pages 193–222. Springer, 2004.

[83] Stephen Beveridge and Charles R Nelson. A new approach to decomposition of economic time series into permanent and transitory components with particular attention to measurement of the 'business cycle'. *Journal of Monetary economics*, 7(2):151–174, 1981.

[84] Luis Von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. Captcha: Using hard ai problems for security. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 294–311. Springer, 2003.

[85] 2Captcha. Online captcha solving and image recognition service., 2018. URL https://2captcha.com/.

[86] Anti CAPTCHA. Anti captcha: captcha solving service. bypass recaptcha, funcaptcha, image captcha., 2018. URL https://anti-captcha.com/mainpage.

[87] Suphannee Sivakorn, Jason Polakis, and Angelos D Keromytis. I'm not a human: Breaking the google recaptcha.

[88] Kevin Bock, Daven Patel, George Hughey, and Dave Levin. uncaptcha: a low-resource defeat of recaptcha's audio challenge. In *Proceedings of the 11th USENIX Conference on Offensive Technologies*, pages 7–7. USENIX Association, 2017.

[89] Zi Chu, Steven Gianvecchio, Aaron Koehl, Haining Wang, and Sushil Jajodia. Blog or block: Detecting blog bots through behavioral biometrics. *Computer Networks*, 57(3):634–646, 2013.

[90] Ah Reum Kang, Jiyoung Woo, Juyong Park, and Huy Kang Kim. Online game bot detection based on party-play log analysis. *Computers & Mathematics with Applications*, 65(9):1384–1395, 2013.

[91] Binh Nguyen, Bryan D Wolf, and Brian Underdahl. Detecting and preventing bots and cheating in online gaming, January 29 2013. US Patent 8,360,838.

[92] Zi Chu, Steven Gianvecchio, Haining Wang, and Sushil Jajodia. Detecting automation of twitter accounts: Are you a human, bot, or cyborg? *IEEE Transactions on Dependable and Secure Computing*, 9(6):811–824, 2012.

[93] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Detecting spammers on social networks. In *Proceedings of the 26th annual computer security applications conference*, pages 1–9. ACM, 2010.

[94] Gang Wang, Tristan Konolige, Christo Wilson, Xiao Wang, Haitao Zheng, and Ben Y Zhao. You are how you click: Clickstream analysis for sybil detection. In *USENIX Security Symposium*, volume 9, pages 1–008, 2013.

[95] Jing Zhang, Ari Chivukula, Michael Bailey, Manish Karir, and Mingyan Liu. Characterization of blacklists and tainted network traffic. In *International Conference on Passive and Active Network Measurement*, pages 218–228. Springer, 2013.

[96] Augur Software, . URL https://www.augur.io/.

[97] Athena Stassopoulou and Marios D Dikaiakos. Web robot detection: A probabilistic reasoning approach. *Computer Networks*, 53(3):265–278, 2009.

[98] Dusan Stevanovic, Aijun An, and Natalija Vlajic. Feature evaluation for web crawler detection with data mining techniques. *Expert Systems with Applications*, 39(10):8707–8717, 2012.

[99] Andoena Balla, Athena Stassopoulou, and Marios D Dikaiakos. Real-time web crawler detection. In *Telecommunications (ICT), 2011 18th International Conference on*, pages 428–432. IEEE, 2011.

[100] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[101] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. Understanding variable importances in forests of randomized trees. In *Advances in neural information processing systems*, pages 431–439, 2013.

[102] Octavio Loyola-González, Milton García-Borroto, Miguel Angel Medina-Pérez, José Fco Martínez-Trinidad, Jesús Ariel Carrasco-Ochoa, and Guillermo De Ita. An empirical study of oversampling and undersampling methods for lcmine an emerging pattern based classifier. In *Mexican Conference on Pattern Recognition*, pages 264–273. Springer, 2013.

[103] Simon Bernard, Laurent Heutte, and Sébastien Adam. Influence of hyperparameters on random forest accuracy. In *International Workshop on Multiple Classifier Systems*, pages 171–180. Springer, 2009.

[104] Naoki Takei, Takamichi Saito, Ko Takasu, and Tomotaka Yamada. Web browser fingerprinting using only cascading style sheets. In *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, pages 57–63. IEEE, 2015.

[105] Takamichi Saito, Kazushi Takahashi, Koki Yasuda, Takayuki Ishikawa, Ko Takasu, Tomotaka Yamada, Naoki Takei, and Rio Hosoi. OS and Application Identification by Installed Fonts. *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 684–689, 2016. doi: 10.1109/AINA.2016.55. URL http://ieeexplore.ieee.org/document/7474155/.

[106] ua parser. Python implementation of ua-parser. https://github.com/ua-parser/uap-python, 2018.

[107] Modernizr. Modernizr: the feature detection library for html5/css3. https://modernizr.com, 2019.

[108] Alexis Deveria. Can i use... support tables for html5, css3, etc, 2019. URL https://caniuse.com/.

[109] Brave Software, . URL https://brave.com/.

[110] Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. Measuring price discrimination and steering on e-commerce web sites. In *Proceedings of the 2014 conference on internet measurement conference*, pages 305–318. ACM, 2014.

[111] MDN Web Docs. Mutationobserver api, 2018. URL https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver.

[112] Distil Networks. Distil's bad bot report 2018: The year bad bots went mainstream. https://resources.distilnetworks.com/all-blog-posts/bad-bot-report-now-available, 2018.

[113] Google. Puppeteer, 2019. URL https://pptr.dev/.

[114] Martin Monperrus. Crawler-user-agents, 2019. URL https://github.com/monperrus/crawler-user-agents.

[115] Google. Issue 775911 in chromium: missing accept languages in request for headless mode. https://groups.google.com/a/chromium.org/forum/#!topic/headless-dev/8YujuBps0oc, October 2017.

[116] Erti-Chris Eelmaa. Can a website detect when you are using selenium with chromedriver? https://stackoverflow.com/questions/33225947/can-a-website-detect-when-you-are-using-selenium-with-chromedriver/41220267#41220267, 2016.

[117] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[118] Selenium HQ. What is selenium?, 2019. URL https://www.seleniumhq.org.

[119] Selenium HQ. Selenium ide, 2019. URL https://www.seleniumhq.org/projects/ide/.

[120] Sergey Shekyan. Detecting phantomjs based visitors, 2015. URL https://blog.shapesecurity.com/2015/01/22/detecting-phantomjs-based-visitors/.

[121] Alexis Deveria. Support of typed arrays, 2019. URL https://caniuse.com/#search=Int8Array.

[122] Alexis Deveria. Support of mutation observers, 2019. URL https://caniuse.com/#search=MutationObserver.

[123] Antoine Vastel. Detecting chrome headless, 2017. URL https://antoinevastel.com/bot%20detection/2017/08/05/detect-chrome-headless.html.

[124] MDN Web Docs. Permissions api, 2018. URL https://developer.mozilla.org/en-US/docs/Web/API/Permissions_API.

[125] Alexis Deveria. Support of ogg vorbis audio format, 2019. URL https://caniuse.com/#search=ogg.

[126] Alexis Deveria. Support of mp3 audio format, 2019. URL https://caniuse.com/#feat=mp3.

[127] Alexis Deveria. Support of waveform audio file format, 2019. URL https://caniuse.com/#search=wav.

[128] Alexis Deveria. Support of advanced audio coding format, 2019. URL https://caniuse.com/#feat=aac.

[129] Alexis Deveria. Support of ogg/theora video format, 2019. URL https://caniuse.com/#feat=ogv.

[130] Alexis Deveria. Support of mpeg-4/h.264 format, 2019. URL https://caniuse.com/#search=h264.

[131] Alexis Deveria. Support of webm video format, 2019. URL https://caniuse.com/#search=webm.

[132] Chromium Bug Tracker. Support webgl in headless, 2016. URL https://bugs.chromium.org/p/chromium/issues/detail?id=617551.

[133] Apple Inc. ios sdk release notes for ios 8.0 gm, 2014. URL https://developer.apple.com/library/archive/releasenotes/General/RN-iOSSDK-8.0/.

[134] Distil Networks. Can comic sans detect cyber attacks? https://resources.distilnetworks.com/all-blog-posts/can-comic-sans-detect-cyber-attacks, September 2016.

[135] Xianghang Mi, Ying Liu, Xuan Feng, Xiaojing Liao, Baojun Liu, XiaoFeng Wang, Feng Qian, Zhou Li, Sumayah Alrwais, and Limin Sun. Resident evil: Understanding residential ip proxy as a dark service. In *Resident Evil: Understanding Residential IP Proxy as a Dark Service*, page 0. IEEE, 2019.

[136] Imperva Incapsula. Bot traffic report 2016. http://time.com/12933/what-you-think-you-know-about-the-web-is-wrong/, March 2014.

[137] Renáta Hodován and Ákos Kiss. Fuzzing javascript engine apis. In *International Conference on Integrated Formal Methods*, pages 425–438. Springer, 2016.

[138] Christina Braz, Ahmed Seffah, and David M'Raihi. Designing a trade-off between usability and security: a metrics based-model. In *IFIP Conference on Human-Computer Interaction*, pages 114–126. Springer, 2007.

[139] Google. Devtools: Make it possible to control permissions via protocol, 2016. URL https://bugs.chromium.org/p/chromium/issues/detail?id=631464.

Appendix A

List of fingerprinting attributes collected

This appendix presents the list of fingerprinting attributes monitored to detect scripts that use browser fingerprinting for crawler detection.

A.1 Navigator properties

- userAgent,
- platform,
- plugins,
- mimeTypes,
- doNotTrack,
- languages,
- productSub,
- language,
- vendor,
- oscpu,
- hardwareConcurrency,

- cpuClass,
- webdriver,
- chrome.

A.2 Screen properties

- width,
- height,
- availWidth,
- availHeight,
- availTop,
- availLeft,
- colorDepth,
- pixelDepth.

A.3 Window properties

- ActiveXObject,
- webdriver,
- domAutomation,
- domAutomationController,
- callPhantom,
- spawn,
- emit,
- Buffer,
- awesomium,
- _Selenium_IDE_Recorder,

- ___webdriver_script_fn,
- _phantom,
- callSelenium,
- _selenium.

A.4 Audio methods

- createAnalyser,
- createOscillator,
- createGain,
- createScriptProcessor,
- createDynamicsCompressor,
- copyFromChannel,
- getChannelData,
- getFloatFrequencyData,
- getByteFrequencyData,
- getFloatTimeDomainData,
- getByteTimeDomainData.

A.5 WebGL methods

- getParameter,
- getSupportedExtensions,
- getContextAttributes,
- getShaderPrecisionFormat,
- getExtension,
- readPixels,

- getUniformLocation,
- getAttribLocation.

A.6 Canvas methods

- toDataURL,
- toBlob,
- getImageData,
- getLineDash,
- measureText,
- isPointInPath.

A.7 WebRTC methods

- createOffer,
- createAnswer,
- setLocalDescription,
- setRemoteDescription.

A.8 Other methods

- Date.getTimezoneOffset,
- SVGTextContentElement.getComputedTextLength

Appendix B

Overriding crawlers fingerprints

This appendix presents the code used to override the fingerprints of the 7 crawlers used for the evaluation in Chapter 5. The `page` variable used in the code snippet refers to an instance of a `Page` object of the Puppeteer library.

B.1 Overriding the user agent

The seven crawlers override their user agent to appear like a non-headless Chrome. The code modifies both the user agent sent in the HTTP headers and the user agent contained in the `navigator` object.

```
await page.setUserAgent(userAgent);
```

Listing B.1 Setting the user agent of the crawler to a value contained in a variable `userAgent`.

B.2 Deleting the webdriver property

Crawlers 2 to 7 delete the `webdriver` property of the `navigator` object. Since the `navigator` object is handled differently by the browser, we cannot directly delete it. Instead, we create a reference to the navigator prototype. Then, we delete the `webdriver` property from this reference. Finally, we assign the new prototype reference that does not contain the `webdriver` property to the `navigator` object.

```
await page.evaluateOnNewDocument(() => {
    const newProto = navigator.__proto__;
    delete newProto.webdriver;
    navigator.__proto__ = newProto;
});
```

Listing B.2 Delete the `webdriver` property from the `navigator` object.

B.3 Adding a language header

Crawlers 3 to 6 add an `Accept-Language` header field to all the HTTP requests they send.

```
await page.setExtraHTTPHeaders({
    'Accept-Language': 'en-US'
});
```

Listing B.3 Adding an `Accept-Language` header field to all the HTTP requests sent by the crawler.

B.4 Forging a fake Chrome object

Crawlers 4 to 6 add a realistic `chrome` property to the `window` object.

```
await page.evaluateOnNewDocument(() => {
    window.chrome = {
        app: {
            isInstalled: false,
        },
        webstore: {
            onInstallStageChanged: {},
            onDownloadProgress: {},
        },
        runtime: {
            PlatformOs: {
                MAC: 'mac',
                WIN: 'win',
                ANDROID: 'android',
                CROS: 'cros',
                LINUX: 'linux',
```

```
                OPENBSD: 'openbsd',
            },
            PlatformArch: {
                ARM: 'arm',
                X86_32: 'x86-32',
                X86_64: 'x86-64',
            },
            PlatformNaclArch: {
                ARM: 'arm',
                X86_32: 'x86-32',
                X86_64: 'x86-64',
            },
            RequestUpdateCheckStatus: {
                THROTTLED: 'throttled',
                NO_UPDATE: 'no_update',
                UPDATE_AVAILABLE: 'update_available',
            },
            OnInstalledReason: {
                INSTALL: 'install',
                UPDATE: 'update',
                CHROME_UPDATE: 'chrome_update',
                SHARED_MODULE_UPDATE: 'shared_module_update',
            },
            OnRestartRequiredReason: {
                APP_UPDATE: 'app_update',
                OS_UPDATE: 'os_update',
                PERIODIC: 'periodic',
            },
        },
    };
});
```

Listing B.4 Adding a `chrome` property to the `window` object.

B.5 Overriding permissions behavior

Crawlers 5 and 6 override the way permissions for notifications are handled.

```
await page.evaluateOnNewDocument(() => {
    const originalQuery = window.navigator.permissions.query;
    window.navigator.permissions.__proto__.query = parameters =>
        parameters.name === 'notifications'
```

```
            ? Promise.resolve({state: Notification.permission})
            : originalQuery(parameters);
});
```

Listing B.5 Override the ways permissions are handled.

B.6 Overriding window and screen dimensions

Crawler 6 override several attributes related to the size of the screen and the window using consistent values collected on a non-headless browser.

```
await page.evaluateOnNewDocument(() => {
    Object.defineProperty(window, 'innerWidth', {
        get: function() { return 1919;}
    });

    Object.defineProperty(window, 'innerHeight', {
        get: function() { return 1007;}
    });

    Object.defineProperty(window, 'outerWidth', {
        get: function() { return 1919; }
    });

    Object.defineProperty(window, 'outerHeight', {
        get: function() { return 1007; }
    });

    Object.defineProperty(window, 'pageXOffset', {
        get: function() { return 0; }
    });

    Object.defineProperty(window, 'pageYOffset', {
        get: function() { return 0; }
    });

    Object.defineProperty(window, 'screenX', {
        get: function() { return 1680; }
    });

    Object.defineProperty(window, 'screenY', {
        get: function() { return 0; }
```

```
    });

    Object.defineProperty(screen, 'availWidth', {
        get: function() { return 1920; }
    });

    Object.defineProperty(window, 'availHeight', {
        get: function() { return 1080; }
    });

    Object.defineProperty(screen, 'width', {
        get: function() { return 1920; }
    });

    Object.defineProperty(screen, 'height', {
        get: function() { return 1080; }
    });

    document.addEventListener("DOMContentLoaded", () => {
        Object.defineProperty(document.body, 'clientWidth', {
            get: function() { return 1919; }
        });

        Object.defineProperty(document.body, 'clientHeight', {
            get: function() { return 541; }
        });
    });
});
```

Listing B.6 Override properties related to the size of the screen and the window using real values collected on a non-headless browser.

B.7 Overriding codecs support

Crawler 6 overrides the support for several audio and video codecs.

```
await page.evaluateOnNewDocument(() => {
    HTMLAudioElement.prototype.canPlayType = (t) => {
        const tTrimed = t.replace(/[ "';]/g, '');
        if (tTrimed === 'audio/oggcodecs=vorbis') {
            return "probably";
        } else if (tTrimed === 'audio/mpeg') {
```

```
            return "probably";
        } else if (tTrimed === 'audio/wavcodecs=1') {
            return "probably";
        } else if (tTrimed === 'audio/x-m4a') {
            return "maybe";
        } else if (tTrimed === 'audio/aac') {
            return "probably";
        }
        return '';
    };

    HTMLVideoElement.prototype.canPlayType = (t) => {
        const tTrimed = t.replace(/[ "';]/g, '');
        if (tTrimed === 'video/oggcodecs=theora') {
            return "probably";
        } else if (tTrimed === 'video/mp4codecs=avc1.42E01E') {
            return "probably";
        } else if (tTrimed === 'video/webmcodecs=vp8,vorbis') {
            return "probably";
        } else if (tTrimed === 'video/mp4codecs=mp4v.20.8mp4a.40.2') {
            return "";
        } else if (tTrimed === 'video/mp4codecs=mp4v.20.240,mp4a.40.2')
            {
            return "";
        } else if (tTrimed === 'video/x-matroskacodecs=theora,vorbis')
           {
            return "";
        }
        return '';
    };
});
```

Listing B.7 Override the `canPlayType` function for `HTMLAudioElement` and `HTMLVideoElement`.

B.8 Removing traces of touch support

Crawler 6 removes attributes and events that indicates the presence of touch support on the device.

```
await page.evaluateOnNewDocument(() => {
    document.createEvent = (function (orig) {
```

```
        return function () {
            let args = arguments;

            if (args[0] === 'TouchEvent') {
                throw 'error';
            }

            return orig.apply(this, args);
        };
    }(document.createEvent));

    Object.defineProperty(navigator, 'maxTouchPoints', { get: () => 0
       };
    delete window.ontouchstart;
});
```

Listing B.8 Remove events and functions related to touch screen.

B.9 Overriding toStrings

Crawlers 5 and 6 override the behavior of native functions. In order to hide their changes, they both override the `toString` function of the functions they override. Moreover, they also override `Function.prototype.toString`, the `toString` of the `Function` class.

```
await page.evaluateOnNewDocument(() => {
    const oldCall = Function.prototype.call;
    function call() {
        return oldCall.apply(this, arguments);
    }
    Function.prototype.call = call;

    const nativeToStringFunctionString = Error.toString().replace(/
       Error/g, "toString");
    const oldToString = Function.prototype.toString;

    function functionToString() {
        if (this === window.navigator.permissions.query) {
            return "function query() { [native code] }";
        }

        if (this === HTMLVideoElement.prototype.canPlayType) {
            return "function canPlayType() { [native code] }";
```

```
        }

        if (this === HTMLAudioElement.prototype.canPlayType) {
            return "function canPlayType() { [native code] }";
        }

        if (this === document.createEvent) {
            return "function createEvent() { [native code] }";
        }

        if (this === functionToString) {
            return nativeToStringFunctionString;
        }
        return oldCall.call(oldToString, this);
    }
    Function.prototype.toString = functionToString;
});
```

Listing B.9 Override `toString` functions of functions overridden.