

Leveraging browser fingerprinting to strengthen web authentication

ANTONIN DUREY

14th January 2022

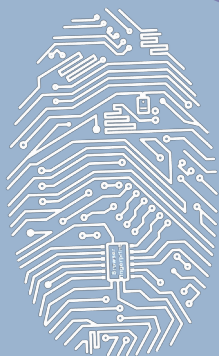
University of Lille & Inria center
CRIStAL laboratory
MADIS doctoral school

Supervisors:

Romain ROUVOY - Professor
Walter RUDAMETKIN - Associate Professor with HDR

Jury:

Marc TOMMASI - Professor, University of Lille - president
Olivier BARAIS - Professor, University of Rennes 1 - reviewer
Sonia BEN MOKTHAR - Director of research, LIRIS, CNRS - reviewer
Isabelle CHRIMENT - Professor, TELECOM Nancy - examiner
Gunes ACAR - Assistant Professor, Radboud University Nijmegen - examiner
Arnaud PÉRILLOUX, Ministry of Armies - invited



Utiliser les empreintes de navigateurs pour renforcer l'authentification sur Internet

ANTONIN DUREY

14 Janvier 2022

Université de Lille & Centre Inria
Laboratoire CRIStAL
Ecole Doctorale MADIS

Directeurs:

Romain ROUVOY - Professeur
Walter RUDAMETKIN - Maître de conférences avec HDR

Jury:

Marc TOMMASI - Professeur, Université de Lille - président
Olivier BARAIS - Professeur, Université de Rennes 1 - rapporteur
Sonia BEN MOKTHAR - Directrice de recherche, LIRIS, CNRS - rapportrice
Isabelle CHRISMENT - Professeur, TELECOM Nancy - examinatrice
Gunes ACAR - Assistant Professor, Radboud University Nijmegen - examinateur
Arnaud PÉRILLOUX, Ministère des Armées - invité

Acknowledgements

I would like to thank all the people that helped me realizing this thesis and writing this manuscript.

First, I would like to thank my directors, Romain Rouvoy and Walter Rudametkin. It was a pleasure to share ideas and receive comments and advices from both of you. Thanks you, Pierre Laperdrix, for all the work done together and all the interesting discussions we had together. I would like to thank all the members in the Spirals team, particularly Lionel Seinturier, Laurence Duchien, Antoine Vastel, Vikas Mishra, Rémy Raes, Guillaume Fieni and Sarra Habchi.

During my Ph.D, I had the chance to work on an authentication solution in a real system. Thank you, Didier Benza, Anne Combe, Jérôme Berthier, Quentin Laize and Florent Derudas for working with my on this project and allowing me to have interesting results for these experiences.

Thank you to all the people I met during the extra-activities I performed during my Ph.D: Yann Secq and Fabien Delecroix when teaching, Ludovic Macaire when being a representant of the Ph.D. students at the Doctoral School and all the Ph.D students I met while being president of the ADSL - Ph.D Students in Science of Lille Association.

Thank you, Pierre Bourhis, Iovka Boneva and Sophie Tison, for supervising me for my internship and my end-of-studies project during my Master when you encouraged me to do a Ph.D. Thank you, Martin Monperrus for also motivating me to do a Ph.D during my Master.

Finally, I would like to thank my sisters, parents and friends. Your support was essential during this adventure and gave extra-motivation all along this journey.

Abstract

Security on the Web is a major concern for any user, and authentication solutions, such as multi-factor authentication, negatively impact the user experience and add cost and complexity that may prevent them from being more accepted by users and more largely deployed. ***Browser fingerprinting*** is a stateless and permission-less technique that collects information about the user's device, OS, browser and configuration to form an identifier. While it has mainly been studied from a tracking perspective, its properties make it interesting for security, and more specifically, for Web authentication.

In this thesis, I provide 3 main contributions:

- I manually browse 1,485 pages on 446 websites and measure fingerprint collection on sensitive pages of websites, such as sign-up and sign-in pages. I evaluate the resilience of these websites against 2 types of attack, stolen credentials and cookie hijacking, and show that fingerprinting, despite its potential, is barely used to protect against these attacks.
- I collect fingerprints in a controlled environment to precisely measure the attributes that offer interesting uniqueness and stability properties. I use this knowledge to design and implement a fingerprints linking algorithm for Web authentication and evaluate it on a dataset of 952,828 fingerprints collected from 64,235 browser instances, and show the algorithm is reliable and relevant to link fingerprints.
- I design and implement an authentication scheme that strengthens web authentication by using browser fingerprinting. I evaluate the scheme on a centralized authentication server with 82 users. I demonstrate that browser fingerprinting strengthens Web authentication while having a minimal impact on the user experience.

With these contributions, I argue that browser fingerprinting improves web authentication and conclude this manuscript by providing short-term and long-term perspectives to improve this work.

Résumé

La sécurité d'un système d'authentification est un élément dont l'importance continue de croître depuis la naissance du Web. De nos jours, c'est un point d'intérêt majeur et des solutions comme l'authentification multi-facteur ont un impact fort sur l'expérience utilisateur qui empêchent ces techniques d'être acceptées par la majorité des utilisateurs et d'être déployées à grande échelle sur Internet. Une ***empreinte de navigateur*** est une technique d'identification sans état et qui ne requiert pas de permission. Elle collecte des informations sur l'appareil, l'OS, le navigateur et la configuration de l'utilisateur. Alors que cette technique a majoritairement été étudiée à des fins de suivi en ligne, ses propriétés en font un élément intéressant pour renforcer la sécurité sur Internet, et notamment l'authentification. Dans cette thèse, je propose 3 contributions relative à l'usage des empreintes de navigateur pour améliorer l'authentification sur Internet:

- Je visite manuellement 1,485 pages Internet appartenant à 446 sites et je mesure que les empreintes de navigateur sont collectées sur des pages sensibles, tel que des pages d'authentification. J'évalue la résistance de ces sites contre 2 attaques, le vol de mot de passe et le vol de cookies, et montre que les empreintes de navigateur sont peu utilisées pour se protéger contre ces menaces.
- Je collecte des empreintes de navigateur dans un environnement contrôlé pour mesurer précisément les attributs qui offrent des propriétés intéressantes en terme d'unicité et de stabilité. J'utilise cette connaissance pour concevoir et implémenter un algorithme pouvant lier des empreintes de navigateur. J'évalue cet algorithme sur un ensemble de données formé de 952,828 empreintes provenant de 64,235 instances de navigateur, et montre que l'algorithme est fiable et pertinent.
- Je conçois et implémente un schéma d'authentification qui renforce l'authentification en utilisant la technique des empreintes de navigateur. J'intègre ce schéma dans un système centralisé d'authentification comportant 82 utilisateurs. Je démontre que la technique des empreintes de navigateur améliore la sécurité tout en ayant un impact minime sur l'expérience utilisateur.

Avec ces contributions, je démontre que les empreintes de navigateur sont légitimes pour renforcer l'authentification sur Internet. Alors que le Web est en constante évolution, je conclus en proposant des perspectives à court et long terme pour améliorer ces travaux et suivre l'évolution de la technique des empreintes de navigateur.

Table of contents

List of figures	xii
List of tables	xiii
1 Introduction	1
1.1 Context	1
1.2 Objectives	2
1.3 List of Scientific Publications	3
1.4 List of Tools and Prototypes	4
1.5 Outline	4
2 State of the Art	7
2.1 Context	7
2.1.1 Birth of the Web	7
2.1.2 Web evolution	8
2.2 Web authentication	10
2.2.1 Concept	10
2.2.2 Threats and attacks	11
2.2.3 Protecting data access	13
2.2.4 Bots protection techniques	14
2.2.5 Improving authentication	15
2.3 Browser fingerprinting	19
2.3.1 Definition	19
2.3.2 Properties	19
2.3.3 Attributes	20
2.4 Browser fingerprinting studies	32
2.4.1 Measuring browser fingerprinting properties	32
2.4.2 Detection and classification	36
2.5 Browser fingerprinting countermeasures	38
2.5.1 Blocking scripts	39
2.5.2 Unifying attributes value	40

2.5.3	Changing attributes value over time	41
2.5.4	Induced information leaks	42
2.6	Browser fingerprinting usages	43
2.6.1	User Tracking	43
2.6.2	Bot Detection	44
2.6.3	User Authentication	45
2.7	Conclusion	47
3	FP-Redemption: Studying Browser Fingerprinting Adoption for the Sake of Web Security	51
3.1	A Dataset of Secure Web Pages	52
3.1.1	Websites Under Study	52
3.1.2	Web Page Acquisition	53
3.1.3	Monitored Fingerprinting Attributes	53
3.1.4	Resulting Dataset Description	54
3.2	Classification of Fingerprinters	56
3.2.1	Incremental Script Classification	57
3.2.2	Script Classification Results	59
3.2.3	Algorithm results validation	60
3.3	Analysis of Secure Web Pages	61
3.3.1	Browser Fingerprinting Attributes	61
3.3.2	Similarities of Browser Fingerprinting Scripts	63
3.3.3	Origins of Browser Fingerprinting Scripts	63
3.3.4	Web page type and website category & country impact	65
3.3.5	Additional Security Mechanisms	67
3.4	Websites resilience against 2 attack models	68
3.4.1	Stolen credentials	68
3.4.2	Cookie hijacking	71
3.5	Discussion	73
3.5.1	Intents in fingerprinting usages	73
3.5.2	Fingerprinting is barely used for security	74
3.5.3	Deficiencies in the state of the art	75
3.6	Conclusion	75
4	FP-Controlink: Studying fingerprinting under a controlled environment to link fingerprints	77
4.1	Methodology	78
4.1.1	Controlled environment	78
4.1.2	Browser versions	79
4.1.3	Attributes	79

4.1.4	Data collection	80
4.2	Causes of fingerprints diversity	82
4.2.1	Desktop evaluation	82
4.2.2	Mobile evaluation	86
4.2.3	Layers responsible for an attribute change	87
4.3	Fingerprints evolution through browser versions	88
4.3.1	Release versions	90
4.3.2	Nightly/beta versions	93
4.3.3	Categorizing attributes	94
4.4	A browser fingerprints linking algorithm	94
4.4.1	Main goal	94
4.4.2	Design	95
4.4.3	Parameters	96
4.5	Evaluation of the linking algorithm	97
4.5.1	Datasets	97
4.5.2	Key performance metrics	97
4.5.3	Parameters values	99
4.5.4	In-the-wild results	100
4.6	Discussion	102
4.6.1	Ethical consideration	102
4.6.2	Choosing parameters value	102
4.6.3	Linking algorithm improvements.	103
4.7	Conclusion	103
5	Advanced risk-based authentication using browser fingerprinting	105
5.1	Authentication scheme	106
5.1.1	Design	106
5.1.2	Challenges	107
5.2	Implementation	109
5.2.1	Legacy Authentication Systems	109
5.2.2	Rising to the challenges	110
5.2.3	Authentication scheme and CAS plugin	115
5.3	Evaluation	116
5.3.1	Dataset constitution	117
5.3.2	Key Performance Metrics	117
5.3.3	Trusted network fingerprints and authentication attempts	118
5.3.4	Linking algorithm scores	118
5.3.5	Collection and analysis time	120
5.4	Discussion	121
5.4.1	Ethical considerations	121

5.4.2	Security versus user experience	122
5.4.3	Client-side-generated information	122
5.4.4	Device management rules	123
5.4.5	Compromised device	124
5.4.6	Adding features to the authentication scheme	124
5.5	Conclusion	125
6	Conclusion	127
6.1	Contributions	127
6.1.1	FP-Redemption: Studying Browser Fingerprinting Adoption for the Sake of Web Security	127
6.1.2	FP-Controlink: Studying fingerprinting under a controlled environ- ment to link fingerprints	128
6.1.3	Advanced risk-based authentication using browser fingerprinting .	129
6.2	Short-term perspectives	129
6.2.1	Discovering new fingerprinting JAVASCRIPT attributes	129
6.2.2	Studying attacks targeting fingerprinting-based authentication sys- tems	131
6.2.3	Investigating WEB ASSEMBLY technology	132
6.3	Long-term perspectives	133
6.4	Concluding note	133
	Bibliography	135

List of figures

2.1	Evolution of the Web device market share from 2009 to 2021.	8
2.2	Evolution of the browser market share from 2009 to 2021.	9
2.3	Examples of captcha techniques	15
3.1	Distribution of the 446 visited websites per country & category.	55
3.2	Distribution of visited pages per type (the sum exceeds 1,485 pages as some pages match several types).	56
3.3	Flow chart representing our incremental script classification algorithm. . .	57
3.4	Iteration distribution of the label put on the scripts of our dataset. . . .	60
3.5	CDF of number of attributes used by scripts.	61
3.6	Distribution of attributes families across fingerprinters.	62
4.1	Number of attribute value changes per version grouped by browser out of a total of 56 attributes in our experiments.	92
4.2	Number of changes per attribute per browser. Only attributes that have changed at least once during a version update are shown on this graph. .	92
4.3	Browser instances identification duration according to the threshold and weights sets	99
4.4	Number of assigned IDs to each browser instance according to the threshold and weights sets	101
4.5	Evolution of the proportion of True Negative (TN) and False Positive (FP) when evaluating our algorithm in attack mode.	102
5.1	Description of the interactions when authenticating on an SSO system. .	109
5.2	Description of the changes required on an SSO when adding a dynamic canvas check step.	111
5.3	Description of the new authentication scheme with our plugin and new components.	116
5.4	Fingerprints number ratio according to the threshold and weight set . . .	119
5.5	Distribution of the fingerprint collection time	120
5.6	Distribution of the linking algorithm analysis time and linear regression .	121

List of tables

2.1	Summary of the techniques used and results obtained by different detection and classification studies.	39
3.1	Distribution of first-party & third-party scripts per website category. . .	64
3.2	Summary of security organizations, with the accessed attributes and the presence in the web pages of our dataset.	65
3.3	Number of pages including a multi-factor authentication mechanism or a bot detection technique, depending on the page type and the presence of a browser fingerprinting script in the page.	67
3.4	Parameters and results concerning the reauthentication experiment. . . .	70
3.5	Number of websites involved in each step of the validation for the session and basket cookies attacks, and results of the attack on the validated subset	72
4.1	Recapitulative table concerning the desktop devices, OS and browsers on which we run our experiments	81
4.2	BrowserStack devices with OS, OS version and browsers used for the mobile dataset.	81
4.3	Number of plugins and unique plugins supported by browsers, on Ubuntu 20.04	84
4.4	Attributes that are different between ANDROID and IOS, and corresponding value(s). Each value has been observed for all browsers tested on the specified OS.	86
4.5	Layer(s) impacting the value of an attribute, and category of our attributes.	89
4.6	Set of weights to be evaluated, and corresponding weights for the canvas attribute	99

Chapter 1

Introduction

1.1 Context

The Web has reshaped our lives. From buying a train ticket to checking a cooking recipe, or watching live events from all around the world, Internet became essential to everyone's professional and personal life. As the Web expanded, users and websites have shared more personal data that needs to be protected from attackers. In this context, web authentication provides users a way to ensure they are the only ones allowed to access private data. Historically, such systems were designed only with a password verification step. These systems have suffered several attacks, such as phishing or dictionary ones. As these attacks have become more sophisticated, password-based systems are no longer sufficient to ensure security on the Web.

On the one hand, websites and regulation authorities encourage users to adopt more secure mechanisms, such as Multi-Factor Authentication (*MFA*). Multi-Factor Authentication consists in verifying a user's identity through multiple channels called *factors*. Commonly used factors are email or mobile devices (e.g., an SMS message), on which the user receives a code to confirm she is in control of the factor. According to the factors used and their characteristics, they require the user to perform more actions to authenticate. Thus, using several authentication factors often implies a degradation of the user experience due to added complexity. The user experience is essential because it influences the acceptance of the authentication system and users are known to degrade security in search of usability. Users might be more willing to perform additional actions during the authentication attempt for sensitive accounts, such as banking. More generally, authentication systems must decide whether they prefer to strongly strengthen the security while degrading the user experience or if they prefer to adopt a security mechanism that have a lighter impact on the user experience but that might be easier to attack.

On the other hand, browser fingerprinting is a stateless and permissionless identification technique based on the collection of attributes. Rather than setting a server-side generated identifier on a client-side storage space, such as cookies, browser fingerprinting collects client-side information about the browser, device, and configuration to form an identifier. The uniqueness property of browser fingerprinting has been observed in several studies [96, 117, 103], which present varying uniqueness percentages (from 30% to 80-90%) due to the different composition of their datasets. As browser fingerprinting collects client-side information, this information can evolve due to changes in configuration or because of a browser update. Because of such evolutions, a fingerprinting changes over time. However, to be reliable at identification, fingerprints collected from the same browser instance at different times should be identified as such, that is, *linked*. This linking problem has been studied in the state of the art to study its use for tracking devices [150, 119]. Other studies have monitored the usage of browser fingerprinting in-the-wild and found the technique is mostly used for tracking [125, 85, 84, 97] and bot detection [151, 111]. Finally, other work has discussed the usage of browser fingerprinting to enhance web authentication. They highlight the potential security improvements introduced by this technique [145, 130, 87]. These studies are mainly theoretical and do not propose an evaluation of such an authentication system.

1.2 Objectives

The main objective of this thesis is to build an understanding of how browser fingerprinting can increase the security of web authentication systems.

In the state of the art, studies that measured the use of browser fingerprinting in-the-wild used automated tools to crawl the Web. They often visit public pages, such as home pages or random pages reachable from the website's home page. By doing so, they miss sensitive pages that require specific interactions, such as filling out sign-up or sign-in forms that process authentication events.

The **first objective** is to measure if browser fingerprints are collected on sensitive pages, and to what end. To this end, I will study the following research questions:

RQ1: Are browser fingerprints collected in the wild, on pages that process sensitive data that needs to be protected?

RQ2: How are browser fingerprints used to protect user accounts and websites against stolen credentials and cookies hijacking?

The **second objective** is to study fingerprinting in a controlled environment to measure precisely the causes of the uniqueness and evolution of a fingerprint, and to use

this knowledge to design a fingerprint linking algorithm for web authentication. Studying these following research questions will help fulfill this objective:

RQ3: How does the change of a hardware or a software component in a device affect its browser fingerprint?

RQ4: What is the impact of a browser update on the fingerprint and can it be anticipated or even predicted?

RQ5: How to design a browser fingerprint linking algorithm for authentication that combines efficiency and reliability?

The **third objective** consists in implementing such an authentication system and evaluating it to measure the security gains and the impact on the user experience. I will study the following research questions to tackle this objective:

RQ6: What are the user experience concerns that should be addressed in a web authentication system that uses browser fingerprinting?

RQ7: Does the usage of browser fingerprinting for authentication provide security improvements and high usability?

Throughout this manuscript, I answer each of these questions by studying the existing literature and technical reports from browser vendors, by developing tools and proof of concepts, and by performing data analyses on controlled and in-the-wild datasets.

1.3 List of Scientific Publications

Parts of this thesis are adapted from the following papers:

[94] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. **FP-Redemption: Studying Browser Fingerprinting Adoption for the Sake of Web Security**. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 18th International Conference, DIMVA 2021* - <https://hal.inria.fr/hal-03212726/>.

[95] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. **An iterative technique to identify browser fingerprinting scripts**. *Arxiv preprint* - <https://arxiv.org/abs/2103.00590>.

Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. **FP-Controlink: Studying browser fingerprinting in a controlled environment to develop a linking algorithm for web authentication**. Under submission.

1.4 List of Tools and Prototypes

During this thesis, I developed several prototypes and tools to collect and classify data. To increase the reproducibility of my results and encourage open research, I publicly share the following implementations:

- A browser extension to monitor browser fingerprinting attributes accesses, used in [Chapter 3](#) [4].
- An implementation of our script classification technique used in [Chapter 3](#) [5].
- The code of our bash scripts and configuration files for our desktop controlled environment in [Chapter 4](#) [3].
- A proof-of-concept authentication scheme leveraging browser fingerprinting [2].

1.5 Outline

This thesis is organized as follows.

[Chapter 2](#) presents the context of this work. I explain the evolution of the Web since its birth and its consequences on diversity of devices and browsers used to browse the Web. I review existing threats targeting authentication systems and sensitive information, such as phishing, data leaks, and present existing techniques to strengthen authentication mechanisms, such as multi-factor authentication and risk-based authentication. I describe browser fingerprinting, what it consist in and what information it collects to identify users. I present the different uses of fingerprinting, from tracking to security. I conclude this chapter by presenting the current limitations of the state of the art and how this thesis will explore new aspects.

[Chapter 3](#) explores the uses of browser fingerprinting to enhance web authentication. I manually browse a dataset made of sensitive pages from various website categories and detect scripts collecting the browser fingerprint of their users. I present the adoption of browser fingerprinting by these pages, as well as its use to complement existing security features, such as multi-factor authentication and bot detection. I evaluate the resistance of websites against 2 attacks that target the authentication process, namely stolen credentials and cookies hijacking.

[Chapter 4](#) leverages a controlled environment to design a browser fingerprint linking algorithm. I collect fingerprints from various browsers, on both desktop and mobile devices. I measure the sources behind the diversity of browser fingerprinting attribute values and evaluate the stability of attributes over browsers updates. I leverage this knowledge to build a linking algorithm and evaluate it against an in-the-wild dataset.

Chapter 5 presents our authentication scheme using browser fingerprinting. I present user experience concerns for authentication systems leveraging browser fingerprinting. I address these as well as the state-of-the-art security requirements by implementing a risk-level authentication scheme based on browser fingerprinting in a real authentication system. I provide an evaluation on both security and user experience of the authentication system and discuss its limitations and potential improvements.

Chapter 6 summarizes key results and insights about the contributions of this manuscript. Finally, I provide short-term and long-term perspectives concerning browser fingerprinting and its evolution.

Chapter 2

State of the Art

2.1 Context

2.1.1 Birth of the Web

The Web was created by Tim Berners-Lee at *CERN* in 1989 to connect researchers and allow them to share resources more easily. It was based on 3 technologies:

- **HyperText Markup Language** (*HTML*): a description language that presents information in a structured way,
- **Uniform Resource Locator** (*URL*): a string that identifies resources on the Web,
- **HyperText Transfer Protocol** (*HTTP*): a protocol for machines to communicate and transfer HTML documents.

The first browser developed to understand and interpret these technologies was *WorldWideWeb* only accessible on NeXT operating system [83]. The first cross-OS browser was launched in 1992 with the first line-mode browser [26].

The Mosaic browser was created at the National Center for Supercomputing Applications (*NCSA*) while Netscape was created in 1994. Mosaic was licensed by Windows, and was used to create **INTERNET EXPLORER** in 1995. These 2 browsers were the 2 major ones at the end of the *90s* and tried to become dominant. Known as the *first browser war*, this period was opportune to develop new technologies to become the best browser. Because of this, the innovative browsers were preferred by web developers, and many browsers tried to pretend to be each other to stay competitive. This instability is reflected in the history of the **User-Agent** string and is still the reason why **User-Agent** strings are formatted strangely [55]. The **JAVASCRIPT** language was created and released by Netscape. It allows the Web to become dynamic by providing APIs to interact with the user. In response, Microsoft created the **CSS** language (*Cascade-Style Sheet*) to add style to the HTML pages. As Microsoft installed **INTERNET EXPLORER** by default within its Windows OS, it allows **INTERNET EXPLORER** to be reached by many users as

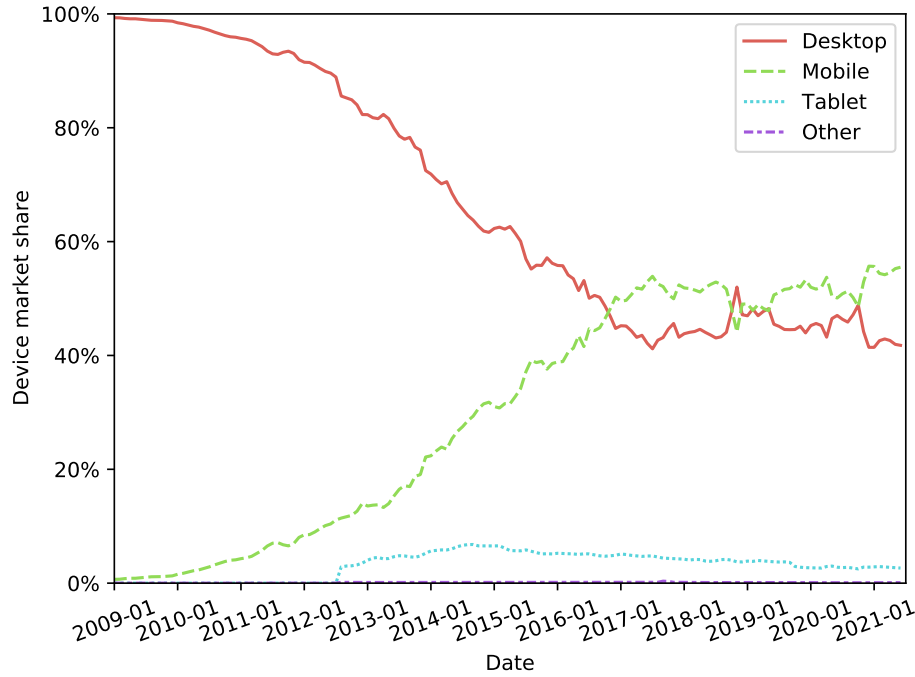


Figure 2.1: Evolution of the Web device market share from 2009 to 2021.

a default choice. By 1999, INTERNET EXPLORER owned 99% of the browser market. In response, Netscape went into making its code open source and created the not-for-profit Mozilla organization, that created the FIREFOX browser in 2002 [63]. Apple launched SAFARI on its MacOS system in 2003 [82]. CHROME was released in 2008 [22] while EDGE was launched in 2015 to replace the aging INTERNET EXPLORER browser.

2.1.2 Web evolution

Increasing diversity of devices. Originally built for desktop devices, the Web is now accessed by more and more diverse devices. Figure 2.1 present the evolution of the distribution of the device market share from 2009 to 2021. With the birth of smartphones in 2007, mobiles started to massively access the Web. Since 2017, they represent 45% to 55% of the devices accessing the Web. Nowadays, many other devices can browse the Web, such as connected watches, TVs and even cars. This increases the need for websites to adapt to users and their devices.

Increasing diversity of APIs. JAVASCRIPT was developed to transfer a part of the logic treatment from the server-side to the client-side. It allows websites to increase the possible interactions with the user. The language keeps evolving along revisions to both adapt to the increasing diversity of devices connecting to the web and to introduce new

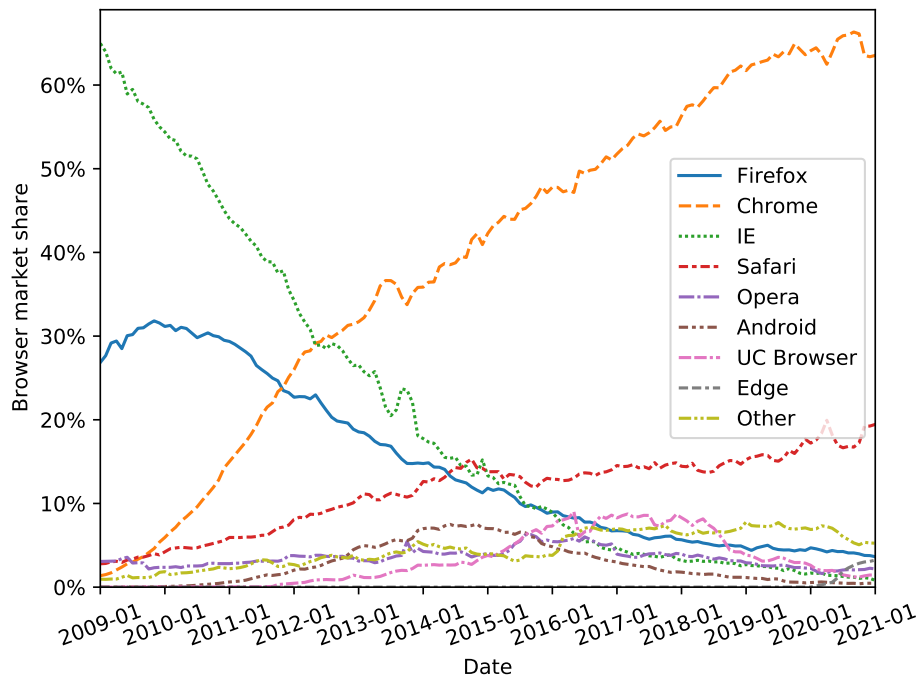


Figure 2.2: Evolution of the browser market share from 2009 to 2021.

APIs to develop new functionalities, such as accessing the network information of the connected device or connecting a VR headset to the device. While the standardization of JAVASCRIPT is ensured by ECMA INTERNATIONAL, browser vendors often develop non-standard APIs to test new functionalities on their users. For example, CHROME implemented the non-standard `Keyboard API` [58] to detect the keyboard layout of the user and get the best of it. This increases the diversity of APIs available in browsers.

Evolution of the browser market share. As we explained in [Section 2.1.1](#), the *first browser war* lead to the birth of CSS and JAVASCRIPT, which are now omnipresent on the Web. More generally, it was the opportunity to develop new features to attract more users. The same situation happened at the end of the *2000s* and during the *10s*, during the *second browser war*. While INTERNET EXPLORER was declining, other browser vendors raced to attract more users and become dominant. The evolution of the browser market share from 2009 to 2021 is presented on [Figure 2.2](#). Since a couple of years, the situation has stabilized. CHROME now represents 65% of the browser market share, followed by SAFARI with 15%, FIREFOX and EDGE with 5% each. The remaining 10% are shared between less-popular browsers.

Consequences and browsers orientation. The abundance of APIs developed by browsers during the *second browser war* lead to many privacy issues for users. Some

browser vendors, such as MOZILLA, APPLE or BRAVE, have centered their browser on privacy. They propose user-friendly mechanisms to manage privacy settings and are leading the removal of functionalities having privacy or security issues for users. For example, BRAVE did not install FLASH by default [16] and FIREFOX disabled FLASH in 2019 [43], more than one year before its official end of life.

2.2 Web authentication

2.2.1 Concept

2.2.1.1 Definition

On the web, *authentication* is a process that allows a system to verify the identity of a user. It consists in requesting an identity proof to the user. If the identity proof can be verified by the system, the authentication is successful.

A *factor* or *authentication factor* refers to the identity proof the user submits to a system to authenticate. The term *credentials* generally includes the factor(s) and the user identifier, which can be the email address of a user, her username or her phone number.

The lifetime of an authentication factor can be summarized in 3 main steps:

- **Enrollment:** This step allows the system to register the factor of the user. During this step, the system stores the factor for future checks. The enrollment is often performed during the account creation. This way, the system has the guarantee that the user that creates the account owns the authentication factor.
- **Verification:** This step is the one happening during the authentication attempt. The system asks for the factor and compares it to the one stored during the enrollment step.
- **Recovery/Revocation:** This step allows the user to change his factor if it has expired or if the user lost it. It ensures the factor can be used for a long period of time. Additionally, the user must be able to revoke a factor if she no longer has access to it.

An *authentication scheme* covers the collection and verification of all the factor(s) and feature(s) used to authenticate the user. While an authentication scheme aims at being secure, it must also compose with the user experience. If users have to perform too many constraining actions, they will hesitate to adopt the authentication scheme. Thus, it is essential for authentication schemes and systems to find a good balance between a satisfying security level and a limited degradation of the user experience. In this context, a *Single-Sign On* (SSO) is an authentication system that allows a user to authenticate once and have access to several services. These services interact with the SSO system to check the authentication state of the user and verify its session, but all these steps

are unnoticeable by the users. The main advantage of a SSO system is to minimize the number of factor to setup and memorize. However, the loss of the authentication factor(s) used by the SSO compromises the whole set of applications available behind the SSO system and is the major drawback.

2.2.1.2 Password and flaws

The Web mostly relies on *Single-Factor Authentication* (SFA), which relies on the collection and verification of one factor to authenticate. On the Web, the most used factor is the *password* [131]. This factor has the advantages of being easy to setup by websites during the enrollment step, easy to use during the verification step, and easy to choose and remember by users. However, the password factor suffers from several security flaws:

Weakness. As the password is a string, users can put anything in it. This can result in weak passwords as users choose short passwords with common Latin characters in it. Florencio *et al.* [100] studied the strength of passwords in the wild. They collected 540,000 authentication data, including passwords, and studied their length and complexity. They showed around 78% of the passwords are only lowercase letters, 20% being only formed of digits.

Predictability. While passwords are created by users, they often choose common patterns, including names, dates or phrases [152, 146]. This creates a social engineering vulnerability because attackers can study their victims by collecting data about them and try to guess the password based on the information they collected.

Reusability. The multiplication of services on the Web increased the number of websites users have to authenticate on. As a consequence of this phenomenon, several studies highlighted and alerted about the reuse of passwords by users. Wash *et al.* [153] conducted a survey over 134 participants and found out the participants tend to reuse each password on 1.7 to 3.4 different websites. Even if passwords are not directly reused, users tend to just modify existing passwords to create new ones [113].

2.2.2 Threats and attacks

The password factor, and more generally the web authentication mechanism, is under constant attack. In this section, we present the most common attacks against web authentication systems.

Compromised credentials. The most common attack against an authentication system relies on the compromising of the authentication factor—in many cases, the password. This can happen in an active or passive way. Active techniques rely on the action of an attacker to steal a password, by using various techniques, such as *phishing*.

Phishing is a technique where an attacker builds a fake authentication page to collect the user credentials. The fake page is designed to mimic the real page where the user usually enters her login information. For example, if an attacker aims at stealing the banking credentials of users, he will mimic the authentication page of the bank by creating a new page that will look and behave similarly. Then, the attacker distributes the URL of his page—often via an email—and makes the user believe she needs to click on the link to authenticate. Several techniques are used to make the user believe the email is a real one, such as a billing problem or an expiration date [75]. The goal of the attacker is to give the user confidence [105]. The user must not suspect an attack or she will change her password immediately, leading to an unsuccessful attack and an increasing attention about this kind of attack. Then, phishing is both a technical and social engineering attack.

While phishing emerged in the 90s [54] and is almost as old as the Web itself, it is still one of the most prevailing attack [6]. As the phishing pages generated gained likeness with the targeted ones, previous techniques that could help the users detecting a phishing page are no longer sufficient. For example, the lack of TLS certificate was previously described as a strong indication that the page was a phishing one. Attackers focused on increasing the percentage of phishing pages with a certificate, that reached 83% in 2021 [6]. Phishing mainly targets financial institutions (24.9%), social medias (23.6%), web mails (19.6%) and payment systems (8.5%). Phishing aims at raking large and targeting a large number of users for a single attack. When the attack specifically targets a user or group of users because of their positions and ability to perform sensitive operations such as banking or financial operations, it is called *spear phishing*.

Passwords can also be compromised in a more passive way. This is the case when a database of passwords is not well protected and can be publicly disclosed or accessed without restriction. When the database is found by attackers, they can query it, find credentials and try to login on the deficient system. Before, this attack required users to often check if their credentials were compromised [53]. Since 2016, the *General Data Protection Regulation* (GDPR) requires the authentication system operator to warn the users about this kind of events [51].

Session hijacking. While many attacks exist to target the authentication process of a website, session hijacking is a different attack. Once a user has authenticated, her valid authentication attempt is kept under a session state that allows the website to know the user owning this session already successfully authenticated previously. Technically speaking, the session state is kept in cookies. In this context, the goal of the session hijacking attack is to intercept the cookies owning the session state to gain access to the session of the targeted user. Several techniques can be used to steal these cookies, such as:

- Network monitoring: The attacker reads the network traffic and collects the desired information,
- Cross-site scripting (XSS): An attacker attacks the website and gain the possibility to send malicious JAVASCRIPT code in the targeted website, and use it to access the cookies of the users.

Sivakorn *et al.* [138] studied the impact of the session hijacking attack against major services. During 30 days, they collected data from a network tap and found 29M requests were exposed, concerning 282,000 accounts. They showed several major services were vulnerable, such as Google, Youtube, or Amazon. They also studied the defenses proposed by browser vendors, and found they can reduce the attack surface, yet not offering a full protection.

Bots. In a web context, a *bot* is an automated software that browse the Web. Several bots have legitimate goals, such as search engine bots. These are bots used by search engines to crawl and index pages and websites [52, 29, 69]. However, many bots on the Web have malicious purposes, being designed to steal data or cause damages to systems. For example, spam bots can be used to harvest emails from websites and contact lists and send spam or phishing emails [81]. Bots can also be used to perform DDoS attacks, or automate the attacks we described previously, making them more dangerous because they can target many users in a short period of time. It is estimated malicious bots represent 25% of the web traffic in 2020 [11]. It highlights the threats bots represent on the Web.

Because of the multiplication of threats on the Web, many security improvements and defenses were proposed and implemented to protect users against them. The remainder of this section presents these improvements and defenses.

2.2.3 Protecting data access

Encryption. The first defenses concern the way data is sent on the Web. First, the HTTPS protocol intends to use the TLS/SSL encryption technology over the classic HTTP protocol [56]. It allows users to exchange data under an encryption layer that protects against cookie hijacking and other threats. Felt *et al.* [98] measured the current adoption of HTTPS on the Web. They collected data from Chrome and Firefox users from 2014 to 2017, and showed the proportion of pages loaded under HTTPS keeps growing, and reached 58 – 90% on Chrome depending on the OS used, and 50-57% on Firefox. They also measured disparities according to the region and country of the user.

In addition to HTTPS, The *HTTP Strict Transport Security* (HSTS) header has been designed to use HTTPS by default. HSTS is an HTTP header a server adds to its response. It tells the client to always reach it with HTTPS for a certain period of time.

The browser receiving the response will store this information and will send the future requests to this domain directly with the HTTPS. Felt *et al.* [98] showed the HSTS header is only available on 3% of websites on the Alexa Top 1M.

Restricting cookies access. Browsers also implemented several defenses to protect cookies against interception and other malicious usages [57]. The **Secure** cookie response header allows websites to define cookies that must only be sent over HTTPS, preventing the interception of cookies via network monitoring. Websites can use the instruction **HTTPOnly** that will prevent the access of cookies via **JAVASCRIPT**, preventing them from being stolen via XSS attacks. Finally, websites can use the **Domain**, **Path**, and the experimental **SameSite** instruction to restrict the access of cookies and protect against CSRF attacks.

2.2.4 Bots protection techniques

Due to the growing importance of bots, several measures tend to control their possibilities and defend against them. The **robots.txt** file allows websites to define a list of folders and files crawlers and bots should not access. Bots should first read the content of this file and adapt their behavior accordingly before starting crawling and indexing the website. While search engine bots often respect the instructions of the **robots.txt** file [52, 29, 69], nothing forces them to do so. Additionally, the **robots.txt** file is available publicly and malicious agents can see if a website wishes to protect resources against crawlers.

IP address reputation. Bot detection techniques can refer to the reputation of IP addresses to classify users. Previously, bots were often run behind proxies and specific IP addresses that were easy to detect and block. Now, bots often use many IP addresses, including residential ones that have an excellent reputation. This evolution highlights the limits of this technique.

Traffic analysis. Other techniques analyze the behavior of users to detect suspicious behaviors that could be the work of bots. They monitor the server logs as well as the user interactions, such as mouse movement or keyboard usages to detect suspicious behaviors [109], such as too many pages browsed in a limited period of time or an absence of mouse movements or click. However, these techniques need a large amount of data before being able to accurately classify the traffic.

Captchas. They are Turing tests that aim at distinguishing bots from humans. When a website has some doubt about the nature of a user, it can expose a captcha to the user whose goal is to solve it. If she can, she is considered as a human, otherwise she might be a bot. Several techniques exist and propose various tests to the user. 3 techniques are presented in Figure 2.3: a textual captcha, a Google's RECAPTCHA, and a Geetest captcha. However, the technique is subject to shortcomings. i) First, Bursztein *et al.* [91] found that users might fail solving captchas. ii) Second, and more

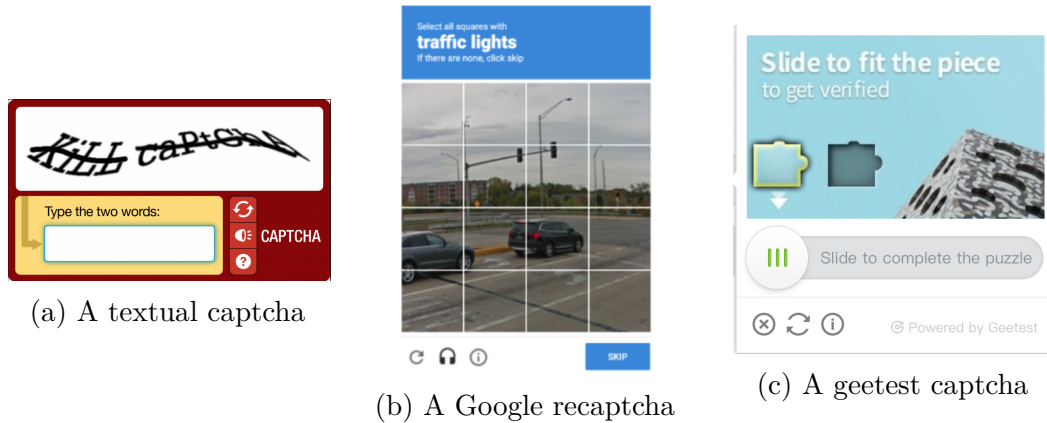


Figure 2.3: Examples of captcha techniques

important, Sivakorn *et al.* [139] showed in 2016 that it was possible to solve Google's RECAPTCHA automatically, again questioning the efficiency of the technique.

2.2.5 Improving authentication

2.2.5.1 Strengthening password creation

As passwords are the most dominant authentication factor, a common technique to improve security of an authentication system relies on the strengthening of the password creation rules. Shay *et al.* [137] conducted a study of 470 students and staff belonging to the *Carnegie Mellon University* (CMU) to measure their opinion about the new password policy of the University. The University system used to have no rule when creating a password. The University updated its policy and the rules to create and update a password:

- The new password must contain at least 8 characters, and include at least one upper-case letter, one lower-case letter, one digit and one symbol,
- After removing the non-alphabetic characters, the password cannot match a dictionary word,
- The password cannot contains 4 occurrences or more of the same character.

In their response, participants say they needed an average of 1.77 tries to create this new password and estimate at 1.25 the number of attempts to authenticate on the system. Similarly to the study presented in Section 2.2.1.2, more than 80% of the participants admit to reuse passwords in different authentication systems. They also studied the values of password thanks to the answer of the participants. In average, the passwords contain 10.49 characters, which is more than the requirement of 8 characters. They are formed by 5.94 lower-case letters, 1.54 upper-case letters, 2.70 digits, and 1.70 symbol.

While the strengthening of the password creation rules is crucial for security, several attacks and threats presented in [Section 2.2.2](#) collect the complete password, such as phishing or data leak. In this context, the single use of password exhibits several flaws.

2.2.5.2 Multi-Factor Authentication

Multi-Factor Authentication (MFA) aims at fixing these flaws. Instead of relying on a single authentication factor, MFA leverages additional factors to enhance web authentication [128]. During a web authentication attempt, each factor must be verified to validate the identity of the user. The authentication system can use as many factors as necessary—2 factors (*2FA*), 3 factors (*3FA*)—but has to find a good balance between user experience and security. If the actions required to authenticate are too complex or too numerous, users will become uncooperative to adopt the authentication scheme.

Authentication factors can be divided into 3 main categories:

- *Knowledge factor*: This is something the user knows, such as a password;
- *Ownership factor*: This is something the user owns, such as an email account, a smartphone or a physical token. To authenticate and prove that she owns this factor, she is often provided a code received on the factor. It can be sent by SMS or via a smartphone application to prove the user owns a mobile phone, or by email to prove she owns the email. The code is often called a *One-Time Password* (OTP), which is only valid for this specific authentication attempt, often with a short validity duration period. When typing this code on the authentication page, the server compares the submitted code to the one sent on the factor. If it matches, then she can authenticate;
- *Inherence factor*: This is something the user is. She does not have anything to do to get this factor because it is based on the nature of the user. It concerns properties of the user, such as a biological fingerprint, or a behavior pattern. These systems often require specific scanners to collect the property of the user during an authentication attempt. This category also includes factors concerning the context of the authentication attempt, such as the IP address of the user.

All the factors presented above do not improve the security in the same ways, and have different impacts on the user experience. Some factors, such as behavior patterns or based on the context of the authentication attempt, are called **implicit** because they do not require a user-specific action to be collected and verified and do not cause a degradation of the user experience during the authentication attempt. By opposition, many factors are said to be **explicit** because they require an action of the user to be collected—typing a password, a OTP, putting a finger on the fingerprint scanner. Explicit factors are often more secure than implicit factors because they cannot be collected as easily, meaning an attacker must rely on a user interaction to collect the explicit factor. Because of this user interaction, the usage of an explicit factor degrades the user

experience. The main challenge for authentication systems relies on the balance between a security improvement and a user experience degradation.

Authentication systems can integrate factors in an immediate or delayed mode. An **immediate** factor will be verified during the authentication attempt, while a **delayed** factor will be checked during the session duration. To minimize the impact on the user experience, the implementation of a factor in a delayed mode often implies the use of an implicit factor, such as an IP address check.

Ometov *et al.* [128] discuss and evaluate the current state of the art of the factors being used in authentication systems. They compared a list of 16 authentication factors on 6 properties: i) Universality - the factor is present for each user, ii) Uniqueness, iii) Collectability, iv) Performance, v) Acceptability, vi) Spoofing. As we explain above, the factors that are easy to setup and use (password, token) are the ones considered as the most easily spoofable. Oppositely, the factors that offer high uniqueness (ocular-based, electrocardiographic recognition, DNA recognition) suffer from low collectability.

Adoption of MFA Several studies tried to quantify the adoption of MFA on the Web. In 2015, Petsas *et al.* [129] studied 100,000 Google accounts and measured an adoption of 2FA of only 6.4%. Quermann *et al.* [131] studied the mechanisms used by 48 different services on the Web. First, they observed usernames are the most common identifier (65%), followed by email addresses (45%). While the primary authentication factor is a password or a PIN code—or a combination of both—they measured 7 different additional factors in the systems they studied: i) an SMS code, ii) a code received via a code generator application installed on a smartphone, iii) an offline code, iv) a smartphone notification, v) a U2F key, vi) a key fob, also named a token, and vii) a biometric proof. They showed the most implemented factors are the SMS code, the generated code and the smartphone notification. They observed a specific behaviour when analyzing the systems set-up by German banks. They found the second factor is only required when performing sensitive operations, such as transferring money, but is not necessary when performing read-only operations, such as checking account balance. Concerning the recovery of the factors, they observed the most used technique to recover the primary factor—the password—is the recovery via an email, where the user is either provided a code to enter on the website or a link which directly provides a form to update the password. The recovery of the second factor is often performed via contacting the support. It has a strong impact on the user experience, as the user has no guarantee to receive a quick response. Otherwise, the researchers measured websites often rely on another factor. For example, a user can use an SMS OTP to generate back her offline codes.

2.2.5.3 Risk-Based Authentication

Risk-Based Authentication (RBA) is a dynamic authentication scheme where the context of the authentication attempt is taken into consideration. It is analyzed and compared to the regular context used by the user to authenticate. Given this information, the system evaluates the risk for this authentication attempt to be fraudulent. If the risk is too high, the system can require the user to provide an additional authentication factor to validate the authentication attempt. Several features can be used to perform RBA, such as: i) The IP address, ii) the HTTP headers (User-Agent, Language), iii) the display resolution, iv) the authentication time, v) canvas fingerprinting.

RBA aims at being implicit: it does not require specific user interaction nor to degrade the user experience, but collects information that can distinguish a suspicious authentication from a regular one. Thus, it increases security while having a much more limited impact on the user experience than MFA.

Evaluation Wiefling *et al.* [155] investigated the adoption of RBA on 8 high-traffic online services: Amazon, Facebook, GOG.com, Google, iCloud, LinkedIn, Steam, and Twitch. They designed 28 virtual identity and created 224 user accounts. They measured the risk levels were computed differently according to the website. GOG.com asked for an additional authentication factor as soon as the IP address was different while Google, Amazon or LinkedIn asked for it when the IP address was coming from a different country than the original authentication. They also tested several combinations of features, and observed the IP address is the most used feature to consider the risk of an authentication attempt, followed by the User-Agent string. They also observed that the email and SMS OTP were the additional factors used the most by websites when the authentication risk was considered too high.

Wiefling *et al.* [154] also conducted a usability survey of RBA on 65 users. They designed a website that proposes 4 types of authentication schemes to the users:

- Password-only,
- 2FA, which asks the user for its password and a code sent via email,
- RBA-location, which uses only a password if the location of the device used for the authentication attempt has been used before. Otherwise, it uses 2FA.
- RBA-device, which uses only a password if the device used for the authentication attempt has been used before. Otherwise, it uses 2FA.

Their users found the 2FA much more annoying and tiring than RBA-location or RBA-device based, with respectively 56% versus 13%/0% and 44% versus 6%/13%. Out of the 3 solutions proposed to enhance password-only, users preferred the RBA-location (94% of acceptance) than RBA-device (87%) and 2FA (75%). They also found large disparities concerning the user acceptance responses for different website categories. Users were much more disposed to accept any of the 3 solutions on a banking website than a social

network. One of the reason advanced for these answers are the level of trust in the website: people tend to trust much more a banking system than a social media website. When asked about the perception of the security level provided by the 4 authentication schemes, users were more satisfied with the 2FA (94%) than the RBA-device (87%), the RBA-location (75%) and the password-only scheme (23%). Their global results perfectly illustrate the balance to find between security and user experience.

2.3 Browser fingerprinting

2.3.1 Definition

A **Browser fingerprint** is a stateless and permissionless technique that can identify a browser [116]. Cookies and other stateful techniques generate an identifier on the server-side and use a storage available on the client side to store it. It means the identification technique relies on the state of the storage used. Because a browser fingerprint is formed by collecting information about the user's browser, it does not require to generate and store an identifier.

In this manuscript, the terms *browser fingerprint* and *fingerprint* represent the set of attributes forming the fingerprint and can be used interchangeably. The terms *browser fingerprinting* and *fingerprinting* represent the whole technique used to collect a fingerprint. When it comes to the goal of the technique, I sometimes stated that it can identify a user. Browser fingerprinting collects information about the browser and the device on which it is installed, but also collects information about the configuration of the user. While these information are insufficient to identify a single user, they provide an added value that helps to distinguish similar browser instances.

The attributes forming a fingerprint are in majority properties and information accessible via JAVASCRIPT. To collect the fingerprint of a user, the server needs to load a script besides the HTML page returned to the user. Once the script is loaded, it accesses the properties and functions required to form a fingerprint. After the collection, the script can either perform some treatment on the client side or send the fingerprint to the server for backend processing.

2.3.2 Properties

To be effective at identifying, the fingerprint needs to have the following properties:

- **Uniqueness:** Because stateful identification techniques generate the identifier that will be stored on the client's device, they can ensure the identifier will be unique among all their users. As browser fingerprinting relies on the information gathered on the user's browser, it does not have this possibility. Instead, its goal is to collect information that identify the device of the user: not only her browser, but

also her OS, her hardware components, and so on. By collecting attributes giving information about various parts of the device, the uniqueness will increase.

To measure the uniqueness of an attribute, studies use the *entropy* as a reference measurement unit for the attribute values. In computer science, entropy consists in measuring the number of bits necessary to represent data. For example, if there are 536 possible data values for a given attribute, the associated entropy is 9.06878.

- **Stability:** Stateful identification techniques can rely on their identifier until it has either expired or been removed by the user. A browser fingerprint is more volatile. It evolves due to changes in the user configuration, browser updates or other events. This will change the value of some attributes and make the fingerprint looks different from the one collected before the changes. As stability is essential for identification techniques, browser fingerprinting aims either at collecting attributes that remain stable for a long period of time or being able to *link* a fingerprint with its evolution.

Depending on the final use of browser fingerprinting, some properties might not be necessary anymore while browser fingerprinting might respect new properties to be efficient. In this manuscript, I consider by default the uniqueness and stability properties for the browser fingerprinting technique. When discussing specific uses of browser fingerprinting, I will clearly explicit the new properties required by browser fingerprinting.

2.3.3 Attributes

In this section, I present the attributes composing a browser fingerprint.

2.3.3.1 HTTP Headers.

Some information is sent within HTTP request and contains useful information for the server to understand what type of device made the request, how the response should be sent, or which languages the user prefers. Consequently, configuration changes in the device or browser impact these headers, and the information in the headers has high entropy, meaning they are highly discriminating [117]. HTTP headers are sent by all browsers and can easily be collected by servers, making them ideal attributes to use in a browser fingerprint. HTTP headers are defined in the Chapter 5 of the RFC 7231 [71]. I present the main HTTP headers used for fingerprinting, sorted by alphabetic order:

Accept. Indicates the **Multipurpose Internet Mail Extensions** (*MIME*) types accepted by the browser. The values are ordered by preference with quality information, represented in the header with the *q* letter. Examples of values are given below.

Accept header value	Source
text/html,application/xhtml+xml,application/xml;q=0.9, image/webp,image/apng,/,;q=0.8	Chrome 73 on Windows 10
text/html,application/xhtml+xml,application/xml;q=0.9, */*;q=0.8	Firefox 66 on Ubuntu 18.04

Accept-Encoding. Indicates the data encodings supported by the browser.

Accept-Language. Indicates the list of languages the user prefers. Similarly to the `Accept` header, each value comes with a quality value `q` between 0 and 1. The quality value indicates the user's languages order preference the server should try to respect. The table below provides value examples for this header.

Accept-Language header value	Description
zh-CN, zh; q=0.8, ru-RU, ru; q=0.6	Chinese preferred, then Russian
fr-FR, fr; q=0.8, ja; q=0.6, en-US, en; q=0.5	French preferred, then Japanese, then American English or English

DNT. This header represents the user's `Do Not Track` setting. The value is set to 1 if the user does not want to be tracked, 0 otherwise.

User-Agent. This parameter contains the browser name, browser version, OS name, OS version and can contain additional information about the hardware or engine used depending on the device. The table below contains a list of `User-Agent` that can be found in the wild.

User-Agent header value	Source
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.86 Safari/537.36	Chrome 73 on Windows 10
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:66.0) Gecko/20100101 Firefox/66.0	Firefox 66 on Ubuntu 18.04
Mozilla/5.0 (Linux; Android 7.0; SM-A510F Build/NRD90M) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Mobile Safari/537.36 OPR/51.2.2461.137690	Opera 51 on Android 7.0
Mozilla/5.0 (PlayStation 4 5.55) AppleWebKit/601.2 (KHTML, like Gecko)	Playstation 4

Beyond the values exposed by headers, several studies [96, 145] demonstrate the order of these headers depends on the browser implementation and can be used to verify a browser is the one it pretends to be.

2.3.3.2 JavaScript Fingerprinting

A) Browser properties. The majority of the attributes are available via JAVASCRIPT. I present these attributes grouped by the JAVASCRIPT parent object exposing the information.

Navigator object. Mulazzani *et al.* [123], Nikiforakis *et al.* [125], and Acar *et al.* [84] showed that the order of the properties of the navigator object, as well as the presence and absence of features, can be used to fingerprint the browser and its version. While this technique does not bring additional information compared to the **User-Agent**, it can be used to verify the nature of the browser.

navigator.appCodeName. Returns the code name of the browser, which is now Mozilla for every modern browser.

navigator.appName. Returns the name of the browser, which is now Netscape for every modern browser.

navigator.appVersion. Returns the version of the browser.

navigator.buildID. Returns the build identifier of the browser. This property is now set to a fixed value in all major browser for privacy reasons.

navigator.cookieEnabled. Returns a boolean which indicates whether cookies are enabled or not.

navigator.deviceMemory. Returns the amount of memory on the device, in gigabytes. This property is only available on CHROME 63 or higher, and Chromium-based browsers.

navigator.doNotTrack. Returns 1 if the user has requested not to be tracked via the Do Not Track setting, 0 otherwise. This value should be equal to the one given by the HTTP header. The parameter is available in SAFARI and INTERNET EXPLORER by calling `window.doNotTrack`.

navigator.getBattery. Returns an object containing information about the battery, such as the `charging`, `chargingTime` or `level` properties. The fingerprintability of the battery object has been explored by Olejnik *et al.* [127].

navigator.hardwareConcurrency. Returns the number of logical processors available in the browser.

navigator.language. Returns the first string of the `navigator.languages` property.

navigator.languages. Similar to the HTTP header `Accept-Language`, it returns a list of languages representing the user's preferred languages. Contrary to the header, it does not contain the quality values but the list is ordered.

navigator.maxTouchPoint. Returns the maximum number of touch contact points supported simultaneously by the device.

navigator.mimeTypes. Returns an array containing the MIME types supported by the browser. For each MIME type, it gives information about the type supported, a description and the filename associated with the type. An example is given in the table below.

Description	Suffixes	Type
Shockwave Flash	swf	application/x-shockwave-flash
Shockwave Flash	spl	application/futuresplash
Silverlight Plug-In		application/x-silverlight-2
Silverlight Plug-In		application/x-silverlight

navigator.platform. Returns the platform the browser is running on. While the information is redundant with the OS element contained in the `User-Agent` parameter, it can be used to verify the consistency of the OS claimed.

navigator.plugins. Returns the list of plugins installed in the browser [120, 96]. For each plugin, it provides the plugin name, version, description and the filename associated. This attribute is one of the most unique, with an entropy between 9.4 and 15 bits [96, 117, 103].

navigator.product. Always returns `Gecko` for every modern browser.

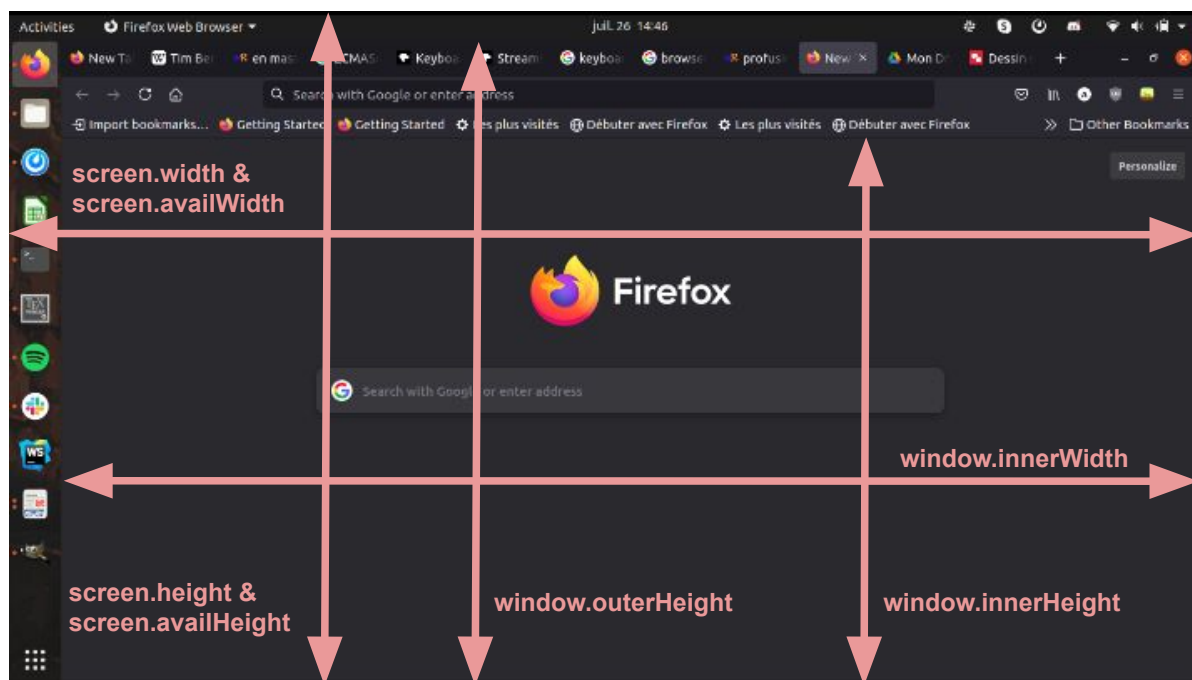
navigator.userAgent. Returns the user agent of the browser, which is the same string as the one sent in the HTTP headers.

navigator.vendor. Returns the vendor of the browser.

Date.getTimezoneOffset(). Returns the difference between the UTC timezone and the timezone of the user, in minutes.

MediaDevices.enumerateDevices(). Returns a list of the available media inputs and outputs of the device, such as a camera or a microphone. The API does not require any permission to collect this information, but provides additional information, such as the media full name if a media is active or if the media permissions are granted.

Screen and window size properties. The browser exposes several properties to measure the page, window, and screen sizes. For example, `window.innerWidth` and `window.innerHeight` returns the width and height of the visible content of the browser page. The complete list of properties used in a browser fingerprint is available through the `screen` and `window` objects. The figure below presents these properties and their signification.



Bot detection attributes. In 2019, Jonker *et al.* [111] showed the presence of specific attributes could tell if a browser was driven by an automated tool. Such attributes are

JAVASCRIPT properties or functions that are added by the automated tool in the browser. Thus, these properties are accessible via the `window` objects. Examples of added properties are: `callPhantom` or `_phantom` by PHANTOMJS, `_selenium` or `selenium_unwrapped` by SELENIUM and `__nightmare` by NIGHTMARE.

Sensors objects. Bojinov *et al.* [90] showed it is possible to use a device's accelerometer and sensors to fingerprint a user. The presence or absence of specific sensors also provides information about the nature of the device. Currently, it is possible to check for the presence or absence of an accelerometer, a gyroscope, a magnetometer, a proximity sensor, and a touch device. The following code checks the presence of a gyroscope and a proximity sensor on the device.

```
1  if (typeof Gyroscope === "function") {  
2      // Gyroscope exists on the device  
3  }  
4  
5  if ("ProximitySensor" in window) {  
6      // Proximity sensor exists on the device  
7  }
```

B) Rendering APIs. Browsers implement rendering APIs to display advanced representations of content to users. In the context of browser fingerprinting, rendering can produce a wide diversity of side effects that are interesting to analyze in order to capture the true nature of the browsing environment.

Font detection. The list of fonts installed on a user's device used to be accessible to the server through the `Fonts.enumerateFonts` Flash API. It was one of the most unique attributes with an entropy between 8 and 14 bits [96, 117, 103]. Since Flash became deprecated and browsers started to disable FLASH by default [43, 74], this technique has become near useless. To compensate for the loss of FLASH and the fonts list, a new technique to detect fonts was proposed in 2013 by Nikiforakis *et al.* [125], namely *font enumeration*. It relies on the fact that font glyphs are rendered with different dimensions according to the environment (OS, browser, etc.). The technique is organized as follows:

1. The script renders a text with a fallback font in a span HTML tag, and measured the width & height of the span;
2. Then, the script renders the same text with the to-be-tested font followed by the font previously used, which will be the fallback font. Similarly, it measured the sizes of the span;
3. If the sizes are different from the sizes measured with the fallback font, it means the to-be-tested font was used to render the text. Then, the font is available on

the system. If the sizes are equal, it means the font used to render the text was the fallback one, hence the to-be-tested font is not present on the system.

As the measured sizes of the to-be-tested font can be similar to the ones measured with the fallback font, it can lead to a *False Negative* (FN). Scripts can use several fallback fonts to avoid this problem. The results can also vary according to the size of the text. A larger rendered text will have more specific sizes, the risk of having false negative will drop, and the quality of the results will increase. A single test is invisible and fast, and can say if a font is installed, but also distinguish several fonts or several versions of the same font. However, this technique tests one font at a time, contrary to FLASH that used to give the full list of installed fonts. The following code tests the presence of the Baskerville font while using the DejaVu font as a fallback.

```
1  const fallbackFont = 'DejaVu';
2  const toBeTestFont = 'Baskerville';
3  const testSize = '72px';
4  const testChar = 'A';
5  const h = document.getElementsByTagName('body')[0];
6
7  // create a span element
8  const s = document.createElement('span');
9  s.style.fontFamily = fallbackFont;
10 s.style.fontSize = testSize;
11 s.innerText = testChar;
12
13 // Measure the size of the span element with the fallback font
14 h.appendChild(s);
15 defaultFontWidth = s.offsetWidth;
16 defaultFontHeight = s.offsetHeight;
17 h.removeChild(s);
18
19 // Measure the size of the span element with the to-be-tested font
    and compare to the sizes given with the fallback font
20 s.style.fontFamily = toBeTestFont + ', ' + fallbackFont;
21 h.appendChild(s);
22 const fontIsPresent = (s.offsetWidth !== defaultFontWidth ||
    s.offsetHeight !== defaultFontHeight);
23 h.removeChild(s);
24
25 console.log(fontIsPresent);
```

Since this technique was discovered, one of the main challenge was to find a small subset of fonts that can uniquely identify most browsers. This is a difficult test in practice because, contrary to the list of fonts available via FLASH, this technique requires to test the fonts one by one, which can be a long process if the script is checking thousands of

fonts. Consequently, the entropy of this attribute will vary a lot depending on the list of fonts.

Fonts rendering. Fifield *et al.* [99] aim at fingerprint users with font metrics. Rather than measuring the presence or absence of a font on a system, they showed similar fonts were rendered differently on different devices. They tested the rendering of 125,000 different Unicode characters on more than 1,000 browsers. Each character was drawn using the five generic CSS font families (**sans-serif**, **serif**, **monospace**, **cursive**, **fantasy**). They showed that 34% of the browsers in their dataset had a unique way of rendering the glyphs. The study revealed that they could deduce the same amount of information with just 43 characters, instead of 125,000, making the test much faster. Their results also highlighted that the most unique glyphs are the most recent ones such as the Indian rupee sign (U+20B9) or the Turkish Lira sign (U+20BA), because their design include more graphical elements that, once rendered, appear to be more unique. It implies this technique could become more effective as new glyphs are added into Unicode in the incoming years.

Emojis rendering. Emojis are Unicode characters that are included in fonts and designed to be rendered as a single glyph [32]. Not all fonts provide emojis, and some systems might not include any fonts that properly support emojis [68]. Furthermore, the rendering of an emoji differs depending on the font, its version, the operating system, the browser, or the graphical libraries [117]. Thus, it is possible to check the presence of a font with emojis by using the font enumeration technique, or to include the emoji in a canvas (see below) and get the value of its rendering using the `toDataURL()` or `getImageData()` functions.

Canvas. The HTML5 canvas API provides a drawing context on a `canvas` tag. It can be used by scripts to draw 2D shapes and render textual content directly in the browser by using the graphical capabilities of the device. The server can tell the browser to render graphical instructions and return the binary value of the rendered image. This allows the server to verify the image and search for pixel differences, or simply hash the image to create an identifier.

Canvas was first studied as a fingerprinting element by Mowery *et al.* [122]. In their study, they performed tests by writing several sentences with specific fonts, font sizes and font styles, as well as a final test that leverages WebGL, an API similar to the canvas API but that allows drawing 3D shapes. They collected data on 300 users and their study shows many differences across users. They obtained around 45 to 50 unique values for each test, which means this fingerprinting technique contains a lot of entropy. Because canvas was shown to be very unique among a large dataset, it has since been used in

many studies. The drawing primitives have also become more complex and include more elements like colors or emojis [117, 19, 103].

The JAVASCRIPT code presented below asks the browser to render graphical instructions that are commonly used for fingerprinting, which generates the image below.

```

1  const canvas = document.createElement('canvas');
2  canvas.height = 60;
3  canvas.width = 400;
4  const canvasContext = canvas.getContext('2d');
5  canvas.style.display = 'inline';
6  canvasContext.textBaseline = 'alphabetic';
7  canvasContext.fillStyle = '#f60';
8  canvasContext.fillRect(125, 1, 62, 20);
9  canvasContext.fillStyle = '#069';
10 canvasContext.font = '11pt no-real-font-123';
11 canvasContext.fillText('Cwm fjordbank glyphs vext quiz, \ud83d\ude03', 2, 15);
12 canvasContext.fillStyle = 'rgba(102, 204, 0, 0.7)';
13 canvasContext.font = '18pt Arial';
14 canvasContext.fillText('Cwm fjordbank glyphs vext quiz, \ud83d\ude03', 4, 45);
15 canvasData = canvas.toDataURL();

```



Cwm fjordbank glyphs vext quiz, 😄

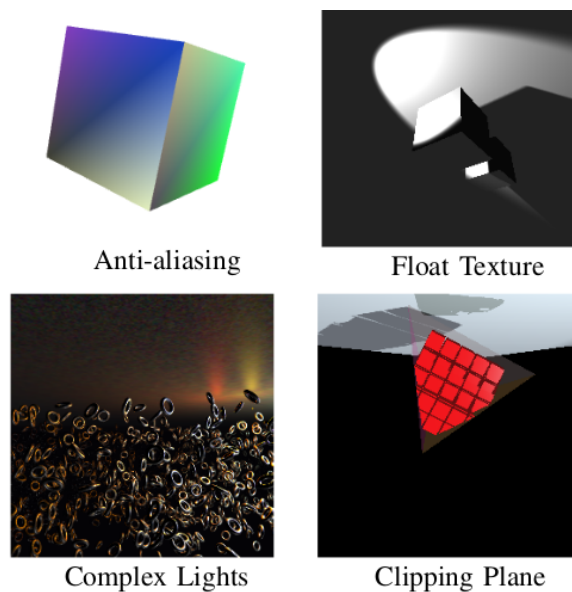
Cwm fjordbank glyphs vext quiz, 😊

In 2016, Bursztein *et al.* [92] used the canvas API to conduct a large study to attempt to identify device families—by opposition to individual devices. They collected 52 million canvas fingerprints that used four drawing primitives (circle, font, bézier curves, and gradients). They were able to distinguish devices across OS families—Windows, Mac OS, Linux, iOS—and across browser families—Chrome, Firefox, Safari. They use this knowledge to identify automated browsers that were using PHANTOMJS to attack Google web accounts.

WebGL. The WebGL API enables the rendering and manipulation of dynamic 3D objects based on the canvas element. It exposes information about the graphical stack of the device to help developers adapt their code to the user device. Two attributes are specifically used in browser fingerprinting, `WebGL vendor` and `WebGL renderer`. The `WebGL vendor` returns the name of the GPU vendor. The following values can be found in the wild: Intel Open Source Technology Center, Google Inc. or ARM. The `WebGL renderer` returns the name of the GPU renderer. The following values can be found in the wild: Mesa DRI Intel(R) Sandybridge Mobile, NVIDIA

Quadro K4000 OpenGL Engine, Mali T-720 or ANGLE (NVIDIA GeForce GTX 750 Ti Direct3D11 vs_5_0 ps_5_0)

The WebGL API also provides a 3D drawing context. The approach is similar to the one used to perform canvas fingerprinting: rendering objects is a complex and difficult task and is based on several hardware and software layers—such as the browser, the OS, the drivers, as well as a graphics card. Thus, sending the same set of drawing instructions to devices with different graphic hardware and software components produces different output values. Laperdrix *et al.* [117] explored the use of the WebGL API for fingerprinting, but did not succeed because the 3D canvas they generated were not stable enough to be integrated into a fingerprint. More recently, Cao *et al.* [93] showed it was possible to generate unique and stable 3D scenes using the WebGL API. They proposed a set of 19 different 3D scenes by varying shapes, models and textures with additional effects on light or camera. Their results show the more complex the 3D scene is, the more unique the values are. The entropy of each scene varied between 1 and 7 bits, but by combining them, they were able to uniquely identify more than 99% of the users. However, their evaluation was conducted using only 3 600 fingerprints, and long-term stability is not well understood. The image below presents some of the scenes they generated.



Audio. In 2016, Englehardt *et al.* [97] crawled the Alexa Top 1M and discovered fingerprinting scripts that processed an audio signal generated by the browser to create an audio fingerprint. This technique is similar to canvas or WebGL fingerprinting because it asks the browser to render an element based on instructions and collect the output. Depending on some hardware or OS feature, the result will differ from one device to another. The researchers built a website to collect data about audio fingerprinting [10].

Out of over 18,500 different submissions, 713 were unique. They estimated the entropy of this technique at 5.4 bits.

WebRTC. In the same study, Englehardt *et al.* [97] also discovered that fingerprinting scripts were using the WebRTC API to collect the public IP address of users behind a VPN or a NAT.

C) Extensions detection. Browser extensions are add-ons installed in the browser that personalize the user's browser. Extensions can add styles to webpages, customize the interface of the browser or add privacy and security functionalities. Starov *et al.* [142] monitored the changes that extensions inject into the DOM (*Document Object Model*) when interacting with the page. Among the Top 10,000 Chrome extensions, 9% of them interacted with the DOM. With a specifically structured DOM, a webpage can infer these extensions from their interactions. Moreover, they collected the configuration of 854 users and showed that 14% of them had a unique combination of browser extensions. This observation shows that the list of browser extensions could be used as a fingerprinting feature.

Sjosten *et al.* [140] proposed another approach to detect browser extensions. They focused on the WAR files (*Web Available Resources*) that extensions load or use, such as images. By testing the presence of these resources on the local file system, it is possible to check if an extension is installed. They downloaded and tested resources for more than 43,000 Chrome extensions and 15,000 Firefox extensions. In total, 12,000 Chrome (28%) and 1,000 Firefox (6.73%) extensions could be identified using resource detection. They also observed that the more popular the extension is—based on the number of users—the more it is likely to be detected. Gulyas *et al.* [104] used this attack to perform a large-scale analysis on 16,000 users. They found 55% users has installed at least one detectable extension and 18% of the users have a unique set of installed extensions. Mozilla fixed this attack in 2017 by randomizing the extension ID when it is installed in FIREFOX [50]. On its side, CHROME proposed a new version of the extension `manifest.json` file, which allows better control over the WAR files that are exposed [21].

Laperdrix *et al.* [118] presented and evaluated a new technique to measure the browser extensions installed by users. They relied on the CSS injected by extensions in a webpage that can modify components, such as divs or buttons. They used the JAVASCRIPT function `window.getComputedStyle` that returns all the CSS properties applied to the targeted element. By comparing the final style of the element and its original value, a script is able to tell if the style has been modified, hence if it has been altered by an installed extension. They analyzed 116,000 extensions for the Chrome Web Store, reported more than 6,500 extensions injected style sheet on pages, and demonstrated they could uniquely identify 4,446 of them.

Karami *et al.* [112] proposed 2 new extension fingerprinting vectors: i) Interception of messages sent from an extension content script and a webpage script, ii) Resources enumeration via HTTP request monitoring. They developed CARNUS, a software system to automatically perform static and dynamic analysis on browser extensions. They fed it with more than 29,000 extensions to measure the vulnerability they defined. They found browser extensions detection techniques are complementary: some techniques can identify an extension while other techniques were unable to.

D) Timing attacks. The goal of timing attacks is to analyze the time taken to perform an operation and use that to infer information. Fingerprinting leverages timing attacks to identify hardware components of the device.

Sanchez *et al.* [136] used this technique to distinguish devices based on a set of CPU-intensive operations. They performed a basic cryptographic operation, generated random values, and repeated the operation dozens of thousands of time. By measuring the small differences between the execution times, and due to processor clock imperfections from production, they claim to extract stable and unique features capable of distinguishing otherwise identical devices—same OS, same hardware. The study was conducted on a set of 176 identical computers. However, they only generated 3 fingerprints from each computer (528 fingerprints in total), and did not look at consistency over time. Furthermore, their best results were based on a prototype built in C, and although they claim that this extends to the cryptography APIs added to browsers, it is unclear from their paper and the results of their prototype in JAVASCRIPT how true this is. Finally, due to the Spectre and Meltdown attacks, the precision of several JAVASCRIPT timers has been reduced [59, 18]. In this context, I could not reproduce Sanchez’s results and believe this attack is not usable in the current state of JAVASCRIPT timers.

Rokicki *et al.* [134] studied the different timing measurement techniques in JAVASCRIPT. More specifically, they measured the efficiency of timers, including `performance.now()` and `SharedArrayBuffer` APIs on several settings, on several browser versions and against various countermeasures. They found the re-enabling of the `SharedArrayBuffer` allow scripts to perform precise timing attacks that were mitigated before.

Timing attacks provide timing measurements that are hardly deterministic. The techniques are often impacted by other parameters such as the CPU usage of the device or the state of a cache, resulting in non-deterministic attributes. To succeed at identifying users, these techniques rely on machine learning pipelines or neural networks. This increases the elements to setup when collecting and classifying the information.

2.3.3.3 CSS fingerprinting

Another way of inferring information about the browser is to use CSS [143]. In particular, CSS provides query strings, which are triggered whenever the associated CSS property

is applied. The code below defines 2 CSS properties, which apply on different device pixel ratios. The URL with the corresponding device pixel ratio will be triggered, thus informing the server about the device pixel ratio. The method does not provide additional information compared to the attributes accessible via JAVASCRIPT (cf. [Section 2.3.3.2](#)), but it helps to verify the information consistency.

```
1 @media screen and (-webkit-device-pixel-ratio: 1){
2     div#dpr{
3         background-image : url("database.php?dpr=1");
4     }
5 }
6
7 @media screen and (-webkit-device-pixel-ratio: 1.5){
8     div#dpr{
9         background-image : url("database.php?dpr=1.5");
10    }
11 }
```

Commonly used attributes. I presented in this section the different attributes that can form a fingerprint. Some of them can be more easily collected because they do not require a specific setup to be integrated into a fingerprinting script, or because they provide a deterministic value that can be directly used to identify the user. Without explicit enumeration, I will refer by the term *attributes* or *browser fingerprinting attributes* to the HTTP headers, JAVASCRIPT browser properties and JAVASCRIPT rendering APIs ([Section 2.3.3.1](#) and [Section 2.3.3.2 -A\) and B\)](#)).

2.4 Browser fingerprinting studies

2.4.1 Measuring browser fingerprinting properties

Dedicated studies. Mayer [120] was the first to talk about the possibility of creating an identifier from browser configurations and properties. He collected 1,298 fingerprints for 2 weeks by concatenating and hashing the values of the `navigator`, `screen`, `navigator.plugins` and `navigator.mimeTypes` JAVASCRIPT objects. He was able to uniquely identify more than 95% of the visitors in his dataset. Moreover, the number of non-unique fingerprints—*e.g.* fingerprint hash collisions—are contained in small clusters. This study was the first attempt to quantify browser fingerprinting.

Eckersley [96] built on the idea of browser fingerprinting and attempted to understand the privacy risks more precisely. He created the <https://panopticlick.eff.org> website and collected 470,000 fingerprints in 3 weeks. His study covered the following attributes:

- HTTP headers: User-Agent, Accept and Cookies enabled,

- JAVASCRIPT elements: timezone, screen resolution, plugin names and versions, and MIME types,
- Fonts obtained via Java or Flash applets.

He measured that 83% of the browser fingerprints collected in his dataset were unique. He also analyzed fingerprint stability to measure the long-term identification possibilities of the technique. Although his dataset was not ideal because he only relied on cookies to re-identify returning users, he found that 37% of the fingerprints changed within the 3 weeks.

Laperdrix *et al.* [117] revisited and updated the findings of the Panopticlick study by including modern web technologies and collecting mobile devices data in 2016. They developed a new website, <https://amiunique.org>, to collect fingerprints. They used all the attributes from the Panopticlick study, with the following additional attributes:

- JAVASCRIPT browser properties: screen color depth, `platform`, `Do Not Track`, use of an ad blocker,
- Canvas: They used primitives to write texts and draw a rectangle with colors and an emoji,
- WebGL: vendor and renderer.

Their results concerning uniqueness were similar to the Panopticlick study. They found that 89.4% of the fingerprints in their dataset (106,327 fingerprints) were unique, 90% on desktops and 81% on mobiles. However, they also went into possible scenarios that could be used to reduce fingerprinting, including the disappearance of FLASH, blocking JAVASCRIPT, or more subtle changes, like unifying User Agents.

In-the-wild study. These three studies have one major drawback: they collect data on a dedicated website that attracts a biased set of users. Due to the precise and technical goal of these websites, users visiting them are often people who care about their privacy, who are more technically capable than an average user on the web, and who are more likely to have specific configurations, browsers or extensions to protect themselves against online tracking. Because these kinds of behaviors are not representative of all web users, the datasets of these websites suffer bias that is hard to study or remove.

In 2018, in an attempt to study an unbiased fingerprints dataset, Gomez-Boix *et al.* [103] setup a fingerprinting script on one of the Top-15 most popular French websites. They deployed the script on 2 pages, a weather forecast page, and a page about politics. They collected around 2 million fingerprints by using the same set of attributes as AMIUNIQUE [117]. They reached a uniqueness percentage of 33.6%: 35% for desktop devices, and 18% for mobile devices, showing a strong reduction in fingerprint uniqueness compared to previous studies. The authors imply that fingerprinting is less of a risk to privacy than previously thought, that previous studies were biased and the datasets were not large enough to find fingerprint collisions. As the AMIUNIQUE study, they also

studied the impact of three scenarios on their dataset: the end of browser plugins, the adherence to standard HTTP headers and the end of JAVASCRIPT. The data representing the percentage of unique fingerprints for each scenario are summarized in the following table:

Device type	Default	Plugins end	HTTP headers	JAVASCRIPT end
Desktop	35.7%	16.5%	31%	0.7%
Mobiles	18.5%	18%	16.2%	4.3%

The end of browser plugins decreases fingerprint uniqueness by a large amount because this parameter is very discriminating in desktops, especially with regards to the Flash plugin that provides a lot of information to fingerprint users. The HTTP headers values does not add a lot of entropy, as their entropy is low compared to JAVASCRIPT attributes. In fact, many of the HTTP header values can be retrieved with JAVASCRIPT. Consequently, the standardization of HTTP headers does not have a huge impact. Finally, the end of JAVASCRIPT strongly reduces the interest of browser fingerprinting, uniqueness is severely reduced, although this is the least likely scenario to become a reality, as the use of JAVASCRIPT-intensive websites is growing.

Controlled environment. Al-Fannah *et al.* [86] studied the fingerprintability of browsers. They studied 5 desktop browsers on 2 desktop OS and 5 mobile browsers on 3 mobile OS. Their work is innovative because they studied browser fingerprinting on a controlled environment with the ground truth concerning hardware, OS and browsers. However, it suffers from several flaws:

- they do not study the complete set of attributes from the state of the art,
- they only focus on browser fingerprintability, but did not study other components such as hardware, OS or browser configuration,
- their results do not measure the reasons behind browser fingerprints uniqueness but only present a global level of fingerprintability for each studied browser.

Stability. Fingerprints evolve over time because of updates and configuration changes, which means it can only be used to identify or track a device if a fingerprint can be linked to its previous version. Vastel *et al.* [150] studied the stability of browser fingerprints over time. They used the AMIUNIQUE web extensions available for CHROME and FIREFOX that collect the fingerprint of their users. The extension collects the fingerprint of the user every 4 hours and links it to a unique identifier (*UUID*) generated during the extension installation. This *UUID* will be used as the ground truth to verify the effectiveness of fingerprint linking algorithms.

They develop 2 algorithms to link fingerprint evolutions:

- The rule-based algorithm takes advantage of a set of hard-defined rules to link fingerprints. For example, one rule claims that the OS, platform and browser family must remain the same for all the evolutions of a fingerprint. Another rule claims that the browser version can only increase or remain identical, but never decreases;
- The hybrid-based algorithm combines the rules defined for the rule-based algorithm and a machine learning pipeline. Based on a random classifier, this step was integrated to measure fine-grained changes that would be difficult to identify and track, such as font or plugin changes, timezone changes, or other hard-to-define changes.

To evaluate the effectiveness of their linking algorithms, they introduced a few new concepts. They created the concept of *fingerprints chain*, which is a list of fingerprints that an algorithm has identified as belonging to the same browser instance. A fingerprint chain may, of course, contain linking errors, but the UUID can be used to retrieve the ground truth. They also propose the concept of *ownership ratio*, which measures the ratio of fingerprints belonging to the right fingerprints chain.

They collected 172,285 fingerprints from 7,965 different browser instances over 2 years (July 2015 to August 2017). Their rule-based algorithm obtained an overall ownership ratio of 0.977, while their hybrid algorithm obtained an ownership ratio of 0.985. They mainly highlight two takeaways from their analysis: i) There is a segment of the population that is difficult to track for extended periods of time using only browser fingerprinting because they use common devices, with popular browsers, and few customizations, making fingerprint collisions more common. In their dataset, this is close to 20% of the browser instances; ii) There is another segment, around 25% of browser instances in the dataset, that is highly trackable and have unique fingerprints with highly identifiable attributes. In general, these browsers could be tracked through the entire duration of the experiment with relative ease. They hypothesize that users that focus more on privacy and have more experience with technology, and tend to use uncommon devices or browsers with extra customization, are those more susceptible to fingerprint tracking.

Li *et al.* [119] performed a large-scale analysis to study the evolution of browser fingerprints in the wild. They defined the term *fingerprint dynamic* to refer to an evolution in the browser fingerprint. They collected common attributes defined in [Section 2.3.3](#) from 1,329,927 browser instances, and measured 960,853 different fingerprint dynamic. They classify fingerprint dynamics into 3 categories:

- Browser or OS updates refers to a newer version of the browser or OS. Browser or OS versions affect the **User-Agent** string, the canvas rendering or the font string. In their dataset, it represents 30% of the observed dynamics, 95% of these being iOS updates, which are reflected in the **User-Agent**;

- User actions refers to every action the user can take in his browser that can change the browser fingerprint, such as zooming in the page, moving to a different timezone or updating her configuration. These dynamics count for 31.07% of the total number of dynamics, but only 13.4% of total browser instances generate these types of dynamics.
- Environment updates concerns the update of software co-located with the browser. For example, the installation or update of Microsoft Office or Adobe Acrobat Readers might introduce new fonts on the device that can be detected through the font enumeration technique.

They evaluated the 2 algorithms from the FP-STALKER paper [150] on their dataset. They first measured the time taken by the hybrid algorithm to match a single fingerprint drastically increases according to the number of fingerprints in the dataset, which reaches 1 second for around 1M fingerprint. For a tracking system which displays ads, this is largely considered as unacceptable for an in-the-wild system. They also showed that very few changes in the fingerprint could lead into *false positive* (FP) or *false negative* (FN).

Finally, they leveraged their observations to provide insights to develop future linking algorithms:

- Browser fingerprints reveal privacy or security-related information,
- Several fingerprint dynamics are related, while the features which cause the corresponding changes are not,
- The timing of some fingerprint dynamics are correlated with real-world events.

2.4.2 Detection and classification

Many studies measured the presence of browser fingerprinting on websites. The first of them was the one conducted by Nikiforakis *et al.* [125]. They analyzed the code of 3 popular fingerprinting script providers: BlueCava, Iovation, and ThreatMatrix. They studied the attributes collected by the scripts generated by these providers and showed they access similar set of attributes. They analyzed the presence of these scripts by crawling the Alexa Top 10k and found 40 websites that were using one of these scripts.

Acar *et al.* [85] designed *FPDetective*, a framework to detecting browser fingerprinting on the Web. They focused on scripts performing font detection, either via FLASH, or via the span measurements technique. More precisely, they considered the script was doing fingerprinting if: i) The FLASH file requests the set of fonts, or ii) the enumeration via the span technique was performed on at least 30 fonts. They crawled the Alexa Top 1M and found 6 websites using FLASH font detection on 69 websites, and 13 scripts performing font enumeration on 404 websites.

Acar *et al.* [84] crawled the home pages of the Alexa Top 100k to measure the collection of the canvas attribute. To avoid detecting canvas uses that are not related to fingerprinting, they included rules into their crawlers: i) the rendering and getting

functions must be called from the same URL, ii) the canvas should contain more than one color and be bigger than 16x16 pixels, and iii) the image content should be required in a non-lossy compression format. They observed 5.5% of the crawled websites were using canvas fingerprinting, 95% of them using a script from a single provider, ADDTHIS. They manually analyzed ADDTHIS’s script and found additional techniques than the ones outlined by Mowery *et al.* [122]. For example, the script used the perfect pangram ‘*Cwm fjordbank glyphs vext quiz*’ as a text to draw in the canvas. This technique allows the script to check-in a short string all the letters of the English alphabet. The goal of these new additions is to check multiple drawing techniques and rendering elements to add entropy to the canvas image returned by the canvas rendering context.

Englehardt *et al.* [97] went further and analyzed the Alexa Top 1M. They measured the presence of canvas fingerprinting, font fingerprinting using canvas, and WebRTC fingerprinting. Their results showing the percentage of websites using these fingerprinting techniques are presented in the table below.

Rank Interval	Canvas	Canvas Font	WebRTC
[0,1k)	5.10%	2.50%	0.60%
[1k, 10k)	3.91%	1.98%	0.42%
[10k, 100k)	2.45%	0.86%	0.19%
[100k, 1M)	1.31%	0.25%	0.06%

To avoid detecting false positive uses of canvas fingerprinting, Englehardt *et al.* made the matching rules harder. Thus, the text in the canvas must be at least 10 characters long or written in 2 colors. They observed several trends since the study conducted by Acar *et al.* 2 years earlier [84]: i) the most used trackers detected by Acar *et al.* [84] stopped employing the canvas fingerprinting technique, ii) the most used trackers are not the same in the two studies, iii) the number of domains using this technique went up, as well as the complexity and variety of the canvas fingerprints, iv) canvas fingerprinting began being used not only for tracking, but also for fraud detection. They studied the percentage of scripts detected earlier and blocked by several blocking tools by comparing the performance of DISCONNECT, a privacy-preserving tool, to EASYLIST and EASYPRIVACY. The results show both solutions blocked scripts, 17.6% for Disconnect and 25.1% for EASYLIST and EASYPRIVACY. However, the percentage of sites with blocked scripts was much higher, with 78.5% for Disconnect and 88.3% for EASYLIST and EASYPRIVACY. It also shades light on the difficulties for tracking protection lists to keep lists up-to-date to be effective at protecting user privacy.

While the study of wellknown fingerprinters or attributes can be useful to understand their prevalence and usage, it cannot propose an exhaustive vision of the usage of the browser fingerprinting technique. This is due to the fact that browser fingerprinting is

permissionless and does not require any user interaction to be triggered. It is then very easy for a script to collect attributes and form a fingerprint. Rather than studying a set of wellknown fingerprints or attributes, the studies in the state of the art started to propose techniques to dynamically detect scripts doing fingerprinting. They often rely on building similarities between a ground truth of fingerprinters and the scripts collected in the wild.

Haanen *et al.* [106] and Rizzo [132] both used machine learning to detect browser fingerprinting scripts. They used static code features to train their machine learning classifier. However, their approach suffers several drawbacks. Obfuscated code can easily bypass this detection technique. Bird *et al.* [89] build a technique that group scripts by similar JAVASCRIPT execution traces and trained a machine learning model. They showed their technique could identify the scripts identified by existing heuristic techniques, but also new fingerprinting scripts that were missed by previous techniques. Still, their ground truth consists in specific attributes or keyword lists that cannot be considered reliable when labeling a real-world dataset. More recently, Iqbal *et al.* [108] and Rizzo *et al.* [133] proposed interesting approaches to classify fingerprinting scripts. Both studies combined static and dynamic analysis to build a machine learning classifier.

- The static part is based on the similarities of *Abstract Syntax Tree* (AST) representation of scripts. This solution is stronger than a textual comparison or representation because it does not take into consideration the writing style of the JAVASCRIPT file;
- The dynamic part relies on the collection of accessed properties and functions that are used for fingerprinting, as well as the number of times these elements were accessed, the parameters and return values in case of a function call. This part helps bypassing obfuscated scripts that cannot be detected through a static analysis.

Both studies use a machine learning model to classify scripts. Finally, they rely on a manual labeling step to improve the model based on elements that can hardly be studied by the classifier, such as the presence or absence of interactions between the fingerprinting and non-fingerprinting parts of the scripts, or the likeness between the to-be-classified script and wellknown fingerprinting library, like **FingerprintJS2** [33].

Table 2.1 presents a summary of the classification techniques used by the studies in the state of the art.

2.5 Browser fingerprinting countermeasures

In this section, we present the 3 major techniques used to defeat browser fingerprinting:

- blocking the loading or execution of fingerprinting scripts,
- unifying attributes value to break the uniqueness of a browser fingerprint,

Study		Analysis techniques			Classification			Evaluation	
Ref.	Year	Static		Dynamic (attributes access)	Rule-based	ML-based	Validation/ Ground truth dataset	Dataset	Overall presence on websites
		Textual	AST						
[125]	2013						Fingerprinters set	Alexa Top 10k	0.4%
[85]	2013			font detec & enum	✓		N/A	Alexa Top 1M	4%
[84]	2014			canvas	✓		N/A	Alexa Top 100k	5%
[97]	2016			font detec & enum & canvas	✓		N/A	Alexa Top 1M	0.07% to 1.43%
[106]	2018	✓	✓			✓	Custom	N/A	N/A
[132]	2018	✓				✓	Custom + [97]	[97]	0.3%
[89]	2020			✓		✓	[97]	[62]	N/A
[108]	2020		✓	✓		✓	[97]	Alexa Top 100k	10.2%
[133]	2021		✓	✓		✓	[97]	Custom - 236k scripts	1.1%

Table 2.1: Summary of the techniques used and results obtained by different detection and classification studies.

- changing attributes value to break the stability of a browser fingerprint.

2.5.1 Blocking scripts

As fingerprinting is mostly based on the attributes collected via `JAVASCRIPT`, one technique to protect against it relies on blocking the loading or execution of scripts. Countermeasures designed to block scripts do not necessarily aim at specifically blocking fingerprinting, but at blocking all malicious scripts, such as scripts used to track users or collect all kinds of information about the users. These defenses are based on lists of scripts that must be blocked. Two of the most popular lists are `EASYLIST` [30] and `EASYPRIVACY` [31]. These lists are included in many browser extensions, such as `ADBLOCKPLUS` [1] and `UBLOCK ORIGIN` [79]. This type of defense is very popular and represents 6 of the Top 10 installed extensions on `FIREFOX` [37]. Several browsers also integrated these defenses, such as `FIREFOX` [34] or `BRAVE` [14]. Because these defenses often rely on static lists to block resources, these defenses are limited and cannot block an unknown malicious script. Then, they failed at detecting all malicious scripts on the Web [97] and required to be constantly updated.

The other approach to block malicious scripts is to block the execution of all `JAVASCRIPT` code. This is a defense implemented by several browser extensions such as `NOSCRIPT` [65] and by browsers, such as `TOR` [77]. While this defense is radical and does not require lists to block resources compared to defenses presented previously, it also prevents regular `JAVASCRIPT` code to run. Because of the overwhelming weight of

JAVASCRIPT in the Web ecosystem, it also prevents the user to have access to a large part of the Web.

2.5.2 Unifying attributes value

This second strategy against fingerprinting is designed to counter one of the properties that define fingerprinting: the uniqueness. As we explained in [Section 2.3.2](#), uniqueness is essential for fingerprinting to identify users. If several users share the same fingerprint, it is impossible for a system to distinguish them and use it for identification. Thus, the usage of fingerprinting is conditioned to an acceptable uniqueness rate.

Several browser vendors use this technique for their users. FIREFOX does not support JAVASCRIPT APIs that are used for fingerprinting, such as the `navigator.deviceMemory` and `battery` APIs. Additionally, browsers started to unify attribute values such as the `navigator.buildID` and `navigator.productSub`. The `buildID` navigator property returns 20181001000000 for all FIREFOX instances, while the `productSub` navigator property returns 20100101 for all FIREFOX instances and 20030107 for all Chromium-based instances. FIREFOX proposes another defense, called `privacy.resistFingerprinting`. This configuration flag increases the number of APIs that are blocked or unified. SAFARI was one of the first browsers to block FLASH because of security concerns [74]. More recently, SAFARI added a system called Intelligent Tracking Prevention (*ITP*) that aims at protecting against tracking, including browser fingerprinting [8]. While the strategy they use is not public, the idea behind it is to unify system attributes, such as fonts, among all MACOS devices to decrease the uniqueness ratio of their users [9].

Saito *et al.* [135] studied the fingerprinting defenses of the TOR browser. The table below presents the common fingerprint attributes and their status in TOR.

Status in TOR	Fingerprint attribute
Available	Local storage availability, Session storage availability
Constant value	User-Agent (JS and HTTP header), <code>Accept</code> , <code>Accept-Language</code> , <code>Accept-Encoding</code> and <code>Accept-Charset</code> headers, <code>Date.getTimezoneOffset()</code>
Modified	Screen resolution
Unavailable	Plugin list, Flash font list

Moreover, since version 5.5 (January 2016), the TOR browser makes JAVASCRIPT fonts enumeration ineffective by building a unique list of basic fonts for all users [78]. This does affect a page's rendering if the selected font is not be available. Additional fonts can be allowed by modifying the `font.system.whitelist` attribute in the `Fonts.conf`

configuration file for TOR, but this is not recommended since it would make users more identifiable. TOR also asks for the user permission to use the canvas API [76]. In their study, Saito *et al.* found that 14% of the Tor browsers running on version 5.5 or higher can be fingerprinted, compared to 70% for versions before 5.5. The reason is, as explained earlier, the list of fonts is highly discriminating.

While these defenses still provide information to distinguish between instances of different browsers, it lowers the differences between users using the same browser. Because all the users of a defense based on breaking the uniqueness share the same attribute value, the defense is more likely to work if many users use this defense. Thus, browser vendors are perfect candidates to implement this strategy as their browsers are used by millions of users.

2.5.3 Changing attributes value over time

As explained in Section 2.3.2, a fingerprint evolves over time. Identification techniques need to be able to link instances of the same fingerprint. Then, it is essential to rely on stable attributes that rarely evolve. Based on this element, the last strategy to protect against browser fingerprinting is based on the changes of a value of some attributes to break the stability of a fingerprint. The frequency of the changes can vary from one defense to another, but it should be high enough to quickly have an effect. Changes that occur each day or during each browsing session start appear to be a relevant frequency.

This stability-breaking strategy is the one chosen by several browser extensions such as RANDOM AGENT SPOOFER [70]. It is an extension that protects against browser fingerprinting using a system of profiles extracted from real devices. Profiles are composed of some of the attributes that constitute a fingerprint, such as attributes from the `navigator` object, the screen's resolution, or the `User-Agent`. The browser fingerprint is frequently updated by randomly selecting a profile among the set of available profiles. As the selected profile will be different from the one previously used, that will break the stability of the fingerprint.

Nikiforakis *et al.* [124] proposed a modified CHROMIUM browser that focuses on two browser fingerprinting attributes: i) The list of plugins, and ii) the list of fonts. Their modified browser randomly adds plugins in the browser. Concerning the list of fonts, they focused on the JAVASCRIPT font enumeration technique. The browser randomizes the `offsetHeight` and `offsetWidth` properties of the HTML span element used to detect fonts. Thus, contrary to other countermeasures that lie on the OS and browser's nature, their approach is less likely to be detected because the lies are minor and difficult to identify. They tested their approach against four fingerprinting scripts:

- two commercial fingerprinting scripts: BLUECAVA [13] and COINBASE [27],
- one open source fingerprinting script: FINGERPRINTJS [33],
- a research fingerprinting script: PETPORTAL [67].

For each script, they showed they could deceive all 4 fingerprinting scripts with their technique.

Virtual machines Laperdrix *et al.* [115] proposed an approach that leverages virtual machines to generate consistent and unique fingerprints. Their solution aims at breaking the stability of a browser fingerprint by changing the user’s OS, browser, font list, and plugin list. The table below presents the different values they use to randomize the fingerprints.

Element	Possible values
OS	Fedora 20 (32 & 64 bits), Ubuntu 14.04 (32 & 64 bits)
Browser	Firefox 28.0, Firefox 29.0, Firefox 30.0, Chrome 34, Chrome 35, Chrome 36
Fonts	2,762 different fonts
Plugins	39 different plugins

Their approach, BLINK, runs inside a *Virtual Machine* (VM). Each time it is launched, it generates an environment that exhibits a new fingerprint by randomly selecting the OS of the VM image, and then randomly selecting one of the browsers, and finally, installing a random selection of fonts and a random list of plugins. Thus, by frequently generating new fingerprints, the technique breaks the stability required for identification. Moreover, contrary to countermeasures that generate inconsistent fingerprints, Blink does not lie since it uses a real OS, real browser, real fonts, and real plugins. This proof-of-concept approach could be extended to randomizing more attributes, like drivers, or also randomizing the browsers’ configurations, the OS configurations, or even the configuration of the virtual machine itself (*e.g.*, hardware acceleration, number of CPUs). Laperdrix *et al.* also proposed a version of BLINK that uses Docker containers instead of VMs [12].

2.5.4 Induced information leaks

While defenses help to protect against browser fingerprinting, they can introduce privacy leaks. Mowery *et al.* [121] showed filterlist-based extensions have side-effects that can be exploited to track users. In particular, they proposed an approach to infer the domains whitelisted by the user in the NoScript extension [65]. Thus, this set of domains is itself an information that can be added in the list of attributes of a fingerprint. Even though they evaluated their approach only against the NoScript browser extension, they explained that similar approaches could be applied to other ad and tracker blocker extensions.

Nikiforakis *et al.* [125] raised a more specific problem: fingerprinting countermeasures tend to generate inconsistent fingerprints, *i.e.* combinations of attributes that cannot be found in the wild. For example, the `JAVASCRIPT navigator` object exposes the `platform` property that returns the OS of the device. If a countermeasure lies on the `User-Agent` and changes its OS, it also has to change the OS given by the `platform` property. If it does not, the 2 attributes will give 2 different OS, which is something that cannot be observed in the wild, hence something more unique. Thus, they argue that these inconsistencies make the users more identifiable and that fingerprinters can use them for tracking.

Vastel *et al.* [149] studied several countermeasures that could generate inconsistencies. They showed that even though `RANDOM AGENT SPOOFER` [70] uses real device profiles to generate consistent fingerprints, it is still possible to detect its presence and recover the real OS and browser. They also studied a particular kind of countermeasure: canvas poisoner extensions, such as `CANVAS DEFENDER`, `CANVAS FP BLOCK` and `FP RANDOM`. These extensions operate by adding noise to the canvas to hide its real value. Vastel *et al.* showed they could easily detect the noise added by these extensions, which can lead to more unique fingerprints since knowing the user has installed one of these extensions increases uniqueness. They showed that, in some cases, it was possible to extract the noise and recover the canvases original value, making the extension counterproductive. Finally, they analyzed `FIREFOX` defenses and showed they introduced inconsistencies that can be used to detect users with fingerprinting protection activated. For example, when fingerprinting protection is active, the `WebGL vendor` and `WebGL renderer` might not be consistent with what the OS claimed. They also detected inconsistencies about the media query `-moz-os-version` and the installed fonts list when the browser was running on Windows. However, since the `FIREFOX` fingerprinting protection might be used by a significant number of users, the privacy gain it provides to its users probably compensates for the information leak it engenders.

These results highlight the challenge of designing efficient and reliable defenses against browser fingerprinting.

2.6 Browser fingerprinting usages

2.6.1 User Tracking

Many studies rely on client-side signals and information to guess the intend of a script. While some signals are insufficient, others express the real goal of the script. Nikiforakis *et al.* [125] analyzed 3 popular browser fingerprinting scripts. As they are provided by tracking companies, the researchers present them as trackers. Acar presented in his 2 studies [85, 84] several results about the presence of browser fingerprinting.

While he does not explicitly conclude about the usages made by these scripts of the browser fingerprinting technique, several results, such as the FLASH and HTTP cookies respawning performed by third-parties, showed these scripts are performing tracking. Englehardt *et al.* [97] used the EASYLIST and EASYPRIVACY filter-lists to classify third-party scripts. The lists themselves have the main purpose to block tracking and are implemented in consequence. Fouad *et al.* [101] used OpenWPM to crawl the Alexa Top 30k to measure cookies respawning thanks to browser fingerprinting. While they measured the IP address is the most used feature to respawn cookies, many fingerprinting attributes such as the **User-Agent**, **Language** and **Do Not Track** headers, or the time-zone JAVASCRIPT property, are also used. They also showed 37 of the respawned cookies are third-party, many of them being generated by tracking and advertising companies.

2.6.2 Bot Detection

Bursztein *et al.* [92] proposed an approach that leverages the unpredictable but stable nature of canvas to detect crawlers and emulated devices. Their approach aims at generating canvas whose values are the same among devices that belong to the same class, *i.e.* same browser and OS, but different among different classes. They deployed their fingerprinting script on real traffic and observed their website suffered two attacks on the authentication page of the website. Their post-analysis of the IP addresses and fingerprint data revealed that the two attacks shared a device signature belonging to the PhantomJS automated browser. However, it is unclear how their detection solution would perform against more advanced and realistic headless browsers, such as CHROME and FIREFOX headless. Furthermore, it requires a significant amount of devices, in the order of millions of devices, to bootstrap and maintain the detection system by calculating each device ground truth, a quantity of data that only large companies like GOOGLE can obtain and use to brute-force their approach. Moreover, they only address a single attribute, canvas rendering.

Jonker *et al.* [111] studied the techniques to detect bots. They analyzed a client-side web bot detection commercial script, and found it tests the presence of several JAVASCRIPT properties. They measured the presence or absence of new JAVASCRIPT properties in 14 automated browsers. These new properties consist in the list of keys of the **document** and **window** objects, and the list of couples key/value of the **navigator** JAVASCRIPT objects. The automated browsers were tested in both regular and headless mode if available, which does not provide a graphic interface. They found all the automated browsers they tested had differences compared to the original browser on which they are based. Their result illustrates the possibility to rely on these attributes to detect more easily bots. Finally, they browsed the Alexa Top 1M and found out 12,8% of the websites were collecting fingerprint-based bot detection information.

Vastel *et al.* [151] studied the usage of browser fingerprinting for bot detection. They crawled the Alexa Top 10k and measured 291 websites were blocking their crawler. They measured 93 of these websites were collecting fingerprinting attributes. They designed 6 different crawlers to study more in-depth these websites. Each of these crawlers has a specific set of attributes modified to detect the attributes used by the websites to detect the crawler. By running these crawlers against the websites that blocked the original crawler, they were able to measure which attribute was causing the blockage and how to bypass the defense. Finally, they explained many automated browsers were now really close to vanilla ones. This leads to more difficulties for scripts to distinguish between vanilla and automated browsers, as well as more facilities for automated browsers to hide themselves by changing the attributes responsible for the detection.

2.6.3 User Authentication

As explained in [Section 2.2](#), the authentication process on a website aims to verify a user's identity. The verification happens when the user enters her credentials or along the user session lifespan. In this context, several studies analyzed and evaluated the security gain provided by the usage of browser fingerprinting in an authentication context. Unger *et al.* [145] proposed an approach to enhance session security with fingerprinting. Their approach works in five steps:

- the server verifies if the session cookie sent by the browser is associated with an existing session,
- the server performs basic checks for static fingerprinting attributes, such as the HTTP headers, their order and the IP address range,
- the server asks the client browser to test the presence of features in the browser, such as CSS and WebGL,
- the browser sends the list of tested features to the server, and
- the server verifies the values sent by the browser.

Their approach protects users against session hijacking, but is vulnerable to man-in-the-middle or *Cross-Site Request Forgery* (CSRF) attacks. Furthermore, they implemented the proposed solution but did not evaluate it.

Spooren *et al.* [141] evaluated the usage of browser fingerprinting to enhance web authentication for mobiles. They integrated a fingerprinting solution in a module of the OpenAM authentication system [66]. It consists in a fingerprinting script that collects the fingerprint of the user that tries to authenticate, and a rule-based algorithm to compare fingerprints. They evaluated their solution by collecting 59 mobile fingerprints. First, they explained a static fingerprint cannot be considered as a robust identity proof in a defense mechanism because many attributes can be easily spoofed. They also explained their dataset has too little entropy, which leads to fingerprints being too similar and not distinguishable enough. While this makes the defense weak because an attacker would

have little effort to mimic any of the fingerprint of the dataset, this statement requires to be confirmed on a larger dataset.

Preuveneers *et al.* [130] were the first to talk about adaptive authentication with browser fingerprinting by using a risk model based on the situation on which the user tries to authenticate. An authentication attempt with the usual device of the user on a usual network can be easily considered as safe while an authentication attempt on an unusual device or an unexpected network can be considered as risky. They discussed 4 security requirements a risk-based system with browser authentication should validate:

- SR1: Ensure fingerprints cannot be compromised,
- SR2: Prevent fingerprints replay-attacks,
- SR3: Support fingerprints revocability,
- SR4: Ensure fingerprints can be strongly linked.

They developed SMARTAUTH, an authentication framework that leverages dynamic fingerprinting. Rather than relying only on static fingerprints that have a deterministic value, they also collect dynamic information such as the geolocation, the time of access and the IP address. They implemented their solution as a new authentication plugin. They evaluated their solution against a controlled environment of 2,000 fingerprints and showed they were in a large majority able to successfully link their dynamic fingerprints.

Alaca *et al.* [87] proposed the usage of browser fingerprinting to enhance user security, both during the initial authentication attempt and during the session. Their idea is to add the user's fingerprint to the request so that the server can perform additional verifications, both during the authentication attempt and during the session duration. During the authentication attempt, the server compares the received fingerprint to the ones already trusted for this account, and allows or denies the attempt accordingly. As explained by Unger *et al.* [145], the browser fingerprinting technique can be used to enhance security during the session duration by checking fingerprinting attributes for every HTTP request sent. Additionally to the properties of browser fingerprinting we defined in Section 2.3.2, they defined additional properties for browser fingerprints to be used in an authentication system: **Low Resource Use** and **Spoofing Resistance**. For each attribute in the state of the art, they describe its level of used resource during the collection and the resistance to spoofing. As browser fingerprinting attributes are the result of a collection of JAVASCRIPT properties and functions return values, the attributes are in a large majority using few resource, which make them good candidates to enhance web security. However, they are very vulnerable to spoofing and can be collected by malicious scripts to be reused by attackers. To this end, Alaca *et al.* also defined 5 threat models against such a system, and discuss their impact on a web authentication system with browser fingerprints.

Goethem *et al.* [148] studied the usage of accelerometer for web authentication. They queried the vibration motor of a mobile via the `navigator.vibrate()` JAVASCRIPT API

for different duration periods, and join them to form a trace. They made the assumption that by varying the length of the vibration, they will generate different vibration periods that will be robust and distinguishable enough. During the account registration, they register a set of traces. When a user tries to login, the system collects a trace of a defined length and compare it to the already registered trace of the same length. They evaluated their techniques against a set of 15 mobiles devices. They explained their defense has low risks of being spoofed compared to static fingerprinting because an attacker using the accelerometer by making vibrate the device would alert the user. Moreover, the device needs to be placed on a hard surface. While this is feasible for authentication, an attacker would have difficulties to confirm this information, or would have to rely on data collected on a non-hard surface, which provides less robust data.

Laperdrix *et al.* [114] explored the use of dynamic challenges to enhance authentication. They proposed to generate two canvas images when the user authenticates: the first canvas is verified for the current authentication, while the second is stored on the server and is used for the device next authentication. This allows the server to use dynamic canvas tests, avoiding replay attacks, while side-stepping the issue of canvas unpredictability (the server doesn't need to know in advance the value of a dynamic canvas, it can collect it directly from the device, one step at a time). Laperdrix's approach is similar to the approach proposed by Bursztein *et al.* [92], where they ask the browser to generate random challenges based on canvas rendering. Given a seed, the challenge consists in drawing randomly selected canvas primitives, such as Bezier curves or polynomial curves, as well as applying random colors and shadow. They evaluated their approach on real traffic and showed that given a random seed, they were able to generate unique and stable canvas fingerprints. Thus, it makes it difficult for an attacker to replay a canvas, or to forge a canvas by predicting its value.

Andriamilanto *et al.* [88] evaluated the properties of browser fingerprinting for web authentication. They list the properties required for using browser fingerprinting for web authentication: *distinctiveness*, *stability*, and *performance*. They evaluated these properties on a dataset of 4M fingerprints collected over 6 months. They measured a global uniqueness percentage of 81%–43% for the mobile dataset. They showed fingerprints were quite stable, but explained stability could still be improved by removing some attributes. Finally, they measured a median time of 2.92 seconds (2.64 for desktop devices and 4.44 for mobiles) taken by their script to collect the fingerprint. Their results illustrate the trade-off to reach between uniqueness, stability, and collection speed.

2.7 Conclusion

Limitations. In this section, we presented the current state of the art of web authentication. We showed a system that only relies on a password to authenticate its users is

vulnerable to many attacks. On one hand, we measured the adoption and acceptance of several multi-factor authentication mechanisms, showing that no particular solution stands out to improve security while having a limited impact on the user experience and being accepted by the users. However, risk-based authentication seems to be promising at it combines security improvements with limited degradation of the user experience. On the other hand, we presented browser fingerprinting, a stateless and permission-less identification technique on the Web. Thanks to its identification power, we believe browser fingerprinting can be a reliable technique for risk-based authentication. Based on this belief, we identify 3 limitations of the state of the art:

- Several works analyzed and measured the uses of browser fingerprinting for tracking and bot detection on the Web, but no study in the state of the art measured the use of this technique to enhance web authentication.
- Current techniques to link fingerprints scale badly when used on a large dataset of fingerprints. Additionally, their design and goals make them inadequate for authentication.
- Several studies covered browser fingerprinting for web authentication from a theoretical point of view, but they lack a complete evaluation of the authentication scheme, from both the security and user experience point of view.

Contributions. In this manuscript, we study browser fingerprinting to enhance web authentication, and tackle the research questions defined in [Section 1.2](#).

In [Chapter 3](#), we address **RQ1** by measuring the collection of browser fingerprints by scripts when visiting sensitive pages—sign-up, sign-in, basket and payments pages. We analyze the providers of fingerprinting scripts and uncover 12 security-centered organization. We also observe additional authentication factors and bot detection techniques being required during authentication attempts. Finally, we answer **RQ2** concerning the current use of fingerprinting to enhance web authentication by designing and evaluating 2 attack models against authentication schemes intended to measure the security gains provided by browser fingerprinting.

In [Chapter 4](#), we design a controlled environment to collect browser fingerprints while knowing the ground truth of the browser and device of our dataset. We analyze these fingerprints and identify the unique and stable attributes, hence answering **RQ3** and **RQ4**. We use this knowledge to design a browser fingerprint linking algorithm for web authentication. We answer **RQ5** by evaluating our algorithm on a dataset of browser fingerprints collected in the wild to demonstrate its ability to reliably and efficiently link browser fingerprints evolution.

In [Chapter 5](#), we study **RQ6** by proposing concerns concerning the user experience of authentication schemes leveraging browser fingerprinting. We design a risk-based authentication scheme with browser fingerprints and implement it on an existing *Single*

Sign-On (SSO) authentication system. We answer **RQ7** by evaluating the security improvements provided by our authentication scheme. While we identify situations that lead to a degradation of the user experience, these situations happen on a much lower frequency than comparable situations on MFA systems. We discuss the potential attacks and threats against our system, and improvements to be incorporated into our scheme.

Chapter 3

FP-Redemption: Studying Browser Fingerprinting Adoption for the Sake of Web Security

Over the years, studies have focused on studying browser fingerprinting on the Web. Many studies measured the presence of browser fingerprinting on the Web by using automated tools to crawl the Top Alexa [85, 84, 97]. Some contributions proposed using browser fingerprinting to improve web authentication [145, 87, 114], but have not evaluated the benefits to secure online websites. In this chapter, we investigate the adoption of browser fingerprinting to reinforce authentication and security on the Web. Through our experiments, we intend to detect if fingerprinting is used to strengthen web security, and in which specific contexts this occurs.

In [Section 3.1](#), we define 4 types of web pages that store and process sensitive user information, namely sign-up, sign-in, basket and payment pages. We manually visit 1,485 pages from 446 websites belonging to 14 different categories with the aim of detecting fingerprinting scripts. In [Section 3.2](#), we design and implement a browser fingerprinting classification technique based on both automated and manual heuristics. When ran on our dataset, we detected that 169 scripts are collecting browser fingerprints. In [Section 3.3](#), we evaluate the presence of these fingerprinting scripts on the pages of our dataset. We show they are present in all type of pages and website categories, and observe 14 fingerprinting scripts being served by security-focused organizations. In [Section 3.4](#), we study the resilience of websites adopting browser fingerprinting for security purposes by simulating two classes of attacks: *stolen credentials* and *cookie hijacking*. We show a single website uses fingerprinting to protect against stolen credentials, and no website are protected against cookie hijacking. We discuss our results and the limitations of our work in [Section 3.5](#) and conclude in [Section 3.6](#).

3.1 A Dataset of Secure Web Pages

This section reports on our methodology to build a dataset of secured web pages to study the use of browser fingerprinting for security purposes.

3.1.1 Websites Under Study

Secured pages. All pages of a website are not equal when it comes to user security. While most web pages do not process sensitive data, some require careful design to deal with user personal information (*e.g.*, emails, credentials, personal details, payment card numbers). On sensitive web pages, any security breach can quickly lead to privacy leaks for the end-users and seriously affect the reputation of the website. We decided to focus on 4 types of web page requesting personal data:

1. **Sign-up**, which may require email, name, password, and additional personal information depending on the website.
2. **Sign-in** usually requests user credentials: email/pseudonym and authentication factor(s).
3. **Payment** is a page containing a specific form requesting the user to input their payment information (*e.g.*, credit card, wallet, banking information).
4. **Basket** refers to any page related to a shopping basket or shopping cart process, starting from adding an item up to, but not including, payment. Such pages may also request additional information, such as billing/delivery addresses.

We call these 4 types of web page *secured web pages*. To assess our results, we also aim at collecting data from other types of pages. From these, we isolated *home pages* as it has been reported they might fingerprint 25% less [147]. We consider pages that are neither *secured* nor *home* pages to be *content* pages.

Website categories. Previous studies crawled the Top Alexa with automated tools, thus studying a large set of home pages and resources reachable by bots. We decided to avoid the bias introduced by bots, preferring to manually browse the websites and reach deeper pages that require user interactions. Moreover, we are interested in studying the adoption of browser fingerprinting on secured pages. In this context, the diversity of websites indexed by the Top Alexa—or other rankings—proved to be unsatisfactory. Thus, we decided to consider a list of website categories that we estimate to be more relevant for the purpose of our study. To build this list of relevant categories, we adopted the following methodology:

- we targeted websites focused on gambling, credit card, financial, and money services;
- we focused on different retail websites, such as event tickets, games, flights and transports, and accommodation booking websites;

- finally, we added to our list job search, social network, adult, dating, institutional and governmental websites as they often request detailed personal information when creating an account.

We used the following list of keywords to get specific website categories: Adult, Airlines, Bank, Bet games, Cryptocurrency, Dating, Ecommerce, Email, Event ticket, Financial, Flight companies, Healthcare, Job search, Metro/train/flight tickets, Money transfer service, News, Online game, Poker, Shopping, Social insurance, Social network, Sport ticket, Stock trading, Streaming, Taxes, Travel agencies, TV channel.

We used the following list of countries for our experiments: China, France, Germany, India, Japan, Russia, Spain, United Kingdom, United States.

We mainly entered a combination of the country name, category, and the word ‘*website*’ into the GOOGLE search engine, and we visited the websites given on the first page of the results. We also translated the search terms into the main language of the country when the results given in English were not suitable according to the country and the category. This was the case for several searches for Russian websites.

3.1.2 Web Page Acquisition

Past studies used automated crawls to observe browser fingerprinting at scale [125, 84, 97]. However, relying on bot crawls introduces bias in the collected data [111, 107, 156] as more and more websites use defenses to block bot access [151]. Automating the registration and payment processes is also a challenge due to the high variability that can be found in related forms [110]. No unique or universal standard exists, and the number of required fields can strongly differ depending on page types and website categories. The coding practices may be different with obfuscated code and custom attributes, making it hard for a bot to automatically fill a field with the right information. Security requirements are also different, including diverse password constraints and security questions. As the scope of our study is not to develop a bot to automatically test these websites, we manually visited them and collected the required data via a custom web extension we developed. This strategy allows us to appropriately locate interesting secured pages and reduce the bias of being blocked by the bot security mechanisms in place.

3.1.3 Monitored Fingerprinting Attributes

This work does not intend to discover new browser fingerprinting techniques, but rather to investigate the adoption of existing ones in the context of web security. As part of our data acquisition campaign, we thus focused on collecting the values of existing attributes reported in the literature. Thus, we consider navigator & screen properties [96, 117, 125, 123, 127], fonts detection via `span`’s width and height measurement [125], canvas [122], audio [97], and WebGL rendering [93] and parameters [117], WebRTC [97]

and bots detection attributes, including `window` properties that were considered by Jonker *et al.* [111]. Our web extension monitors the accesses triggered by scripts to these attributes by overriding getters of selected properties and functions. Whenever one of these attributes is accessed, the web extension collects the function or property name, the list of arguments passed in the case of a function, the property's value or the function's return value, the script accessing the property or function, and the page's URL.

We manually visited the selected websites and we used a single identity we created on a popular email provider. For each visited page, we stayed at least 10 seconds, and manually filled each form. When asked for proof of identity, such as a valid phone number, an ID or a credit card, we provided one of the phone numbers used to create the email account. As our identity was fake, we were not able to provide a real ID (*e.g.*, a passport) when required by some websites. Given that payment pages require filling out credit card information, we used a fake credit card generator¹ to be able to validate online payment forms and make sure that we trigger most of the scripts embedded in the page. Even though the generated cards were fake and the payment processes were not completed, we bypassed many client-side verifications thanks to this technique. Yet, using fake payment data raises several issues, such as, our account could be considered suspicious and prevented from performing additional actions, and our IP address could be blacklisted and blocked for the rest of our experiment. To reduce suspicion, we used several residential IP addresses during our data collection. Although we provided websites with fake payment data, we believe the low number of payment attempts we performed on each website has had minimal impact on their operations. We did not try to harm them in any way and we canceled our baskets if any information received after the payment attempt indicated the website could validate the basket and ask for a future payment.

3.1.4 Resulting Dataset Description

We performed our data collection campaign from December 2019 to January 2020. We used a fresh install of CHROME 79 on UBUNTU 19.04. We always accepted the default cookie settings from pop-ups, but refused all other types of solicitations, such as geolocation, notifications, or newsletters. In total, we visited 1,485 pages across 446 websites.

Website category and ranking. We used the *category* keyword put into the search engine to categorize the website. We specifically targeted bank and money-related services because of the sensitivity of the data they manipulate, visiting 85 of these websites (see Figure 3.1). The country tag represents the main country the website operates in. We assign the country tag by following the result of two observations: i) Is the website

¹<https://www.creditcardvalidator.org/>

Figure 3.1: Distribution of the 446 visited websites per country & category.

Category	Bank & related	85	11	10	6	5	10	14	4	7	12	5	1
	Ecommerce	38	11	0	6	2	0	1	5	6	4	0	3
	Flights & related	37	11	5	2	2	3	4	4	4	1	1	0
	Adult	33	22	0	4	1	2	0	1	0	0	3	0
	Event ticket	32	4	4	0	5	1	1	1	4	1	3	8
	Technology	29	16	8	1	0	0	3	1	0	0	0	0
	Institutional	27	2	3	4	5	0	3	1	2	0	4	3
	Dating	26	11	0	5	4	1	2	0	1	1	1	0
	Media	23	11	4	0	3	0	0	0	3	0	1	1
	Accommodation	22	11	1	2	1	1	1	3	1	0	1	0
	News	22	1	2	1	2	4	0	2	0	4	1	5
	Financ. & crypto.	20	12	0	0	2	4	0	0	0	0	2	0
	Social network	19	3	0	4	0	4	0	6	0	2	0	0
	Job search	18	3	3	4	2	3	0	0	0	0	3	0
	Games	15	13	0	0	0	0	0	0	0	1	0	1
	Total	446	142	40	39	34	33	29	28	28	26	25	22
		Total	International	UK	Russia	France	Germany	Other	China	Japan	India	Spain	US
		Country											

available in English or in multiple languages and translated into the user's preferred language? ii) Are the services proposed by the website available in a single country or geographic zone? If the website is available in multiple languages or served in English, and if the website provides services to multiple countries, we use the *International* tag. Otherwise, we specify the country. If the website does not operate in a listed country, we use the *Other* tag. With these rules, we tagged 142 *International* websites. The resulting distribution of visited websites per country and category is depicted in Figure 3.1. We did not aim to build an exhaustive manual dataset. However, we checked the Top Alexa rankings of the websites in our dataset. We find that our dataset is relatively well balanced across the less-than-1k (18%), 1k-10k (29%), 10k-100k (27%) and higher-than-100k (26%) Top Alexa rankings.

Page type. We also tagged each page according to its type. By default, a page is associated to a single tag, with the exception of *home* or *content* pages that have a **sign-up** or **sign-in** embedded form that allows the account creation or authentication without going to a specific page (44 occurrences in our dataset), and single pages that handle both account creation and authentication processes (3 occurrences in our dataset). In the case of pages containing both basket-like content and a payment form, we tagged the page as **payment**. If no tag matched, we used the *content* tag. Figure 3.2 presents the distribution of the page types among our dataset.

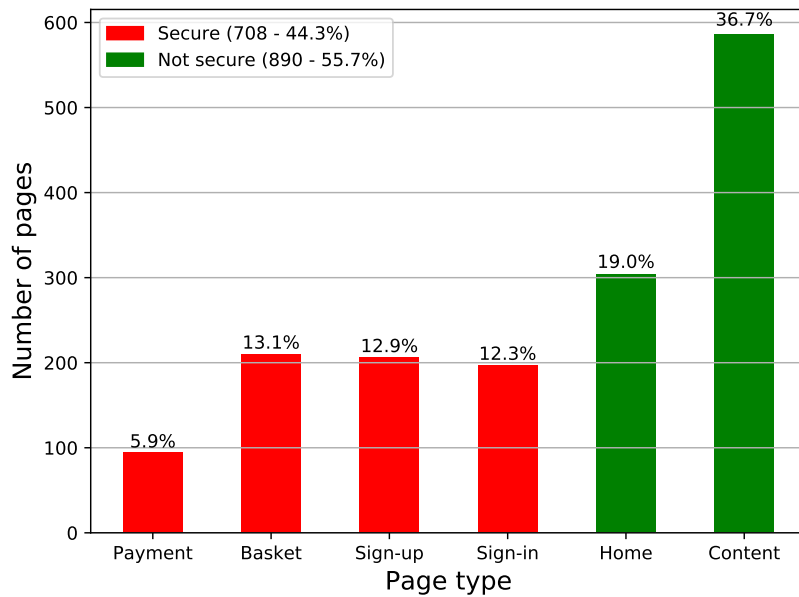


Figure 3.2: Distribution of visited pages per type (the sum exceeds 1,485 pages as some pages match several types).

3.2 Classification of Fingerprinters

The efficiency of browser fingerprinting relies on the design and implementation of a wide combination of stateless attributes, which can uniquely identify a user who visits a web page. However, this combination of fingerprinting attributes is not formally defined and constantly changes due to the evolution of JAVASCRIPT APIs. In [Section 2.3.3](#), we presented the current state of the art concerning browser fingerprinting attributes. The list of attributes we collected covers all the major attributes currently reported, but this list might evolve as some API might become deprecated or being removed by browsers vendors. Oppositely, new APIs introduced in browsers might be reported by the community as being usable in a fingerprinting context. This makes browser fingerprinting challenging to detect at large. As we mentioned in [Section 2.4.2](#), some techniques exist [\[108, 133, 89\]](#) to classify fingerprinting scripts, but their implementation is not publicly available, leading to a technique hardly reusable to label a real-world dataset using all the attributes a browser fingerprint contains. This section, therefore, proposes to apply a supervised classification technique that leverages a ground truth of known browser fingerprinting scripts to label a list of scripts collected in the wild as *fingerprinter* or *non-fingerprinter*, based on the combination of APIs and parameters they access. We use similar heuristics and methodology reported in the literature [\[108, 133, 89\]](#).

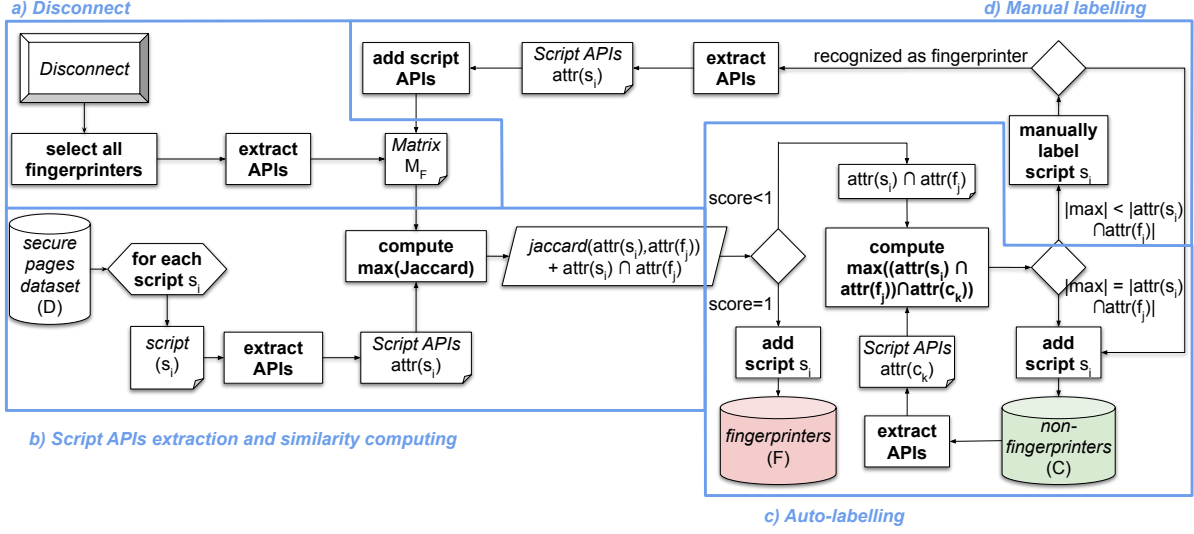


Figure 3.3: Flow chart representing our incremental script classification algorithm.

3.2.1 Incremental Script Classification

Given the lack of a formal model to identify browser fingerprinting scripts, we propose to adopt an incremental classification process. In particular, we leverage a list of scripts classified as *fingerprinter* by DISCONNECT to explore similarities in the accessed APIs. Therefore, we compute the similarity score (Jaccard index) to automatically classify our scripts. If a script cannot be labeled automatically, we go through a manual analysis to classify it. We continuously update the similarity scores whenever a script is classified as *fingerprinter* or *non-fingerprinter*. This way, we reduce the number of iterations required to label all the scripts included in our dataset by exploiting scripts similarities. Figure 3.3 provides an overview of our incremental script classification approach.

3.2.1.1 Learning fingerprinting attributes from Disconnect

We bootstrap our approach with the DISCONNECT project that provides a list of fingerprinters whose behaviors have been analyzed by experts [28]. DISCONNECT has the advantages of being public, constantly updated, and recognized by the community to be reliable [89, 108, 133].

Our first step consists in extracting all the JAVASCRIPT attributes $attr(f_i)$ accessed by each of the fingerprinters $f_i \in F$ reported by DISCONNECT to identify the discriminating features of known fingerprinters. We bypass minification and obfuscation techniques by instrumenting and monitoring the runtime behavior of each script loaded in an empty web page. Each signature of a fingerprinting attribute is structured as a 3-tuple $\langle name, args, N \rangle$, where *name* is the name of the accessed API, *args* is the list of parameters used to retrieve this attribute (empty if none) and *N* is the number of

times that the pair $\langle name, args \rangle$ has been observed along with the execution of script f_i . This results in a matrix of attributes M_F that characterize the fingerprinters listed by DISCONNECT.

3.2.1.2 Computing the fingerprinter similarity for unknown scripts

We start an incremental classification process that takes in input the classification matrix of known fingerprinters attributes M_F and the list of scripts S from our dataset of secure web pages. For each script $s_i \in S$:

1. we extract the set of attributes for each script ($attr(s_i)$),
2. we compute the similarity score (Jaccard index) between $attr(s_i)$ and the attributes $attr(f_j)$ of each known fingerprinter $f_j \in F$, and
3. we keep the tuple with the maximum similarity score and its corresponding intersection $\langle s_i, jaccard(attr(s_i), attr(f_j)), attr(s_i) \cap attr(f_j) \rangle$.

We order the scripts from S by decreasing similarity score to iteratively find the closest script to a known fingerprinter.

3.2.1.3 Auto-labelling

If the maximum similarity score of s_i equals 1, s_i implements a browser fingerprinting feature and we automatically label it as a *fingerprinter*. However, if the score is less than 1, we need to compare the script s_i to all of the non-fingerprinting scripts already labeled to take a decision. For each *non-fingerprinter* $c_k \in C$, we calculate a new intersection (that reuses the previous one we saved), specifically $attr(c_k) \cap (attr(s_i) \cap attr(f_j))$, and we keep the result that maximizes the size of the new intersection. If the attributes we obtain from our intersection with fingerprinters are the same attributes we obtain from our intersection with non-fingerprinters—*i.e.*, the intersections are equal—the script’s fingerprinting attributes are not discriminating enough to be considered a fingerprinter, so we label the script s_i as *non-fingerprinter*. However, if the intersections differ, the fingerprinting attributes $attr(s_i)$ are new in the classification, and our algorithm cannot automatically label the script s_i .

3.2.1.4 Manual labeling

As pointed out by previous studies [89, 108, 133], manual labeling is necessary when the automatic tools used to classify are unable to decide the label of a script. If our algorithm cannot automatically label the script, we manually analyze the code and label it as either *fingerprinter*, *non-fingerprinter*, or *unknown*. To do so, we use the following criteria:

- the script is blocked by EASYLIST [30] or EASYPRIVACY [31],

- the script contains obvious keywords that reveal its goal, such as *fingerprinting* or *deviceFingerprint*,
- the attribute values are forwarded to a remote server, which can reveal the need for the server to compute similarities with previously saved fingerprints, or
- the privacy policy, when available, of the company owning the script mentions *fingerprinting*, *stateless identification technique*, *device* or *browser identification*.

As soon as the manual evaluation matches 2 of the above criteria, we label the script as *fingerprinter*. If at most 1 criteria is matched, we label the script as *non-fingerprinter*. If we cannot perform a manual analysis, for example due to the script being too obfuscated, we label the script as *unknown*. Whenever we label a *fingerprinter*: i) the fingerprinting attributes of s_i are added to the detection matrix M_F ; ii) we rerun the classification process in a new iteration and only stop when all the scripts of our dataset have been labeled.

3.2.2 Script Classification Results

We ran the algorithm on the 4,665 scripts embedded in the 1,485 web pages of our dataset. At the time we collected the data, DISCONNECT labeled 82 scripts as implementing browser fingerprinting. We cleaned the list provided by DISCONNECT by removing the scripts matching one of the following rules:

- browser fingerprinting scripts without a URL (12 scripts) or with a URL that does not point to a JAVASCRIPT file (15 scripts),
- scripts that changed and do not include the fingerprinting attributes they were initially labeled for (7 scripts),
- scripts we could not run, either because they were too obfuscated and we could not find the triggering events, or because we were lacking additional resources (13 scripts).

After these steps, we kept a list of 35 fingerprinters and integrated each fingerprinter in a blank page to collect the fingerprinting attributes as described in [Section 3.1](#). We filtered duplicate scripts that collect the same attributes, and bootstrapped our classification algorithm with a classification matrix M_F composed of 19 distinct sets of fingerprinting attributes. Finally, we ran our algorithm to label our dataset. The distribution of the accumulated number of scripts labelled as *fingerprinter*, *non-fingerprinter*, *unknown*, as well as not-yet labeled scripts is reported in [Figure 3.4](#). Fingerprinting scripts were mainly labeled in the first iterations, non-fingerprinting scripts until the 35th. The manual effort only represented 7.7% of the total number of scripts in our dataset, showing the benefits of our approach. Overall, out of 4,665 scripts, we identified 199 browser fingerprinting scripts.

Finally, we removed duplicate fingerprinting scripts. We removed the URL parameters of the scripts when they did not change the attributes collected, and removed the resulting

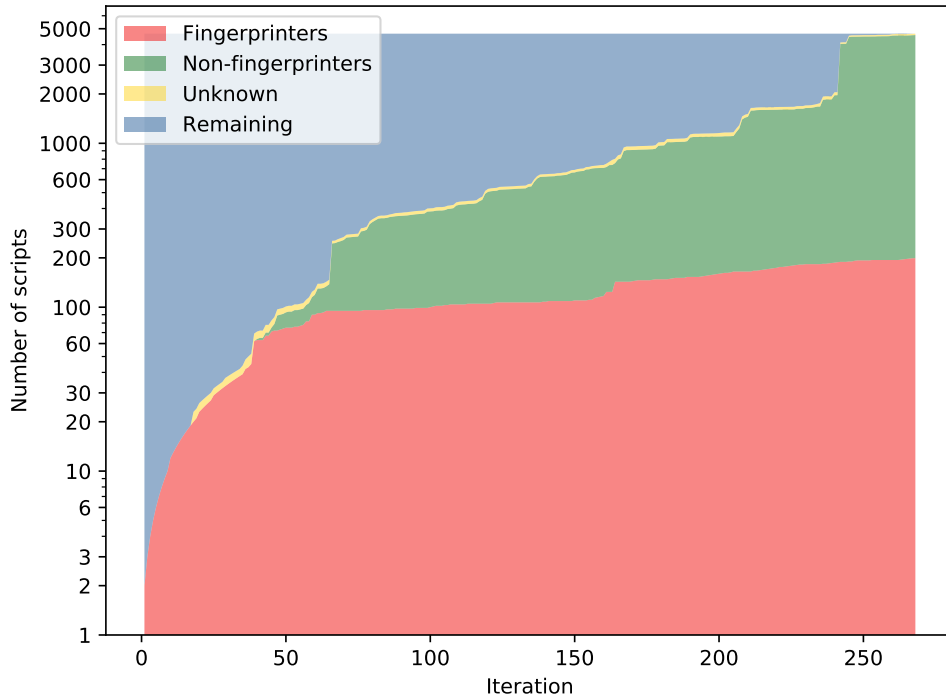


Figure 3.4: Iteration distribution of the label put on the scripts of our dataset.

duplicates. We also removed duplicate scripts when they had a similarity score of 1 with only small differences in their names, URLs, or domains. We believe these changes are due to domain or URL updates, but the scripts are essentially the same. After removing the duplicates, we obtain a final dataset of 169 browser fingerprinting scripts.

3.2.3 Algorithm results validation

We computed a sample size to validate our algorithm. Choosing a confidence level of 95% with a margin error of 5%, our sample contains 355 elements.² We randomly chose 355 scripts from our dataset that were automatically labeled. We labeled these scripts manually, leading to 349 correct and only 6 incorrect labels compared to the results of our algorithm. Given the low proportion of differences between the label given by our automatic method and the results of our manual validation, we believe our classification method is efficient to detect browser fingerprinting scripts.

In this section, we propose a new scripts classification algorithm. Applied on our dataset, we detected 169 browser fingerprinting scripts. The following section will study these browser fingerprinting scripts, as well as their inclusion the pages and websites of our dataset.

²<https://www.checkmarket.com/sample-size-calculator/>

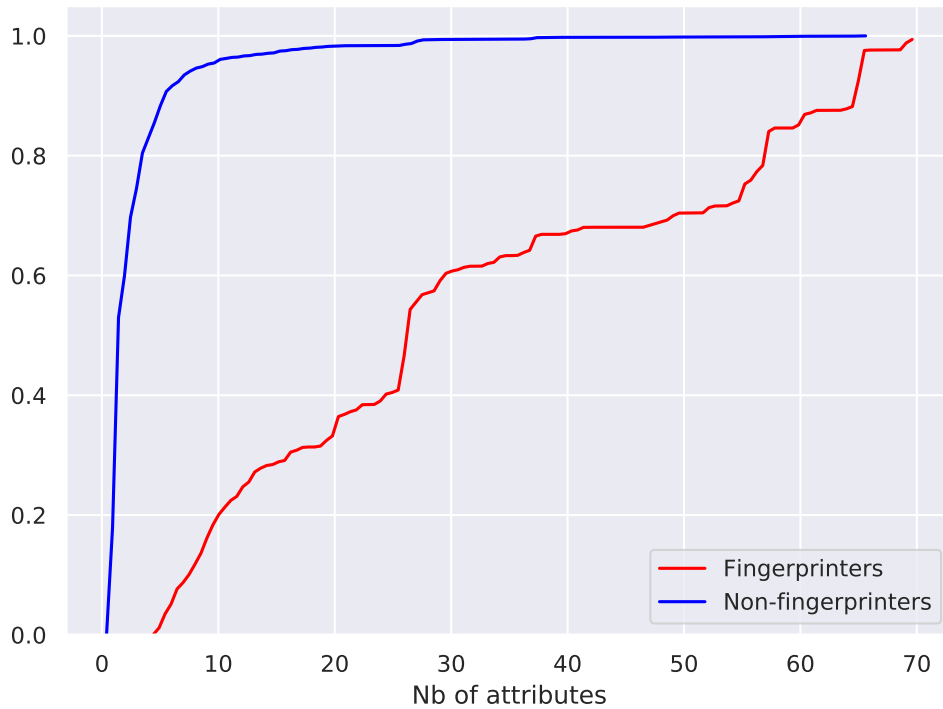


Figure 3.5: CDF of number of attributes used by scripts.

3.3 Analysis of Secure Web Pages

3.3.1 Browser Fingerprinting Attributes

We first observed the collected attributes. [Figure 3.5](#) reports on the *Cumulative Distribution Function* (CDF) of attributes collected by the scripts of our dataset. The lowest number of attributes collected by a fingerprinter is 5, showing fingerprinting attributes can be used selectively. For example, we detected a script that clearly indicated their intention to identify the user thanks to specific function names, such as `deviceFingerprinting` or `getFingerprint`,³ yet only accessed 8 attributes.

Of the 169 browser fingerprinting scripts we classified, we observed 132 distinct fingerprinting attributes that we organized into 8 families. The family of an attribute is the parent `JAVASCRIPT` object calling the attribute; except for the bot attributes where we used Jonker *et al.* list [111]. [Figure 3.6](#) presents the distribution of the attributes per fingerprinter grouped by family, showing that all attribute families are exploited in the wild. The most accessed attributes are the `User-Agent`, screen `width` and `height`, `plugins` list, and `timezone`. Even if it would be tempting to rely on these to detect fingerprinting scripts, they can be used for many other purposes, such as analytics or adjusting a website’s UI to the device. In our dataset, these attributes are used

³<https://jsak.mmtcdn.com/flights/assets/js/desktop.3410609e.js>

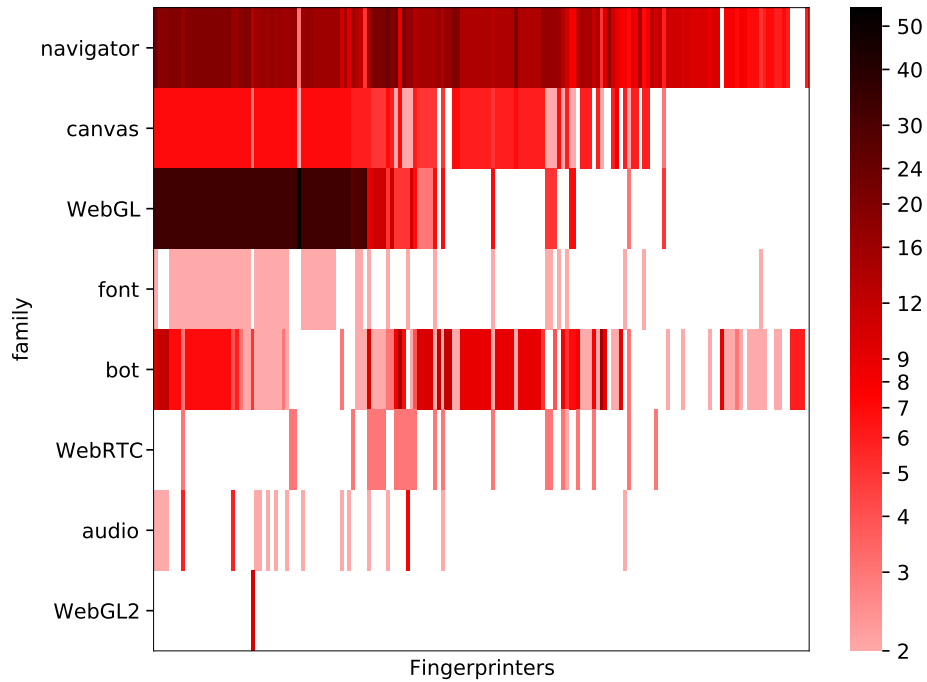


Figure 3.6: Distribution of attributes families across fingerprints.

by respectively 81%, 20%, 19%, 6% and 13% of non-browser fingerprinting scripts, respectively. This illustrates the difficulty of identifying reliable attributes to detect browser fingerprinting scripts.

We analyzed the scripts that use canvas or WebGL fingerprinting. 120 scripts fingerprint browsers using canvas drawing primitives, using between 2 and 14 different drawing instructions. We found 44 different drawing sequences. Concerning WebGL fingerprinting, 54 scripts draw with WebGL primitives, using between 17 and 20 distinct drawing instructions. Only 7 drawing instruction sequences are different. Moreover, one single sequence is used by 46 scripts. Concerning fonts enumeration, we measured 54 fingerprinters used this technique. The number of fonts tested ranges from 66 to 594, with 19 different sets of fonts. We observe 2 sets of fonts being largely checked by fingerprinters: a set of 82 fonts tested by 17 scripts, and a set of 66–69 fonts used by 18 scripts. Thus, even though there is a potentially unlimited combination of testable fonts, a majority of scripts use similar sets. We believe this is due to these font sets being copied from one fingerprinter to another, as well as, being sufficient to capture enough uniqueness. We observed 107 fingerprinters collect at least one bot attribute. The average number of bot attributes is 5. **PhantomJS** attributes are the most collected (41% of all scripts), followed by those that detect **Headless Chrome** (18–33%) and **Selenium** (12–16%).

Finally, we observed that the most used attributes belong to the earliest identified for browser fingerprinting, such as the navigator and screen properties (Eckersley *et al.* [96] in 2010) and the canvas (Mowery *et al.* [122] in 2012). More recent attributes are less present in our dataset, such as audio and WebRTC (Englehardt *et al.* [97] in 2016). We also found one fingerprinting script accessing 9 properties and functions from the `WebGL2RenderingContext` object, which is part of the WebGL2 APIs and has not been studied by the community yet.

3.3.2 Similarities of Browser Fingerprinting Scripts

As we explained previously, our algorithm labeled 169 browser fingerprinting scripts. DISCONNECT classifies as browser fingerprinting the inclusion of any piece of code from the FINGERPRINTJS library [33]. We therefore wanted to study the influence of FINGERPRINTJS on the browser fingerprinting scripts of our dataset. We ran the FINGERPRINTJS script in a blank page and monitored the attributes it collects. We measured the proximity between this script and our browser fingerprinting scripts by computing the similarity score (Jaccard index) between each of them and FINGERPRINTJS. We iterated over our browser fingerprinting scripts to see if they could be related with FINGERPRINTJS by evaluating the following rules:

- the script has a similarity score of 1 with FINGERPRINTJS;
- the script has exactly the same subset of fonts, canvas drawing primitives or WebGL drawing primitives. As we said in the previous section, the potential possible sets of these elements are infinite. Thus, having the same sets of font or drawing primitives uncover a strong link between these scripts;
- The script contains one of the following keyword: *fingerprintjs*, *fingerprint2*, or functions with a specific name: *hasLiedBrowser*, *hasLiedLanguages*, *hasLiedResolution* or *hasLiedLanguages*. As a non-negligible proportion of the scripts in our dataset were obfuscated, the number of *fingerprinters* matching this rule will probably be underestimated.

Among the 169 distinct browser fingerprinting scripts included in our dataset, one has a similarity score of 1—the raw FINGERPRINTJS library present in our dataset. Then, 6, 7, and 45 browser fingerprinting scripts had the same sets of fonts, canvas drawing primitives and WebGL drawing primitives, respectively. Finally, the monitored keywords were found in 60 distinct fingerprinting scripts. After removing the duplicate scripts—those which were matching several rules—we ended up with a list of 61 browser fingerprinting scripts having string similarities with FINGERPRINTJS. This result demonstrates the impact and the central role of this library on the fingerprinting ecosystem.

3.3.3 Origins of Browser Fingerprinting Scripts

We studied the origins of the browser fingerprinting scripts by looking for their hosting domain and additional hints in the scripts, such as comments. From our 169 distinct browser fingerprinting scripts, 82 are first-party, meaning they are hosted by the same domain as the pages they appear in, and 87 are third-party. We did not detect any browser fingerprinting scripts being served both as first and third-party. Concerning the third-party browser fingerprinting scripts, we then uncovered the owner of the script by browsing on the website, by using the WHOIs tool, or by looking for clues in the script comments.

The results are presented in [Table 3.1](#). We failed to detect the website category for 24 third-party fingerprinting scripts. Moreover, it is difficult to uncover the owner of scripts being hosted on CDNs (11 of the scripts we identified). We also noticed browser fingerprinting scripts being hosted on domains with explicit objectives, such as security-oriented (12) or ads/analytics companies (17). Finally, we detected 121 domains that host fingerprinting scripts. 30 domains host at least 2 distinct scripts. We believe several reasons can explain this behavior:

- script variants hosted by the same domain yet with a few changes in the attributes (*e.g.*, 2 scripts hosted by GEETEST^{4,5} show this),
- several scripts can serve different purposes—ads, bot detection, authentication—even if they are hosted by the same domain,
- scripts hosted by CDN domains. In our dataset, CLOUDFRONT hosts 9 of our browser fingerprinting scripts.

Regarding the adoption of browser fingerprinting for security purposes, we analyzed the scripts hosted by domains whose main goal is security. We identified 14 fingerprinting scripts from 12 security-focused organizations. For each of the organizations, we extracted their main purpose, and analyzed the presence of their scripts on the sensitive page types we defined. [Table 3.2](#) reports on these results. All of these security scripts are present in at least one of our 4 sensitive page types.

We analyzed the attribute families collected by these browser fingerprinting scripts. All major techniques are being actively used. The navigator and screen properties are

Table 3.1: Distribution of first-party & third-party scripts per website category.

Domain category	# of fingerprinters	
	First-party	Third-party
Flights & transports	24	1
Bank & money transfer	19	1
Ecommerce	9	8
Ads & analytics		17
Security	2	12
CDN		11
Event tickets	8	
Travel	7	1
Governmental	7	
Technology	1	6
Dating	3	
Cryptocurrency & trading		3
Social network	1	1
Job search		2
Adult	1	
Unknown		24
Total	82	87

⁴<https://static.geetest.com/static/js/fullpage.8.6.1.js>

⁵<https://static.geetest.com/static/js/fullpage.8.8.9.js>

Table 3.2: Summary of security organizations, with the accessed attributes and the presence in the web pages of our dataset.

Organization goal	Organization name	Script # of attributes	Script presence on					
			# domains	# pages	Sign up	Sign in	Basket	Payment
Payment platform	ADYEN	47	1	1				✓
	CENTROBILL	14	1	1			✓	
	PROBILLER	29	1	4			✓	✓
	RAZORPAY	10	1	1				✓
	SECURED TOUCH	73	1	6		✓		
Fraud prevention	IOVATION	8	1	1				✓
	NUDATA SECURITY	29	2	3	✓	✓		
	SIFT SCIENCE	26	10	26	✓	✓	✓	✓
	SIMILITY	49	2	3				✓
Bot protection	DATADOME	33	1	1		✓		
	GEE TEST	64-65	4	7	✓	✓		
	PERIMETERX	69	1	3				✓

the most collected (included in 14 scripts), followed by canvas (12), bot (11), WebGL parameters (10), WebGL drawing (6), audio and font enumeration via span (5), canvas font enumeration and WebRTC (3). Access to `navigator.userAgent`, `navigator.platform` and `navigator.vendor` was found in 13 scripts. Because these navigator attributes overlap, we believe that they are used to detect spoofing. Moreover, we observed 13 scripts where `screen.width`, `screen.height`, `screen.availWidth`, and `screen.availHeight` are collected. These attributes can also be used to detect spoofing, as the available sizes should be smaller than the width and height. Jonker shows this invariant can detect bots [111]. The 3 organizations that claim to protect against bots naturally collect bot attributes. PERIMETERX collects 10 of them, and DATADOME 5, both covering all major bot types. However, GEE TEST only collects 2 bot attributes, both for detecting PHANTOMJS.

3.3.4 Web page type and website category & country impact

Secured vs non-secured web pages. We analyzed the ratio of web page types that include a browser fingerprinting script. We found browser fingerprinting scripts on all types of web pages we studied. **Basket** (33.8%) and **Sign-up** (31.1%) pages fingerprint more than the average, followed by *content* (25.6%), **Payment** (25.3%), **Sign-in** (23.4%), and *home pages* (23.0%). Other studies have not targeted these specific page types and have generally relied only on home pages. Consequently, we are—to the best of our knowledge—the first study to observe the prevalence of browser fingerprinting in sensitive and secure web pages. We compared fingerprinting in secured to non-secured pages. We found 54 scripts exclusive to secured pages, 68 scripts exclusive to non-secured pages, and 47 in both.

We counted the number of fingerprinting scripts included on a page. Out of 405 pages that fingerprint, 339 pages include 1 script, 51 pages had 2 scripts, and 15 pages had 3 scripts. Out the 66 multi-script pages—*i.e.*, 51+15—10 had first-party fingerprinting scripts, 27 served fingerprinting scripts from a different domain, and 29 served fingerprinting scripts from both first and third-party domains. We make several hypotheses based on our observations: i) the browser fingerprinting scripts have different purposes, such as advertising or security services, and likely do not share the fingerprints they collect, ii) in the case of pages from websites being developed by several teams, they may integrate multiple browser fingerprinting scripts unintentionally. A majority of the pages with multiple fingerprinting scripts are secured pages (35 secured versus 31 non-secure pages) although secured pages represent only 44.2% of our dataset. This result supports the statement that secured pages fingerprint more aggressively than non-secured pages.

Website category and country impact. We studied the presence of browser fingerprinting scripts according to the website category. The results report on a wide disparity in the adoption of browser fingerprinting scripts. The flights & transports (49%), accommodation & travel booking (40%) and the business websites (37%) are the websites categories having the highest percentage of integration of at least one browser fingerprinting script. Oppositely, dating (7%), institutional (11%), and adult websites (12%) fingerprint the least. For bank and money transfer services, 21% fingerprint their users, which is close to average (24.4%). Combined with our previous analysis about the adoption of browser fingerprinting scripts by page type, it seems that the presence or absence of a browser fingerprinting scripts is driven more by website category than page type.

Regarding the adoption of browser fingerprinting scripts per website country, we also observed some variations. Japanese (7%) and Spanish (6%) websites are the ones embedding the fewest browser fingerprinting scripts. On the other side, 62% of the Russian websites integrate at least one browser fingerprinting script, far ahead from American websites (32%), the second category in this ranking. We evaluated the browser fingerprinting scripts embedded on those Russian websites, and found 22 out of the 24 Russian websites integrating a browser fingerprinting script used at least one of the two browser fingerprinting scripts provided by Yandex.^{6,7} We do not know the reasons behind such a high presence on Russian websites.

⁶<https://mc.yandex.ru/metrika/tag.js>

⁷<https://mc.yandex.ru/metrika/watch.js>

3.3.5 Additional Security Mechanisms

During our data acquisition, we also marked the usage of any additional authentication factor or bot detection techniques we found. The results are reported in Table 3.3. The total number of pages in this table is higher than the 1,486 web pages we reported initially, because some pages can refer to several types, as we explained in Section 3.1.

Authentication factors. We observed

39 pages with 2FA schemes, the majority being **sign-up** pages. 3 distinct factors were used during our collection: i) an email OTP or confirmation (used 19 times), ii) an SMS OTP (17), and iii) a phone call (2), in which the code to enter on the website is the last x digits of the calling number. The usage of an additional factor for **sign-up** pages implies a stronger requirement for proof of identity. Because browser fingerprinting could fulfill this requirement, we measured the number of browser fingerprinting scripts included in these pages, compared to other pages. We observe only 1 **sign-up** page that contains an additional factor also embedding a browser fingerprinting script. Moreover, this web page also included a bot detection mechanism. This means the website used on the **sign-up** page both an authentication confirmation and a bot detection technique. Thus, the presence of a browser fingerprinting script might be used to serve either of these purposes, as we are only observing from the client-side, we cannot conclude.

Table 3.3: Number of pages including a multi-factor authentication mechanism or a bot detection technique, depending on the page type and the presence of a browser fingerprinting script in the page.

All pages		Pages with 2FA		Pages with bot detection	
		Without FP	With FP	Without FP	With FP
Sign-up	206	20	1	18	11
Sign-in	197	8	0	7	3
Payment	95	0	0	0	0
Basket	210	1	0	2	0
Home	304	3	0	5	2
Content	586	4	0	3	2
Total	1,598	38	1	35	18

Bot detection mechanisms. Concerning bot detection, we found 51 scripts using bot detection mechanisms: 41 RECAPTCHA, 6 GEETEST puzzles,⁸ and 4 textual captchas. As for the additional authentication factors, they were mainly observed in **sign-up** pages. 2 pages were using 2 bot detection techniques: a **sign-in** page on an adult website and a **sign-up** page on an event ticket website. Half of the pages with a bot detection mechanism embed a browser fingerprinting script. This result shows the interest of websites in the browser fingerprinting technique to detect bots.

SYNTHESIS. In this section, we explored the browser fingerprinting scripts of our dataset, their presence on the different page types we considered and the adoption of fingerprinting

⁸<https://www.geetest.com/en/demo>

in combination with additional security mechanisms. Our results allow us to answer **RQ1**. We show that browser fingerprints are effectively accessed for all the types of web pages covered by our study. We finally observed that browsers are fingerprinted slightly more aggressively on secured pages.

3.4 Websites resilience against 2 attack models

In this section, we define 2 attack models and evaluate the resilience of the websites of our dataset against them.

3.4.1 Stolen credentials

Extracting the protected websites. We observed in [Section 3.3](#) that browser fingerprints are collected by websites during the authentication process. We are interested in observing any security improvements brought by fingerprinting in the case of stolen credentials. In our attack model, we assume that a hacker steals a user’s credentials—through a data leak, a phishing page, or any other technique described in [Section 2.2.2](#)—and tries to authenticate into the targeted website. We reproduced this attack behaviour by trying to authenticate into the accounts we created for this experiment. It is worth mentioning that we used a phone number or any additional information needed to create the account, but we skipped anything that was not mandatory. We assume that the attacker may use a different browser instance on a different OS, with different cookies than the victim’s browser, while browsing from a different network than the network associated with the original accounts. We also assume that the attacker can solve captchas when using stolen credentials. Therefore, bot detection mechanisms are not a reliable protection in this context. We ran this attack on the 42 websites we were able to create accounts on—12 of them use a browser fingerprinting script on the **sign-in** page. The 42 websites are well balanced concerning the *Country* tag we defined, and mainly concern cryptocurrencies, money transfer, e-commerce, adult, event and sport tickets content. Among these websites, 16 of them belong to the Alexa Top 1k, 8 between 1k and 10k, 11 between 10k and 100k, and 7 above the Top 100k. We expect websites that collect browser fingerprints to use this information to protect the accounts from stolen credentials. Our attempts to authenticate to the accounts with different fingerprints fell into the following 3 cases:

- we were able to authenticate into 37 websites without facing additional multi-factor authentication mechanisms or security warnings;
- 3 websites sent a warning message about an unknown authentication attempt to our account. These messages contained the IP address, the OS and the browser we tried to connect with;

- 2 websites asked for an additional proof of identity. The first one sent an *One-Time Password* (OTP) via email with additional information about the ongoing connection. The other sent an OTP via SMS to validate the connection attempt. Those 2 websites also proposed a security panel where the user can check their trusted devices.

We observe that only 5 out of the 42 websites react in a manner that strongly suggests fingerprint-based detection of known devices and browsers is being used to secure the account, namely GOOGLE, WETRANSFER (files transfer service), SKRILL, CRYPTO and BINANCE (cryptocurrencies, finances or money transfer websites). 4 of them have a security panel with the authorized devices with their characteristics and all the authentication attempts made to authenticate on the account.

Isolating the triggering features As we noticed during our previous experiment, several details, including the OS, browser and IP address, were provided to the user to explain the warnings or requirements for additional information to authenticate. The IP address can be used to extract the approximate geolocation of the user and to detect authentication attempts from unusual networks (*e.g.*, through a cloud provider). While the usages of cookies were not indicated in the information given by websites, we believe the presence of previous session cookies might also be used by the website to decide to authenticate the users more easily. Their absence might reveal a device or browser change. As we mention in [Section 2.2.5.3](#), these 4 elements are often used as features for *Risk-Based Authentication* (RBA). Thus, we believe we trigger additional security mechanisms because the risk of our authentication attempts were considered as too high. As the 5 websites of our previous results seem to use RBA, we tried to isolate the feature or set of features used by each website to measure the risk level of the authentication attempt. To do so, we tried the following 6 combinations:

1. we re-authenticated with the same conditions as the ones the account was created with, to get the ground-truth,
2. we signed-in with the same browser, but using a different IP address than the one used for the account creation and previous authentication attempt,
3. on a different IP address, using the same browser, we authenticated in using the browser's private mode to navigate without reusing any cookies previously set,
4. on the same device, but using a different browser, with a different IP address,
5. we then tried to re-authenticate with a different device and browser, and a different IP address,
6. finally, we tried to log again on the same device and the same browser as the previous combination, without deleting cookies or any other stateful data, but we changed our IP address to reuse the original IP address used to create the account and authenticate for the ground truth combination.

Table 3.4: Parameters and results concerning the reauthentication experiment.

Website	MFA on Sign-up	FP on		Features combinations					
		Sign-up	Sign-in	Ground truth	Diff. IP	Diff. IP & no cookies	Diff. IP, browser & no cookies	Diff. IP, browser, device & cookies	Diff. browser, device & cookies
GOOGLE	☞			☑	☑	☞	☞	☑ + ▲	☑ + ▲
SKRILL				☑	☑	☞	☞	☞	☞
CRYPTO	☞	✓	✓	☑ + ▲	☑ + ▲	☑ + ▲	☑ + ▲	☑ + ▲	☑ + ▲
WETRANSFER				☑	☑	☑ + ▲	☑ + ▲	☑ + ▲	☑ + ▲
BINANCE	@	✓	✓	☑	☑	☑ + spec. ▲	@	@	☑ + spec. ▲

We ran the above combinations in order. Browser and OS changes make the fingerprint different, contrary to IP addresses and cookies that do not affect the browser fingerprint. If fingerprinting is used as a feature for RBA, as proposed by several studies [130, 87], we expect to observe different behaviors on the combinations where the fingerprint changes, namely combinations n°4, n°5 and n°6.

Table 3.4 summarizes up the results we observed according to the changes we applied to the browsing features. GOOGLE, SKRILL, and WETRANSFER seem to be based on cookies. When they are not present, the first 2 ask for an OTP, while WETRANSFER sends an alert. CRYPTO always has the same behavior: it allows the authentication attempt but sends an alert. Finally, BINANCE have the most advanced system. First, it sends an alert about the IP address when we changed it, when browsing without cookies (combination n°3). The message did not contain any reference to device or browser changes, so we believed the website knew we were on the same device and browser. The behavior changed when we used another browser on the same device: BINANCE sent us an email containing an OTP and basic information about the new browser used (combination n°4). It also sent an email with an OTP when using a different device (combination n°5). When staying on the browser chosen on this second device and using the same IP address as the one used, we received an alert (combination n°6). Based on this last experiment, we make 2 observations:

- the alert message is different from the one when we changed the IP address. The message now mentions a change in the browser and device;
- we received an email alert, but we were still able to authenticate. We believe this is because the cookies set by the browser were the same as the ones in the previous combination when we needed to provide an email OTP to validate the authentication attempt.

In our dataset, we observe browser fingerprinting being used as a feature for RBA only once, in combination with other identifying techniques, to resist stolen credentials. As this attack is similar to a user trying to authenticate from a fresh browser, it also illustrates the additional steps users need to complete to authenticate with a new browser or a new device.

3.4.2 Cookie hijacking

3.4.2.1 Attack design

Cookie hijacking can lead to vulnerable accounts and data leaks [138]. As browser fingerprinting can be used to identify a browser, we make the hypothesis it can be used to verify if a cookie has been hijacked and used by a different browser. Our goal is not to study the existing ways to perform a cookie hijacking. In our attacks, we assume an attacker was able to steal cookies, no matter the method used—XSS vulnerability, insecure network exchanges, malicious JAVASCRIPT injection. Instead, we aim at studying the resilience to cookie hijacking by websites in our dataset if browser fingerprinting is used to protect the accounts. We designed 2 attacks to study cookie hijacking.

Our first attack is **session hijacking**. As explained in [Section 2.2.2](#), it consists in trying to authenticate using cookies stolen from an existing user session. We authenticated to the target site on a first browser, then we extracted the cookies and authentication page URL and inserted these into a second browser. If the attack works, the second browser will be authenticated and the session will be in the same state as on the first browser. If not, the second browser will be stuck on the authentication page.

Our second attack is **basket hijacking**. The goal is to obtain the same basket as a user by hijacking their cookies. We filled a basket with a commercial item, and visited the page summarizing the basket and its content. Similarly to our session hijacking attack, we then extracted the URL and cookie, and put them in another browser. If the basket content is the same on the 2 browsers, the attack is successful.

3.4.2.2 Methodology

For each website, we automated the browsing to the required pages with a PUPPETEER instance. We automate the insertion of cookies and the navigation to the URL with a second PUPPETEER instance. We lower the possibility to be detected as a bot by changing the fingerprint of the PUPPETEER instances for them to look like CHROME 84. To do so, we re-used the value of each attribute collected by fingerprinters during our manual data collection and integrated them into an extension in the PUPPETEER instance that returns the corresponding value when an attribute is accessed. We also added a delay of at least one second between each action on a page.

Before studying the impact of fingerprint modification, we performed a preliminary run with and without the collected cookies to make sure that sessions could be stolen from the PUPPETEER instance and that no other parameters, like `localStorage` or a hidden parameter in the URL, would impact our measurement. This way, we created a subset of websites where our attacks are successful. Finally, we ran our attack on all the websites of this subset and collected the cookies and URLs. We used different parameters and configurations for the second PUPPETEER instances by running them on

Table 3.5: Number of websites involved in each step of the validation for the session and basket cookies attacks, and results of the attack on the validated subset

	# websites			
	Session		Basket	
	FP	no FP	FP	no FP
Original dataset	12	30	33	51
Cannot automate	0	5	8	8
Anti-bot defense triggered	2	1	3	3
Impacted by other parameters	3	5	6	6
Nothing sold	0	0	7	16
Validated subset	7	19	9	18
Attack works	7	19	9	18

a different device on a different network with a different IP address. We also changed all the fingerprint attributes we monitored during our data collection by giving them values from a FIREFOX 72 instance with the same extension as described earlier in this section. As explained in [Section 2.6.3](#), browser fingerprinting can be used to enhance session security by checking the cookies are always associated with the browser instance which first received the cookies [145, 87]. Should a website be protected and detect the different fingerprint, we rerun the attack by modifying parts of the fingerprint to detect which attributes or combinations trigger the defense mechanism.

3.4.2.3 Results

We ran these experiments in July and August 2020. We used the 42 websites we were previously able to create an account on for the session hijacking attack. Concerning the basket hijacking attack, we used the 84 websites of our dataset containing at least a basket page—33 of them contain at least one fingerprinter on a basket page. We then ran each step of our validation process to make sure the cookies were the only variable needed to retrieve the basket or session state. The results are presented in [Table 3.5](#). Because of the time gap between the data collection explained in [Section 3.3](#) and this experiment, we were unable to fill baskets for several websites with a single item as some of them were not selling anything anymore. We believe this is likely due to the economic and societal restrictions following the Covid-19 pandemic. We ended up with a validated subset of 26 and 27 websites for our session and basket hijacking attacks, respectively. We then ran our attack and inserted the cookies on a different device on a different network with a different fingerprint and HTTP headers. With these parameters, the attacks worked on every website of our validated subset. These behaviors imply no

defense mechanism was being used. Thus, browser fingerprinting is not used to protect against a session or basket hijacking on the websites of our dataset.

3.4.2.4 Usages of cookies protections mechanisms

As we did not detect any usage of additional security mechanisms, we studied the way *HTTPS* and *HSTS* are deployed and how cookies are configured to observe if their settings were secure enough to protect against traffic sniffing. If these elements are properly set, it lowers the attack surface on cookies by complicating their extraction via *JAVASCRIPT* and avoiding their theft from *HTTP* requests, as explained in [Section 2.2.3](#). Over the 53 websites (42 + 84 minus duplicates) we tested our attacks on, 52 were redirecting their traffic through *HTTPS* and 30 of these 52 websites were setting the **Strict Transport Security** response header in the browser. During the experiments, we collected and injected 1,080 cookies. Respectively 198 (18%) and 305 (28%) were **HTTPOnly** and **Secure**. We also looked at the **SameSite** parameter, observing 11 (1%) and 109 (10%) cookies have a **Strict** and **Laxist** **SameSite** policy, respectively. Even if the **SameSite** parameter is now set by default to **Laxist** since *CHROME 80/FIREFOX 69*, few websites were setting it to a secure value, indicating they were added before to all requests because of the default **None** **SameSite** policy.

Based on these observations, we conclude that developers put a lot of trust in cookies as their presence alone in our tests leads to direct user authentication. This trust is only possible thanks to strong security mechanisms in browsers that have grown and matured a lot in the past decade. The rise of *HTTPS* coupled with a lot of control over what can be executed on a webpage (through *CSP*, *CORS* and all their derivatives) have changed the way we come to reason about cookie hijacking and how much harder it is to pull off such an attack today. Yet, our experiment shows that if indeed cookies are stolen, none of the tested websites have any mechanisms in place to detect any irregularities. We can only hypothesize at this point that this may not be in the scope of their threat model.

SYNTHESIS. In this section, we designed 2 attack models and tested them to measure the effectiveness of fingerprinting to protect users on web pages in our dataset. We answer **RQ2** by observing fingerprinting is successfully used to improve security on one website against our first attack. Concerning our second attack, we did not detect any website in our dataset that used browser fingerprinting to protect against cookie hijacking.

3.5 Discussion

3.5.1 Intents in fingerprinting usages

In the case of browser fingerprinting, analyzing why a script is included in a web page and why it is accessing specific attributes is complex as there is little indication of what

will be done with the collected data once transferred to a server. Still, it is possible to rely on some signs to capture the intent behind a fingerprinting script, such as:

- **Accessed APIs:** depending on the goal of the script, some APIs may be picked over others. For example, anti-bot companies access well-identified bot attributes, while others interested in cross-browser fingerprinting access OS and hardware-specific attributes;
- **Number of collected attributes:** while a very high number of attributes can often be linked to a fingerprinting behavior, the numbers vary. As seen in [Table 3.2](#), some third parties, like IOVATION, build on only 8 attributes, while others, like SECURED TOUCH, collect up to 73; As many studies in the state of the art are interested in either uniquely identifying devices or detecting inconsistencies, it makes sense to collect as many attributes as possible. Yet, as seen with IOVATION, if you have a clear goal in mind, collecting very few attributes can be enough for your purpose;
- **Execution context:** where the fingerprinting script is located can reveal intent. If a fingerprinting script is included in all web pages, it is probably linked with an anti-bot system but, if it is only present on a payment page, then it is likely used for fraud prevention.

Considering the above signals, it is possible to estimate how the collected information will be used, but it does not provide certainty without having access to the backend where the browser fingerprints are analyzed.

3.5.2 Fingerprinting is barely used for security

In [Section 3.3.3](#), we identified third-party actors who collect a wide range of data to implement bot protection and fraud prevention. They protect a website globally against external threats. Yet, when looking at what is offered to protect user accounts, the story seems to be very different. Based on our experiments detailed in [Section 3.4](#), there is little evidence that fingerprinting is currently being used to protect individual accounts. As we detected fingerprinting scripts delivered by 12 security-oriented organizations, we would have expected them to add an additional security layer to protect users. This raises the question of the relevance of using such a script from a security organization if the final usage is not security. More generally, we tested the defenses of 42 websites by creating accounts and authenticating with several contexts and parameters. Apart from some warning messages with few details on the new device, we found only a single website blocked access to their services when the browser fingerprint did not match. Moreover, we have not detected any usage of browser fingerprinting to protect against our second attack, the cookie hijacking. We believe these are negative results of our paper and deserve further discussion.

First, these results raise the question of why we observed such behaviors. One concern could be the accuracy of the browser fingerprinting algorithm. While cookies and IP addresses send strong signals that websites have relied on for years, a fingerprint is, in contrast, more volatile. It can change due to a minor modification to the browser's configuration or an update. Some attributes may be deemed too unstable to be included for verification, while others are much more reliable and even predictable. As detailed by Vastel *et al.* [150], browser fingerprinting techniques require constant adaptation to maintain their robustness. Another concern is user experience, as having an overly sensitive algorithm could prompt for additional checks too often, even if the user did not change their device or browser.

3.5.3 Deficiencies in the state of the art

As we identify concerns about the use of browser fingerprinting in an authentication system, we believe the state of the art currently lacks studies to measure the effectiveness and reliability of RBA or MFA with browser fingerprinting. First, as mentioned by Preuveneers [130], users would need a way to add a new fingerprint to their account to be able to authenticate with another device. Websites in our dataset seem to use an OTP email. We believe other options should be studied because each authentication system is different and has its own trade-offs. Also, fingerprints evolve over time, and an authentication system should be able to tell if a fingerprint is an evolution of an already registered one or not. While solutions have been proposed to compute a fingerprint evolution [150], it has been shown to not be fast enough when confronted with a large dataset [119]. Used in an authentication context, it would require a quick decision to have a negligible impact on the user experience. An interesting study has been proposed by Alaca *et al.* [87] about the requirements of such a system, but due to the rapid evolution of the web ecosystem, the study might be outdated. Finally, the state of the art lacks an evaluation of the user experience, satisfaction, and confidence when using this kind of system.

3.6 Conclusion

In this chapter, we studied the adoption of browser fingerprinting for security applications. More specifically, we analyzed 4 types of secured web pages—sign-up, sign-in, basket, and payment—that process sensitive personal data. We considered the state-of-the-art JAVASCRIPT attributes and developed an extension to monitor browser fingerprinting attribute accesses. To avoid biases introduced by automated crawlers and bots, we manually visited 1,485 pages published by 446 websites, and created accounts, authenticated to verify authentication procedures, and went through the payment processes

where available. We designed and implemented a script classification algorithm based on accessed APIs, and labeled 169 distinct browser fingerprinting scripts. We observed these fingerprinting scripts being served by all types of secured pages and various website categories, which answers our **RQ1**. We analyzed the script providers and found 12 security-focused organizations that use browser fingerprinting in secured web pages. We measured the use of additional authentication factors and bot detection mechanisms, showing fingerprinting is used in combination with several bot detection techniques. We defined 2 attack models, stolen credentials and cookies hijacking, and evaluate websites in our dataset against them. Finally, we answer **RQ2** by observing very little usages of fingerprinting to secure websites against these 2 attacks.

Chapter 4

FP-Controlink: Studying fingerprinting under a controlled environment to link fingerprints

One of the main challenges for an authentication scheme using browser fingerprinting is to properly link fingerprints sent by the same browser instance over different sessions. As fingerprints evolve over time, it is essential to take into account possible changes and anticipate them. Vastel *et al.* [150] have developed an algorithm to link browser fingerprints, but they did not focus on the authentication challenges, nor did they take into account the real-time constraints required by an authentication system like speed or precision. Finally, the lack of understanding of browser fingerprints semantics is also a strong obstacle to relevant and reliable linking solutions. In this chapter, we perform the first complete measurement study of browser fingerprint diversity and evolutions in a controlled environment. We reproduce the experiments made by Al-Fannah *et al.* [86] and study more in-depth the different components responsible for browser fingerprints diversity and evolution. This notion of *controlled environment* is a keystone, as it gives us the ability to attribute a precise change in a fingerprint to a very specific hardware or software component. By knowing the exact composition of a system from its software to its hardware, we are able to understand what is causing a specific change in a fingerprint and include it in an authentication logic to better link fingerprints together.

In [Section 4.1](#), we design an environment using desktop and mobile devices to study browser fingerprints under controlled parameters. We assemble a ground truth from 4 desktop and 23 mobile devices and collect a total of 1,160 fingerprints. In [Section 4.2](#), we provide a desktop and mobile evaluation of browser fingerprint uniqueness. We show that the fingerprints uniqueness is a consequence of the diversity of the devices, from the OS and browser components to the user configuration. In [Section 4.3](#), we follow the evolution of browser fingerprints through browsers versions. We show evolutions are inducing

side effects that can be explained and monitored, and demonstrate these evolutions can be anticipated via browsers nightly and beta versions weeks before being applied on release versions. In [Section 4.4](#), we leverage this dataset to design a rule-based browser fingerprints linking algorithm that is tailored for web authentication. In [Section 4.5](#), we evaluate it using 952,828 fingerprints collected from 64,235 browser instances. We measure our algorithm ability to correctly link a browser fingerprint to its predecessors that originate from the same browser instance. We observe our algorithm to be both relevant and reliable to serve on an authentication system using browser fingerprinting. We discuss our results and their limitations in [Section 4.6](#) and conclude in [Section 4.7](#).

4.1 Methodology

4.1.1 Controlled environment

4.1.1.1 Different layers

A fingerprint carries information about the device and browser installed by a user. Users can configure their device and browser according to their needs and preferences. This leads to a large pool of devices and associated configurations. The uniqueness property of the browser fingerprinting technique derives from this diversity. When previous work studied the uniqueness property of browser fingerprints [\[96, 117, 103\]](#), they relied on a dataset collected in the wild for their measurements. In this context, they can study the global uniqueness percentage of browser fingerprints of a given population, but cannot explain the semantic of fingerprints and the causes responsible for this fingerprints diversity. Similarly, studies covering browser fingerprints stability or building linking tools observed and measured changes on fingerprints collected in the wild [\[150, 119\]](#). These studies can evaluate linking tools, but cannot explain the reasons behind browser fingerprints evolution. We propose to study several elements—namely *layers*—that are responsible for this diversity. We believe this can help understand the differences between two browser fingerprints and guide the selection of attributes to reinforce authentication.

Hardware. Several studies already mentioned the link between hardware and browser fingerprints [\[117, 93\]](#). Two fingerprints can be different because the hardware of the devices on which the fingerprints were collected are different. More specifically, we believe the canvas, audio, and WebGL attributes are CPU and GPU-sensitive. Thus, we wanted to collect fingerprints on devices with different hardware for our experiments.

OS. An operating system manages the hardware and software resources of a device. Using one OS rather than another could impact the way resources are managed, leading to a different behavior when using browser APIs. This could be materialized by different

values for the same attributes. We collected fingerprints on both desktop—WINDOWS, MACOS and LINUX—and mobile—ANDROID and IOS—OS.

Browser. As browsers can have different API supports, we naturally checked several browsers. We first built our list of browsers by including the browsers with more than 1% browser market share in January 2021,¹ namely FIREFOX, CHROME, EDGE, SAFARI, OPERA, and SAMSUNG BROWSER. We also added BRAVE, which has specific anti-fingerprinting defenses [17].

User configuration. Users can alter their browser fingerprints. First, they can configure their browser and device with their preferred language, timezone, and specific permissions. They can also use browser defenses to hide values or change them to reduce the identification surface [36, 17]. For these reasons, we evaluated the browser fingerprinting defense of FIREFOX and BRAVE. We also had a look at browser and device configurations and their impact on browser fingerprinting.

4.1.2 Browser versions

As we mentioned in Section 2.3.2, fingerprints evolve over time. This cannot be considered as a layer of a fingerprint as the change is temporal. It is due to an evolution over time rather than a difference in the environment. However, it is a key element to take into consideration, as browsers are updated every 4 to 6 weeks [25, 38]. Changes browsers introduce during updates must be reflected in the fingerprint. In this context, we studied different versions in order to rely on stable attributes to link fingerprints. We targeted FIREFOX and CHROME for this experiment as they are the only browsers with easily available versions history and packages.

4.1.3 Attributes

We considered the attributes from the state of the art for our experiment. Thus, we considered the navigator and screen properties [96, 117, 125, 123], the fonts enumeration via `span`'s width and height measurement [125], canvas [122], audio [97] and WebGL rendering [93], WebGL parameters [117] and WebRTC [97].

We also collected several additional attributes, because we believe they can be relevant for fingerprinting. To the best of our knowledge, these attributes are not reported in the literature. We believed they could have different values on different contexts and devices, leading to an identification and consistency gain for the browser fingerprinting technique. The additional attributes we collected are:

¹<https://gs.statcounter.com/browser-market-share>

- **audio & video formats.** Browsers support `audio` and `video` formats to display multimedia content to users. We believed different browsers could support different formats. A format support test is performed via the `HTMLAudioElement` and `HTMLVideoElement` `JAVASCRIPT` APIs. We tested 21 audio and 8 video formats.
- **permissions.** This `JAVASCRIPT` API helps websites to know if it can use a specific API. It allows users to control the access to some sensitive APIs, such as the `geolocation`, `notification` or `clipboard` APIs. The API can return 3 possible values: i) `granted` which indicates the API is accessible, ii) `prompt` meaning a usage of this API is subject to user validation, and iii) `denied` which indicates the API is not accessible. We tested 15 permission names.
- **audio parameters.** Besides audio data [97], we collected various audio parameters via the `AudioContext` and `AnalyserNode` APIs. We believe these audio parameters could differ from one device or OS to another.
- **additional WebGL parameters.** Laperdrix *et al.* [117] collected the WebGL vendor and renderer. We collected additional WebGL parameters, such as the extensions available via the `WebGLRenderingContext`, as we think they could be different between two graphic cards or two OS.
- **navigator & window properties.** We collected the list of properties of the `navigator` and `window` objects. As they carry the APIs available in the browser, we believe they can differ from one browser to another.

4.1.4 Data collection









Desktop automation. On desktop, we setup an infrastructure to automate our data collection. We first develop a bash script to collect the ground truth of the device, including the CPU, GPU information, the OS name and version, the device brand and model. We then developed other bash scripts to perform browser download, installation, and start without user interaction. Each information of the ground truth is formatted into a URL parameter and concatenated at the end of the URL of our server we start our browsers on. Once the browser has loaded the page of our server, a script automatically collects the associated fingerprint, parses the URL parameters containing the information of the device and sends the fingerprint and the device information to the server that stores them in a database. While the browser is technically launched without user interaction, it runs the binary of the browser as if it was started by the user. Thus, our browsers are the vanilla ones and are not headless browsers, which have been studied to have a different fingerprint [111].

Environment. The desktop environments on which we ran our experiments are presented in Table 4.1. We ran the majority of our experiments on 2 desktop devices running Ubuntu 20.04. We completed this setup by collecting data on CHROME, FIREFOX and

Table 4.1: Recapitulative table concerning the desktop devices, OS and browsers on which we run our experiments

Device	CPU & GPU	OS	Browser		
			Name	Version	Configuration
Dell Latitude E6510	Intel Core i7 CPU M 640 Intel HD Graphics	Ubuntu 20.04	Chrome	rel: 76 to 90 beta: 80 to 90	Default + custom
			Firefox	rel: 29 to 88 b/n: 74 to 83	Default + custom + FP flag
HP-ZBook-15u-G2	Intel Core i7-5500U Intel HD Graphics 5500		Opera	74	Default
			Edge	90	Default
			Brave	89	Default + FP blocking
MacBook Air	Intel Core 2 Duo L9600 NVIDIA GeForce 320M	MacOS 10.13.6	Chrome	90	Default + custom
			Firefox	88	Default + custom
			Safari	13	Default
Acer Aspire V Nitro	Intel Core i7-6700HQ Intel HD Graphics 53	Windows 10	Chrome	90	Default + custom
			Firefox	88	Default + custom

Table 4.2: BrowserStack devices with OS, OS version and browsers used for the mobile dataset.

OS	Brand	Model	OS ver.	Brw.
	Samsung	Galaxy S21	11	  
		Galaxy S20	10	
		Galaxy S20+	10	
		Galaxy S20 Ultra	10	
		A51	10	
		Note 20	10	
		Note 20 Ultra	10	
		A11	10	
	Google	Pixel 5	11	 
		Pixel 4	10-11	
		Pixel 4L	10	
	OnePlus	OnePlus 8	10	
	Xiaomi	Redmi Note 8	9	
	Huawei	P30	9	
iOS	Apple	iPhone 12 Mini	14	 
		iPhone 12 Pro max	14	
		iPhone 12 Pro	14	
		iPhone 12	14	
		iPhone 11	13-14	
		iPhone 8	13	
		iPhone XS	12-13-14	
		iPad Pro 12.9 2020	12-13-14	
		iPad Pro 11 2020	13	

SAFARI on MACOS and WINDOWS. On FIREFOX and CHROME, we ran additional configuration tests including changing the languages and the `Do Not Track` parameter.

We also ran our experiments on mobile devices using the BROWSERSTACK online infrastructure.² The complete set of devices we used is available in Table 4.2. We collected a total of 1,160 fingerprints from 4 desktop and 23 mobile devices. The following sections aim at studying this dataset to build a browser fingerprints linking algorithm.

4.2 Causes of fingerprints diversity

In this section, we present the results observed concerning the diversity observed on our fingerprints dataset. Rather than being explicit about all the changes we observed, we aimed at explaining the causes of fingerprints diversity. We believe this will help us understand the semantic of the browser fingerprints to better build our linking algorithm.

We measured many differences between desktop and mobile fingerprints. The changes observed concern APIs that are not supported on both environments—represented by the `window` and `navigator` properties list—or APIs that return different values—`maxTouchPoint`, `plugins` or `mimeTypes`. For these reasons, we decided to split this evaluation in two parts: the desktop and mobile evaluations.

4.2.1 Desktop evaluation

We first present our observations about the fingerprints diversity in our desktop dataset. We collect a total of 121 fingerprints on the desktop devices of our controlled dataset.

4.2.1.1 Hardware

The diversity of attribute values due to hardware either concerns rendering attributes or attributes giving information about the capabilities of the machine. The rendering attributes include the audio, canvas and WebGL rendering, as their names state. In the second category, we observe the `hardwareConcurrency` navigator property, the screen sizes, the WebGL renderer, vendor and parameters and the audio parameters. The WebGL renderer and vendor attributes were not only impacted by a change in the hardware, but also when changing from a browser to another. On Chromium-based browsers, the WebGL vendor always starts with `Google Inc.` whether the graphic card was an NVIDIA or an INTEL GRAPHICS. Similarly, the WebGL vendor is either `Google SwiftShader` or starts with `ANGLE`. SWIFTSHADER is an implementation of the Vulkan API [73], while ANGLE is an engine to convert OpenGL graphic calls to one of the API supported by the current platform. These values illustrate the importance of the software layers above the hardware for graphic rendering.

²<https://www.browserstack.com/>

More generally, we did not observe any attribute whose value is only impacted by a difference in the hardware. Among the ones impacted by the hardware, all of them depend on at least another layer which is often the browser. For example, while a different GPU will be directly reflected in the WebGL renderer attribute, a change of browser will modify the backend rendering system and modify the content of the renderer string. We believe this is an interesting challenge for browser fingerprinting to find attributes that identify the hardware and are not impacted by any other device component.

4.2.1.2 OS

In our dataset, we observe the OS layer being responsible for different values on several aspects. First, it changes the attributes giving explicit information on the OS of the device, represented in our dataset by 4 properties of the `navigator` object: `appVersion`, `platform`, `userAgent` and `oscpu`—the last one being deprecated and only available on FIREFOX.

A different OS also provides different APIs in the browsers of our dataset. This change can be measured via the properties included in the `window` object. On the devices running UBUNTU, FIREFOX was missing the VR-related events. These APIs were implemented on version 55 on WINDOWS and 64 for MACOS, but are now deprecated [61] and replaced by the `WebXR` API. Until this API is removed, it represents another fingerprint difference caused by a difference in the OS of the device. Similarly, Chrome supports different APIs depending on the OS:

- on WINDOWS, we found the `BluetoothUUID` property and the APIs to easily manipulate audio data such as `AudioDecoder`, `AudioEncoder`, `AudioFrame` and `EncodedAudioChunk`,
- on MACOS, we only detected the `BluetoothUUID` property,
- we did not detect any of these properties and APIs on LINUX.

This illustrates the differences in API support on different OSs and shows how an algorithm can leverage these differences to easily distinguish OSs and better link fingerprints with one another.

4.2.1.3 Browser

Browser fingerprinting relies on the APIs provided by browsers to collect information. Thus, it is not surprising to observe a large number of changes from one browser to another. We distinguish 3 major causes responsible for a fingerprint to change between 2 different browsers: i) a fingerprinting defense altering the collected value, ii) implementation differences, and iii) differences in APIs support.

Fingerprinting defenses. As explained in [Section 2.5](#), fingerprinting defenses alter the attribute value to make the user less identifiable, either by reducing the uniqueness

of a value or by breaking its stability over time. In this paragraph, we study the defenses deployed by browsers by default. BRAVE developed several countermeasures to its millions of users based on randomization [15]. By default, BRAVE randomizes the content of the `plugins`, `mimeType`s, and `hardwareConcurrency` properties and the result of canvas and WebGL rendering by flipping few dozens of pixel to a different value. These changes are not visible as the generated pixels are extremely similar to pixels before the changes. Only a single color of each pixel is changed by mitigating its float representation with an offset of ± 0.4 .

FIREFOX and SAFARI also protects their user bases by not supporting the `getBattery` API. Researchers showed this API exposes a fingerprinting surface that can be leveraged to identify users in short periods of time [126, 127]. This can be used to track users, or for identifiers respawning. In consequence, the API has been revised and removed. Finally, SAFARI does not implement the `hardwareConcurrency` navigator property.

Implementation differences. These differences concern APIs that are supported by browsers, but the value has not been defined by an official standard, leading to unique implementations. By default, the `doNotTrack` property is set to `unspecified` on FIREFOX, `true` on SAFARI and is an empty string on Chromium-based browsers. The `productSub` property has been set to a fixed number for privacy reasons, but the value differs from FIREFOX (20100101) to other browsers (20030107).

Table 4.3: Number of plugins and unique plugins supported by browsers, on Ubuntu 20.04

Browser	# plugins	Unique plugin names
FIREFOX	0	
CHROME	3	
SAFARI	3	WebKit built-in PDF
OPERA	3	News feed handler
EDGE	3	Microsoft Edge PDF Plugin Microsoft Edge PDF Viewer
BRAVE	4	Brave PDF and PS plug-in

Similarly, the `plugins` property is implemented on all browsers, but returns an empty list on FIREFOX, and a list with different elements for other browsers. In fact, we observed the `plugins` property being unique on the 6 browsers we tested on Ubuntu 20.04. The values observed are presented in Table 4.3. We made a similar observation for the `mimeType`s. These attributes are no longer needed for non Chromium-based browsers, as Flash was removed from browsers [24]. However, they

will continue to help identify the user’s browser until their removal.

The audio rendering is only supported by default by FIREFOX and EDGE. For the other browsers we tested, the `AudioContext` API must be triggered with a user gesture and is not usable directly when the users lands on a page. This has been promoted to prevent autoplay and improve the user experience [20, 72].

Differences of APIs support. Several differences are due to the support or non-support of APIs. This is generally because the API is not standard, either because it is

deprecated or because it is still experimental. Among the deprecated APIs, FIREFOX implements the properties `top` and `left` on the `screen` object and the property `oscpu` on the `navigator` object.

The `Permission` API is still considered as experimental. SAFARI does not support it, FIREFOX supports a limited number of permissions—`geolocation`, `notifications`, `persistent-storage` and `push` among those we tested—and Chromium-based browsers support 7 additional permissions compared to FIREFOX.

Finally, we also observed differences due to API names. FIREFOX implements the `mozRTCPeerConnection` API, while CHROME implements the `webkitRTCPeerConnection` API. Both of them were experimental APIs for the `RTCPeerConnection` API, which is now supported by both browsers and an official recommendation. Browser vendors often prefix experimental APIs to avoid a compatibility issue during the standardization process [60]. However, these differences can easily be used to check the real browser name without relying on easily-spoofable attributes.

4.2.1.4 User configuration

While the diversity of fingerprints is mainly caused by the natural diversity of hardware, OS and browser combination, it is increased by the user's configuration. Fingerprints can be changed by the usage of a fingerprinting defense or a change in the browser or device configuration.

Defenses. Browsers have developed defenses to protect against fingerprinting. We already presented the default defenses of FIREFOX, BRAVE and SAFARI, but FIREFOX and BRAVE propose additional defenses. On FIREFOX, users can activate the `resistFingerprinting` flag to unify several attributes and block access to others [36]. It randomizes the canvas and WebGL rendering output, sets the number of media devices, `hardwareConcurrency`, `User-Agent`, WebGL parameters to fixed values, and prevents to access to the WebGL vendor and renderer. The fingerprinting defense of BRAVE has 3 possible levels:

- the `default` level uses the defenses presented earlier in this section,
- the `disabled` one removes the protection of the `default` level,
- the `strict` mode goes further than the `default` level. It gives back an empty value for the WebGL rendering, does not support WebGL extensions, renderer and vendor and fixes all WebGL precisions to 0.

Configuration. We also observed device configuration to have an impact on the browser fingerprint. When first installed, the browser gets the languages and the timezone of the device. When the timezone is updated on the device, it is automatically updated in the browser. Oppositely, the languages in the browser are not updated when they are changed

on the device. Finally, configuring the browser may also change the browser fingerprint. Users can update their preferred languages, `doNotTrack` setting and permissions. These changes are reflected on the corresponding attributes.

Context. Finally, the user context can change the fingerprint. When a user plugs her device to a screen, the fingerprint of the browser can be changed. As more and more external devices can be connected to a device, such as a gamepad, a VR headset or screens, APIs developed to support these devices will continue to add entropy to browser fingerprints [23].

4.2.2 Mobile evaluation

We proceed to present our results concerning our mobile dataset.

4.2.2.1 OS and browsers

We observe the OS has a major influence on browser fingerprints on our mobile dataset. We first aim at studying the differences between devices running on `ANDROID` and `iOS`, no matter which browser is running on them. Besides the `appVersion` and `userAgent`, we detected 8 techniques to easily distinguish the mobile OS of the devices in our dataset. The results are presented on Table 4.4.

iOS. Almost all attribute values are unified between `CHROME` and `SAFARI` on `iOS`. Apart from the `appVersion` and `userAgent` attributes, we did not identify a single attribute that could distinguish between `CHROME` and `SAFARI`. This behaviour was expected, as all browsers on `iOS` must use the `WEBKIT` rendering engine [7].

Browser fingerprints also change when the `iOS` version differs. Among others, the `maxTouchPoints` property was not supported on `iOS 12` and returns 5 on `iOS 13` and 14. `iOS 14` also supports 4 additional `WebGL` extensions compared to `iOS 12` and 13. Some of these changes may be directly linked with an upgrade in hardware components between models where

Table 4.4: Attributes that are different between `ANDROID` and `iOS`, and corresponding value(s). Each value has been observed for all browsers tested on the specified OS.

Attribute	ANDROID	iOS
Audio & video formats	audio/3gpp audio/3gpp2 audio/basic video/webgm	audio/ogg audio/webm video/mp4
Audio params	Supported	Not supported
nav.platform	Linux armv8l Linux aarch64	iPhone MacIntel iPad
nav.vendor	Google Inc. <empty value>	Apple Computer, Inc
permissions	Supported	Not supported
colorDepth pixelDepth	24	32
webGL renderer	Adreno(TM)506 Mali-G76,....	Apple GPU Apple A12 GPU Apple A12X GPU
webGL vendor	ARM Qualcomm	Apple Inc.

more powerful chips offer more modern features. In the end, these results illustrate additional functionalities brought by a newer OS version are reflected in the browser fingerprint, increasing the fingerprints diversity.

Android. Oppositely to iOS, we detected many attributes being different between our 3 mobile browsers on ANDROID. We measured an additional change than the ones observed on the desktop environment between FIREFOX and CHROME. FIREFOX does not activate the **anti-aliasing** WebGL parameter. This information can be observed in both the WebGL parameters itself and on the WebGL rendering attribute value. This property is activated by default on all the desktop browsers, and all the other mobile browsers we tested. Additionally, when we activate this property in the `WebGLRenderingContext` API, the value returned by the WebGL rendering attribute is equal to the value given by default by CHROME and SAMSUNG BROWSER on the same device. We make 2 observations from this result: i) configured similarly, the WebGL rendering attribute value is cross-browser, and ii) this implementation difference can easily identify a mobile FIREFOX instance.

4.2.2.2 Hardware

The attributes being impacted by a change in the hardware were the same as the one presented for the desktop environment, except from the `hardwareConcurrency`, which does not vary according to the hardware of the mobile device. We observed 25 canvas distinct values and 15 WebGL values among the 23 devices we tested. While the WebGL is always cross-browser—except for FIREFOX because of the **anti-aliasing** parameter we described earlier, the canvas often changes from a browser to another leading to more diversity from the canvas than WebGL. This result might be linked to the particular sets of instructions of the canvas and WebGL rendering attributes of our experiment. Other sets of instructions could trigger different drawing components, leading to different results.

4.2.3 Layers responsible for an attribute change

We summarized our findings about the *layer(s)* responsible for an attribute change. [Table 4.5](#) represents our results to measure the layer(s) impacting the value of an attribute, or differently formulated, it represents the layers *identified* by an attribute. ✓ means the value is impacted by this layer on both desktop and mobile devices. 📱 and 💻 specifies the value changes only on mobile and desktop devices, respectively. While we discuss 4 layers—namely hardware, OS, browser and user configuration—in [Section 4.1.1.1](#), we added the *Other* layer for the font attribute, which has been observed to vary when

installing new software on the desktop device. Concerning the *user configuration* layer, we made the distinction between the 3 major events that can cause a change in an attribute value: i) browser/device configuration, ii) context, or iii) a fingerprinting defense. The configuration and context increase the diversity of the fingerprint because of the number of possible values it offers. Oppositely, a fingerprint defense changes the value of an attribute, generally by setting the attribute to a defined value, as we explained in [Section 4.2.1.4](#). This leads to a decrease of the possible values for the attributes that can be impacted by a fingerprinting defense. Then, we consider an attribute identifies the user configuration only if the attribute identifies the browser/device configuration or the context.

The more layers an attribute identifies, the more interesting the attribute is for an authentication system with browser fingerprinting because it helps the system relying on diverse attributes to have a fingerprint as unique as possible. In this context, we define $L(a)$ as a function taking an attribute as a parameter, and returning the number of layers it identifies. In addition, we define $L_m(a)$ and $L_d(a)$ being two derivative functions that only consider the mobile and desktop results, respectively. [Table 4.5](#) also presents these functions when given each browser fingerprinting attribute.

SYNTHESIS. In this section, we treat **RQ3** by understanding how the different elements—hardware, OS, browser and configuration—of a device were responsible for the fingerprints diversity. We observed browser fingerprinting to be an engineering side effect, caused by various factors: differences of APIs support or implementation, different default values or configuration changes. Understanding the causes of the diversity will help us rely on attributes that identify several components of the device for our browser fingerprints linking algorithm.

4.3 Fingerprints evolution through browser versions

As browser fingerprints evolve over time, stability is a key aspect to consider when using browser fingerprinting in an authentication system. In this section, we proceed to study the evolution of browser fingerprints. We continue the work started by Li *et al.* [119] concerning the causes of the evolution of a fingerprint. Our controlled environment allows us to be more accurate, as we control the software and hardware layers of our devices. In this context, we observe the changes brought by natural updates of FIREFOX and CHROME. In order to better understand where these changes come from, we also look into the FIREFOX and CHROME bug trackers to find resources related to the fingerprint changes we observed.

Table 4.5: Layer(s) impacting the value of an attribute, and category of our attributes.

	Hw	OS	Bw	Oth	User config.			$C(a)$	$L_d(a)$	$L_m(a)$	$L(a)$
					cfg	ctx	def				
canvas	✓	✓	✓				✓	<i>stable</i>	3	3	3
webGL data	✓	✓	✓				✓	<i>stable</i>	3	3	3
webGL parameters	✓	✓	✓				✓	<i>stable</i>	3	3	3
webGL vendor	✓	✓	✓				✓	<i>stable</i>	3	3	3
webGL renderer	✓	✓	☐				✓	<i>stable</i>	3	2	3
nav.maxTouchPoint	✓	☐						<i>stable</i>	1	2	2
screen.colorDepth	✓	☐						<i>stable</i>	1	2	2
screen.pixelDepth	✓	☐						<i>stable</i>	1	2	2
audioData	✓		✓					<i>stable</i>	2	2	2
screen.height	✓		☐			✓	✓	<i>volat.</i>	2	3	3
screen.width	✓		☐			✓	✓	<i>volat.</i>	2	3	3
mediaDevices	✓		☐				✓	<i>stable</i>	2	1	2
font		✓	☐	☐			✓	<i>stable</i>	3	1	3
nav.appVersion		✓	✓				✓	<i>evolv.</i>	2	2	2
nav.userAgent		✓	✓				✓	<i>evolv.</i>	2	2	2
window.properties		✓	✓				✓	<i>evolv.</i>	2	2	2
audio parameters		✓	✓					<i>stable</i>	2	2	2
nav.platform		✓	✓					<i>stable</i>	2	2	2
nav properties		✓	✓					<i>evolv.</i>	2	2	2
nav.hardwareConc		✓	☐					<i>stable</i>	2	1	2
permissions		☐	✓		✓			<i>volat.</i>	2	3	3
audioFormats		☐	✓					<i>stable</i>	1	2	2
navigator.vendor		☐	✓					<i>stable</i>	1	2	2
videoFormats		☐	✓					<i>stable</i>	1	2	2
nav.oscpu		☐	✓					<i>stable</i>	2	1	2
nav.mimeTypes		☐	☐					<i>stable</i>	2	0	2
screen.left			✓			✓	✓	<i>volat.</i>	2	2	2
screen.top			✓			✓	✓	<i>volat.</i>	2	2	2
battery			✓					<i>stable</i>	1	1	1
nav.buildID			✓					<i>stable</i>	1	1	1
nav.productSub			✓					<i>stable</i>	1	1	1
screen.availHeight			☐			✓	✓	<i>volat.</i>	1	2	2
screen.availLeft			☐			✓	✓	<i>volat.</i>	1	2	2
screen.availTop			☐			✓	✓	<i>volat.</i>	1	2	2
screen.availWidth			☐			✓	✓	<i>volat.</i>	1	2	2
nav.doNotTrack			☐		✓		✓	<i>volat.</i>	2	1	2
nav.deviceMemory			☐					<i>stable</i>	1	0	1
plugins			☐					<i>stable</i>	1	0	1
timezone					✓	✓	✓	<i>volat.</i>	1	1	1
nav.language					✓		✓	<i>volat.</i>	1	1	1
nav.languages					✓		✓	<i>volat.</i>	1	1	1

4.3.1 Release versions

We first study the evolution between release versions. As explained in [Section 4.1](#), we study FIREFOX between versions 29 and 88 and CHROME between versions 76 and 90. All changes in the collected fingerprints were observed on the release of a major version, except for one. The only exception is a bug disabling WebGL on FIREFOX 68.0 on our 2 devices running UBUNTU. It was quickly reported and fixed in version 68.0.2 [\[49\]](#). As stated by the FIREFOX release policies, major versions contain additional features while minor versions only fix issues and bugs [\[35\]](#). This accentuates the statement that browser fingerprinting is coming from an engineering side effect rather than from issues and bugs.

Rather than grouping the changes by major version number, we group them by root cause. We define 4 root causes that can be responsible for a browser fingerprint evolution on a release version: i) the support of a new API, ii) the removal of a deprecated API, iii) a privacy change, mainly to decrease the entropy of an attribute value, and iv) an evolution in the implementation.

We do not aim at being exhaustive on every change and focus on specific ones that we deem to be the most interesting. At the end of the section, we provide a more global overview of the observed changes.

New API support. We observed that the number of supported WebGL extensions increases from 18 to 25 extensions on FIREFOX and from 23 to 33 on CHROME. This illustrates the addition of new features to allow more complex WebGL rendering techniques. We also observed that FIREFOX added support for the following audio and video formats:

- audio/flac and audio/ogg; codecs="flac" on version 51 [\[42\]](#),
- audio/mpeg on version 71 [\[40\]](#),
- audio/aac, audio/mp4; codecs="mp4a.40.2" and video/mp4; codecs="flac" on version 86.

Finally, CHROME and FIREFOX added properties in the `navigator` object and APIs in the `window` object in almost every new release versions. For example, the `getBattery` API was added on FIREFOX on version 43.

Remove deprecated/non-standard APIs. We also measured the removal of deprecated properties on the `navigator` and `window` objects. For example, we observed the `navigator.battery` API being removed on version 50 because it was no longer needed as the new API `getBattery` was available since version 43 [\[44\]](#). The `registerContentHandler` was removed from the `navigator` object on version 62, because it was non-standard and FIREFOX was the only browser implementing it [\[45\]](#).

Privacy change. We explained in [Section 4.2.1.3](#) that FIREFOX does not support the `getBattery` API to protect its users. We detected that this change happened in version 52 [\[46\]](#). Additionally, we observed the `navigator.buildID` attribute to be related to the major version from version 29 to 63. From version 64, it has been set to a fixed value for all versions to decrease its entropy [\[41\]](#).

Implementation evolution. Finally, changes in the rendering elements—audio, fonts, canvas—are more difficult to understand and we decided to group them under a generic *implementation evolution* category.

On FIREFOX, we observed 4 versions changing the list of fonts: versions 34, 43, 44, and 45. 3 of these versions—34, 43 and 45—considered 3 fonts being added and/or removed (`Arial Narrow`, `KacstDecorative` and `Padmaa`). On version 44, 14 fonts were added while 4 were removed. Several fonts added or removed seem to be related to emoji rendering, such as `Apple Color Emoji` or `Segoe UI Emoji`, but we did not find any resource about these changes to confirm our thoughts.

The canvas value changed 9 times. For 7 of them, the rendered text changed. We assume it is a font update or upgrade, but we were not able to find any resource related to these changes to verify this hypothesis. For the last 2, the rendered emoji presented differences because of an emoji font change. First, on version 59, the emoji changes from a non-colored to a colored emoji. A colored emoji font—namely `EMOJIONE`—was present on FIREFOX since the version 50 [\[48\]](#) but the fonts used to render text—in our case, `DEJAVU` and `ARIAL`—had the rendering priority and rendered a non-colored emoji until version 59. From that point, FIREFOX defined a list of fonts called in priority to render emojis [\[39\]](#). The second change occurs when updating to version 61. The `EMOJIONE` font had a new version with a license being non-compatible with MOZILLA’s standards [\[47\]](#). While several solutions were proposed, FIREFOX developers decided to replace the emoji font by `TWEMOJI`. This change illustrates browser fingerprinting can even be impacted by a license change on a web component.

General observations. We observed a total number of 212 attribute values changes on 59 version updates of FIREFOX and 58 attributes changes on 15 version updates of CHROME. We present the number of changes per version in [Figure 4.1](#). The number of changes is quite stable, mostly varying from 1 to 5 since version 64, out of a total of 56 attributes. When building our browser fingerprints linking algorithm, it shows we can expect to be able to link 2 browser fingerprints up to 2 following versions. The only exception is version 85 on CHROME on which we observed 9 changes compared to version 84. It is mainly related to a huge change on WebGL that impacted the 4 WebGL attributes. While we expect this kind of change to be occasional, it shows we cannot fully consider an attribute as immutable, as the WebGL renderer and WebGL vendor changed in this specific case.

Figure 4.1: Number of attribute value changes per version grouped by browser out of a total of 56 attributes in our experiments.

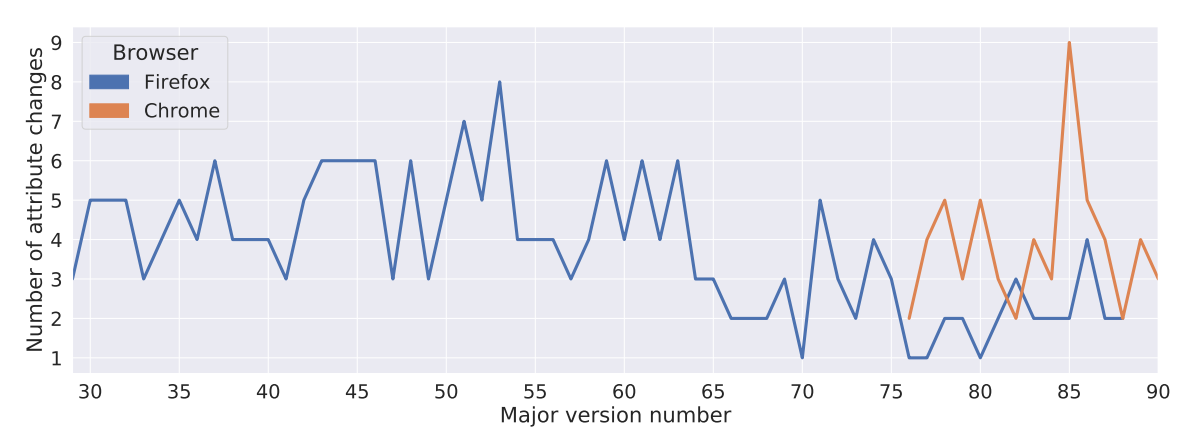
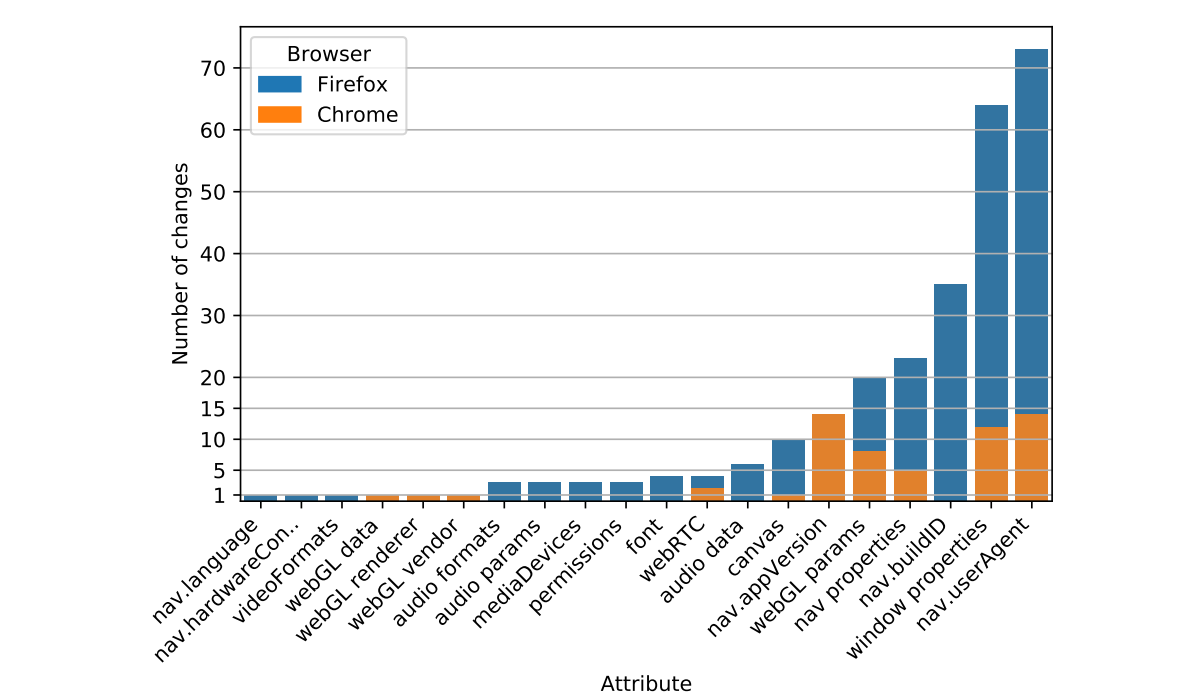


Figure 4.2: Number of changes per attribute per browser. Only attributes that have changed at least once during a version update are shown on this graph.



We continue our observations by measuring the attributes changing the most among the browser versions. The results are presented in [Figure 4.2](#). The most changing ones are the `navigator.userAgent` for both browsers and `navigator.appVersion` for CHROME as they are version-related. The `buildID` attribute was also version-related on FIREFOX, but is now statically defined, and we do not expect any further changes on this attribute. While these attributes update often, their values are highly predictable and can easily be managed on a browser fingerprints linking algorithm. We also observed that the `navigator` and `window` property lists are changing quite often. As we explained earlier, this is because these 2 objects carry the available properties and APIs of the browsers, which are constantly subject to changes. The rest of the attributes we studied either rarely change or are never modified. These results show how often we can expect changes on attributes and how we can rely on stable attributes to build our browser fingerprints linking algorithm.

4.3.2 Nightly/beta versions

Besides our experiments on the evolution among browser release versions, we also ran nightly and beta versions to better understand when the changes appear in the browser, and to study the possibility to anticipate them. We ran all nightly and beta versions on FIREFOX between release versions 74.0 to 83.0 and all beta versions of CHROME between release versions 80 and 90. The nightly build is a FIREFOX pre-release version packaged twice a day for testing purposes. The beta version of both FIREFOX and CHROME are pre-release versions that aim at providing a stable base for future releases. Generally, only bug fixes are allowed on beta versions. On both CHROME and FIREFOX, we observed 2 categories of changes:

- Changes observed on a given release version can often be seen from the very first corresponding beta version. For example, the error message given by FIREFOX when querying for a non-supported permission changed on release version 75. The first beta version corresponding to this release already included this change;
- The other changes are not directly included on the incoming release version but on a near-future release version. For example, we first observed the window properties `originAgentCluster` and `NavigatorUAData` on CHROME beta version 88.0.4324.27, but they were only included on the release version 90.

We made 2 observations from these results: i) Changes can be observed before release versions, when they are engineering contributions that need to be tested and eventually shipped to a fraction of the users before being widely deployed. Similarly to our other results, it shows browser fingerprinting is an engineering side-effect; ii) Weeks or even months can pass between a change made on a nightly or beta version and its appearance in a release version. Browser fingerprints linking algorithm developers could easily manage

nightly and beta versions to observe future incoming changes on release versions and adapt their algorithm for their users.

4.3.3 Categorizing attributes

Based on our results, we classified the stability property of browser fingerprinting attributes into 3 categories:

- We re-used our results in [Section 4.2](#) to measure the attributes identifying the user configuration, namely the `screen sizes`, `permissions`, `navigator.doNotTrack`, `timezone` and `navigator.language(s)`. By essence, these attributes are very unstable: they can change anytime and it is impossible to anticipate their values. We defined these attributes as *volatile*;
- We observed that the `nav.userAgent`, `nav.appVersion`, `navigator properties` and `window properties` attributes change very often, especially over the last release versions of the studied browsers. We defined these attributes as *evolving*;
- We showed that the remaining of the browser fingerprinting attributes never or rarely change during browser updates. We define these attributes as *stable*.

Finally, we defined $C(a)$ as the function to retrieve the category of an attribute. [Table 4.5](#) also presents the category of each of our attributes.

SYNTHESIS. In this section, we answered **RQ4** by measuring the evolution of browser fingerprints through browser versions. We showed attributes value changes are engineering side-effects and concern only a small proportion of attributes, meaning a large number of attributes are stable. We also demonstrated browser nightly and beta versions can be used to predict and anticipate future changes in browser release versions.

4.4 A browser fingerprints linking algorithm

As we explained in [Section 2.7](#), existing techniques to link browser fingerprints are not satisfying for web authentication. In this section, we build on our previous results on the causes of fingerprint diversity and evolutions among browser versions to design a tailored linking algorithm.

4.4.1 Main goal

In an authentication system, browser fingerprinting acts as an additional layer of protection where the system verifies that the device being used for the authentication attempt has been seen before. In this context, the goal of a linking algorithm is to match the fingerprint submitted on the authentication form with one on the fingerprints already registered by the user in a previous session. If the new fingerprint is close enough to a

Algorithm 1 Fingerprints linking algorithm

```

1: function LINK(submittedFP, registeredFPs, threshold, W)
2:   scores  $\leftarrow \langle \rangle$ 
3:   for each registeredFP  $\in$  registeredFPs do
4:     score  $\leftarrow 0$ 
5:     totalWeight  $\leftarrow 0$ 
6:     attrNames  $\leftarrow$  GETATTRS(submittedFP)
7:     for each attrName  $\in$  attrNames do
8:       submittedAttrValue  $\leftarrow$  GETVALUE(attrName, submittedFP)
9:       registeredAttrValue  $\leftarrow$  GETVALUE(attrName, registeredFP)
10:      scoreAttr  $\leftarrow$  COMPUTESCORE(submittedAttrValue, registeredAttrValue)
11:      attrWeight  $\leftarrow$  W(attrName)
12:      score  $\leftarrow$  score + scoreAttr * attrWeight
13:      totalWeight  $\leftarrow$  totalWeight + attrWeight
14:    end for
15:    score  $\leftarrow$  score / totalWeight
16:    scores  $\leftarrow$  scores  $\cup$   $\langle$  registeredFP, score  $\rangle$ 
17:  end for
18:  registeredFP, maxScore  $\leftarrow$  MAX(scores)
19:  if maxScore  $\geq$  threshold then
20:    return registeredFP
21:  else
22:    return false
23:  end if
24: end function

```

previous one with changes that are in an acceptable envelope, the identity check will pass and the user will be granted access. Otherwise, the check will lead to a failure.

An authentication system has strong constraints on the execution speed. The linking algorithm must provide an answer in few milliseconds not to block or slow down the authentication attempt. As the machine-learning algorithms proposed in the state of the art scale badly, we believe a rule-based algorithm is more adapted to this use case. The general idea is to get a similarity score between the submitted fingerprint and each of the registered ones, keeping the highest. If the score is higher than a defined threshold, the user is then authenticated.

4.4.2 Design

We present our algorithm in [Algorithm 1](#). It takes 4 different inputs:

- The *submitted fingerprint* is the fingerprint of the current authentication attempt that has just been collected,
- The *registered fingerprints* are the fingerprints already stored. The algorithm will try to link the submitted fingerprint to one of the registered fingerprints,
- The *threshold* is the minimum score to reach to validate the link between the submitted fingerprint and one of the registered fingerprints,

- The *weight computing function*, named W , which—given an attribute—returns its weight corresponding to the global importance of the attribute when computing the score.

We first take each registered fingerprint (line 3). For each attribute (line 7), the *score computing function* (line 10) computes a score reflecting the similarity between the attribute value of the submitted fingerprint and the attribute value of the registered fingerprint. The algorithm collects the weight of this attribute (line 11). Given this information, it computes a weighted average of the scores of all the attributes (line 12 – 15). The algorithm stores all the similarity scores, keeps the highest one (line 18), and checks if it is higher than the *threshold* (line 19 – 23).

In our algorithm, the *score computing function* computes a score between 0 and 1, reflecting the similarity between the 2 attribute values. 0 means the values have nothing in common, while 1 means the values are identical. We used different score computing function according to the type of the attribute value. i) For primitives type values—string and number values, the function is a simple equality test that returns 0 if the values are different and 1 if the values are equal, ii) We used the Jaccard index to compute the score for arrays attributes, namely `navigator.languages`, `navigator properties` and `window properties`), iii) For attributes having key-values tuples—`audio parameters`, `audio and video formats`, `fonts`, `mediaDevices`, `permissions`, `webGLParameters`, `navigator.mimeTypes` and `plugins`, we computed the Jaccard index on the tuples.

4.4.3 Parameters

The first parameter, the *submitted fingerprint*, is naturally obtained by the system. The 3 others can be customized to optimize the algorithm.

Registered fingerprints The registered fingerprints represent the fingerprints that are trusted as belonging to the user. With one of them, the user can authenticate.

Weights. Each attribute has its own properties. Previously, we studied the causes of the diversity of our attributes among our controlled environment and the causes of the stability among browser version updates. In our algorithm, the *weight* of an attribute, noted $W(a)$ represents its global uniqueness and stability. The more unique and stable an attribute value is, the higher its weight will be. Considering the **uniqueness** of an attribute, noted $W_u(a)$, we use our results when measuring the number of layers an attribute identifies and our functions $L(a)$, $L_d(a)$ and $L_m(a)$ defined in [Section 4.2.3](#). Concerning the **stability**, we use our attributes stability classification made in [Section 4.3.3](#). For each category, we define a weight that represents the stability of the attributes in the category c , noted $W_s(c)$. Thus, the stability weight of an attribute can be obtained with $W_s(C(a))$, where $C(a)$ is the function we defined in [Section 4.3.3](#) that, given an attribute,

returns its category. As *volatile* attributes are less stable than *evolving* attributes, themselves being less stable than *stable* attributes, we set the following constraint: $W_s(\text{volatile}) < W_s(\text{evolving}) < W_s(\text{stable})$. The weight of an attribute is the product of the uniqueness weight and the stability weight, noted $W(a) = W_u(a) * W_s(C(a))$.

Threshold. It represents the tolerance of the algorithm. A low threshold allows more differences between fingerprints, which increases the chances for a user to authenticate after her fingerprint changes. However, it reduces the difficulty for an attacker to mimic a fingerprint, and be mistakenly allowed to authenticate. Oppositely, a high threshold reduces the risk for an attacker to authenticate, but also the chances for a user to do so.

SYNTHESIS. In this section, we used our knowledge on the causes of browser fingerprint diversity and evolution to design a rule-based browser fingerprint linking algorithm. We leveraged the attributes properties concerning stability and uniqueness to define weights that represent the importance of each attribute in the linking algorithm.

4.5 Evaluation of the linking algorithm

4.5.1 Datasets

We used the data collected by the AMIUNIQUE web extension, whose goal is to collect and study fingerprints for research purposes. The extension is available on both CHROME and FIREFOX. During the installation, the extension generates a browser instance unique identifier. Every 4 hours, the extension collects the browser fingerprint of the device and sends it to a server with the extension instance identifier. The identifier is used to link all the fingerprints coming from the same instance browser and will be used as a ground truth when evaluating our algorithm. From August 2020 to March 2021, we collected 952,828 fingerprints from 64,235 extension instances.

4.5.2 Key performance metrics

We ran our algorithm in 2 modes: a *safe* mode and an *attack* mode.

Safe mode. First, we considered a *safe* mode, where the only user that tries to authenticate is the rightful one. Then, the registered fingerprints are hers, and the challenge is to correctly link all the fingerprints coming from a single browser instance.

For this mode, we used several terms and metrics proposed by the state-of-the-art linking algorithm, which is FP-STALKER [150]. We renamed the term *tracking chain* as *identification chain*, which still corresponds to a list of fingerprints that have been linked together. Each identification chain is assigned an *ID*. In the case of a perfect

linking algorithm, each browser instance would have a unique chain, composed of all the fingerprints coming from that instance. The *identification duration* (named *tracking duration* in the FP-STALKER study) represents the period during which the linking algorithm correctly links the fingerprint to the correct identification chain. The *average identification duration* corresponds to the average identification duration of all the browser instances. The *number of assigned IDs* corresponds to the number of different identifiers assigned to a browser instance, which corresponds to the number of identification chain belonging to a browser instance. With a perfect linking algorithm, each browser instance should only be assigned one identifier. Then, we tried maximizing the *identification duration* and minimizing the *number of assigned IDs*. In this context, the list of registered fingerprints is formed by taking the last fingerprint of each identification chain. Our algorithm is used as follows. First, we sort the fingerprints F of the browser instance by increasing timestamps, and create one identification chain composed of one fingerprint, the very first fingerprint sent by the browser instance. Then, for each fingerprint f of F :

- We run our algorithm with f as the submitted fingerprint, our registered fingerprints list as being the list formed by collecting the last fingerprint of each identification chain, our weight sets and threshold (see below for parameters settings),
- If the algorithm links the fingerprint to one of the registered one, the fingerprint is added to the corresponding identification chain, at the end,
- If the algorithm does not link our fingerprint to any of the registered one, we create a new identification chain with one fingerprint, the submitted one.

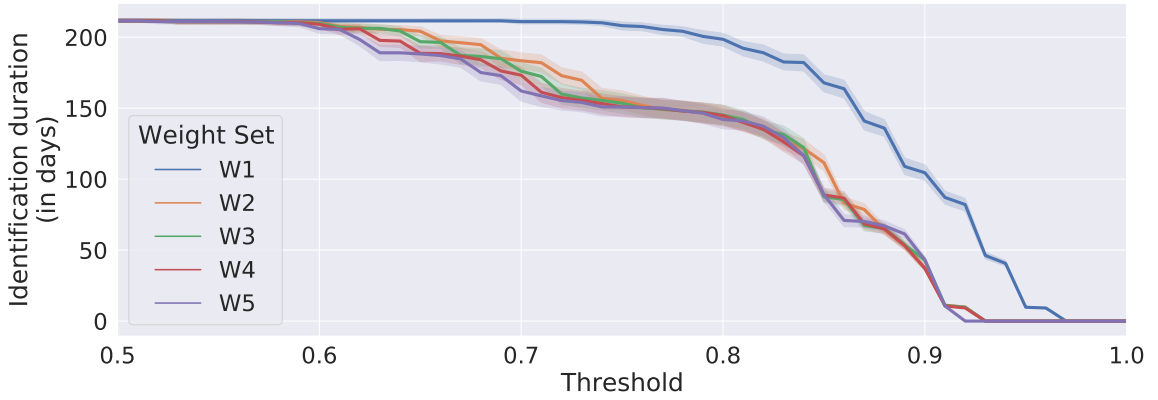
Attack mode. Our second mode considers the possibility for an attacker to mimic a fingerprint and submit it to authenticate. While our algorithm only takes as input the registered fingerprints for the user trying to authenticate, we also aim at measuring the behavior of our algorithm when provided fingerprints that do not belong to the same browser instance. In this context, we are interested in measuring whether our algorithm links two fingerprints coming from different browser instances.

We ran our algorithm with: i) the *submitted fingerprint* being each fingerprint of our dataset, ii) the *registered fingerprints* as an array of one element containing each fingerprint of our dataset which does not belong to the same browser instance. This operation will compute the equivalent of a Cartesian product, which will be time-consuming. We reduce the size of our dataset by splitting our dataset in days and by only using the data collected on the first day of each month. Here, our metric will be the proportion of *True Negative* (TN) and *False Positive* (FP). A False Positive is dangerous because it links a fingerprint to an instance it does not belong to. In the case of a perfect algorithm, the TN should represent 100% of the results, and the FP should represent 0% of the results. Thus, we aimed at finding the best set of parameters that minimizes the proportion of False Positives.

Table 4.6: Set of weights to be evaluated, and corresponding weights for the canvas attribute

$W_s(c)$	Category weight value			$W_u(a)$	$W(a)$	Canvas weight (stable cat.)		
	$W_s(volatile)$	$W_s(evolution)$	$W_s(stable)$			$W_s(c)$	$W_u(a)$	$W(a)$
$W_{s1}(c)$	1	1	1	1	W1	1	1	1
$W_{s2}(c)$	1	2	3	$L_d(a)$	W2	3	3	9
$W_{s3}(c)$	1	2	4	$L_d(a)$	W3	4	3	12
$W_{s4}(c)$	1	3	6	$L_d(a)$	W4	6	3	18
$W_{s5}(c)$	1	3	9	$L_d(a)$	W5	9	3	27

Figure 4.3: Browser instances identification duration according to the threshold and weights sets



4.5.3 Parameters values

As we mentioned in [Section 4.4.3](#), the weight of an attribute is defined by its uniqueness weight multiplied by its stability weight. As our dataset is only made of desktop devices, we defined the uniqueness weight of an attribute $W_u(a)$ as being equal to $L_d(a)$, which we defined in [Section 4.2.3](#) and corresponds to the number of layers an attribute identifies on a desktop browser. Concerning the stability weight, we followed our constraint concerning the categories weight defined earlier ($W_s(volatile) < W_s(evolution) < W_s(stable)$) and tried various ratio between the category weights to evaluate. [Table 4.6](#) summarizes our weights and values to be evaluated, and provides an example of the different weights for the canvas attribute. Concerning our threshold, we will evaluate our metrics when progressively increasing its value from 0.5 to 1.

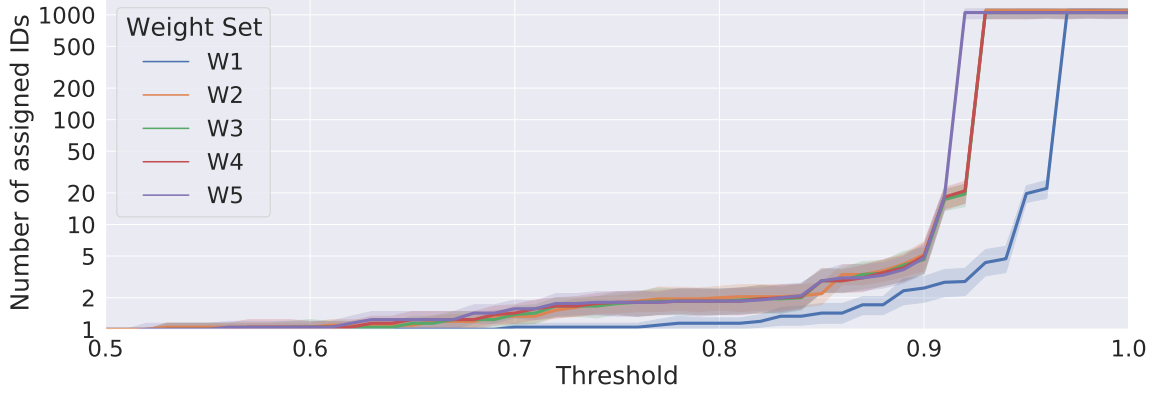
4.5.4 In-the-wild results

4.5.4.1 Safe mode results

We first evaluate the linking capability of our algorithm by running our algorithm in the *safe* mode previously defined. To evaluate our algorithm in this mode, we cleaned our dataset by keeping only browser instances that send browser fingerprints during the entire duration of our data collection. More precisely, we will only consider the browser instances that send i) a fingerprint during the first week of collection, and ii) another fingerprint during the last week of collection. This simplifies the analysis of our results as the longest identification duration is approximately the same for all the browser instances: 210 days. With this configuration, we evaluated 427,702 fingerprints belonging to 748 browser instances. The *average identification duration* for each of our weight sets and threshold values are presented on [Figure 4.3](#). The colors strip represents a confidence interval of 0.95. Our *average identification duration* remains at more 200 days for all our weight sets while the threshold is below 0.62. From this point, weight sets behave differently. The weight sets W2 to W5 are quite similar and provides an identification duration of 150 to 190 days for a threshold of 0.7, 145 days for 0.8 and 40 to 50 days for 0.9. These weight sets loose all identification power when the threshold is above 0.9. Oppositely, the weight set W1 can link browser instances for longer periods when the threshold is similar. This might be an unexpected result because we defined our weights according to the stability of attributes. In our stability weight definition, *stable* attributes have a higher stability weight than *evolving* attributes, themselves having a higher stability weight than *volatile* attributes. In fact, *volatile* attributes concern attributes that identity the configuration or the context of the user. If the user does not change her configuration, these attributes will not change. When designing our weights, we assumed these attributes could change anytime, because we cannot understand the behaviours behind a configuration change. While we do not observe such behaviours in this dataset, we still believe it can happen anytime. Thus, we prefer anticipating the worst scenario and use a weight set that is still able to link fingerprints after configuration or devices changes.

Our second metric for the *safe* mode is the *number of assigned IDs*. It represents the number of *chains* on which fingerprints coming from the browser instance have been put. These results are presented on [Figure 4.4](#). It also shows weight sets W2 to W5 behaves similarly, and put fingerprints coming from the same instance in around 5 different *chain* for a threshold of 0.9. It means for the duration of our experiment, which is 210 days, 5 chains were required per browser instance. A threshold higher than 0.9 does not seem relevant as the *number of assigned IDs* grows exponentially. Similarly to what has been observed on the *identification duration*, the weight set W1 requires a lower *number of assigned IDs* for the same threshold.

Figure 4.4: Number of assigned IDs to each browser instance according to the threshold and weights sets

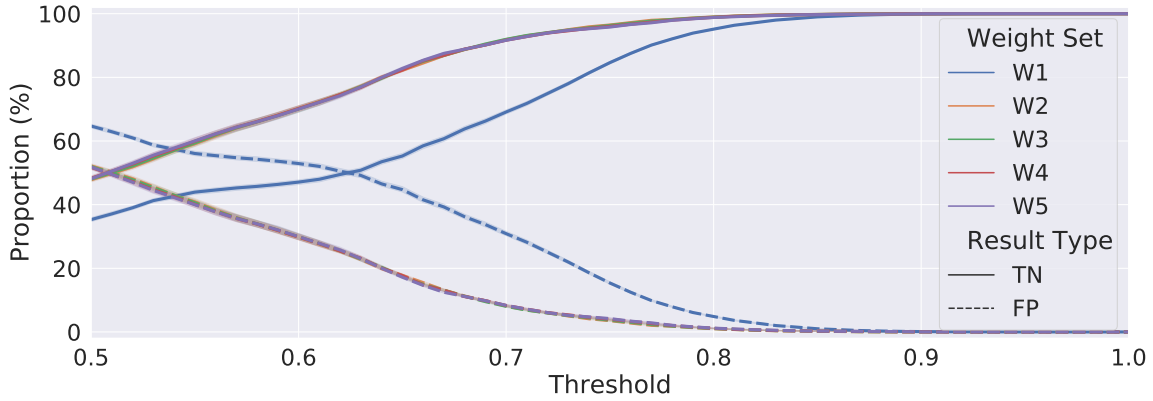


4.5.4.2 Attack mode results

Our second evaluation focuses on measuring the distances between the fingerprints of our dataset, and understanding if fingerprints coming from different instances are linked by our algorithm. We computed the number of *True Negative* (TN) and *False Positive* (FP) over more than 18 billion comparisons. Our results are presented on [Figure 4.5](#). We first observe the weights set *W1* globally links more fingerprints coming from different browser instances than other weight sets with identical threshold values. This is an undesired behavior as an attacker has a higher probability to be linked when trying to authenticate on the user’s account. Additionally, if an attacker is not directly linked to a registered fingerprint of the user, it shows he has less effort to do to mimic the fingerprint of the user he attacks. Because the weight set *W1* defines a constant weight of 1 for all the attributes, it highlights the need to setup different weights for attributes having different properties. The other weights sets we evaluated behaves better and provides similar results. With this dataset, a threshold of 0.85 to 0.88 would limit the possibility for an attacker to be linked to a registered fingerprint.

SYNTHESIS. In this section, we answer **RQ5** by evaluating our linking algorithm defined previously on a dataset of 952,828 fingerprints collected on 64,235 different browser instances. We evaluated our algorithm in 2 modes: a *safe* mode to measure the ability of our algorithm to correctly link the browser fingerprints coming from the same instances, and an *attack* mode where we estimated the probability for 2 fingerprints coming from different instances to be linked. We demonstrate our algorithm is reliable and relevant at linking fingerprints when provided fingerprints collected on in-the-wild browser instances.

Figure 4.5: Evolution of the proportion of True Negative (TN) and False Positive (FP) when evaluating our algorithm in attack mode.



4.6 Discussion

4.6.1 Ethical consideration

We collected our data in the wild with the AMIUNIQUE extensions between August 2020 and March 2021. Before installing the extension, we clearly informed users about its goal and the data collected on the installation page. The collected fingerprints are only linked to a random identifier generated during the extension installation. Contributing participants can access or delete all their data at any time by submitting their extension ID. Finally, the extension and the handling of collected data conform to the IRB recommendations we received.

4.6.2 Choosing parameters value

As our evaluation shows, the parameters have a direct impact on the results when evaluating our algorithm on both safe and attack modes. The weights set $W1$ appears to be unusable in real life because of his tendency to link fingerprints coming from different browser instances. On our dataset, other weight sets behave similarly. Concerning the threshold, the safe mode results encourage to lower the threshold to increase the identification duration while the attack mode results encourage to rise up the threshold to limit the risk for an attacker to be able to be linked to a registered fingerprint. According to our observations, we believe a threshold of 0.8 to 0.85 is interesting for our dataset because it provides an identification period of more than 100 days while having an extremely reduced proportion of false positives (FP): $0 \sim 2\%$.

These results need to be confirmed and/or adapted for other datasets. As already highlighted by several studies [96, 117, 103], different fingerprints datasets have different

properties, which could impact the parameters choice and the reliability of our algorithm. When possible, we advise users of our algorithm to evaluate it on their dataset and adapt the parameters accordingly.

4.6.3 Linking algorithm improvements.

As Li *et al.* [119] suggested, we took advantage of our controlled environment to improve the knowledge about the semantic of browser fingerprints. It allowed us to build relevant rules and set reliable weights. Our algorithm could still be improved on several points.

First, we did not take into consideration real-event timing, such as the major browsers release calendars or the switch to summer/winter time. While we agree on the interest of leveraging these information, these real-event could not occur on the same day or week for all the users. Thus, admins should have a great knowledge of their user and device bases before adding verifications based on these events.

Second, we observed that several changes are correlated and happen in the same browser update. On CHROME, the `appVersion` and `User-Agent` are linked to the current browser version. However, we did not use this information to ensure the changes were consistent with one another. Thus, it is possible to add stronger verification requirements where the values in such attributes should be updated at the same time. This would improve our algorithm by increasing the linkability between fingerprints.

4.7 Conclusion

In this chapter, we collected browser fingerprints inside a controlled environment. We studied the state-of-the-art attributes, as well as new ones and observed the possible values among a set of desktop and mobile devices. We answered **RQ3** about the cause of fingerprints diversity by demonstrating several hardware and software components—hardware, OS, browser configuration—are responsible for this diversity, on both desktop and mobile devices. Concerning the impact of a browser update on the fingerprint, questioned in **RQ4**, we provided a better understanding of the evolution of browser fingerprints from one browser version to the next, and show many attributes does not change or evolve towards a predictable value. We increased our knowledge on the semantics of browser fingerprints and showed that browser fingerprinting is an engineering side-effect caused by differences in hardware capabilities, API support and implementation, and even license issues. We observed few attributes concentrating the majority of the changes, meaning that many attributes are kept stable across dozens of browser versions. We leveraged this knowledge to design a browser fingerprints linking algorithm whose goal is to determine if a fingerprint is close enough to a set of registered fingerprints for web authentication. The design of this algorithm answers **RQ5**. We

implemented and evaluated our algorithm on a dataset collected with a browser extension. We evaluated our algorithm on a dataset of 952,828 fingerprints collected from 64,235 browser instances. Our results showed that our algorithm is able to link fingerprints coming from the same browser instance with high precision and speed. Finally, we also evaluate our algorithm when provided fingerprints coming from different instances, and demonstrate its reliability in this type of attack.

Chapter 5

Advanced risk-based authentication using browser fingerprinting

We explained in [Section 2.2.2](#) the different threats and attack models that target authentication systems. As we mentioned in [Section 2.2.5.2](#), additional authentication factors in a multi-factor authentication system impact the user experience. Among the websites we observed that have adopted authentication factors to strengthen web authentication in [Chapter 3](#), we were often requested to type a *One Time Password* (OTP), received via email or SMS to prove our identity. These techniques have a strong impact on the user experience because the user is required to perform several actions to authenticate, such as getting her phone or opening her email and retyping the code within the allotted time. We also observed browser fingerprinting was barely used to protect websites against attacks. As explained in [Section 3.5](#), we believe this is because: i) no technique existed to properly link browser fingerprints in a web authentication context, and ii) browser fingerprinting is perceived as a weak proof of identity that can easily be spoofed. Concerning fingerprint linking, [Chapter 4](#) aims at providing a browser fingerprint linking algorithm for web authentication. We evaluated our algorithm and observed it properly links fingerprints with a low risk of linking fingerprints from different browser instances. Concerning fingerprint spoofing, we observe several browser fingerprinting attributes are already used by real-life systems for *Risk-Based Authentication* (RBA), such as the HTTP headers [\[154\]](#), as explained in [Section 2.2.5.3](#). In their study, Wiefeling *et al.* [\[154\]](#) highlighted the positive perception of users concerning RBA schemes that only require an additional proof of identity when the device and/or the location has changed or is otherwise unknown to the authentication service. Browser fingerprinting is in itself a technique to identify devices. And while we agree on some of the limitations of this technique in regards to attribute spoofing, it is much more complicated to spoof a complete fingerprint than just the HTTP headers as used in some real-life authentication schemes. We believe it can be used as an additional and complementary technique

to verify the user's identity. Regarding the balance of *security* versus *user experience* discussed in [Section 2.2.5](#), we believe fingerprinting can strengthen an authentication scheme in interesting ways while having a negligible impact on the user experience, which ultimately will result in a higher level of acceptance from users and improved security.

According to these observations and the current state of the art, we believe browser fingerprinting can be used as a feature in a risk-based authentication scheme to strengthen authentication. This chapter aims at exploring the possibilities. In [Section 5.1](#), we complete existing work presented in [Section 2.6.3](#) by designing an authentication scheme that uses browser fingerprinting and explores the challenges it entails concerning security and usability. In [Section 5.2](#), we implement our scheme in the main authentication system at INRIA, which is a *Single-Sign On* service (SSO). We explain how challenges are addressed according to the specifications of the existing SSO and the constraints given by INRIA's *Chief Information Officer* (CIO) and the IT management department. In [Section 5.3](#), we evaluate our scheme and its implementation on a dataset of 82 users and 250 fingerprints. Our results show the scheme is reliable to improve web security while having a limited impact on the user experience. We discuss our results and the possibilities to improve the scheme in [Section 5.4](#) and conclude in [Section 5.5](#).

5.1 Authentication scheme

5.1.1 Design

We detailed in [Section 2.2.2](#) the different attack models that target authentication systems, such as phishing or stolen credentials. Several techniques presented in [Section 2.2.5](#) exist to strengthen web authentication. Among them, *Risk-Based Authentication* (RBA) collects information about the authentication context of the user to provide a risk level about the authentication attempt. This information—such as the IP address or the HTTP headers—are named *features* and can be collected without the user noticing. As stated by Preuveneers *et al.* [130], browser fingerprinting can be used as a feature in an RBA system. We explained in [Section 2.2.5.3](#) that several features already used to compute a risk level are in fact included in a browser fingerprint, such as the HTTP headers or the screen resolution. In this section, we define browser fingerprinting as an advanced feature for a *Risk-Based Authentication* (RBA) scheme. First, it can be used in an *immediate* mode where the fingerprint is collected on the authentication page and checked simultaneously with the username and password. In this context, it would counter attacks based on stolen credentials, such as phishing. The fingerprint can also be checked in a *delayed* mode. After the authentication is completed, each request sent to the server can include the fingerprint to check whether the session cookies are associated with the same fingerprint. Our intuition being that if the fingerprint changes mid-session,

it might reveal that an attacker has stolen the user session. Used this way, it would increase the security against the session hijacking attack described in [Section 2.2.2](#).

As mentioned in [Section 2.2.1](#), the lifespan of an authentication factor or a feature for RBA has 3 main steps, *enrollment*, *verification* and *recovery/revocation*.

Enrollment. We explained in [Section 3.4](#) and [Section 4.4](#) that a device fingerprint needs to be registered and associated with a user before being used for verification. We define a *registered device* as a device having at least one of its fingerprints registered. In Wiefling’s study [155], it is unclear how the features for RBA are registered. We believe the original values for the features are stored during account creation step. As discussed in [Section 4.6](#), a second option would be to use a secure way to register a feature, such as a MFA mechanism (an SMS or email OTP), as mentioned by Quermann *et al.* [131]. We believe both these techniques are suitable to associate a browser fingerprint to a user.

Verification. During an authentication attempt, the system collects the fingerprint of the user, named *submitted fingerprint*, and compares it to a list of *registered fingerprints* for this account. The list of registered fingerprints is obtained by taking the last fingerprint associated to each *registered device*. The fingerprint comparison outputs a risk level RL_F . If the submitted fingerprint is equal or close enough to a registered fingerprint, the risk level RL_F for this feature is considered *low*. Otherwise, the risk level RL_F is considered *high*. In this case, the server can take several actions, such as: i) denying the authentication attempt, ii) using a fallback technique (e.g., MFA SMS or Email verification), iii) allowing the user to authenticate but in read-only mode, and prevent any action that modifies the account.

Recovery/Revocation. The user cannot lose their browser fingerprint, as it is an inherent part of their browser. Thus, the recovery step is not necessary when using browser fingerprinting as an RBA feature. However, users must be able to revoke fingerprints belonging to a device they no longer have access to. For example, a user might change their computer or smartphone, or have one of their devices stolen.

Using browser fingerprinting to identify the device on which the user tries to authenticate is similar to the RBA-device explained by Wiefling *et al.* [154], with the exception they seem to have the device’s ground truth values.

5.1.2 Challenges

Several works have studied the challenges of proposing an RBA scheme being reliable and relevant. As mentioned by Preuveneers [130], the browser fingerprinting feature must fulfill several *Security Requirements* (SR):

- **SR1.** Fingerprints cannot be compromised.
- **SR2.** Prevent fingerprint *replay attacks*.
- **SR3.** Support fingerprint revocation.

- **SR4.** Fingerprints should have strong similarity checks.

Additionally, Alaca *et al.* [87] presented several *Properties* (P) an authentication scheme using browser fingerprinting might have:

- **P1.** Each device has a unique fingerprint that can be associated with a user's account.
- **P2.** Fingerprints obtained via the same browser instance are either identical or linkable.
- **P3.** It is difficult for an attacker to spoof a device.

These studies did not mention the need to reduce the impact on the user experience. If the authentication scheme has a strong impact on the user experience, it will frustrate users and prevent them from adopting the scheme. Concretely, we define the following User Experience (*UE*) concerns that should be addressed:

- **UE1.** First, the time taken by the browser fingerprinting script to collect the fingerprint should not be higher than the time required by users to enter their credentials. In the case of auto-filled forms through, for example, the use of password managers, the collection time should be really short—a few hundred of milliseconds.
- **UE2.** Second, the time taken by the system to compare the submitted fingerprint to the registered ones will be part of the total time taken by the system to authenticate the user. Thus, the time taken by the system to compare the submitted fingerprint to the registered ones should be short because it will increase the total time taken by the system to authenticate the user.
- **UE3.** Finally, the number of actions required to add or remove a fingerprint to a user's account should be low to facilitate the acceptance of the scheme by the users.

By addressing the *SR* and *P* from the state-of-the-art, and the *UE* concerns we propose, we argue browser fingerprinting can strengthen web authentication while having a negligible impact on the user experience. More specifically, we formulate the following **challenges** an authentication scheme using browser fingerprinting should try to rise up to be reliable and relevant:

Challenge n°1: Spoofing the fingerprint of a user should be hard for an attacker.

Challenge n°2: The authentication scheme should support the registration of devices with a user-friendly mechanism.

Challenge n°3: Fingerprints from the same browser instance should be linked together by the authentication scheme.

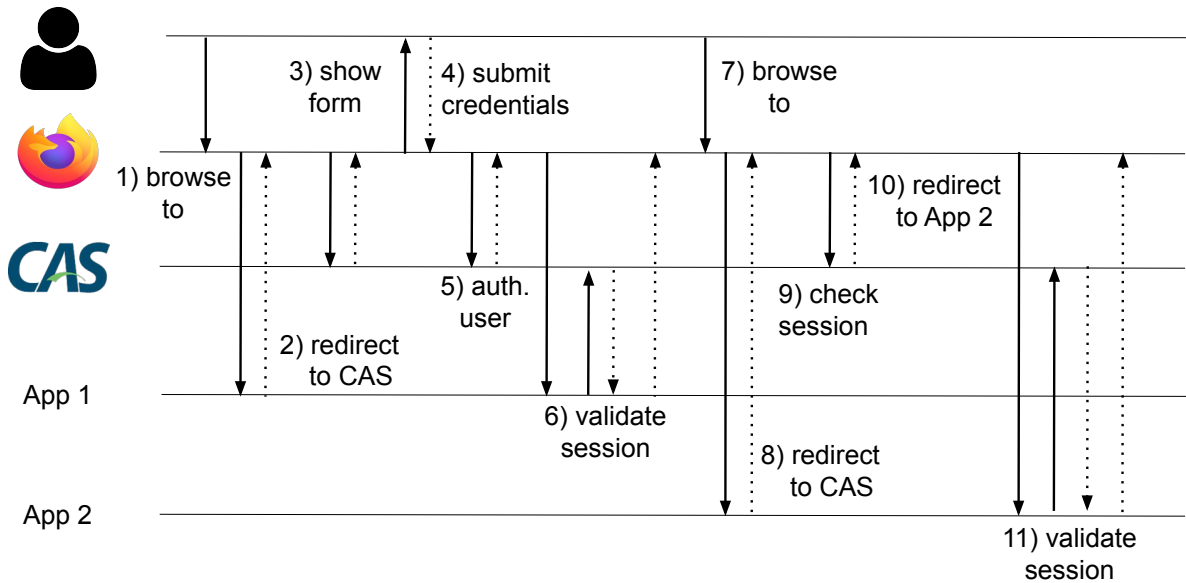
Challenge n°4: The authentication scheme should support the revocation of fingerprints.

Challenge n°5: The authentication scheme should minimize the impact to the user experience.

5.2 Implementation

5.2.1 Legacy Authentication Systems

Figure 5.1: Description of the interactions when authenticating on an SSO system.



INRIA is a French research institute that has 9 research centers across France. The main authentication system INRIA uses is the *Central Authentication Service* (CAS). CAS is a *Single Single-On* (SSO) system that provides a unique authentication point for users to access most of INRIA's applications, as explained in [Section 2.2.1](#).

[Figure 5.1](#) presents the interactions between the user's browser, the applications she attempts to access and the SSO authentication system. When a user accesses an application behind the SSO system for the first time (1), the application redirects the user to the SSO (2), which provides the authentication page to the user (3). The user submits her credentials (4), the system validates (or denies) the authentication attempt (5), and redirects the user to the service she originally accessed with a session identifier. The service queries the SSO system to validate the user session identifier (6). Finally, the service creates the session and the user can now browse the service. When the same user browses a second application behind the same SSO system (7), this application also redirects the user to the SSO system (8). The SSO checks the user session, sees she is already authenticated (9), and redirects the user to the application with a session identifier (10). The service validates the user session identifier with the SSO system (11) and creates a new session. For this second service, the user is not prompted with a form, she is authenticated via a double redirection mechanism plus a session check that are almost unnoticeable. More generally, a user can usually notice only a fraction of

the interactions happening in this kind of system, those which occur between the user and the browser. Several session states are kept by the applications to maintain the authenticated state of the user. For a user that browses n services, she will have $n+1$ sessions:

- n session for the n application she browses. These sessions are initialized by an exchange between the application and the SSO system and allow the applications to maintain an active session without querying the SSO system each time they receive a request from the user.
- One session maintained by the SSO system. It prevents the SSO system to present the user an authentication page with a form each time an application asks for session verification.

We aimed at improving the **immediate** authentication scheme when the user enters her credentials on the form. Before our additions, the authentication system of INRIA worked exactly the way described above. The authentication scheme only used the password as an authentication factor and did not include any additional factor or feature for Risk-Based Authentication (*RBA*).

By integrating our solution in the existing system, we added a fingerprint collection script on the authentication form and a fingerprint verification step during the authentication attempt process on the server. The next section will precise these changes and present the resulting scheme.

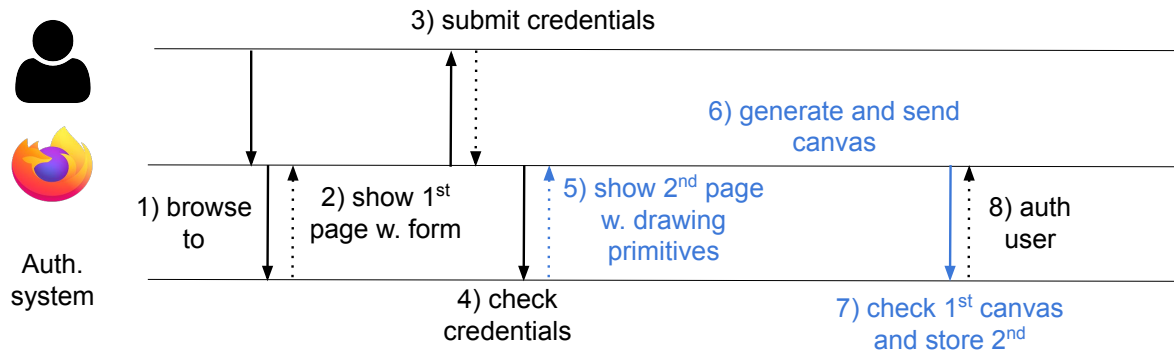
5.2.2 Rising to the challenges

In this section, we explain how the challenges defined in [Section 5.1.2](#) are addressed in our implementation.

5.2.2.1 Spoofing should be hard for an attacker

Challenge n°1 concerns the collection of the fingerprint. It should be difficult for an attacker to collect the fingerprint of the targeted user as it can be used to spoof the user's fingerprint and gain access to the account. The common attributes used to build a fingerprint can be collected by an attacker through JAVASCRIPT APIs as soon as the visitor executes a script on a page under his control. This is the case of the phishing attack, presented in [Section 2.2.2](#). If an attacker is aware of the use of browser fingerprinting to strengthen web authentication, he could update his attack by collecting the fingerprint of the users that visit the phishing page. When the attacker tries to authenticate on the legitimate authentication page with the credentials he stole, he could replay the collected fingerprint to make it look like the legitimate one. With this new

Figure 5.2: Description of the changes required on an SSO when adding a dynamic canvas check step.



attack model, an attacker could bypass the additional security layer provided by browser fingerprinting.

We explained in [Section 2.3.3.2](#) the canvas attribute generates an output whose value depends on the drawing primitives used. As studied in [Chapter 4](#), due to the differences in the hardware and software components between devices, devices that are given the same set of drawing primitives might generate different values. In [Section 2.6.3](#), we described a technique proposed by Laperdrix *et al.* [116] that leverages this unpredictability to generate a pair of dynamic fingerprints. By chaining dynamic fingerprints, it is possible to build a trusted fingerprinting chain that cannot be replayed by an attacker. In this case, the first dynamic fingerprint is used to verify the user's identity, while the second fingerprint is stored for the next authentication. While an attacker learning the set of primitives used for a given user could not directly know the canvas generated by the user's device as it might have different hardware and software components, the attacker could generate canvas on a pool of devices he owns to try generate the same canvas as the user's device would. In this context, the secret information that must be protected from an attacker is not only the value of the canvas, but also the randomly generated drawing instructions. Because they are part of the secret information, the drawing instructions should be protected and cannot be provided to the user without receiving a first proof of identity.

[Figure 5.2](#) presents how such an authentication would work. Steps in blue are new steps compared to an existing scheme and would require to be added. When a user wants to authenticate (1), the server provides the user a first HTML page with a login form (2). The user enters her username and password and sends them to the authentication server (3). If the given password matches (4), the server sends back the user on a second HTML page with the pair of drawing primitive sets to collect the canvas rendering data (5). The second page computes two canvas renderings on the user's device, each with a

different set of drawing instructions, and returns the result to the server (6). The first set of drawing instructions corresponds to an image the server previously asked for from the device, and the second set is a new, random, unique set of instructions. The server compares the image from the first drawing primitives set to the one stored during the previous authentication attempt. If it matches, the second set of drawing primitives is stored to check the next authentication attempt (7), and the user gains access (8).

This authentication scheme with unpredictable canvas values would have the same challenges as an authentication scheme with static browser fingerprints. Additionally, the drawing primitives are a secret information and are specific to a user. Then, a first proof of identity needs to be checked before sending the drawing primitives to the user. Otherwise, the set of primitives used for this user can be learned by the attacker, which leads to a partial loss of interest for this technique compared to using the same set of primitives for all the users. Then, it implies having 2 HTML pages provided to the user at different moments of the authentication flow to complete the data collection without compromising her security. The current protocol authentication system on which we will integrate our solution only provides one HTML form to the user, which is the one collecting the username and the password. Presenting a second HTML page to the user, even without a form, would change the entire authentication flow and split the factors/features checks. The system should be sure the user is following the complete flow from the beginning to the end to avoid attackers that could manage to shortcut some parts of the flow and gain access to the account. It would require major changes on any authentication scheme, and especially on the CAS one. Thus, we decided to let the implementation and evaluation of the Challenge n°1 for future work, and to use the usual static attributes adopted for browser fingerprinting. More precisely, we collected the navigator, screen and window properties, font enumeration, canvas and WebGL rendering, WebGL properties and WebRTC data to form a fingerprint, as explained in [Section 2.3.3](#).

5.2.2.2 The authentication scheme should support the registration of a new device

As mentioned in our **Challenge n°2**, the system needs to provide users a way to register devices to perform comparisons between the submitted fingerprint and registered fingerprints coming from registered devices. We explained in [Section 4.6](#) that several techniques can be used to register a device, such as an SMS or email OTP. As the email service of INRIA is itself behind the CAS system, it cannot be used to register a device. The usage of an SMS OTP would either require 1. employees to use their personal mobile devices, which has been excluded for privacy reasons, or 2. INRIA to provide professional mobiles to its employees, which has been excluded for cost reasons. The use of hardware token has also been excluded for similar reasons. Additionally, accounts are automatically

created and we cannot rely on the account creation process to register a first device to the account.

Instead, we decided to rely on IP networks for our registration. As explained in [Section 2.2.5.3](#), IP networks are a feature for *Risk-Based Authentication* (RBA). Thus, rather than using an authentication factor to register devices, we used an RBA feature. We name the risk-level of this feature RL_N . We define a **trusted network** as a network providing *trusted IP addresses*. All other networks are *non-trusted networks* and do not provide trusted IP addresses. A user with a *trusted IP address* will have the fingerprint of her device automatically registered, leading to a registered device. In our implementation, *trusted networks* are Wifi and wired networks setup by INRIA and available in each research center, plus the VPN network that allows remote connections. The research center's networks require a password to authenticate, plus the geographical constraint of being physically present at one of the research buildings (or very close to it) to be able to use these networks. The authentication scheme for the VPN only requires a password. Consequently, a user who wants to connect on a *trusted network*—internal networks or VPN—must provide her password in all cases, and be in a research center if she does not connect on the VPN. In our implementation, an authentication attempt happening on a *trusted network* will get assigned a *low* risk level RL_N . Otherwise, the risk level RL_N is *high*. The usage of *trusted networks* to assign a risk-level concerning the location of the user for the authentication attempt corresponds to the RBA-location defined by Wiefeling *et al.* [154].

Employees have a physical office in one of the 9 INRIA centers. We believe they are more likely to authenticate from the network of their center or from the VPN. We believe the use of a limited number of trusted networks could help us increase the security by reducing the trusted networks on which devices can be registered. Thus, we make the hypothesis **H1** that the registered devices of a user might be registered on at most 2 trusted networks.

5.2.2.3 Fingerprints coming from the same instance should be linked together

Concerning **Challenge n°3** and the linkability of browser fingerprints coming from the same browser instance, we use our algorithm presented and evaluated in [Chapter 4](#). As advised in [Section 4.6](#), we will first use parameter values computed on the AMIUNIQUE dataset that have been found to be the best. However, we will evaluate the best weights set and threshold for our new dataset, because each dataset has its own particularities, which could change the computed scores and behaviors of the algorithm. When a user tries to authenticate, the algorithm tries to link the submitted fingerprint to one of the registered fingerprints. We explained in [Section 5.1.1](#) the list of registered fingerprints is formed by collecting the last fingerprint associated to each *registered device*. When

comparing the submitted fingerprint to the registered ones, the scheme returns a *risk level* RL_F based on the comparison result:

- **low.** If the submitted fingerprint is equal or close enough to a registered fingerprint, the *risk level* RL_F for the authentication attempt made with this fingerprint is considered as *low*, and the user proceeds to the next step of the authentication scheme;
- **high.** If the submitted fingerprint cannot be linked to any of the registered fingerprints for this user, the *risk level* RL_F is *high*.

The global risk level RL_G of the authentication scheme is computed by taking the highest risk given by the 2 risk levels RL_N and RL_F . Thus, RL_G is *low* only if both RL_N and RL_F are *low*. In this case, the user will be able to authenticate—if the password is validated. Otherwise, the risk level is *high*. In this case, users can have different ***risk policies***:

- **Permissive:** the authentication scheme allows the authentication attempt to proceed despite the risk level being high, or
- **Strict:** the authentication scheme stops the authentication attempt. The user is redirected on the authentication page with an error message. The error message provided is generic to prevent potential attackers to know which factor or risk level caused the authentication attempt to fail.

5.2.2.4 The authentication scheme should support the revocation of fingerprints

We developed a management application for users to manage their accounts and fingerprint data. The available actions are similar to the ones observed in existing applications presented in [Section 3.4.1](#). First, users are able to check their authentication attempts to monitor any suspicious activity on their account. Additionally, they can manage their registered devices and, to address **Challenge n°4**, remove them—for example if they no longer have access to the device.

The application also allows users to configure their *risk policy*, which can either be *strict* or *permissive*. As we do not want to degrade the user experience without the users being informed, the default value is *permissive*. We also offer the possibility for users to receive emails whenever an event requiring the attention of the user happens. We define 3 such events:

- A new fingerprint is registered on a *trusted network*.
- An authentication attempt is made using a fingerprint that cannot be linked to a registered one, on a non-trusted network—no matter the result of the authentication attempt.
- A registered device is removed via the interface of the management application.

Users can configure if they want to receive email alerts concerning these events, or not. By default, users receive alert emails for all these events.

We also provided the possibility for administrators to manage user accounts. Administrators have access to the authentication attempts, registered devices, fingerprints and configuration for all users. Additionally, they can revoke a device or re-allow a device that has been previously revoked.

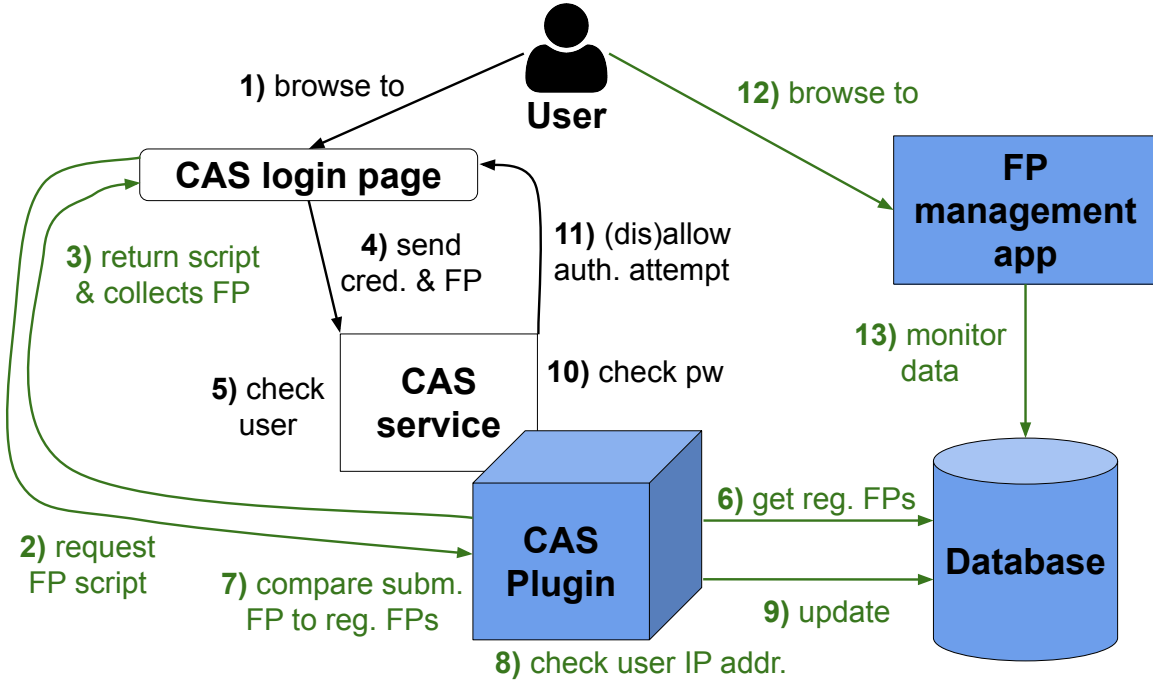
5.2.2.5 The authentication scheme should minimize the impact to the user experience

We aimed at limiting the impact on the user experience to validate **Challenge n°5**. We monitored the time taken by the script to collect a fingerprint from various devices and we estimated its duration to 400–500 milliseconds. Users could validate the form before the fingerprint is collected, for example if they use a password manager that automatically fills the username and password. While the validation of the form before the collection of the fingerprint will cause the authentication attempt to fail, we believe this is unlikely to happen in practice. In a first step, we monitor the collection time for real users, before adapting our strategy according to the results.

5.2.3 Authentication scheme and CAS plugin

Globally, our authentication scheme can be seen as a Risk-Based Authentication scheme with 2 features: i) the device feature, which corresponds to the collection and verification of the fingerprint of the device used by the user to authenticate, and ii) the location feature which corresponds to the collection and verification of the IP address of the user. When authenticating, the user must validate either the device feature or the location feature. If she can, the risk is low, and she is able to authenticate—if the submitted password is correct. If she cannot validate any of these 2 features, the risk for the authentication attempt is high. We implemented this authentication scheme in a CAS plugin. **Figure 5.3** presents the way our plugin integrates with the existing system and how it modifies the authentication scheme. The existing components and steps are presented in black, the new components added are presented in blue, and the new steps in the authentication scheme are depicted in green. When the user browses the CAS login page (1), she enters her username and password. During this period, the page loads the fingerprinting script (2) that collects the user's browser fingerprint (3). When the user validates the form, it sends the username, password and fingerprint to the server (4). First, the server checks if the username exists in the base (5). Then, the plugin gets the registered fingerprints for this user from the database (6) and compares the submitted fingerprint to the registered fingerprints (7). If the submitted fingerprint is linked to a registered fingerprint, the database is updated and the process continues (9-11). If the

Figure 5.3: Description of the new authentication scheme with our plugin and new components.



submitted fingerprint cannot be linked to any registered fingerprint, the IP address of the user is checked to determine if the user is connected on a *trusted network* (8). If she is, its fingerprint is registered (9), and the plugin ends its process. The CAS system processes the password check (10) and allows the user to authenticate (11). Once the user is authenticated, she can access the fingerprint management application (12) where she can access her fingerprints, connection attempts and parameters (13).

SYNTHESIS. In this section, we answer **RQ6** about the user experience concerns of a web authentication system using browser fingerprinting by presenting 3 main concerns. Combined with the existing security requirements and properties from the state of the art, we defined 5 challenges any authentication system leveraging browser fingerprinting should respect. We implemented and integrated our authentication scheme in a CAS plugin that aims at being integrated into a real-life authentication system. The following section will evaluate the security gains of our solution and the impact on the user experience.

5.3 Evaluation

We integrate our CAS plugin that implements our authentication scheme into a real system and analyze the results in this section.

5.3.1 Dataset constitution

We first deployed our plugin in the *certification* environment of INRIA. The *certification* environment is a test environment for engineers before deploying software or updates to the *production* environment. Thus, the users using the *certification* environment are engineers and employees from the IT management department. Our plugin was integrated in the environment in May 2021.

We only analyzed pseudonymised data from users who agreed to have their data analyzed by INRIA researchers. We discuss in more detail the conditions to exploit the data in [Section 5.4](#). The dataset, which we named *certification* dataset, is composed of 82 users, 250 fingerprints and 331 authentication attempts. The distribution of the number of fingerprints and authentication attempts by users is presented in [??](#). 35 users have one authentication attempt and one fingerprint, which means they only authenticated once on the system. Other users have various numbers of authentication attempts and fingerprints.

5.3.2 Key Performance Metrics

Concerning our challenges, we cannot evaluate our **Challenge n°1** as we did not implement it. Additionally, the **Challenge n°4** is implemented and the system supports revoking fingerprints. As the system has only been in the *certification* environment for a limited duration (3 months) with a limited set of users, we believe the evaluation of **Challenge n°4** is not relevant with this dataset. Thus, it remains **Challenges n°2, 3 and 5**.

Challenge n°2 focuses on the device registration. As we leveraged *trusted networks* to register devices, we aim at measuring the *number of fingerprints registered on a trusted network*. This metric indicates how often users connect on a *trusted network*, and how often users could have their authentication attempt blocked if they try to authenticate from an *non-trusted network* and the linking algorithm cannot link their fingerprint. Additionally, we measure the *the number of trusted networks* used by users to register their fingerprints. According to this analysis, we might be able to confirm our hypothesis **H1** detailed in [Section 5.2](#) about the number of *trusted networks* used by users to register devices.

Challenge n°3 concerns the linkability of browser fingerprints. We will evaluate our algorithm on this dataset the same way we did in [Chapter 4](#). However, our *certification* dataset was collected over a much shorter period with much less users. Only 47 users authenticated more than once, and very few authenticated over several weeks. Because of this, the *average identification duration* is 17 days with a standard deviation of 26 days. In this case, the *identification duration* metric does not seem appropriate. Instead, we define a new metric, the *fingerprints number ratio* which corresponds to the number of

fingerprints in the longest *identification chain* divided by the number of total fingerprints of the browser instance. Because we do not have the ground truth of the devices in the context, we will consider that fingerprints belonging to the same user and sharing the same OS and same browser come from the same instance.

Challenge n°5 aims at limiting the impact on the user experience. We already partly addressed this challenge by leveraging *trusted networks*, which is a transparent technique to register fingerprints. We identify 2 metrics to evaluate the impact of our scheme on the user experience: the *fingerprint collection time*, which is measured on the HTML page providing the authentication form to the user, and the *linking algorithm computation time*, which is the time taken by our linking algorithm to provide a result.

5.3.3 Trusted network fingerprints and authentication attempts

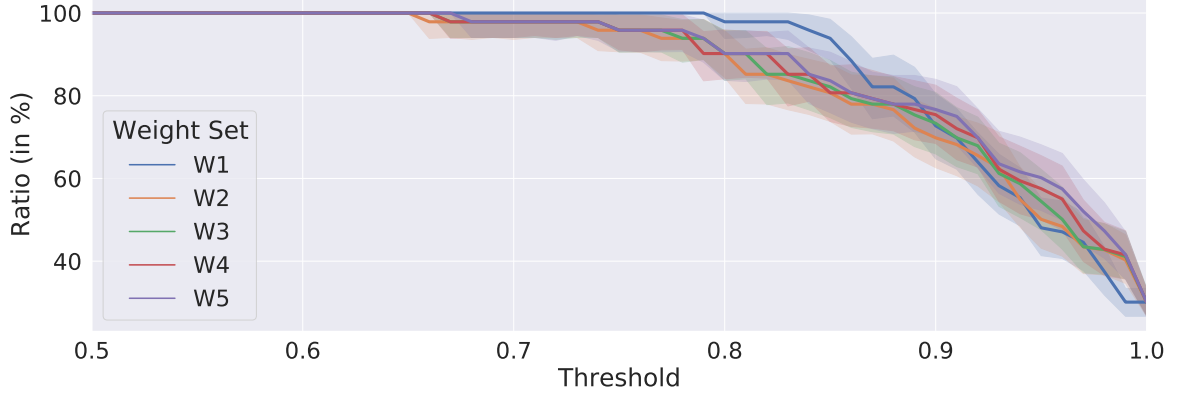
In our dataset, all 331 authentication attempts happened on a trusted network. This can be explained because the *certification* environment is only available behind an INRIA network, which corresponds precisely to the *trusted networks* we defined. This behavior prevents us from conducting further analysis on this part, such as estimating how often users authenticate on a *trusted network*.

Concerning the number of *trusted networks* used by users to register fingerprints, 54 users used a single *trusted network* while the remaining 28 users used 2 *trusted networks*. Based on these observations, we confirm our hypothesis **H1** made in [Section 5.2.2.2](#) stating users might only use at most 2 *trusted networks*: the *trusted network* of the research center they have their office in, and the VPN. This behavior will help us to imagine more strict rules to register fingerprints, as detailed in [Section 5.4.4](#).

5.3.4 Linking algorithm scores

We reran our linking algorithm on our dataset to estimate the best set of parameters for this dataset. We grouped all the fingerprints of each user that share the same OS and browser in a list F_{ud} , and sorted F_{ud} by timestamp. Then, for each fingerprint f of F_{ud} , we defined e as the fingerprint being right before f in F_{ud} , and we ran our algorithm with f as the *submitted fingerprint* and e as the registered fingerprint. [Figure 5.4](#) presents the *fingerprints number ratio* computed from the results given by our linking algorithm, according to the chosen threshold and weight set. We can see all the weight sets behave similarly: the ratio is 100% until a threshold of 0.65, which means there is only one *chain* to this point that contains all the fingerprints F_{ud} . This means that they all have been correctly linked together. After, the ratio decreases slowly, reaching around 90% for a threshold of 0.8, 75% for a threshold of 0.9 and 20% for 1. These results are quite different from the ones we observe in [Chapter 4](#).

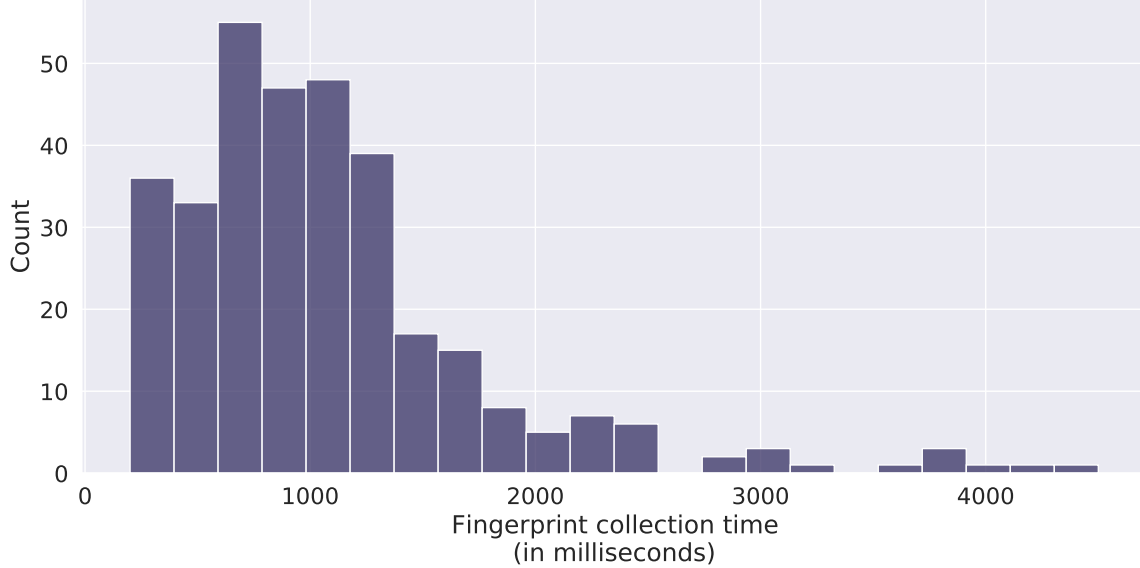
Figure 5.4: Fingerprints number ratio according to the threshold and weight set



First, the results of the weight set $W1$ are similar to the results of the other weight sets. This is due to the fact the attribute that changes the most is the `screen` attribute. As we explained in [Chapter 4](#), we anticipated this behavior by designing our weight sets because some attributes, such as the `screen`, are unpredictable and can change if the user connects an external screen to her computer. Because our weight sets $W2$ to $W5$ have been designed to take into consideration these changes, we expected the weight set $W1$ to behave similarly or worse than the other weight sets.

Second, while the metric used is different, it appears our *identification chains* last longer with this dataset than with the one described in [Section 4.5](#). Thus, our results seem better than the results we got in [Chapter 4](#). Several parameters differ from the previous evaluation, which can be explained by the following factors: i) the dataset described in [Chapter 4](#) is collected via a web extension, whose goal is to specifically study browser fingerprints. In this context, users might want to monitor their fingerprint and its evolution by modifying their browser and/or device configuration and by testing fingerprinting defense. This could lead to a dataset with many fingerprint changes, which are harder to link. Oppositely, the fingerprints of our *certification* dataset were collected from users that do not have a specific interest to change their fingerprint on purpose. Then, with less changes, fingerprints are easier to link. ii) The collection period for the dataset used in [Section 4.5](#) is 7 months compared to the 3 months of our *certification* dataset. A longer collection period implies more difficulties to link fingerprints due to the increased number of changes that can occur. While our results appear to be better, they need to be confirmed with a larger dataset collected over a longer period.

Figure 5.5: Distribution of the fingerprint collection time



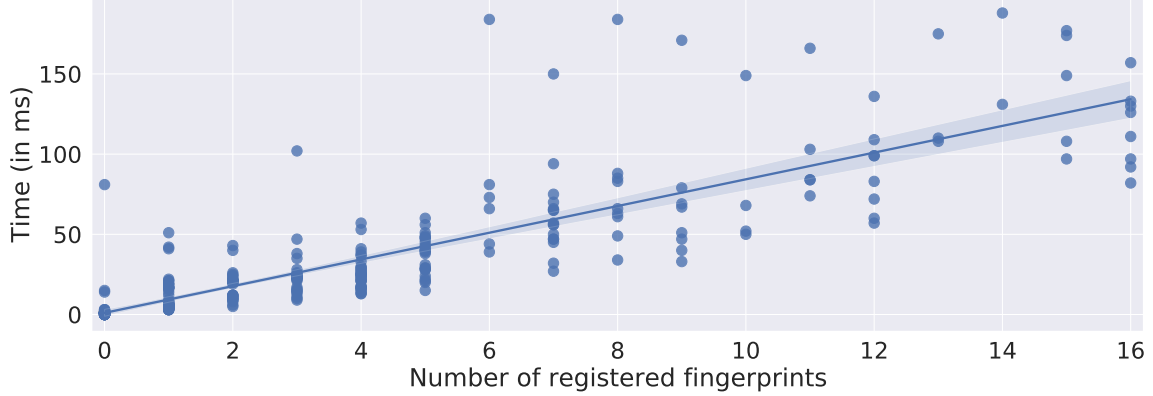
5.3.5 Collection and analysis time

The distribution of the fingerprint collection time for our 331 authentication attempts is presented in [Figure 5.5](#). The median is 887 milliseconds, which is higher than the time we estimated in [Section 5.2.2](#). It still appears to be a decent collection time that should not impact the user experience when filling and validating the authentication form. The 90th percentile corresponds to 1,879 milliseconds while the 95th percentile corresponds to 2,396 milliseconds. According to the time needed by users to fill the form combined with the eventuality users could have password managers that can autofill the authentication form, admins might want to set a higher time threshold to collect the fingerprint to ensure it is available to be checked server-side. We discuss this possibility in [Section 5.4.3](#).

As explained in [Section 5.1.2](#), the analysis time required to link fingerprints is critical because this time is added to the global server response time. A high response time will decrease the user experience. It is important to minimize the time taken by our algorithm to link the submitted fingerprint to one of the registered ones. The distribution of this time, combined with a linear regression, is presented on [Figure 5.6](#). The time grows linearly according to the number of registered fingerprints, which is a logical result according to the design of our algorithm (rule-based). This shows the importance of only taking the last fingerprint of each device to reduce the number of fingerprints in the list of registered fingerprints, and so, the time taken by the linking algorithm.

SYNTHESIS. In this section, we answer **RQ7** about the security improvements and high usability of a web authentication system using browser fingerprinting by proposing an

Figure 5.6: Distribution of the linking algorithm analysis time and linear regression



evaluation of our risk-based authentication scheme using browser fingerprints. This evaluation shows the interest of our technique concerning the limited impact on the user experience. We look forward to integrate our CAS plugin into the *production* environment to enhance the security for all INRIA agents. This should allow us to benefit from a larger dataset with several thousands of users, which will increase our knowledge about the impact and benefits of our authentication scheme.

5.4 Discussion

5.4.1 Ethical considerations

We designed our authentication scheme and CAS plugin while being in constant communication with INRIA’s *Chief Information Officer* (CIO) and the IT management department. They imposed several constraints on the plugin, such as the use of *trusted networks* to register devices and the default values given to the configuration for new users. They validated each step and tested the prototype in several environments similar to the *production* one before considering and validating the use of the plugin in *production*.

Concerning the data treatment aspects, we wrote and submitted several documents to both INRIA’s approval commissions and our national data protection authority (*Commission Nationale de l’Informatique et des Libertés* - CNIL) to validate our experiments, the use of browser fingerprinting to strengthen web authentication and the fact researchers could have access to the pseudonymized data. The use of browser fingerprinting for authentication was validated by both commissions. Concerning the transmission of user data to researchers, CNIL specifically asked to collect the explicit and knowledgeable consent of users. Thus, we set up an additional parameter in the configuration of each account, which indicates if the user agrees to share her pseudonymized data with INRIA

researchers. The default value for this parameter element is *false*, which means a user has to explicitly go to the management application and activate this parameter for us to access her pseudonymized data. Finally, personal information is pseudonymised before we receive it. This includes the username, the nickname users can give to their registered devices, and the name of the *trusted network* on which each device was enrolled.

5.4.2 Security versus user experience

We explained in [Section 2.2.5](#) the *security versus user experience* balance that every authentication scheme has to consider. RBA features, such as browser fingerprinting, have a lighter impact on the user experience because: i) they can be collected without any interaction from the user, ii) they only trigger more secure features, such as OTP, when the risk level is estimated to be high. However, the features are collected on the client-side and are more vulnerable to attacks. In our implementation, we used *trusted networks* to register devices. Other techniques can be used to register devices, such as SMS or email OTP. They have a stronger impact on the user experience but can be considered as more secure, especially if no *trusted networks* exist, as is the case for many online services. Additionally, the default value for the *risk policy* configuration is *permissive*, which does not stop the authentication attempt when the global risk level RL_G is *high*. In our scheme, all the default configuration values and implementation choices were meant to limit the impact on the user experience. While it leads to a less-secure scheme compared to other possibilities we considered, our scheme is still an improvement of the scheme that was only using passwords.

5.4.3 Client-side-generated information

Our **Challenge n°1** mentions the fact that fingerprint spoofing should be hard for an attacker. We explained in [Section 5.2.2](#) that we used static browser fingerprinting attributes to form a fingerprint. Static attribute values can more easily be collected by an attacker, for instance on a phishing page. Several mechanisms can be set up to harden the collection for unauthorized parties:

- We discussed the use of Laperdrix’s proposal to create dynamic canvas’ for web authentication [\[116\]](#). While we did not use dynamic fingerprints for our proof of concept, this solution could improve the authentication scheme and harden the collect of the fingerprint for an attacker.
- During fingerprint collection, we could use APIs that require the users permission to access, such as the geolocation, as experimented by Preuveneers *et al.* [\[130\]](#). It would strengthen the fingerprinting feature as several attacks rely on collecting user credentials, such as phishing, should also have to collect the user’s consent to access the data. The use of these techniques would lower the risk for users to have

their account compromised by these kind of attacks, but it would not completely remove it in case of a similar phishing page and a naive user. Additionally, it would degrade the user experience.

5.4.4 Device management rules

In our current implementation, the user can register any device while being connected on any *trusted network*. This behavior might be too permissive and unnecessary as very few users might need to authenticate or add new devices from multiple research centers. We propose the 3 following configuration elements to improve security and limit the registration of unwanted devices:

- **Trusted networks list.** Users might want to configure on which *trusted network(s)* they allow device registration. According to our observations performed on [Section 5.3.3](#), the use of more than 2 *trusted networks*—the research center they are located in, plus the VPN—should be rare. This improvement should reduce the attack surface of our authentication scheme, as it lowers the geographic possibilities for an attacker to be connected on a *trusted network*;
- **Forbidden devices.** Users could also restrict the type of devices that can be registered on *trusted networks*. If a user does not own any APPLE devices—and do not intend to—she has a limited interest in allowing such devices to be registered on her account. The management application could propose device classes, brands, models, or OSes that should never be registered for the account, and rely, for instance, on the **User-Agent** JAVASCRIPT attribute of the browser to determine if the device should be registered. That would suppose users are not modifying their fingerprint on purpose [150].
- **Reading & writing rights.** Once a device has its fingerprint registered, the user has can perform sensitive operations with it, such as removing registered devices or modifying the configuration of the account to adopt a *permissive* risk policy. With this system, an attacker who gained access to the account can remove the registered devices of the legitimate user. To mitigate the risk of users loosing access to their account, we imagine a system of *reading* and *writing* rights for the management application. The first registered device gains automatically both *reading* and *writing* rights while all other devices only obtain *reading* rights when registered. With this system, the user can only change her data—revoking a device, updating her configuration or changing devices rights—on a device with *writing* rights. In the case of a loss of the only device with *writing* rights, we believe the user could reach the administrators to request an update of another device’s rights by providing a proof of her identity. This is also the current process used by Inria when a password is lost.

5.4.5 Compromised device

Several authentication factors can be changed when they are compromised. For instance, users of web authentication systems are often requested to update their password if they observe suspicious activity, and to disconnect their account from all the devices. This way, the session of the attacker is also closed and he has to learn the new password of the user to re-gain access to the account. This behavior can be easily setup for *knowledge factors*, but is difficult for *ownership factors* as described in [Section 2.2.5.2](#). While a user can technically replace a phone—which is an *ownership factor*—if hers is stolen or compromised, it costs her money and cannot be seen as an easy way to maintain the security level of the authentication scheme. A worse situation happens when an *inherence factor* is compromised. As the information provided by the user is inherent to the user itself, she cannot replace it at all. Concerning browser fingerprinting, an attacker can authenticate if he provides a fingerprint that is linked to a registered device. In this situation, the attacker is able to authenticate and knows the fingerprint he provided is close enough to a registered device. This could mean the whole device is compromised. In fact, when we talk about a *registered device*, we talk about the combination of the layers that provide the values of the browser fingerprinting attributes—hardware, OS, browser, configuration, and maybe more. As a change of browser impacts the fingerprint in many ways (see [Section 4.2.3](#)), a solution to be able to continue using this device for authentication would be to switch browsers. This way, the user can still use her device while the attacker has to learn how to get close to the fingerprint generated by the new browser used by the user. While this solution is clearly not ideal because it impacts the user and her habits—and does not work in the case of a stolen device, for instance—it can still be acceptable for some users.

5.4.6 Adding features to the authentication scheme

We believe we could use additional features to compute our global risk-level RL_G . Features, such as mouse and keystroke dynamics [144] and login time [102], could reinforce our risk-based model and improve the security of our authentication scheme. However, each of these features has specific constraints concerning their collection or analysis and might not be used all together at the same time. For instance, when a user updates her password, its keystroke dynamics need to be learned again. Thus, it requires several successful authentication attempts for both the user to get used to typing it and for the authentication scheme to learn enough to build a stable model that can distinguish legitimate attempts from fake ones originating from attackers. Additionally, several RBA features might be useless according to the authentication context and device used, such as the mouse dynamics on a mobile device or the keystroke dynamics if the user uses a password manager that autofills passwords.

5.5 Conclusion

In this chapter we studied the feasibility of using browser fingerprinting as a feature for risk-based authentication. We answered **RQ6** about the user experience concerns of an authentication scheme using browser fingerprinting by defining 3 concerns focused on the user experience that web authentication systems with browser fingerprinting should address. We extended the state of the art by designing an authentication scheme using browser fingerprinting and defining several challenges such schemes should meet to be secure and have a negligible impact on the user experience. We addressed many of our challenges by proposing innovative solutions, such as the use of *trusted networks* to register devices. We implemented our authentication scheme in INRIA's SSO authentication system. We evaluated the benefits of our technique on the *certification* environment. We showed our linking algorithm designed in [Chapter 4](#) performs well and is able to link fingerprints. We monitored the time taken by both fingerprint collection and fingerprint linking and show the impact on the user experience is negligible in most cases. These results demonstrate the relevance of browser fingerprinting when it comes to strengthen web authentication, which answers our **RQ7**. Finally, we discussed several options to strengthen our authentication scheme, such as restricting the list of *trusted networks* or the types of device that can be registered, and other features that could be used in our authentication scheme.

These experiments demonstrate browser fingerprinting can be used as a feature for risk-based authentication to improve security while having a negligible impact on the user experience.

Chapter 6

Conclusion

6.1 Contributions

The main objective of this thesis was to study how browser fingerprinting can strengthen the security of web authentication systems. I reviewed the state of the art related to mechanisms that improve the security of web authentication systems. I also presented browser fingerprinting and its current uses. I identified interesting properties concerning a browser fingerprint that are relevant for web authentication. In this thesis, I proposed 3 contributions to increase the state of the art on the uses of browser fingerprinting for web authentication.

6.1.1 FP-Redemption: Studying Browser Fingerprinting Adoption for the Sake of Web Security

Previous studies on the adoption of browser fingerprinting on the Web had not studied sensitive pages because they are often available after a certain precise set of actions, such as filling forms or adding an element to a basket. With this contribution, I investigated the following research questions:

RQ1: Are browser fingerprints collected in the wild, on pages that process sensitive data that needs to be protected?

RQ2: How are browser fingerprints used to protect user accounts and websites against stolen credentials and cookies hijacking?

I targeted 4 types of pages that process sensitive information: **Sign-in**, **Sign-up**, **Payment** and **Basket** pages. I manually studied 1,485 web pages from 446 websites, and monitored the browser fingerprinting attributes accessed by scripts via an extension. I designed and implemented a technique to classify scripts and I detected 169 fingerprinting scripts in my dataset. I showed as many fingerprinting scripts are included in sensitive pages as in

non-sensitive pages. This result allows me to answer **RQ1** in the affirmative. I analyzed the providers of these scripts and found 12 security-centered organizations that collect browser fingerprints. I designed 2 attack models, stolen credentials and cookie hijacking, and evaluated the uses of browser fingerprinting for security on these pages by testing our attack in real-world conditions. I observed one website used browser fingerprinting to enhance web security against stolen credentials by requiring an additional proof of identity when the user's fingerprint was different, but I found no website being protected against cookie hijacking. These results, which answer **RQ2**, drove my 2 other contributions to show that browser fingerprinting can be used to strengthen web authentication.

6.1.2 FP-Controllink: Studying fingerprinting under a controlled environment to link fingerprints

In this contribution, I studied browser fingerprinting in a controlled environment to increase the knowledge concerning the causes of diversity and evolution of fingerprints, and leverage this knowledge to build a browser fingerprints linking algorithm. I tackled the following research questions:

RQ3: How does the change of a hardware or a software component in a device affect its browser fingerprint?

RQ4: What is the impact of a browser update on the fingerprint and can it be anticipated or even predicted?

RQ5: How to design a browser fingerprint linking algorithm for authentication that combines efficiency and reliability?

I used desktop devices with all major OSes—WINDOWS, MACOS, LINUX—and more than 20 different mobile devices from various vendors with the major OSes—MACOS & ANDROID. I collected the state of the art attributes, as well as new ones, to study controlled fingerprints while having the ground truth concerning the hardware and software of the device. I answered **RQ3** by showing several attributes are impacted by several components of the device—hardware, OS, browser, configuration—which make them interesting to increase browser fingerprint uniqueness. I demonstrated many attributes rarely or never change when updating browsers, which make them interesting to collect stable fingerprints to link over time. Through these results, I answered **RQ4**. I used this knowledge to define weights for each attribute based on their stability and uniqueness properties. I built a linking algorithm for web authentication that compares attribute values and computes a similarity score leveraging the previously-defined weights. I collected a dataset of 952,828 fingerprints generated from 64,235 browser instances and defined 2 modes to evaluate the algorithm. The *safe mode* evaluation demonstrated the algorithm is able to properly link browser fingerprints generated by the same browser

instance. The *attack mode* evaluation showed the algorithm distinguishes browser fingerprints coming from different instances, which reduces the risk that an attacker gains access to the account. These results show a rule-based algorithm is relevant to link fingerprints in an authentication context, which answers **RQ5** in the affirmative.

6.1.3 Advanced risk-based authentication using browser fingerprinting

My third contribution studies the use of browser fingerprinting when integrated into a risk-based authentication scheme. I answered the following research questions:

RQ6: What are the user experience concerns that should be addressed in a web authentication system that uses browser fingerprinting?

RQ7: Does the usage of browser fingerprinting for authentication provide security improvements and high usability?

I designed an authentication scheme using browser fingerprinting. To answer **RQ6**, I defined several challenges regarding the security requirements and the impact on the user experience any implementation should rise up to. I implemented our authentication scheme in the existing authentication system of INRIA and explained our answers to all the challenges. I evaluated the operation of the authentication scheme on a dataset of 82 users and 250 fingerprints. I showed the linking algorithm designed in [Chapter 4](#) behaves correctly in an authentication system, and demonstrated the impact on the user experience is negligible in most cases. Based on my results, I explained browser fingerprinting is suitable to enhance web authentication with limited impact on the user experience, which answers **RQ7**. I presented the limitations of the authentication scheme, such as the possibility for an attacker to collect the browser fingerprint of the user. Finally, I discussed security improvements to strengthen the authentication scheme and counter its limitations.

6.2 Short-term perspectives

These contributions open new research questions and provide short-term perspectives that would be interesting to study.

6.2.1 Discovering new fingerprinting JavaScript attributes

New features are constantly being implemented in browsers to adapt the Web to new types of devices or to allow new types of applications. Thus, the constant evolution of browser APIs requires being continuously monitored to discover new browser fingerprinting

attributes. We believe several APIs that are in the design, development or testing phases could increase the identification potential of browser fingerprints. We provide 3 examples:

- The **Keyboard** API allows scripts to access the keyboard layout of the user. Examples of values are presented below. First, it could be used in combination with the language to detect spoofers. We guess few people would have a keyboard layout that is not configured to easily type in their language. Second, users with very specific keyboard layouts, such as DVORAK or BEPO could be identified.

Listing 6.1: Azerty keyboard results

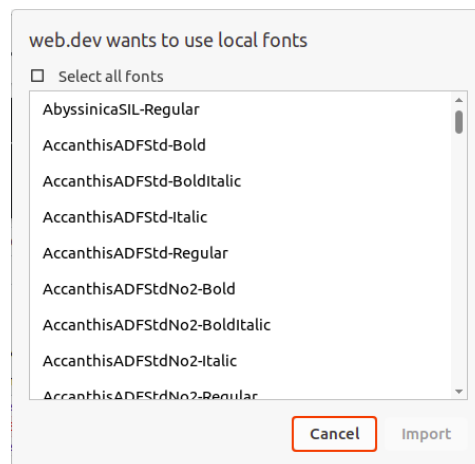
```
1 > const lm = await
    navigator.keyboard.getLayoutMap
    ();
2 > for(e of l.entries()) {
3 >   console.log(e)
4 > }
5 ["KeyQ", "a"]
6 ["KeyZ", "w"]
7 ...
```

Listing 6.2: Qwerty keyboard results

```
1 > const lm = await
    navigator.keyboard.getLayoutMap
    ();
2 > for(e of l.entries()) {
3 >   console.log(e)
4 > }
5 ["KeyQ", "q"]
6 ["KeyZ", "z"]
7 ...
```

- Several APIs are being proposed to increase functionality concerning screens, such as the **Screen Fold** API or the **Visual Viewport** API [64]. These APIs could provide additional information concerning screens and their configurations. It could help distinguish different browsing contexts for users that have several screens. This API is only available in Chrome for the moment, behind a configuration flag.
- Browser fingerprinting currently uses font enumeration via **span**'s width and height measurements. As we described in [Section 2.3.3](#), the technique can only test one font at a time. Since version 87, Chrome has integrated a **Local Font Access** API that can be used to query the list of fonts installed on the device. At the moment, the API is only available when activating the flag **#font-access**. When activated, it gives access to the **navigator.fonts.query()** function that provides a popup message to the user (see below) to allow a set of fonts to be returned. The code below queries the list of fonts and prints the information of the first font.

```
1 > const pickedFonts = await
  navigator.fonts.query();
2 // Process is blocked until the user
  allows a set of fonts to be
  accessible (see popup on the right)
3 > console.log(pickedFonts.length)
4 907
5 > console.log(pickedFonts[0])
6 FontMetadata {
7   family: "Abyssinica SIL"
8   fullName: "Abyssinica SIL"
9   italic: false
10  postscriptName: "
      AbyssinicaSIL-Regular"
11  stretch: 1
12  style: "Regular"
13  weight: 400
14 }
```



These attributes—among others—are currently under development but it would be interesting to start studying them to understand their properties and how fingerprinting can benefit from them. In particular, the questions to be answered are:

- Are they providing highly distinguishable values for a given population?
- Are they stable enough to improve the linkability of fingerprints coming from the same browser instance?
- Can these attributes be used in combination with existing attributes to detect inconsistent fingerprints?

6.2.2 Studying attacks targeting fingerprinting-based authentication systems

I mentioned in [Section 2.2.2](#) the threat phishing represents for web authentication. A user not paying attention can easily believe being on the legitimate authentication page when entering her credentials. Additional factors, features for risk-based authentication, or technique to check the user's identity are getting more used on the Web to strengthen authentication systems. In this context, I think they will be more targeted by attackers in the future. Particularly, static browser fingerprints can be easily collected without the user noticing. While phishing attacks are known to be large-scale attacks that require little effort, I believe it could be interesting to study if phishing pages show an interest in browser fingerprinting. In this context, I believe phishing can be adapted into 2 attack models when targeting an authentication system that uses browser fingerprints:

- The attacker is not aware of the use of browser fingerprinting to enhance web authentication. According to the resources he copies from the targeted authentication page, he might collect the fingerprint of the targeted user, but will not use it when trying to authenticate on the targeted website.
- The attacker knows browser fingerprinting is used to strengthen web authentication. He collects the username, password and the fingerprint of his victim and replays her fingerprint when trying to authenticate on the targeted website with the stolen username and password.

A comprehensive study on this topic should monitor fingerprint collection, their forwarding to the phishing server, and all the accesses to the compromised account to understand how these attack models are used and deployed by attackers. More formalized, it could answer the following questions:

- Are phishing attacks collecting browser fingerprints?
- Are the collected fingerprints used to bypass the fingerprinting verification setup by websites?

Attackers might use other techniques to collect users' credentials, such as data leaks. In this attack, we can imagine both passwords and fingerprints can be compromised. According to the age of the leak, the fingerprints contained in the database might have evolved since. Thus, as the attacker does not have access to the device to recover the fingerprint, he would have to calculate the evolution of the fingerprint present in the leak to successfully attack the authentication system. Based on this attack model, the following question could be studied:

- Does the evolution of a fingerprint can be calculated without having access to the device?

6.2.3 Investigating Web Assembly technology

We explained in [Section 4.2.1.1](#) that there is currently no attribute whose value only changes when the hardware of the device changes. Formalized differently, there is currently no attribute in the state-of-the-art that only identifies the hardware layer of the device. The values of attributes that identify hardware are at least impacted by other layers of the device—OS, browser, configuration. I believe WEBASSEMBLY (*Wasm*) could tackle this problem. It is a binary instruction format for stack-based virtual machines that can be deployed on the web for client applications [80]. WEBASSEMBLY can be compiled from many languages, such as C, C++ or RUST. Its integration into a web page is simple and JAVASCRIPT can be used to interact with the WEBASSEMBLY code. Its main advantage is its quickness to perform computations compared to JAVASCRIPT. I believe the following questions would be interesting to study to understand the benefits browser fingerprinting can get from this technology:

- Can the WEBASSEMBLY technology collect identifying information about the device that can be included into a browser fingerprint?
- Are the information collected via WEBASSEMBLY only hardware-related and are these information cross-browser?

6.3 Long-term perspectives

The long-term perspectives about browser fingerprinting are uncertain. First, the position of web actors towards browser fingerprinting differs. Some browser vendors, such as GOOGLE, continue to implement new APIs in their browsers to allow new applications and possibilities on the Web. While these are leveraged to propose new features to users, the privacy and security risks do not seem to be a key argument when these APIs are developed. This aspect, as well as tracking, is often driven by *economic aspects* because some actors generate profit from the Web. Oppositely, other Web actors—such as MOZILLA, BRAVE or the W3C—do not include APIs that can have unwelcome privacy effects. These actors often prioritize the *privacy and security aspects* of browsing and insist on designing "privacy-friendly" features when developing new APIs. More globally, these 2 aspects of the Web are difficult to reconcile. As fingerprinting relies on these APIs to exist, the direction taken by Web actors will determine the future of the technique. Which of these aspects will gain importance over the other in the upcoming years? The direction Web actors will take and how fingerprinting will evolve is all but certain.

Another concern is the evolution of Web security. Users are now encouraged to create accounts on many of the websites they browse. In this context, attackers will continue to challenge authentication systems and other security measures taken to protect users. As the use of secure authentication schemes, including multi-factor or risk-based approaches, is not dominant on the Web, users will continue to be vulnerable to password-based attack models. With the democratization of tools to automatically control browsers, attacks will manage to mimic more precisely human behaviour by simulating user interactions such as keystrokes, mouse movements or finger sliding on a mobile. Attacks will also be launched at larger scale, and affect many more users. In the future, what will be the major web attacks affecting users? What role, if any, will browser fingerprinting play in the security of the Web of the future?

6.4 Concluding note

The major state of the art about browser fingerprinting concerns its ability to track people. Tracking actors are using browser fingerprinting to follow users across websites, to collect information about them. These behaviors are seen as harmful and cause Web actors to react. The technique is getting more and more attention from browser vendors

themselves and regulation authorities. FIREFOX, SAFARI and BRAVE have now deployed several defense mechanisms based on filter lists or explicit authorization from the user for scripts to access attributes. Additionally, a browser fingerprint is considered as personal data. Thus, regulation authorities increase the rules around this technique to ensure user data are protected and collected with the users' consent. This global state of mind of the Web actors about the harmful uses of browser fingerprinting might be partly erroneous. While it has been shown that fingerprinting is used for tracking, I demonstrate it can be used for virtuous purposes by extending the state of the art concerning the use of browser fingerprinting for web authentication. I hope these contributions will help to consider browser fingerprinting not only for bad purposes but also for relevant and privacy-friendly ones.

Bibliography

- [1] Adblockplus. <https://adblockplus.org/>.
- [2] Adurey gitlab - authentication with fingerprinting demo. <https://gitlab.com/adurey/demo-fp-authentication>.
- [3] Adurey gitlab - controlled environment. <https://gitlab.com/adurey/controlled-environment>.
- [4] Adurey gitlab - fingerprinting monitoring extension. <https://gitlab.com/adurey/fp-monitor>.
- [5] Adurey gitlab - scripts classification technique. <https://gitlab.com/adurey/scripts-classification-technique>.
- [6] Anti-phishing working group - phishing activity trends report - 1st quarter 2021. https://docs.apwg.org/reports/apwg_trends_report_q1_2021.pdf.
- [7] Apple - app store review guidelines – apple developer. <https://developer.apple.com/app-store/review/guidelines/#software-requirements>.
- [8] Apple - intelligent tracking prevention. <https://webkit.org/blog/7675/intelligent-tracking-prevention>.
- [9] Apple introduces macos mojave. <https://www.apple.com/newsroom/2018/06/apple-introduces-macos-mojave>.
- [10] Audiocontext fingerprint test page. <https://audiofingerprint.openwpm.com/>.
- [11] Bad Bot Report 2021: The Pandemic of the Internet. <https://www.imperva.com/resources/resource-library/reports/bad-bot-report/>.
- [12] Blink docker. <https://github.com/plaperdr/blink-docker>.
- [13] Blue cava. <http://bluecava.com>.
- [14] Brave - a long list of ways brave goes beyond other browsers to protect your privacy. <https://brave.com/privacy-features/>.
- [15] Brave - fingerprinting protections v2: Farbling and cross-origin. <https://github.com/brave/brave-browser/issues/8787>.

- [16] Brave - how do i manage flash audio & video. <https://support.brave.com/hc/en-us/articles/360018163151-How-do-I-manage-Flash-audio-video->.
- [17] Brave, fingerprinting, and privacy budgets. <https://brave.com/brave-fingerprinting-and-privacy-budgets/>.
- [18] Bug 1517: Reduce precision of time for JavaScript. <https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517>.
- [19] Canvas fingerprinting on the web. <https://antoinevastel.com/browserfingerprinting/2019/02/19/canvas-fingerprint-on-the-web.html>.
- [20] Chrome - autoplay policy changes. <https://developers.google.com/web/updates/2017/09/autoplay-policy-changes>.
- [21] Chrome - manifest : Web accessible resources. https://developer.chrome.com/docs/extensions/mv3/manifest/web_accessible_resources/.
- [22] Chrome - stable release: Google chrome is out of beta! <https://chromereleases.googleblog.com/2008/12/stable-release-google-chrome-is-out-of.html>.
- [23] Chrome status - feature: Multi-screen window placement. <https://chromestatus.com/feature/5252960583942144>.
- [24] Chrome status - return empty for navigator.plugins and navigator.mimetypes. <https://www.chromestatus.com/feature/5741884322349056>.
- [25] Chromium - speeding up chrome's release cycle. <https://blog.chromium.org/2021/03/speeding-up-release-cycle.html>.
- [26] Cli browser. <https://line-mode.cern.ch/>.
- [27] Coinbase. <https://www.coinbase.com>.
- [28] Disconnect tracking protection. <https://github.com/disconnectme/disconnect-tracking-protection>.
- [29] Duckduckbot. <https://help.duckduckgo.com/duckduckgo-help-pages/results/duckduckbot/>.
- [30] Easylist. <https://easylist.to/easylist/easylist.txt>.
- [31] Easyprivacy. <https://easylist.to/easylist/easyprivacy.txt>.
- [32] Everything you need to know about emoji. <https://www.smashingmagazine.com/2016/11/character-sets-encoding-emoji/>.
- [33] Fingerprintjs. <https://github.com/Valve/fingerprintjs>.
- [34] Firefox - enhanced tracking protection in firefox for desktop. <https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop>.

- [35] Firefox - patch uplifting rules. https://wiki.mozilla.org/Release_Management/Uplift_rules.
- [36] Firefox - protection against fingerprinting. <https://support.mozilla.org/en-US/kb/firefox-protection-against-fingerprinting>.
- [37] Firefox - public data report. <https://data.firefox.com/dashboard/usage-behavior>.
- [38] Firefox - release process. https://wiki.mozilla.org/Release_Management/Release_Process.
- [39] Firefox bugzilla - add font.name-list.* for emoji. https://bugzilla.mozilla.org/show_bug.cgi?id=1032671.
- [40] Firefox bugzilla - add mp3 decoding support to ffvpx. https://bugzilla.mozilla.org/show_bug.cgi?id=1582271.
- [41] Firefox bugzilla - don't reveal navigator.buildid to every site on the web. https://bugzilla.mozilla.org/show_bug.cgi?id=583181.
- [42] Firefox bugzilla - flac support / create flac mediadatademuxer. https://bugzilla.mozilla.org/show_bug.cgi?id=1195723.
- [43] Firefox bugzilla - remove "always activate" and "remember this decision" flash options in firefox 69. https://bugzilla.mozilla.org/show_bug.cgi?id=1519434.
- [44] Firefox bugzilla - remove deprecated navigator.battery api. https://bugzilla.mozilla.org/show_bug.cgi?id=1259335.
- [45] Firefox bugzilla - remove registercontenthandler(). https://bugzilla.mozilla.org/show_bug.cgi?id=1398169.
- [46] Firefox bugzilla - remove web content access to battery api. https://bugzilla.mozilla.org/show_bug.cgi?id=1313580.
- [47] Firefox bugzilla - replace emoji one with a free emoji font. https://bugzilla.mozilla.org/show_bug.cgi?id=1358240.
- [48] Firefox bugzilla - ship an emoji font on windows xp-7. https://bugzilla.mozilla.org/show_bug.cgi?id=1231701.
- [49] Firefox bugzilla - webgl is (accidentally?) blacklisted for gtx1060/nouveau/ubuntu 19.04 while webrender is running fine. https://bugzilla.mozilla.org/show_bug.cgi?id=1563854.
- [50] Firefox webextensions may be used to identify you on the internet. <https://www.ghacks.net/2017/08/30/firefox-webextensions-may-identify-you-on-the-internet/>.
- [51] Gdpr - article 34 : Communication of a personal data breach to the data subject. <https://www.privacy-regulation.eu/en/34.htm>.
- [52] Googlebot. <https://developers.google.com/search/docs/advanced/crawling/googlebot>.

- [53] have i been pwned? <https://haveibeenpwned.com/>.
- [54] History of phishing. <https://www.phishing.org/history-of-phishing>.
- [55] History of the browser user-agent string. <https://webaim.org/blog/user-agent-string-history/>.
- [56] Http over tls. <https://datatracker.ietf.org/doc/html/rfc2818>.
- [57] Mdn - cookies. <https://developer.mozilla.org/fr/docs/Web/HTTP/Cookies>.
- [58] Mdn - keyboard api. https://developer.mozilla.org/en-US/docs/Web/API/Keyboard_API.
- [59] Mdn - performance.now(). <https://developer.mozilla.org/en-US/docs/Web/API/Performance/now>.
- [60] Mdn - vendor prefix. https://developer.mozilla.org/en-US/docs/Glossary/Vendor_Prefix.
- [61] Mdn - webvr api. https://developer.mozilla.org/en-US/docs/Web/API/WebVR_API.
- [62] Mozilla - overscripted web: Data analysis in the open. <https://github.com/mozilla/overscripted>.
- [63] Mozilla - the history of web browsers. <https://www.mozilla.org/en-US/firefox/browsers/browser-history/>.
- [64] Multi-screen window placement on the web. <https://github.com/webscreens/window-placement>.
- [65] Noscript. <https://noscript.net/>.
- [66] Openam. <https://github.com/OpenIdentityPlatform/OpenAM>.
- [67] Petportal. <http://fingerprint.pet-portal.eu>.
- [68] Project Emoji: The complete redesign. <https://blogs.windows.com/windowsexperience/2016/08/04/project-emoji-the-complete-redesign/>.
- [69] Qwant web crawler. <https://help.qwant.com/bot/>.
- [70] Random agent spoofer. <https://github.com/dillbyrne/random-agent-spoofers>.
- [71] Rfc 7231 - hypertext transfer protocol (http/1.1): Semantics and content. <https://tools.ietf.org/html/rfc7231>.
- [72] Safari blog - auto-play policy changes for macos. <https://webkit.org/blog/7734/auto-play-policy-changes-for-macos/>.
- [73] Swiftshader. <https://github.com/google/swiftshader>.
- [74] Thoughts on flash. https://en.wikipedia.org/wiki/Thoughts_on_Flash.

- [75] Top 10 types of phishing emails. <https://www.securitymetrics.com/blog/top-10-types-phishing-emails>.
- [76] Tor - canvas test. <https://people.torproject.org/~brade/tests/canvasTest.html>.
- [77] Tor - noscript. <https://support.torproject.org/glossary/noscript/>.
- [78] Tor release - tor browser 5.5 is released. <https://blog.torproject.org/tor-browser-55-released>.
- [79] ublock origin. <https://ublockorigin.com/>.
- [80] Web assembly. <https://webassembly.org/>.
- [81] What is a spambot. <https://www.cloudflare.com/learning/bots/what-is-a-spambot>.
- [82] Wikipedia - safari version history. https://en.wikipedia.org/wiki/Safari_version_history.
- [83] Worldwideweb browser. <https://worldwideweb.cern.ch/browser/>.
- [84] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juárez, Arvind Narayanan, and Claudia Díaz. The web never forgets: Persistent tracking mechanisms in the wild. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 674–689. ACM, 2014.
- [85] Gunes Acar, Marc Juárez, Nick Nikiforakis, Claudia Díaz, Seda F. Gürses, Frank Piessens, and Bart Preneel. Fpdetector: dusting the web for fingerprinters. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1129–1140. ACM, 2013.
- [86] Nasser Mohammed Al-Fannah and Wanpeng Li. Not all browsers are created equal: Comparing web browser fingerprintability. In Satoshi Obana and Koji Chida, editors, *Advances in Information and Computer Security - 12th International Workshop on Security, IWSEC 2017, Hiroshima, Japan, August 30 - September 1, 2017, Proceedings*, volume 10418 of *Lecture Notes in Computer Science*, pages 105–120. Springer, 2017.
- [87] Furkan Alaca and Paul C. van Oorschot. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 289–301. ACM, 2016.
- [88] Nampoina Andriamilanto, Tristan Allard, and Gaëtan Le Guelvouit. "guess who?" large-scale data-centric study of the adequacy of browser fingerprints for web authentication. In Leonard Barolli, Aneta Poniszewska-Maranda, and Hyunhee Park, editors, *Innovative Mobile and Internet Services in Ubiquitous Computing - Proceedings of the 14th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2020), Lodz, Poland, 1-3 July, 2020*,

- volume 1195 of *Advances in Intelligent Systems and Computing*, pages 161–172. Springer, 2020.
- [89] Sarah Bird, Vikas Mishra, Steven Englehardt, Rob Willoughby, David Zeber, Walter Rudametkin, and Martin Lopatka. Actions speak louder than words: Semi-supervised learning for browser fingerprinting detection. *CoRR*, abs/2003.04463, 2020.
- [90] Hristo Bojinov, Yan Michalevsky, Gabi Nakibly, and Dan Boneh. Mobile device identification via sensor fingerprinting. *CoRR*, abs/1408.1416, 2014.
- [91] Elie Bursztein, Steven Bethard, Celine Fabry, John C. Mitchell, and Daniel Jurafsky. How good are humans at solving captchas? A large scale evaluation. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, pages 399–413. IEEE Computer Society, 2010.
- [92] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. Picasso: Lightweight device class fingerprinting for web clients. In Long Lu and Mohammad Mannan, editors, *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM@CCS 2016, Vienna, Austria, October 24, 2016*, pages 93–102. ACM, 2016.
- [93] Yinzhi Cao, Song Li, and Erik Wijmans. (cross-)browser fingerprinting via OS and hardware level features. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.
- [94] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-redemption: Studying browser fingerprinting adoption for the sake of web security. In Leyla Bilge, Lorenzo Cavallaro, Giancarlo Pellegrino, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 18th International Conference, DIMVA 2021, Virtual Event, July 14-16, 2021, Proceedings*, volume 12756 of *Lecture Notes in Computer Science*, pages 237–257. Springer, 2021.
- [95] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. An iterative technique to identify browser fingerprinting scripts. *CoRR*, abs/2103.00590, 2021.
- [96] Peter Eckersley. How unique is your web browser? In Mikhail J. Atallah and Nicholas J. Hopper, editors, *Privacy Enhancing Technologies, 10th International Symposium, PETS, 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, volume 6205 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010.
- [97] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1388–1401. ACM, 2016.

- [98] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring HTTPS adoption on the web. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1323–1338. USENIX Association, 2017.
- [99] David Fifield and Serge Egelman. Fingerprinting web users through font metrics. In Rainer Böhme and Tatsuki Okamoto, editors, *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, volume 8975 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2015.
- [100] Dinei A. F. Florêncio and Cormac Herley. A large-scale study of web password habits. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 657–666. ACM, 2007.
- [101] Imane Fouad, Cristiana Santos, Arnaud Legout, and Nataliia Bielova. Did I delete my cookies? cookies respawning with browser fingerprinting. *CoRR*, abs/2105.04381, 2021.
- [102] David Freeman, Sakshi Jain, Markus Dürmuth, Battista Biggio, and Giorgio Giacinto. Who are you? A statistical approach to measuring user authenticity. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.
- [103] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In Pierre-Antoine Champin, Fabien L. Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 309–318. ACM, 2018.
- [104] Gábor György Gulyás, Dolière Francis Somé, Nataliia Bielova, and Claude Castelluccia. To extend or not to extend: On the uniqueness of browser extensions and web logins. In David Lie, Mohammad Mannan, and Aaron Johnson, editors, *Proceedings of the 2018 Workshop on Privacy in the Electronic Society, WPES@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 14–27. ACM, 2018.
- [105] Surbhi Gupta, Abhishek Singhal, and Akanksha Kapoor. A literature survey on social engineering attacks: Phishing attack. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pages 537–540, 2016.
- [106] Sjors Haanen and Tim van Zalingen. Detection of browser fingerprinting by static javascript code classification. 2018.
- [107] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean Michel Picod, and Elie Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 743–758. IEEE Computer Society, 2016.

- [108] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. *CoRR*, abs/2008.04480, 2020.
- [109] Grégoire Jacob, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. PUB-CRAWL: protecting users and businesses from crawlers. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 507–522. USENIX Association, 2012.
- [110] Hugo Jonker, Jelmer Kalkman, Benjamin Krumnow, Marc Slegers, and Alan Verresen. Shepherd: Enabling automatic and large-scale login security studies. *CoRR*, abs/1808.00840, 2018.
- [111] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. Fingerprint surface-based detection of web bot detectors. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, volume 11736 of *Lecture Notes in Computer Science*, pages 586–605. Springer, 2019.
- [112] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.
- [113] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords and people: measuring the effect of password-composition policies. In Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare, editors, *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 7-12, 2011*, pages 2595–2604. ACM, 2011.
- [114] Pierre Laperdrix, Gildas Avoine, Benoit Baudry, and Nick Nikiforakis. Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, volume 11543 of *Lecture Notes in Computer Science*, pages 43–66. Springer, 2019.
- [115] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, volume 10379 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2017.
- [116] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Trans. Web*, 14(2):8:1–8:33, 2020.

- [117] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 878–894. IEEE Computer Society, 2016.
- [118] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *30th USENIX Security Symposium*, Virtual, France, August 2021.
- [119] Song Li and Yinzhi Cao. Who touched my browser fingerprint? a large-scale measurement study and classification of fingerprint dynamics. In *Proceedings of the ACM Internet Measurement Conference, IMC '20*, page 370–385, New York, NY, USA, 2020. Association for Computing Machinery.
- [120] Jonathan R Mayer. Any person... a pamphleteer”: Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, page 85, 2009.
- [121] 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.
- [122] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In Matt Fredrikson, editor, *Proceedings of W2SP 2012*. IEEE Computer Society, May 2012.
- [123] 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.
- [124] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 820–830. ACM, 2015.
- [125] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 541–555. IEEE Computer Society, 2013.
- [126] Lukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Díaz. The leaking battery - A privacy analysis of the HTML5 battery status API. In Joaquín García-Alfaro, Guillermo Navarro-Arribas, Alessandro Aldini, Fabio Martinelli, and Neeraj Suri, editors, *Data Privacy Management, and Security Assurance - 10th International Workshop, DPM 2015, and 4th International Workshop, QASA 2015, Vienna, Austria, September 21-22, 2015. Revised Selected Papers*, volume 9481 of *Lecture Notes in Computer Science*, pages 254–263. Springer, 2015.

- [127] Lukasz Olejnik, Steven Englehardt, and Arvind Narayanan. Battery status not included: Assessing privacy in web standards. In José M. del Álamo, Seda F. Gürses, and Anupam Datta, editors, *Proceedings of the 3rd International Workshop on Privacy Engineering co-located with 38th IEEE Symposium on Security and Privacy, IWPE@SP 2017, San Jose, CA, USA, May 25, 2017*, volume 1873 of *CEUR Workshop Proceedings*, pages 17–24. CEUR-WS.org, 2017.
- [128] Aleksandr Ometov, Sergey Bezzateev, Niko Mäkitalo, Sergey Andreev, Tommi Mikkonen, and Yevgeni Koucheryavy. Multi-factor authentication: A survey. *Cryptogr.*, 2(1):1, 2018.
- [129] Thanasis Petsas, Giorgos Tsirantonakis, Elias Athanasopoulos, and Sotiris Ioannidis. Two-factor authentication: is the world ready?: quantifying 2fa adoption. In Juan Caballero and Michalis Polychronakis, editors, *Proceedings of the Eighth European Workshop on System Security, EuroSec 2015, Bordeaux, France, April 21, 2015*, pages 4:1–4:7. ACM, 2015.
- [130] Davy Preuveneers and Wouter Joosen. Smartauth: dynamic context fingerprinting for continuous user authentication. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 2185–2191. ACM, 2015.
- [131] Nils Quermann, Marian Harbach, and Markus Dürmuth. The state of user authentication in the wild. USENIX Association, 2018.
- [132] Valentino Rizzo. Machine learning approaches for automatic detection of web fingerprinting. 2018.
- [133] Valentino Rizzo, Stefano Traverso, and Marco Mellia. Unveiling web fingerprinting in the wild via code mining and machine learning. *Proc. Priv. Enhancing Technol.*, 2021(1):43–63, 2021.
- [134] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers. In *6th IEEE European Symposium on Security and Privacy (EuroSec’21)*, Vienna, Austria, September 2021.
- [135] Takamichi Saito, Kazushi Takahashi, Koki Yasuda, Kazuhisa Tanabe, Masayuki Taneoka, and Ryohei Hosoya. Tor fingerprinting: Tor browser can mitigate browser fingerprinting? In Leonard Barolli, Tomoya Enokido, and Makoto Takizawa, editors, *Advances in Network-Based Information Systems, The 20th International Conference on Network-Based Information Systems, NBIS 2017, Ryerson University, Toronto, ON, Canada, August 24-26, 2017*, volume 7 of *Lecture Notes on Data Engineering and Communications Technologies*, pages 504–517. Springer, 2017.
- [136] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. Clock around the clock: Time-based device fingerprinting. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1502–1514. ACM, 2018.

- [137] Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Encountering stronger password requirements: user attitudes and behaviors. In Lorrie Faith Cranor, editor, *Proceedings of the Sixth Symposium on Usable Privacy and Security, SOUPS 2010, Redmond, Washington, USA, July 14-16, 2010*, volume 485 of *ACM International Conference Proceeding Series*. ACM, 2010.
- [138] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 724–742. IEEE Computer Society, 2016.
- [139] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. I am robot: (deep) learning to break semantic image captchas. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 388–403. IEEE, 2016.
- [140] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita, editors, *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 329–336. ACM, 2017.
- [141] Jan Spooren, Davy Preuveneers, and Wouter Joosen. Mobile device fingerprinting considered harmful for risk-based authentication. In Juan Caballero and Michalis Polychronakis, editors, *Proceedings of the Eighth European Workshop on System Security, EuroSec 2015, Bordeaux, France, April 21, 2015*, pages 6:1–6:6. ACM, 2015.
- [142] Oleksii Starov and Nick Nikiforakis. XHOUND: quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 941–956. IEEE Computer Society, 2017.
- [143] Naoki Takei, Takamichi Saito, Ko Takasu, and Tomotaka Yamada. Web browser fingerprinting using only cascading style sheets. In Leonard Barolli, Fatos Xhafa, Marek R. Ogiela, and Lidia Ogiela, editors, *10th International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2015, Krakow, Poland, November 4-6, 2015*, pages 57–63. IEEE Computer Society, 2015.
- [144] Issa Traore, Isaac Woungang, Mohammad S. Obaidat, Youssef Nakkabi, and Iris Lai. Combining mouse and keystroke dynamics biometrics for risk-based authentication in web environments. In *2012 Fourth International Conference on Digital Home*, pages 138–145, 2012.
- [145] Thomas Unger, Martin Mulazzani, Dominik Fruhwirt, Markus Huber, Sebastian Schrittwieser, and Edgar R. Weippl. SHPF: enhancing HTTP(S) session security with browser fingerprinting. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, pages 255–261. IEEE Computer Society, 2013.

- [146] Blase Ur, Fumiko Noma, Jonathan Bees, Sean M. Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. "i added '!'" at the end to make it secure": Observing password creation in the lab. In Lorrie Faith Cranor, Robert Biddle, and Sunny Consolvo, editors, *Eleventh Symposium On Usable Privacy and Security, SOUPS 2015, Ottawa, Canada, July 22-24, 2015*, pages 123–140. USENIX Association, 2015.
- [147] Tobias Urban, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. Beyond the front page: Measuring third party dynamics in the field. In Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 1275–1286. ACM / IW3C2, 2020.
- [148] Tom van Goethem, Wout Scheepers, Davy Preuveneers, and Wouter Joosen. Accelerometer-based device fingerprinting for multi-factor mobile authentication. In Juan Caballero, Eric Bodden, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, volume 9639 of *Lecture Notes in Computer Science*, pages 106–121. Springer, 2016.
- [149] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-scanner: The privacy implications of browser fingerprint inconsistencies. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 135–150. USENIX Association, 2018.
- [150] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. FP-STALKER: tracking browser fingerprint evolutions. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 728–741. IEEE Computer Society, 2018.
- [151] Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Xavier Blanc. FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers. In Oleksii Starov, Alexandros Kapravelos, and Nick Nikiforakis, editors, *MADWeb'20 - NDSS Workshop on Measurements, Attacks, and Defenses for the Web*, San Diego, United States, February 2020.
- [152] Rafael Veras, Christopher Collins, and Julie Thorpe. On semantic patterns of passwords and their security impact. In *21st Annual Network and Distributed System Security Symposium, NDSS, 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.
- [153] Rick Wash, Emilee J. Rader, Ruthie Berman, and Zac Wellmer. Understanding password choices: How frequently entered passwords are re-used across websites. In *Twelfth Symposium on Usable Privacy and Security, SOUPS 2016, Denver, CO, USA, June 22-24, 2016*, pages 175–188. USENIX Association, 2016.
- [154] Stephan Wiefeling, Markus Dürmuth, and Luigi Lo Iacono. More than just good passwords? A study on usability and security perceptions of risk-based authentication. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, pages 203–218. ACM, 2020.

- [155] Stephan Wiefeling, Luigi Lo Iacono, and Markus Dürmuth. Is this really you? an empirical study on risk-based authentication applied in the wild. In Gurpreet Dhillon, Fredrik Karlsson, Karin Hedström, and André Zúquete, editors, *ICT Systems Security and Privacy Protection - 34th IFIP TC 11 International Conference, SEC 2019, Lisbon, Portugal, June 25-27, 2019, Proceedings*, volume 562 of *IFIP Advances in Information and Communication Technology*, pages 134–148. Springer, 2019.
- [156] David Zeber, Sarah Bird, Camila Oliveira, Walter Rudametkin, Ilana Segall, Fredrik Wollén, and Martin Lopatka. The representativeness of automated web crawls as a surrogate for human browsing. In Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 167–178. ACM / IW3C2, 2020.

