Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Browser fingerprinting

Master's Thesis

2018
Peter Hraška

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Browser fingerprinting

Master's Thesis

Study programme:	Informatics
Study field:	2508 Informatics
Department:	Department of Informatics
Supervisor:	RNDr. Michal Forišek, PhD.

Bratislava, 2018
Peter Hraška

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Peter Hraška
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Browser Fingerprinting
Browser fingerprinting

Anotácia: Browser fingerprinting je súbor metód používaných na identifikáciu unikátnych používateľov webovej stránky pomocou údajov poskytovaných použitým prehliadačom, a to vrátane postranných kanálov, teda údajov, ktoré neboli na tento účel zamýšľané.

Táto diplomová práca má nasledujúce ciele:
1. Preskúmať a spracovať existujúci výskum v oblasti browser fingerprintingu.
2. Vybrať vhodnú podmnožinu používaných prístupov, prípadne ju doplniť vlastnými návrhmi.
3. Vybrané prístupy implementovať a vhodne prakticky otestovať ich spoľahlivosť a robustnosť.
4. Vyvodiť závery o vhodnosti jednotlivých testovaných prístupov
5. Navrhnúť spôsoby, akými sa používateľ vie proti technikám browser fingerprintingu brániť. Analyzovať vhodnosť navrhnutých spôsobov.

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 15.12.2016

Dátum schválenia: 16.12.2016

prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

....

študent vedúci práce

Acknowledgement

I would like to thank my supervisor, Michal Forišek, for the time and insightful feedback, my family and friends for always being there for me, my classmate Askar Gafurov for the brainstorming sessions, Matej Kraj ovi for always sharing relevant articles with me, and my sister, Lucia Hrašková, for reviewing my grammar.

Abstrakt

Browser fingerprinting je metóda identifikácie unikátnych webových prehliada ov pomocou údajov, ktoré webové prehliada e poskytujú. Príkladmi takýchto údajov sú rozlíšenie obrazovky, zoznam nainštalovaných pluginov, jazyk systému a mnohé alšie. Zbieraním a následným porovnávaním hodnôt browser fingerprintov vieme ur i , i pochádzajú z toho istého prehliada a alebo nie. V tejto práci popisujeme našu implementáciu skriptu, ktorý zbiera informácie o webovom prehliada i vrátane najpokro ilejších údajov, akými sú audio fingerprint a canvas fingerprint. Pomocou tohto skriptu sme zozbierali 566,704 browser fingerprintov a na týchto dátach sme skúmali efektivitu tejto metódy identifikácie. 65% nami zozbieraných browser fingerprintov pochádza z mobilných zariadení, o z našej práce robí prvú, ktorá dokázala analyzova efektivitu tejto metódy na mobilných zariadeniach. V tejto práci ukazujeme, že miera presnosti identifikácie prehliada ov na po íta och je vä šia, než na mobilných zariadeniach.

K ú ové slová: fingerprinting, browser, fingerprint, ochrana súkromia, identifikácia

Abstract

Browser fingerprinting is a method of web browser identification based on information provided by each web browser, such as the screen size, the list of installed plugins, system languages, and others. Collecting and comparing browser fingerprints allows us to determine whether they come from the same browser or not. In this thesis, we implement a state-of-the-art browser fingerprinting script that also includes extraction of the canvas fingerprint and the audio fingerprint. We collected a dataset consisting of 566,704 browser fingerprints, and used it to analyze the accuracy with which users can be identified using this method. The fact that 65% of the fingerprints in our dataset originate from mobile devices allowed for this thesis to be the very first large-scale analysis of how e ective browser fingerprint identification is on mobile devices. We discovered, that browser fingerprint identification is more e ective at identifying desktop browsers than browsers used in mobile devices.

Keywords: browser fingerprinting, browser, fingerprint, online privacy, identification

Contents

Introduction 1

1 Related work and theory 3
- 1.1 Browser identification . . . 3
- 1.2 Browser features . . . 5
- 1.3 Fingerprinting smartphones . . . 6
- 1.4 Definitions . . . 8

2 Aim of this work 10

3 Features 12
- 3.1 Display properties . . . 14
- 3.2 Browser features . . . 17
- 3.3 System properties . . . 22
- 3.4 Hardware properties . . . 27
- 3.5 HTTP Headers . . . 30
- 3.6 Orthogonal features . . . 31
- 3.7 Omitted features . . . 37

4 Datasets and feature collection 39
- 4.1 Sources of data . . . 39
- 4.2 Implementation . . . 41

5 Browser fingerprinting prevention 44
- 5.1 Fingerprint with common values . . . 44
- 5.2 Randomizing browser values . . . 45
- 5.3 Blocking fingerprinting scripts . . . 46
- 5.4 Response of browser developers . . . 46
- 5.5 GDPR in context of browser fingerprinting . . . 47

6 Results and discussion 49
- 6.1 Dataset description . . . 49

6.2 Entropy 50
6.3 Anonymity set sizes 53
6.4 Change of fingerprints in time 56
6.5 Entropy in error descriptions 58
6.6 Minimal fingerprint 58

Conclusions 61

Appendix Appendix A - Most typical fingerprint 66

Appendix Appendix B - Contents of the attached CD 68

List of Figures

3.1 Pixel ratio example 16
3.2 Detecting fonts by comparing their width and height 26
3.3 Example of di erent font families 26
3.4 Final canvas fingerprint image 32
3.5 13 ways to render 20px Arial 33
3.6 Demonstration of anti-aliasing 34
3.7 A shape filled using the even-odd rule. 35
3.8 12 styles for a single emoji 36
3.9 Audio fingerprint configuration 37

6.1 Distribution of devices in our dataset 50
6.2 Distribution of fingerprints, as observed in our dataset 54
6.3 Anonymity set sizes of various devices 55
6.4 Fingerprint change as a function of time 56
6.5 Fingerprint change for each device type separately 57

List of Tables

3.1 Overview of all browser features used in our browser fingerprinting script and their inclusion in a previous large-scale browser fingerprint analysis. For features inspired by other work, we also present the source. 13
3.2 List of all the features with example values 15
3.3 Example of di erent date formats in web browsers 23

6.1 Normalized entropy of all features for each device type 52
6.2 Entropy with and without error description 58
6.3 Highest achievable entropy for the given number of features 59

Introduction

Browser fingerprinting is a method of web browser identification using the features most browsers make available, such as the screen size, the list of plugins, system languages and others. Collecting and comparing browser fingerprints allows us to determine whether they come from the same web browser on the same device or not. The same browser on the same device is typically used exclusively by a specific user. Browser fingerprinting is therefore often considered as a form of user identification.

This work implements a browser fingerprinting script that contains the most advanced browser fingerprint features, including audio and canvas fingerprint. By combining our knowledge and ideas with existing work, we have created a script that reveals more information about web browsers than all existing implementations. With 566,704 browser fingerprints collected, our dataset is larger than the datasets analyzed in any other study on the topic of browser fingerprinting.

While most of the datasets found in other studies were collected from websites spreading awareness about browser fingerprinting and online privacy, which might have introduced a bias to their data, our dataset was collected from a real world web application. 65% of the fingerprints in our dataset originate from mobile devices, which enabled us to perform the first large-scale analysis on the use of browser fingerprinting on mobile devices.

Older studies agree that browser fingerprints on mobile browsers are more uniform and therefore harder to identify. A recent study by Laperdrix et al.. [24], however, suggested that using more advanced browser features, such as the canvas fingerprint, might make mobile browsers easier to distinguish than browsers on desktop devices. Our work shows that many browser features do, indeed, provide more information on mobile devices than on desktops. Nevertheless, fingerprints on mobile browsers are, overall, more uniform, which makes them harder to identify. For comparison, 73% of fingerprints we collected from Windows browsers were unique, while only 34% of iPhone browser fingerprints were unique.

Our results demonstrate, inter alia, that the change rate of fingerprints is significantly higher on iPhone browsers than on any other device, and that 12 out of 31 browser features can be removed from our dataset with almost no loss in entropy of the fingerprint distribution.

The following chapter summarizes the results of previous studies of browser fingerprinting. Chapter 2 outlines the goals of this work. In Chapter 3, we list all the features we use to identify browsers, and explain how we have implemented their extraction. In Chapter 4, we describe the datasets we used. Chapter 5 provides an overview of how identification by browser fingerprinting can be prevented. Chapter 6 presents our results and findings. Lastly, we conclude accomplishments of this thesis in Chapter 6.6.

Chapter 1

Related work and theory

The first research paper that described browser fingerprinting as an identification technique was written by Eckersley [17] in 2010. To demonstrate to the public how this form of identification can be used and abused, he created an online project called Panopticlick. This project gained a lot of attention through social media sites like Twitter, Facebook and Reddit, collected 470,161 browser fingerprint samples and sparked a lot of interest in the topic of web privacy. In this chapter we are going to describe a few studies related to our work and explain a few concepts and terms that will be used in the following parts of this thesis.

1.1 Browser identification

We only found three works that were able to collect and analyze the order of magnitude in hundreds of thousands of browser fingerprint samples. These works helped us choose a direction that had not been explored before and provided guidance on which methods to facilitate, and what to avoid.

Panopticlick

First of these works was a paper written by Eckersley [17]. With 470,161 browser fingerprint samples collected from `panopticlick.eff.org`, Eckersley was able to observe that the distribution of these fingerprints contains at least 18.1 bits of entropy. This means that if we pick a random browser fingerprint, only one in 286,777 other browsers will have the exact same fingerprint. He also observed that among the browsers with

Flash or Java plugins installed, the distribution can contain up to 18.8 bits of entropy. Eckersley was able to achieve these results by collecting just 10 different browser feature values.

Eckersley continued by analyzing how accurately browsers can be identified despite differences between two fingerprints of the same browser. The data in this project was collected anonymously, unless the user allowed it to store a unique cookie in their browser to pair all of the user's browser fingerprints together. In total, 8,833 visitors of the website agreed to do that and Eckersley found that even a simple algorithm can be accurate in recognizing changes of fingerprints. His algorithm compared two browser fingerprints as strings and if they were at least 85% similar, they were treated as if they came from the same browser. This algorithm had 99.1% accuracy, while the false positive rate was 0.86%.

AmIUnique.org

Laperdrix et al. launched this website in 2014 and published their first scientific publication in 2016 [24]. They were able to collect and analyze almost 119,000 fingerprint samples. While Panopticlick was their main inspiration, these researchers introduced a few major improvements with their implementation. In contrast to Panopticlick, they collected 17 rather than 10 browser feature values, including canvas fingerprint, HTTP headers, and hardware information. They were the first to collect and analyze canvas fingerprint on a large scale and were able to prove that this technique works better on smartphones than on desktop machines. Prior to this study, the consensus was that smartphone browsers are far less unique than desktop browsers. Laperdrix et al. observed that, with their implementation, 81% of mobile fingerprints were unique. They also mentioned that while Java and Flash are significant sources of entropy in browser fingerprinting, they are slowly disappearing from the web. That might affect browser fingerprinting in the future.

A study with real-world dataset

The third and final work we would like to mention in this section is a paper written by Erik Flood and Joel Karlsson from Sweden [20]. To our knowledge, their dataset is the only one containing real-world data and not data collected from a website informing about browser fingerprinting and privacy issues connected to this technique. They used a large network of corporate websites to collect browser fingerprints and used machine learning to identify web browsers. Their set of collected features was very similar to

the one used in Panopticlick, with a few additions like calculating round-trip time and clock error. All of these new features, however, turned out to have an insignificant impact on identification accuracy.

They attempted to use machine learning for more accurate identification. However, their main conclusion was that while machine learning can bring a slight improvement in accuracy of browser identification, it requires a lot of additional computational power, and the static comparison of browser fingerprints ultimately seems to be the best choice. They also noted that if one decides to use machine learning to identify browsers, it is a good idea to partition the data, so that a custom classifier can be used for each OS. Lastly, according to their results, browser features that identify browsers with the highest accuracy include system fonts, browser plugins, and user-agent strings.

1.2 Browser features

There are a few browser features that caught our interest, namely audio and canvas fingerprint, and a few projects related to browser features and JavaScript APIs that are not necessarily related to browser fingerprinting but helped us understand various browser features. Thanks to these, we were able to implement them in our browser fingerprinting script, and improve existing implementations.

In 2016, in a work called "A 1-million-site Measurement and Analysis" [19], Steven Englehardt and Arvind Narayanan crawled 1 million websites to measure and analyze the methods used to track users online at that time. Most notably, in their research they discovered a new technique of browser fingerprinting being used on several websites that uses Audiocontext API, a method that had never been described in a paper before (see 3.6 to learn more about audio fingerprint). To demonstrate this new technique, they created `https://audiofingerprint.openwpm.com/`, a website that collects and displays the audio fingerprint of the visitor's browser, and they also collect these fingerprints. As of 31st March, 2018, they do not store fingerprints anymore, though they have not published any kind of results from using audio fingerprints yet.

Another of their discoveries was that, to identify browsers, many sites use the canvas fingerprint alone. This indicates how powerful information gained through this technique can be. See Section 3.6 for a detailed explanation of how canvas and audio fingerprint works, and how we implemented them in our browser fingerprinting script.

There are several online sources that helped us understand and implement browser APIs and features. Firstly, MDN Web Docs [8], Mozilla's take on documenting web

technologies such as JavaScript, CSS, and HTML. This documentation contains information about all of the browser features, APIs, and interfaces, including links to W3C specification documents related to the feature.

Another great source of information about specific differences in browsers is Modernizr [9], a JavaScript library for detecting available browser features. Its intended use is to help developers show relevant content on a website if a certain feature is not available in a browser. For example, if a website wants to display SVG graphics but SVG is not available, this website can choose to display the same content in a different way, or notify the user that their browser lacks some essential features. We did not use Modernizr this way, but thanks to its detailed documentation and open-sourced code, we were able to understand how browsers behave in certain situations, and use this information to make our browser fingerprinting script more accurate.

The last project we would like to mention in this section is called Fingerprintjs2 [2]. It is an open source browser fingerprinting script that contains detection of many browser features useful for the purpose of browser fingerprinting. Parts of this project inspired our implementation and we made sure to contribute back to this project whenever we found a part of it that we knew how to improve.

1.3 Fingerprinting smartphones

Ever since the first studies on browser fingerprinting, smartphones and smartphone browsers have been considered more difficult to fingerprint than other devices and their browsers. Eckersley [17] concluded his study by saying that iPhone and Android browsers are significantly more uniform and harder to fingerprint than desktop browsers, for the following reasons: the small variety of plugins on smartphone browsers, the lack of cookie control options, more uniform user-agent strings, and the absence of Flash on iPhone devices, which meant that he could not collect the list of fonts from these devices.

Similarly, Erik Flood and Joel Karlsson [20] stressed that the lack of Flash on iPhones also makes them significantly harder to fingerprint. They believe that the reasons why handheld devices are harder to fingerprint are: limitations in what software can be installed, automatic system and application updates and frequent changes in screen resolution due to screen rotation.

A study that introduced the canvas fingerprinting technique to the public performed by Mowery et al. [25] was only able to collect 294 samples of browser fingerprints, with

only 3 of them collected from smartphones (1 Android and 2 iPhone devices).

Laperdrix et al. [24] were able to implement detection of all the browser features from Panopticlick research, add features like canvas fingerprint to their list, and collect tens of thousands of samples. In their study, they were able to confirm that the list of fonts and the list of browser plugins - the two features that are the most powerful in identifying desktop browsers - are practically unusable for smartphone browser identification. Conversely, they found that smartphones have very rich and revealing user-agent strings, and that the canvas fingerprint technique works better on smartphones than on desktops in terms of identification. The latter is mainly due to the diversity of emojis on smartphones, which they included in the canvas fingerprint. 35% of the 7,416 Android browser fingerprint samples, and 9% out of 5,335 iOS samples were unique. According to Laperdrix et al., this significant di erence is due to the wealth of Android smartphone models available on the market.

Sensor fingerprinting

A number of studies took advantage of hardware sensors, present on almost every smartphone, for their identification. Nakibly et al. [26] point out that the emergence of the HTML5 standard provides an opportunity to identify smartphones using their hardware properties. Using GPU, camera, microphone, motion sensor, battery, and GPS information, they were able to develop a fingerprinting technique that yields 5.14 bits of entropy. However, they do not specify how they collected their dataset, and how big it is. Using a similar approach, but expanding the list of sensors being used, Bojinov et al. [15] from Stanford University were able to correctly identify 58.7% of the 3,583 devices in their dataset. They estimated that their approach can yield 7.5 bits of entropy, making it a robust one. Lastly, Jakobsson et al. [21] introduced a notion of implicit authentication for mobile devices using data such as the typing pattern and rhythm, location, times active, and voice. They clustered the data they had collected to explain how it can be used to implicitly authenticate users based on their actions.

All of these studies suggest that the use of hardware information can greatly improve the accuracy of browser fingerprints. Nevertheless, collecting most, if not all, of this data requires user consent prior to being able to access them via the web browser. For example, a user needs to explicitly allow a website to use their GPS data when trying to determine their location. Because most websites and web applications do not have a real use for such data, other than for browser fingerprinting purposes, requiring users to accept prompts to access their hardware sensors is not feasible in the real world.

Since we are collecting data from a real-world web application, we cannot ask users to accept the use of these sensors. We thus decided not to use any sensor data in our fingerprint implementation.

1.4 Definitions

This section explains few terms used throughout the thesis that readers may not be familiar with.

Document Object Model

Document Object Model (DOM), is a cross-platform interface for describing HTML and XHTML documents as tree structures consisting of objects. In HTML DOM, Elements represent paragraphs, headings, divs, and other HTML elements.

Browsers and operating systems

Whilst we expect readers to be familiar with the terms "web browser" and "operating system", we would like to list the most popular ones in order to make sure that we can refer to them by their names. As for operating systems, we will most frequently talk about Windows, MacOS, Android, and iOS. MacOS and iOS are desktop and mobile operating systems, respectively, created by Apple. Windows is a popular operating system mostly used on desktop machines, and Android is an operating system for smartphones currently maintained by Google.

Among the most popular web browsers are Chrome (by Google), Firefox, Opera, and Safari (by Apple). All of them are available on both desktop and smartphone operating systems. Safari, however, is only available on MacOS and iOS after its Windows version was discontinued in 2012.

Entropy

To measure how accurate a browser fingerprinting method is within a given dataset, we will use Shannon's information entropy and refer to it as entropy. The higher the entropy, the better the method is at identifying unique browsers. When the entropy is lower, it means that two or more distinct browsers were identified as the same browser.

The entropy H of a discrete random variable X, with possible values $\{x_1, x_2, ..., x_n\}$, and a probability mass function $P(X)$ is:

$$H(X) = \sum_{i=1}^{n} P(x_i) log_b P(x_i)$$

Chapter 2

Aim of this work

First research on the topic of browser fingerprinting was performed by Peter Eckersley [17] in 2010. However, internet evolves fast and a lot of new studies have emerged since. Several important trends and changes with a significant effect on the future of browser fingerprinting have appeared.

Many studies, such as the one by Peter Eckersley, have shown that the use of browser and system information provided by Flash can greatly improve the accuracy of browser identification via fingerprinting. However, Adobe - the company behind Adobe Flash technology - officially announced that end of support for Flash will take place at the end of the year 2020.

Adobe Flash, a web technology used in interactive web applications even before the HTML5 era, is known for its major flaws in core architecture. The latter have resulted in numerous security holes that required regular patches. Flash had long not been considered safe, and was hard to maintain for Adobe. The end of support for this technology therefore did not come as a surprise.

In browser fingerprinting, Adobe Flash is currently the only technology that can extract the full list of fonts installed on the device. Several research papers have suggested that the list of fonts of a device is a powerful piece of information when it comes to browser identification. Using this technique, Pierre Laperdrix et al. [24] were able to collect 31,007 unique lists of fonts from 36,202 unique devices. Not only can a list of fonts be extracted via Adobe Flash, Pantopticlick research [17] also discovered that this list of fonts is always returned in the same order on the same machine. However, the order is not sorted alphabetically or sorted in any other rational way. This is a considerable source of entropy because of the number of different ways in which hundreds of fonts can be ordered. This source of entropy, however, is an unnecessary one.

While full list of fonts is a powerful piece of information, techniques for obtaining them will become deprecated in upcoming years. Several browsers already require user approval before Adobe Flash can be executed, which is not a valid option when trying to invisibly fingerprint web browsers.

Instead of Flash, developers choose JavaScript more and more often as the go-to technology for creating rich web applications. JavaScript frameworks such as Angular, React, Vue.js, and others, have recently seen an explosion in popularity. In their 2017 report, Github, a popular git-repository hosting service, reported JavaScript to be the most popular programming language in terms of opened pull requests [6]. We believe that, in the foreseeable future, JavaScript will maintain and strengthen its popularity among web application developers. It is therefore sensible to study how accurate JavaScript can be in browser identification.

Another visible trend in the use of web applications is the ever increasing percentage of users that use smartphones to access the web. As mentioned in Section 1.3, most of the cited studies have stated that fingerprinting on smartphones is less accurate than fingerprinting desktop computers. However, a recent analysis of data from smartphone fingerprints performed by Laperdrix et al. [24] has found that using advanced features like canvas fingerprints might, in fact, work better on smartphones. In their dataset, they only had around 17,370 samples of smartphone fingerprints. We, on the other hand have a dataset with 566,704 browser fingerprint samples, 65% of which are from smartphone devices. This gave us an excellent opportunity to focus on the use of browser fingerprinting techniques on smartphones.

Our work is currently the most accurate analysis of its kind, thanks to having access to the biggest dataset of browser fingerprints to date. Key contributions of our work include:

- Extensive research on the use of browser fingerprinting techniques on smartphones.
- Implementation of a state-of-the-art browser fingerprinting script based on knowledge from previous studies, improved with our own findings.
- First public results on the use of audio fingerprints for browser fingerprinting.
- A review of the state of online privacy.
- Unlike the datasets of all existing studies, the dataset analyzed in this thesis was collected from a real-world environment.

Chapter 3

Features

In this chapter, we will describe all the data we collect as features in our browser fingerprint implementation. An extensive review of related studies, as well as relevant projects, libraries, and articles on the topic of browser fingerprinting, allowed us to put together a list of the most promising browser fingerprinting features. We omitted features that were proven to be unreliable, or required too much computing power or processing time to be acquired. Many of the browser features we have included had not been tested on a large scale before. Including them enabled us to determine their e ciency. With each feature implemented in our browser fingerprinting script, we iterated through many di erent implementations, always starting with our own, and testing their e ciency on a small data sample. We then incorporated the results of our observations in the final implementation of the browser fingerprinting script. Table 3.1 shows whether each individual browser feature found in our final script had been tested on a large scale before. When there was an external source that inspired us while we were looking for the most e cient implementation of a browser extraction script, we made sure to list it in this table. When building our browser fingerprinting script, we made sure to stay within the boundaries of what a real-world implementation of such a script might look like.

All methods of browser fingerprinting rely on the fact that some methods are implemented in a di erent way on di erent systems, and some may not be implemented at all. When a feature is not available, all of the implementations of browser fingerprint we have seen will simply discard any information the browser responded with, and store their value as `undefined`, or `null`. Conversely, based on the assumption that, under the same conditions, di erent browsers will respond di erently, we decided to always store the error message generated by the browser.

Table 3.1: Overview of all browser features used in our browser fingerprinting script and their inclusion in a previous large-scale browser fingerprint analysis. For features inspired by other work, we also present the source.

	Panopticlick	AmIUnique	Source
Display properties			
Screen size	✓	✓	
Available size			
Color depth	✓	✓	
Pixel ratio			
Browser features			
AdBlock		✓	
Cookies enabled	✓	✓	
Do Not Track (DNT)		✓	
Plugins	✓	✓	
IE plugins			FingerprintJS2 [2]
Indexed database			
Local storage		✓	
Session storage		✓	
Binary Behaviors			FingerprintJS2 [2]
User-agent	✓	✓	
System properties			
CPU class			
Timezone	✓	✓	
Languages	HTTP only	HTTP only	FingerprintJS2 [2]
Installed fonts	Flash only	Flash only	FingerprintJS2 [2] - fixed and improved
Date format			
Tanh			Browserprint [1]
Hardware properties			
Hardware concurrency			
Touch compatibility			Panopticlick
WebGL vendor		✓	AmIUnique [24]
WebGL renderer		✓	AmIUnique [24]
Platform		✓	
HTTP headers			
Accept	✓	✓	
Accept encoding	✓	✓	
Accept language	✓	✓	
User-agent			
Orthogonal features			
Canvas		✓	FingerprintJS2 [2] - improved
Audio			openwpm.com [28] - fixed for iPhones and updated deprecated parts of JavaScript code

In order to develop a state-of-the-art script for browser fingerprint extraction, we combined our findings with knowledge from all related studies and libraries we were able to find. We designed our script to be precise, consistent, lightweight, and quick.

The complete list of the 31 features we have implemented, along with an example of their values, can be found in Table 3.2.

3.1 Display properties

In this category, we will list all display-specific and visual features. Access to this information is useful when websites and web applications want to respond to user-specific viewing options. An example of the use of this information is showing images with higher resolution to users with higher density displays, and saving bandwidth for users with lower density displays by providing them with lower resolution images.

This category provides us with information that can reliably be used as browser fingerprint features, since these rarely change during regular use of a web browser. With a few exceptions, the value of these features can only be changed by changing the display or changing low-level display settings.

Screen size

Trough JavaScript, the total width and height of the user's screen, in pixels, can be accessed. This also includes the width and/or height of the taskbar (for Windows users) and application dock (for MacOS users) or similar parts of the OS. This size is therefore not representative of what is accessible to the web browser window, unless the latter is in full-screen mode. Except for Internet Explorer, zooming of the window does not a ect these measures.

The width and height of the window can be accessed through `window.screen.width` and `window.screen.height`.

Available size

`window.screen.availWidth` and `window.screen.availHeight` are similar to screen size. They return the dimensions of the portion of the screen available to the web

Table 3.2: List of all the features with example values

	Value example
Display properties	
Screen size	1440x900
Available size	1440x827
Color depth	24
Pixel ratio	2
Browser features	
AdBlock	true
Cookies enabled	true
Do Not Track (DNT)	false
Plugins	{name: Chrome PDF Plugin, fileName: internal-pdf-viewer, description: Portable Document Format, mimeType: ...}, {...}
IE plugins	empty
Indexed database	true
Local storage	true
Session storage	true
Binary Behaviors	false
User-agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 ...
System properties	
CPU class	undefined
Timezone	-120
Languages	en-US, sk-SK, sk, en
Installed fonts	Andale Mono; Arial; Arial Black; Arial Hebrew; Arial Narrow; Arial Rounded MT Bold; Arial Unicode MS; Comic Sans MS;
Date format	01/01/1970, 01:00:00
Tanh	-1.4214488238747245
Hardware properties	
Hardware concurrency	4
Touch compatibility	0,false,false
WebGL vendor	Intel Inc.
WebGL renderer	Intel(R) Iris(TM) Graphics 540
Platform	MacIntel
HTTP headers	
Accept	application/json, text/plain, */*
Accept encoding	gzip, deflate, br
Accept language	en-US,en;q=0.9,sk;q=0.8
User-agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) Apple...
Orthogonal features	
Canvas	875f14dcfa55c0f534b7809b0b5109d1
Audio	124.04877783898846

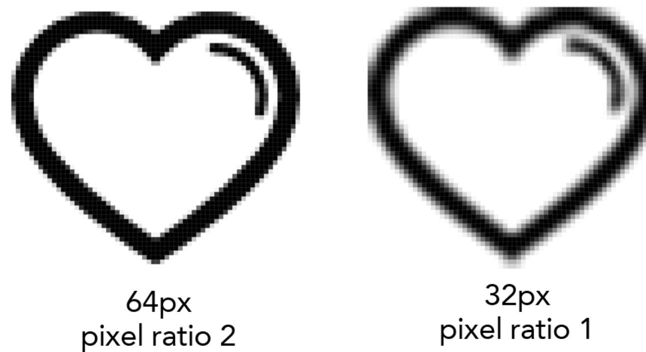

Figure 3.1: Demonstration of what two images would look like, if their displaying size was the same, but their pixel ratio was different.

browser window, in pixels. Available size thus amounts to screen size with the width and/or height of taskbars, docks etc. excluded.

Similarly to screen size, zooming the browser in or out affects available size in Internet Explorer alone.

Color depth

Color depth, also known as bit depth, is the number of bits that represent the color of each pixel on the display that contains the browser window. For example, 1-bit color depth would mean a black and white screen, 8-bit would mean each pixel can be represented with one of 256 colors. Most modern displays use 24-bit color representation, also known as True color.

The color depth of a display is accessible through `window.screen.colorDepth`.

Pixel ratio

The pixel ratio of a display is a size ratio between physical and logical pixels. For example, a pixel ratio of 2 would mean that 4 physical pixels represent a single logical pixel. Using more pixels on a screen to represent fewer logical pixels results in sharper objects for users. Figure 3.1 demonstrates what two images with the same displaying size but different pixel ratios look like.

The pixel ratio of the user's device can be accessed through `window.devicePixelRatio`.

3.2 Browser features

This section describes those browser fingerprint features collected by us that are closely tied to the web browser itself. These reflect the capabilities, settings, plugins, available data storage APIs, and similar properties of the browser.

User-agent string

The user-agent string in HTTP is a list of tokens describing the system and the browser that are being used to view the website.

Information contained in the user-agent string includes:

- system information
  - platform
  - operating system
  - CPU
- rendering engine compatibility
- browser information
  - name
  - version
  - build number

Websites can adjust content that is being sent according to this information. This often means that older browsers are served with less complex content, even though they are able to handle it properly. Web browsers thus make their user-agent strings easy to spoof, to enable users to request the entire content of a website when necessary. This might decrease browser fingerprint accuracy if the user chooses to change their user-agent to a commonly used one. By doing so poorly, however, users might create a unique user-agent capable of identifying them precisely.

AdBlock

Many users install ad-blocking plugins into their web browsers to filter advertisements and similar website content [16]. Page elements identified as advertisements are then hidden from all pages, to provide users with an uninterrupted web experience.

A report on the usage of AdBlocks [16] shows that around 11% of all internet users use some sort of an ad-blocking plugin. This number varies across di erent device types and regions, but is significant enough to be used as a browser fingerprint feature.

One example of an ad-blocking plugin is AdBlock Plus, a widespread, open-source project. Although currently available for all major web browsers across all device types, AdBlock Plus is just one among thousands available ad-blocking plugins.

While it is not possible to retrieve information about what plugins a user has installed, it is possible to detect the presence of an ad-blocking plugin by mimicking the behavior of a web advertisement. We do so by creating a page element that acts as a web advertisement (through its content, class names, and ID attribute), inserting it into the DOM, and finally checking whether it is actually present in the DOM or not. With an ad-blocking plugin is installed, such page element will be filtered out upon insertion. This allows us to determined whether the web browser has an ad-blocking plugin installed or not.

This method will not detect all ad-blocking plugins. However, on the same web browser instance, its results will be consistent, as long as the user does not add or remove plugins. Since consistency is a key property of a browser fingerprint, this method fits our needs perfectly.

Do Not Track Header

Do Not Track (DNT) is a proposed HTTP header sent with each HTTP request. It indicates the user's tracking preference, with its value being either `true` or `false`. By setting it to `true`, the user expresses their preference not to be tracked for purposes of online advertisement and personalized content.

Although still not fully standardized, the DNT header has been present in all major web browsers for a few years now. The decision to respect this user setting or not is currently up to web developers, but can be expected to become a web privacy standard in the foreseeable future.

The DNT property value can be accessed using `window.navigator.doNotTrack`.

Cookies

HTTP cookies, also known as cookies, are pieces of data stored in the browser that can be accessed by both the server and the web browser. They usually hold user-specific data, such as the login authentication token, or the items in their shopping cart.

Users may decide to disable cookies in their browser for a number of security and privacy reasons. Many users seem to believe that the latter outweigh the inconveniences and disadvantages for web personalization this brings.

By trying to store a cookie in a user's browser, we can detect whether the user has cookies enabled or not, and then use this information for browser fingerprinting purposes. Accessing this information through `navigator.cookieEnabled` is also possible, albeit not reliable, as it can easily be spoofed, and may not work in certain site-specific cases.

Local Storage

Local storage is very similar to cookies. While both are used to store data on the client-side, there are a few key di erences.

- Local storage can only be accessed by the client-side, while cookies can be read by both server and the client-side.
- Unike cookies, data stored in local storage never expires.
- Cookies can store up to 4KB of data per domain, while local storage usually has a limit of 5 MB.

We determine whether the user does or does not have local storage allowed by storing and, subsequently, removing a key-value pair into this storage. If these operations are successful, local storage is available. If they return an error, it means it is not.

Session Storage

Session storage works almost the same as Local storage, and uses the same API, with one important di erence. Session storage data is only stored until the browser window or tab is closed.

We test whether it is available on the browser using the same method we use for Local storage detection (see 3.2).

Indexed Database

An indexed database is a transactional database that works entirely within a browser. While local storage and session storage are useful for storing small amounts of data, indexed database is ideal for large amounts of structured data. It supports more complicated operations, such as search, and is useful for complex web applications, such as web email clients. Instead of transferring considerable amounts of data over the internet with each request, the data can be stored and operated on the client-side.

The availability of an indexed database in a browser can be tested by calling `window.indexedDB`.

Binary Behaviors

Binary behaviors, also known as user data, are a predecessor of local storage used in Internet Explorer. They can be used to store and load persistent data on the client-side by setting special attributes on the DOM elements. However, this data storage is unique for Internet Explorer, and obsolete since Internet Explorer version 10.

We detect the availability of binary behaviors by trying to create an element, and adding behaviors to it by calling `document.createElement("div").addBehavior`.

Plugins

Plugins are third-party libraries that can be used by the web browser to embed `<object>` or `<embed>` tag into the web page. They are mostly used for displaying animations, applets, or PDF files inside web pages.

Commonly used plugins include:

- **Adobe PDF Reader** - viewing PDF files
- **Shockwave Flash** - interactive applications and games
- **Java Applet Plug-in** - interactive components

- **Apple Quicktime** - a multimedia framework

Browser plugins are often mistaken for browser extensions, from which they, however, differ in several aspects. Browser plugins cannot affect browser behavior, cannot add browser menus, do not automatically process the content of the web page, and have to be inserted into websites. In contrast, browser extensions can affect browser behavior by filtering or altering website content, or adding new functionality to the browser.

Examples of commonly used browser extensions include:

- **AdBlock Plus** - advertisement content filtering
- **Grammarly** - checking the spelling and grammar of user input on websites
- **Momentum** - to-do list and welcome screen
- **Google Translate** - text translation

While the list of browser extensions is not accessible through JavaScript, the list of browser plugins is. Developers can use this information to show a meaningful message if a certain necessary plugin is missing. This can be useful when a website is trying to display a PDF document but the user has no plugin that enables the browser to display PDF files installed.

The list of plugins is available through `navigator.plugins`, which returns an iterable array. As mentioned above, the main reason for these plugins being exposed via JavaScript is for developers being able to check whether a user's browser can handle content on their website. This iterable array, however, exposes much more than just a list of plugin names.

`navigator.plugins` contains the following information about each plugin:

- name
- filename
- description
- supported mime types
- supported mime suffixes

It is worth mentioning that a plugin's description often contains its version number. Supported mime types is a list of types that can be handled by the plugin. We combine all of this information from all available plugins into a single JSON, which we then store for browser fingerprinting purposes.

As Eckersley [17] noticed, the amount of plugin information provided by a browser appears to be exhaustive. While knowing the exact version of the plugin may be useful for debugging purposes, it may also be dangerous in terms of privacy and security. This study also shows that information about plugins is the most powerful browser feature among the 8 they tested. According to their calculations, it provides around 15.4 bits of entropy.

Firefox decided to tackle this privacy issue by allowing users to disable enumeration of `navigator.plugins`. Applications that attempt to check the presence of a browser plugin have to query for the plugin name or mime types by exact names, rather than iterating through all of the plugins.

IE Plugins

Surprisingly, Internet Explorer conceals its plugin information similarly to Firefox. It does not return an iterable array. Instead, all the plugins have to be queried for.

However, a reasonable amount of data can still be obtained by querying for many common plugins using a predefined list. We used an existing list from the PluginDetect JavaScript library [11] for purposes of this plugin detection. We query for each one of them, and use the list of available plugins as a browser fingerprint feature.

Although the development of Internet Explorer has been discontinued, it remains one of the most popular web browsers. The extra e ort needed to distinguish Internet Explorer browsers is therefore justified.

3.3 System properties

In this section, we describe browser fingerprint features that do not depend on JavaScript or the web browser, and are closely tied to the operating system and hardware properties instead. These include various hardware information and low-level system settings that a ect the values of these features we collect.

Timezone

We can read the time zone information of a device by accessing the `getTimezoneOffset()` method on any `Date` objectt, i.e. by using `new Date().getTimezoneOffset()`. This

Table 3.3: Example of di erent date formats in web browsers

USA	12/20/2012, 8:42:47 AM
Great Britain	20/12/2012, 08:42:47
Korea	2012. 12. 20. 오전 8:42:47
Japan	2012/12/20 8:42:47
Slovakia	20. 12. 2012, 8:42:47
Czech republic	20. 12. 2012 8:42:47

method returns the time zone di erence between UTC and the date set in the host device in minutes.

For instance, if the current locale is `UTC+1`, we will get `-60` as a result. Setting the device clock to adjust for daylight savings, also a ects timezone o set, when applied. The latter is important when searching for all possible edge cases of browser fingerprint identification. Users can also easily change their timezone o set by changing the time in their system settings.

Date format

The method `toLocaleString()`, executed on any `Date` object, returns a date string in a format that respects the browser's locale (i.e. language preference). In older implementations, the format of the date string returned by this method depends entirely on its implementation.

In Table 3.3, we demonstrate how the date format di ers for di erent locales. Both implementation and string format di erences are helpful in browser identification.

Languages

A user's language preference may a ect the language in which a website will serve its content. For historical reasons, there are many di erent ways of how these languages can be retrieved. We thus try to read and store all of them in a single JSON object. If a certain implementation is not available, we simply store the `undefined` value returned by the browser in such a case instead.

Language attributes that we try to collect are:

- `window.navigator.language`

```
window.navigator.languages
window.navigator.userLanguage
window.navigator.browserLanguage
window.navigator.systemLanguage
```

Languages are always described using BCP 47 [14] language tags (e.g. en_US, ja_JP, sk_SK). They always return a single language or a list of languages ordered from most to least preferred.

Tangent

We were not able to find a study that tried calculating and storing the result of the tangent as a browser feature. We found this feature implemented in Browserprint [1], an open-source project, the aim of which is to provide the same - and better - functionality as the Panopticlick research by Eckersley [17], and to spread awareness about web privacy. We decided to implement it as one of our features, to determine how powerful it can be.

The idea behind this feature is that different browsers implement mathematical functions in different ways, and can produce similar, yet not quite the same results. This should reflect in rounding and precision differences in the returned values of the mathematical functions.

JavaScript calculates the value for the tangent using Mathematical Markup Language (MathML), a low-level specification of mathematical content on the web. However, the specification of MathML is not yet complete. Many browsers thus do not support this functionality and, to calculate its value, a CSS fallback is used instead. This brings even more diversity into the values of this feature.

We use the result of `Math.tan(-1e300)` as a value of this feature.

Fonts

As mentioned in Section 2, getting the full list of fonts installed on a system is not possible via JavaScript. However, there is a way to detect whether a given font is or is not installed on the system by only using JavaScript and CSS. Our implementation of collecting fonts is inspired by the one found in Fingerprintjs2 [2], an open-source browser fingerprinting library. We found and fixed one major flaw in this method,

which we will discuss at the end of this subsection.

The idea behind this method is to use a prepared list of fonts with various standard and non-standard fonts and to check, one by one, whether the font is or is not available in the current system. This allows us to create a list of all available fonts within our prepared list.

The technique of detecting the availability of a single font, we take advantage of several important properties of fonts in CSS:

- The width and height of a `<div>` element adjusts to the size of its inner content, by default, meaning that the exact dimensions of this element can be read.
- Any amount of fallback fonts can be set for an element. If the first one fails to apply, the second one is used and this continues until it reaches an end of its fallback fonts and then the default system fallback font is applied.
- A non-specific, generic-family fallback fonts can be used, such as `sans-serif`, which is not a font, but rather a type of a font. This way, we let the browser substitute this fallback with its preset option for these generic font families. Browsers will always substitute the same generic-family fallback with the same font.
- It is very unlikely for two texts written with the same attributes except for font family, to be the same size.

The single-font detection method works as follows:

1. Create a list of fallback fonts against which the availability of a single font can be tested. In our case, we use `monospace`, `sans-serif`, and `serif` as our fallback fonts, since they are rarely unavailable in a browser. We will refer to these as default fonts.
2. For each default font, create an element with default styles applied, and store the width and height of this element.
3. Get the width and height of an element, but with the font set to dimension that is being tested.
4. Compare the dimensions of all 3 default fonts against the font that is being tested, one by one. If these dimensions differ in one or more cases, we declare that this font is available on the system. Figure 3.2 demonstrates what such a difference looks like.

Figure 3.2: Example of two di erent font families on top of each other, with all font attributes except for font family set to the same value. One of them is Arial, a popular sans-serif font,the other one is Andale Mono, a monospaced font.

font-family: serif

font-family: sans-serif

font-family: monospace

Figure 3.3: Example of di erent font families, with all of their font attributes, except for font family, set to the same value.

There are some extremely unlikely conditions under which this method will not detect a font accurately. Nevertheless, the results are always consistent when the test ran multiple times. Getting a false negative, would require all three default fonts, as well as the font that is being tested, to have the same dimensions on a system, which is very unlikely to occur.

Figure 3.3 shows the di erence between the 3 font families we used as default fonts. A monospaced font is a font where all letters and characters occupy the same amount of horizontal space. A serif font is a font with serifs or "legs" decorating the ends of its strokes, and a sans-serif is a serif-less font. With serif and sans-serif fonts, horizontal space is proportional, not monospaced, therefore letters like "i", "l", and "I" occupy less horizontal space. For this reason, we included these letters in our testing string, mmmmmmmmmmml l i .

We found that a similar implementation, inspired by Fingerprintjs2 project, was used in a number of prior studies, and various projects spreading awareness about browser fingerprinting. When testing this method of font detection, we found the results of this method to be inconsistent. We first thought that changing the string used for font

detection might help, since it was only using 3 di erent characters. While increasing the diversity of the characters in this string improved the accuracy of our results, the latter remained inconsistent. We later noticed that the results are always consistent within a given URL address, but inconsistent between di erent websites or di erent pages of a single website. This finding led us to the discovery that di erent styles from outside of the fingerprinting script were applied to our testing element, and a ected the results of this test. To fix this, we now always perform a CSS font reset on the element before we start working with it, so no external code will ever a ect the results of this test. We discussed our findings with the creator and maintainer of Fingerprintjs2, and implemented this fix in this project as well.

Font detection is the most time-consuming task within our browser fingerprinting script. Aiming to keep our script as light and fast as possible, we reused the list of fonts used by Fingerprintjs2, which consists of 65 fonts. To increase the entropy of this feature, while keeping the number of fonts limited, we suggest finding an ideal subset of fonts to be used in this method.

Such list can be built by detecting a substantial number of fonts on a large number of devices, and analyzing this data.

3.4 Hardware properties

This section describes another category of browser fingerprint features - hardware properties. A lot of information about the hardware used to run the browser can be accessed through JavaScript and these values are often hard to change. For example, users are not able to manually replace a CPU inside their laptop. However, it is not impossible to spoof the results of these tests.

Platform

We use `navigator.platform` to get a string that represents the platform of the browser. The current specification allows for this string to be empty, but any information, including an empty string, obtained through any attribute, is useful.

Websites use this information to display appropriate content on certain devices. For example, a website might serve a simpler version of the UI for TV platforms, and a more advanced UI for desktop platforms. Another common use case is suggesting the

right version of software on the website when downloading it (i.e. a Windows build for Windows platforms and a MacOS build for MacOS platforms).

Examples of platform strings include:

MacIntel
Win32
Android
WebTV OS

CPU class

The navigator.cpuClass property returns a string that represents the class of the CPU. This property, however, is implemented in Internet Explorer only, and will only recognize these CPU classes:

68K - Motorola processor
Alpha - DEC processor
PPC - Motorola processor
x86 - Intel processor
Other - Unknown processor type

Hardware concurrency

CPUs with multiple cores and threads have existed for several years, yet not many developers have taken advantage of this additional power. In recent years, this trend has finally started to change, and software developers are now trying to take advantage of all threads when executing code, where possible. JavaScript is no di erent. JavaScript developers can facilitate multiple logical cores by splitting demanding tasks between multiple Web Workers. However, it is not useful to use more Web Workers than the number of threads available on the system.

For this reason, the navigator.hardwareConcurrency property is implemented in most modern web browsers. This property returns the number of logical cores that are available on the system. Browsers may choose to report a lower number because the browser assumes it will occupy several logical cores by itself. The value of this property will, however, always be the same on one system.

CPUs usually have multiple cores, and each core has multiple threads. The number

of cores multiplied by the number of threads on each core equals to the number of logical cores, and that is the value we get by calling `navigator.hardwareConcurrency`.

Touch compatibility

As of 2018, roughly half of web traffic is served to smartphone or tablet devices [18], which are typically fitted with touch displays. In order to be able to take advantage of touch and multi-touch gestures (gestures done with multiple fingers), it is useful for websites to know that the device supports them.

Web applications handle such events by listening to event listeners, such as `onClick` or `onTouchMove`. We try to read all available information about the touch support of the current browser, and store it as a single browser fingerprint feature.

The data we collect includes:

- `navigator.maxTouchPoints` - the maximum number of separate touch points that the touch screen is able to detect
- `msMaxTouchPoints` - same as above, but for Internet Explorer and Edge browsers
- `"ontouchstart" in window` - detecting whether "onTouchStart", the most basic touch event, is available in the browser; returns value `true` or `false`

WebGL

WebGL (Web Graphics Library) API is a JavaScript API used for rendering interactive 2D and 3D content. This allows GPU-accelerated image processing to take place inside a web browser without the use of plugins. Graphics can be displayed in a <canvas> element specified by the HTML5 standard.

In addition to the GPU-accelerated graphics processing power, the WebGL API provides JavaScript with GPU-related information, which can be used as browser fingerprint features. We read and store this data:

- Vendor - a string of the graphics card driver that contains the name of its manufacturer (e.g. `Intel Inc.`)
- Renderer - a string of the graphics card driver describing the model of the graphics card in detail (e.g. `Intel(R) Iris(TM) Graphics 540`)

In order to read this information, we need to create a temporary <canvas> element, and read it from its properties.

3.5 HTTP Headers

HTTP headers are part of the Hypertext Transfer Protocol (HTTP). They are pieces of additional information sent with each HTTP request between the server and client (where, in our situation, the client is a web browser), used to communicate the operating parameters of HTTP transactions.

This is the only category of browser fingerprint features that we collect on the server side. In our implementation, we use the HTTP POST request sent from the client side to transfer all browser fingerprint data. We read the HTTP headers of this HTTP request, and concatenate them with the rest of the features prior to storing them in a database.

In this section, we will briefly describe which HTTP headers we collect.

Accept

The HTTP Accept header is used by the client to advertise which content types (MIME types) it is able to handle. Content types can also contain a quality factor, which defines an order of preference for each content type represented by a number between 0 and 1. The server selects and uses one of these content types when serving content to this client.

Accept Encoding

To save bandwidth, content sent via HTTP can be sent in a compressed form by the server. This will only happen if both the server and the client support the same compression algorithm. The client advertises what compression algorithm it supports by setting an HTTP accept-encoding header. The server then selects and uses one of the accepted encodings.

Accept Language

Similarly to the languages mentioned in Section 3.3, the client may choose to advertise its preferred languages by setting a HTTP accept-language header. This header will contain a list of language tags with the quality factor representing the order of preference for these languages.

User-agent

Lastly, HTTP requests contain user-agent information. This header should contain information analogical to the user-agent string described in Section 3.2. However, this two are often not the same, which is why we decided to also collect HTTP user-agent information.

3.6 Orthogonal features

In this section, we describe two browser fingerprint features that we call "orthogonal". This name is appropriate because a single result of each of these two tests is a ected by many di erent factors. All previous features were, in fact, quite straightforward. Most of them simply required us to read and store a single piece of information.

If we were using all but the orthogonal features, we would join them all into a single string and generate its hash. Comparing their hashes would allow us to know whether two browser fingerprints are the same or not. The orthogonal features are similar to this but work as a single package. We use a variety of behaviors of di erent parts of these methods to bring as much entropy into a single feature as possible. The result of both of these features is a hash of their values, as their full result would be unnecessarily long.

Laperdrix et al. [24] suggested that the canvas fingerprint alone could substitute for all other features, while decreasing complexity and maintaining the entropy of such the browser fingerprinting method.

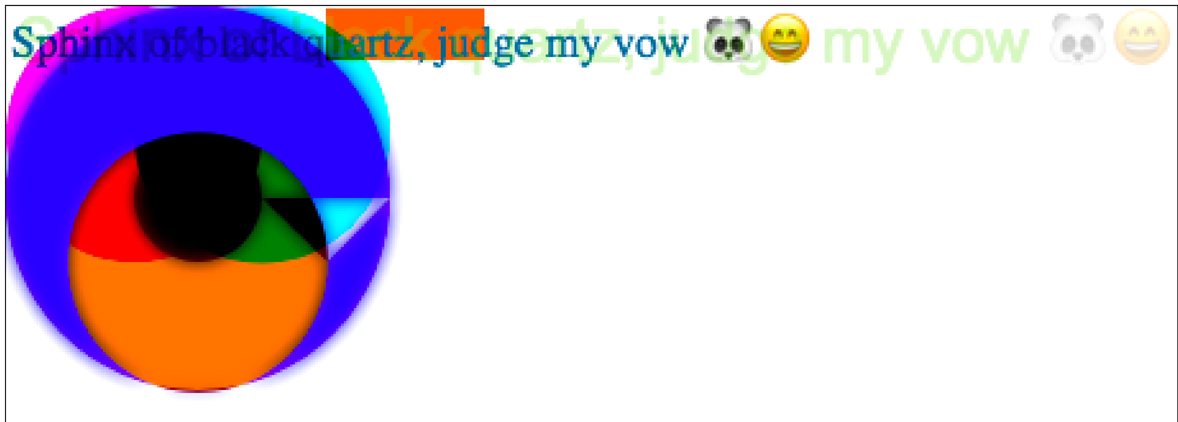

Figure 3.4: A visual representation of the image that we used in our canvas fingerprint implementation.

Canvas fingerprint

Canvas fingerprinting is browser fingerprinting by generating images using the same rules on di erent browsers, and comparing them. Rather than comparing images pixel by pixel, we compare the hash of their bitmaps, exported in base64 format. This allows us to determine whether they came from the same browser or not. The first study describing this method was published in 2012 by Keaton Mowery and Hovav Shacham [25]. Canvas fingerprinting works by o setting the canvas element far from the edges of the website's viewport, thus taking place out of the user's sight. Figure 3.4 shows an example of a final image used in our implementation.

The di erent results of the canvas fingerprinting method are due to inconsistencies between di erent systems, browsers, and implementations. We reviewed existing research and included our own findings to get as much entropy from canvas fingerprints as possible. The following text summarizes the key causes of these di erences.

Typeface inconsistency

Several typefaces (or fonts), such as Arial, Times, Helvetica, or Georgia, can be found on almost every system because they are usually part of the operating system itself. On di erent systems, however, these typefaces may di er slightly. On each system, there is at most one typeface with a given name, which will be used whenever requested by its name. To make sure we are rendering every letter of the alphabet, we insert a pangram - a sentence containing every letter of the alphabet at least once - into our image.

Keaton Mowery and Hovav Shacham collected 297 images with renders of Arial

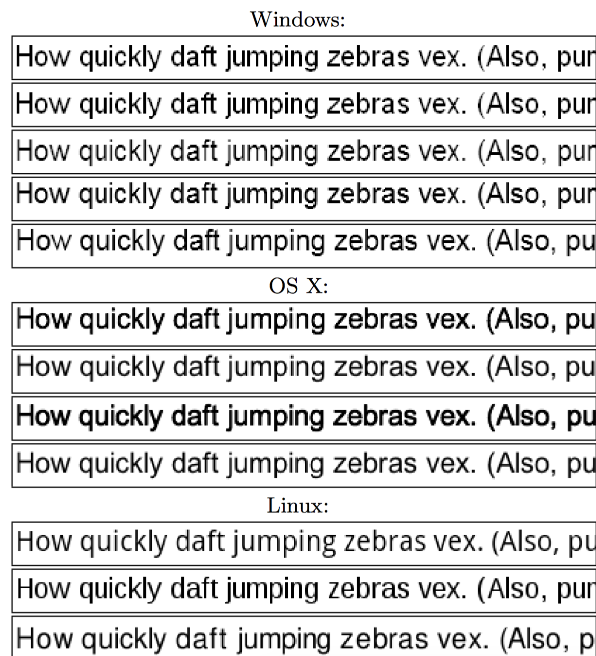

Figure 3.5: 13 ways to render 20px Arial [25]

from 297 distinct users, and found 50 di erent groups of results . This suggests that a typeface may di er considerably across browsers, and may increase the entropy of canvas fingerprints.

Typeface fallback

When a font is not available on a system, a fallback font is used instead. Attempting to render a text with a fictional font allows us to guarantee that the system's fallback font will be applied to this text instead. Fallback fonts are OS and browser specific. Using this method therefore enables us to increase the entropy of the canvas fingerprint.

Sub-pixel font smoothing

Displaying a font on a computer display means using a few squared pixels to represent a vector image visually. No standard definition of how this should be achieved exists. Companies like Apple, Microsoft, Adobe, and many others thus use di erent font-rendering engines with di erent algorithms for this task. Whilst they all have the same goal - to render a human-readable text - they use di erent approaches to achieve it. Their results therefore di er on pixel level. While Apple believes that the design of a font should be preserved, even at the cost of blurriness, Microsoft believes that fonts should be rendered with as much sharpness as possible, to improve readability.

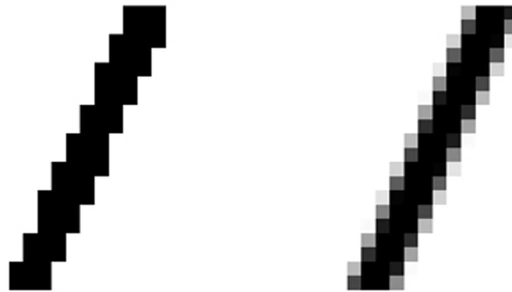

Figure 3.6: A shape without anti-aliasing (left), and a shape with anti-aliasing (right).

Di erent techniques of font-smoothing and sub-pixel rendering are employed on di erent systems, which increases the entropy of canvas fingerprints.

Anti-aliasing

Anti-aliasing is a method of minimizing the distortion of shapes when representing vector graphics or high-resolution images as smaller images. It is similar to font smoothing, but can be applied to any graphic, not just fonts. An example of what an anti-aliased and a non-anti-aliased shape looks like is presented in Figure 3.6

Anti-aliasing in the HTML5 canvas element is controlled by the browser, and is turned on on some browsers and turned o on others. Implementations of anti-aliasing algorithms may di er slightly across browsers. This accounts for the di erences on images drawn on di erent browsers and systems.

To increase the possibility of detecting di erences in the anti-aliasing algorithms that take place on the HTML5 canvas, we draw multiple objects of di erent shapes and colors. Unlike all other implementations of canvas fingerprinting we have seen, we also make sure that all of these objects overlap, set a di erent transparency for each of them, apply shadows to some of them, and set their size parameters in float numbers to take advantage of rounding di erences of browsers.

Canvas winding

Winding and even-odd rules are algorithms for filling vector shapes. When applying an even-odd filling rule, shapes with more than one closed outlines, which are overlapping. The color of each point is then determined by the parity of the number of closed outlines that are covering it.

Figure 3.7: A shape filled using the even-odd rule.

Since not all browsers support winding and even-odd fill rules in the HTML5 canvas, we include it in the browser fingerprint. Figure 3.7 demonstrates what an image filled using the even-odd rule looks like.

Emojis

Emojis are ideograms and smileys used on web pages. Whilst originally used in mobile phone messaging applications, they were added to the Unicode Standard in October 2010, due to their popularity. As of June 2017, there are a total of 2,666 emojis in the Unicode Standard.

Since emojis are transmitted in a non-graphical way, as Unicode characters, it is up to the browser or the operating system that is running the browser to decide what design an emoji will have. For brand and design purposes, emojis are drawn in di erent styles on various systems. For an example of a panda face emoji drawn on 12 di erent systems, see Figure 3.8.

Audio fingerprint

An audio fingerprint is analogical to a canvas fingerprint in many ways. The main di erence, of course, is that instead of an image, an audio signal is generated. To generate it, we take advantage of AudioContext interface, which works by linking modules, called AudioNodes, together into a graph. These modules can generate, process, play, or store an audio signal. This is, in many ways, similar to real-life musical instruments where, for example, an electric guitar generates an audio signal, which is then processed by e ects like an echo or a phaser, and finally played through speakers.

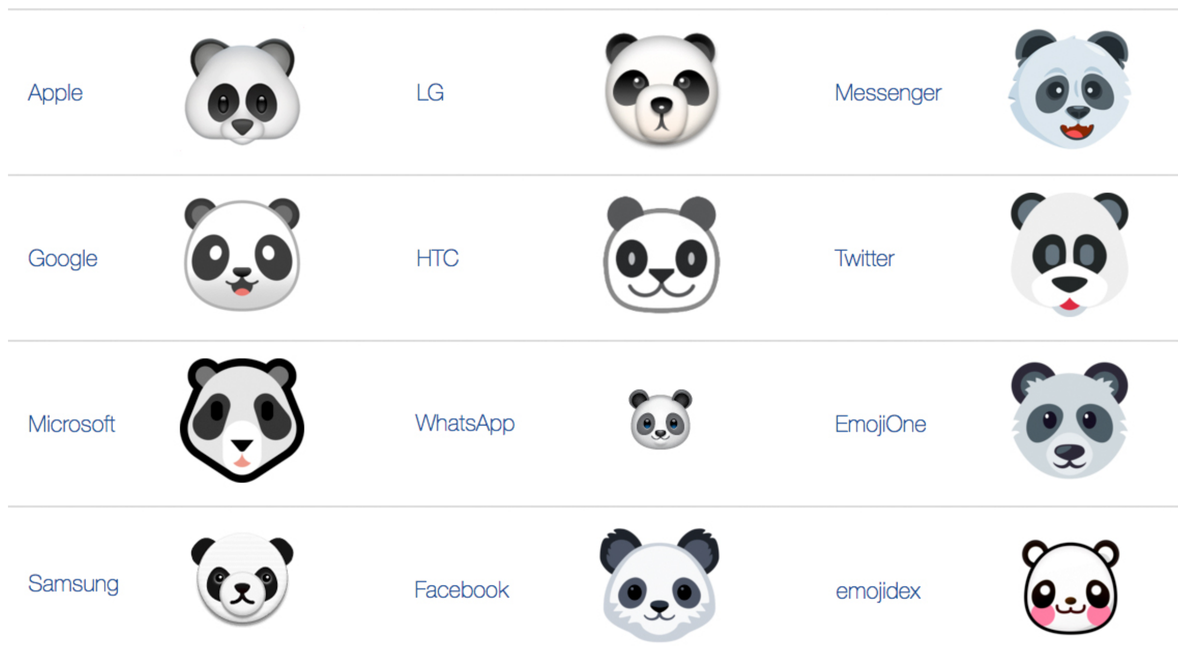

Figure 3.8: 12 styles for a single emoji

In our setup, as shown in Figure 3.9, we use an `OscillatorNode` to generate a sine, or triangular wave, and send it through a `DynamicsProcessorNode` with exactly one input and one output. The latter lowers the volume of the loudest parts of an audio signal to prevent clipping and distortion that would be unpleasant for the human ear. We then send this processed signal to an `AudioContextBuffer`, used to store an audio signal so it can be played at a later time. We also set the length, the number of channels, and the sample rate used to represent our audio signal.

This sound does not have to be played at all. To fingerprint a browser, only the sampled data stored in the `AudioContextBuffer` is necessary. We read this data frame by frame, and add the values up into a single number (or a checksum) which presents the final audio fingerprint. To check whether two audio fingerprints are the same, all we need to do is compare their checksums.

The first mention of using AudioContext to fingerprint browsers that we were able to find is from a paper called "A 1-million-site Measurement and Analysis" [19]. In this paper, the researchers used a semi-automated approach to analyze 1 million websites with the goal of finding out what methods are being used to track users online. For purposes of future research, they also created a website with an audio fingerprint demonstration. The website attracted 18,500 visitors with distinct cookies and the researchers were able to collect 713 different audio fingerprints in total. They also confirmed that a browser will always generate the same audio fingerprint.

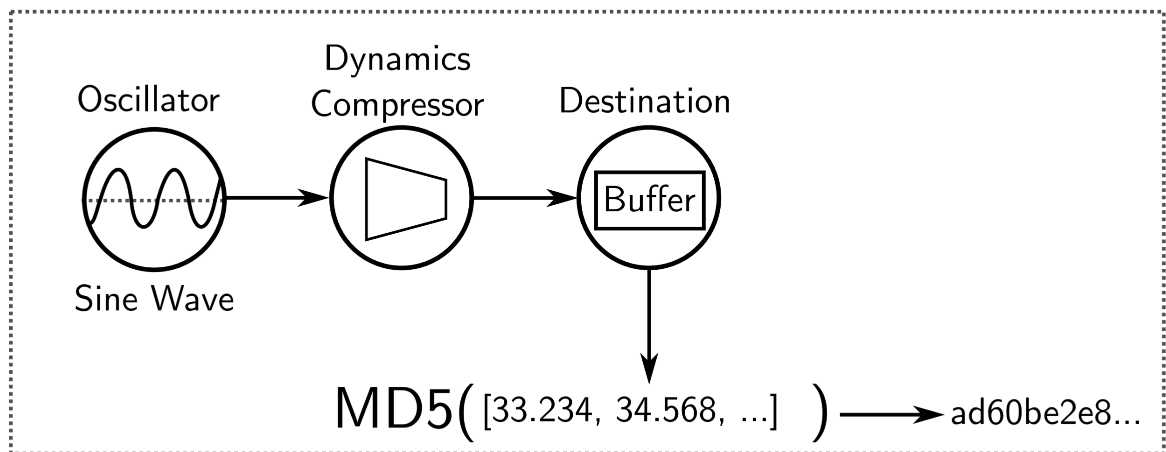

Figure 3.9: Configuration of nodes in AudioContext API fingerprint implementation [19]

We used the same implementation of the audio fingerprint as the one found by the Princeton University researchers, with minor tweaks to ensure that the code compatible with the newest specification of the AudioContext API.

Since our research is focused on smartphones, we noticed an interesting phenomenon when trying to generate the audio fingerprints of iPhone and iPad devices. When checking the data collected by our first iteration of the browser fingerprinting script, we noticed that no iPhone or iPad fingerprint contained an audio fingerprint. Upon further research, we found that Apple's iOS documentation mentions that no Web Audio API code will be executed, unless triggered by an explicit user action. For purposes of the analysis of browser fingerprinting, we decided to implement a workaround that would trigger the browser fingerprinting script on first touch or mouse event, rather than on page load. This ensured that an audio fingerprint was also collected from iPhone and iPad devices.

This behavior is specific to Apple, and we have not seen it mentioned in any other related work or implementation. Whilst we assume that Apple did not do this intentionally, limiting the use of Web APIs to reasonable user actions is a very good practice that might improve privacy in web applications.

3.7 Omitted features

We did not include all of the browser fingerprint features that we have found in other implementations and studies, or discovered by ourselves, mainly due to our decision to create an implementation that could be used in the real web environment. This

means we do not want to interrupt user experience with user consent pop-up windows, or execute any code that would significantly decrease the performance of the user's browser. For this reason, we, for example, limited the number of fonts we are able to detect with our font detection method (see 3.3).

A lot of powerful fingerprint data can be obtained from hardware sensors, such as the accelerometer, the GPS, the camera or the microphone. All of these sensors, however, require user consent prior to being accessed. Most web applications do not have a real use for such information, and a pop-up window requiring the user to allow access to their GPS data would be quite suspicious. We therefore omitted all the features that would require user consent prior to being accessed.

In Chapter 2 we explained that, since support for Flash will be halted by 2020, there is no reason to keep using it in browser fingerprints. Therefore, we decided not to use any data obtained by Flash in our implementation.

Among other omitted features, there is a method of estimating video RAM by repeatedly allocating textures until it is full, a method for estimating server-client round-trip delay time, clock skew detection methods and similar. F. Alaca and P.C. van Oorschot [13] investigated these features and discovered that they either use too many hardware resources or are too unreliable to be used as browser fingerprint features.

Lastly, we considered using Battery API data and user interaction data as browser fingerprint features. However, Battery API has been deprecated, and we found user interaction data, such as the size of the browser's scroll jumps, to be too unreliable in our tests.

Chapter 4

Datasets and feature collection

Since browser fingerprint data can be quite sensitive, there is no public dataset of collected browser fingerprints. In order to analyze browser fingerprints, we had to collect a dataset by ourselves, which also means that we can only compare our methods by comparing our results with the results published by other researchers. Although comparing data side by side would, arguably, be quite interesting, we will only be able to compare the final results.

This chapter describes the datasets and sources of data used for the purposes of this thesis, and implementation details of the methods we used.

4.1 Sources of data

For the purposes of our research, three di erent sources of data, each implemented in a di erent project/website, were used to collect three di erent datasets. In this section, we will describe each of them and describe the implementation of browser fingerprint collection in more detail. Using several sources of data may seem unnecessary. The numbers of visitors of these websites were, however, vastly di erent. With more users, the development cycle of a project is longer, the implementation is harder, and the level of responsibility is higher. Using several sources of data proved to help increase the speed of our development, and helped us detect problems sooner rather than later. For instance, it helped us detect an audio fingerprint problem with iPhones and iPads (see 3.6) before implementing it in our largest source of data, and presumably saved us several weeks worth of work.

Promotional website

For the purposes of this thesis, we created a simple informative website that displays the values of browser fingerprint features to users who allow us to store this data for analysis purposes. We first expected this website to be the main source of data for our research. However, we soon realized that a website with no added value, other than demonstrating a browser fingerprint, will not bring much tra c, and therefore will not collect a su cient amount of data.

If readers of this are interested in seeing what their browser fingerprint looks like, they can visit fp.virpo.sk. Through this website, we shared our work with colleagues, classmates, and friends, to be able to quickly test our method on any device. This allowed us to iterate our code quickly in the early stages of development, and to detect several issues with our implementation. On this website, we used JavaScript to extract the browser fingerprint, PHP to process it and append HTTP headers to it, and MySQL database to store it.

Over a period of 10 weeks, we were able to collect 355 fingerprints with 52 distinct browser IDs. Browser IDs were generated and stored in the browser's cookies upon first visit of the website. Because this dataset consists of browser fingerprints of di erent versions, we did not analyze this data any further, as we had a much better data collected elsewhere.

Medium-sized website

Another source of browser fingerprints we used was a medium-sized business website built on the WordPress content management system. We will refer to this source of data as the "WordPress website". Since we were not sure whether we would be allowed to collect data from the third - and biggest - dataset, we implemented our browser fingerprinting script on this website to have a backup dataset with a reasonable amount of data. In order to collect browser fingerprints on this website, we created a custom WordPress plugin that stores data in a/the MySQL database.

Over a period of 2 months, we iterated 6 di erent versions of our script, and collected 3,436 fingerprints with 2,600 unique browser IDs. 1,186 of these fingerprints were collected using the latest version of our script, and 930 of them had a unique browser ID.

Whilst we did not analyze this data thoroughly, we monitored this dataset for any

unexpected behaviour, and managed to detect several important cases of a browser fingerprint not behaving as expected. The findings worth mentioning include an audio fingerprint bug on iOS (see 3.6), inconsistent font-detection (see 3.3), and conversion to daylight savings time (see 3.3). To our knowledge, none of these had been described before.

Large-scale JavaScript web app

By far the largest source of data on which we were able to apply our browser fingerprinting script was a web application used for enhancing audience interaction on events by allowing them to ask questions or answer polls. Since this web app is mainly used by event participants, most of them will use their smartphone to access it. Indeed, around 65% of its, approximately, 200,000 active users per week use mobile phones/smartphones, and 4% use tablets to access it. These attributes make this dataset ideal for an analysis of browser fingerprinting methods on mobile phones and smartphones on an unprecedented scale. In the rest of this thesis, we will refer to this data set and web application as the "web app".

We were able to collect 566,704 browser fingerprints with 323,746 distinct browser IDs. It is also important to note that, unlike most other researchers working on this topic, we collected and analyzed real-world data from an application used all around the world. Eckersley [17] analyzed data collected on `https://panopticlick.eff.org/`, and Laperdrix et al. [24] analyzed data collected on `https://amiunique.org/`. Both of these are websites dedicated to browser fingerprinting, which inform their visitors what browser fingerprinting is, and that it can be used to track their presence, prior to collecting their browser fingerprint. These datasets might therefore be biased, since many of their visitors will attempt to change their browser fingerprint, thus a ecting the data. In contrast, our dataset, reflects how accurate browser fingerprinting can be in the real world.

4.2 Implementation

As already mentioned, we used a combination of JavaScript, PHP, and MySQL in order to collect fingerprints from our promotional website and from the WordPress website. However, implementation in the web app was di erent and more complex.

The code

In order to be able to collect fingerprints from the web app, our implementation had to respect its internal code standards, terms of service, and privacy policy. The web app is running on AngularJS, a JavaScript open-source front-end web application framework maintained by Google. Its codebase is written in TypeScript, a strict syntactical superset of JavaScript, which is needs to be compiled to JavaScript and served to users.

In order to be able to collect browser fingerprints from all devices, including iPhones which, as we discovered, are problematic (see 3.6), the fingerprinting script is triggered on the first interaction of the user with the web app (touch or mouse event). The script is ran asynchronously and when it finishes, the results are sent to an API endpoint.

The API runs on Node.js, a JavaScript runtime environment that can execute JavaScript code on the server side. The codebase of this API is also written in TypeScript, which is then compiled to JavaScript. We have created a simple endpoint that receives fingerprint data and the browser ID, and stores them in a MySQL database together with HTTP header values and a timestamp.

Most modern JavaScript applications use polyfilling techniques. Polyfill is a code that implements features that are not supported on some browsers, and makes the application compatible for all or most browsers. However, pollyfilling should be treated with caution when implemented together with browser fingerprinting scripts. Since information about di erent implementations and unsupported features is exactly what we utilize as identifying information, we made sure that no polyfill interferes with our browser fingerprinting script.

Errors as a source of additional entropy

If a feature is not available on a browser, the test for this feature will either continue without executing or, in some cases, crash and stop the execution of the entire fingerprinting script. For this reason, we handle errors separately for each feature, so that the execution of the script will always finish. Unlike all previous implementations we are aware of, we decided to store the error message, rather than simply skip unsupported features. Our assumption that di erent browsers will throw an error for di erent reasons, and with di erent descriptions, proved to be right. For example, in a small sample of 60,000 fingerprints, our script failed to extract WebGL Vendor information 1,095 times, and collected 14 distinct error messages. These error messages were always the same on a given browser.

Browser ID

We wanted to be able to pair all the browser fingerprints collected from a browser. Doing this required storing this ID in the browser, on the client-side, and sending it to our API endpoint together with each fingerprint. We used a randomly generated string consisting of 64 characters long as an ID. We made sure that storing of the browser ID is as robust as possible. In our implementation, we check whether cookies or local storage are available, and store the browser ID in either of the two. This way, the browser ID will be stored even if the cookies are unavailable, which occurs frequently.

Chapter 5

Browser fingerprinting prevention

Given that this thesis is about browser identification and user identification, it will, naturally, also touch a few points related to privacy and identification prevention. Being on the web goes hand in hand with leaving some kind of a trace behind. On the web as we know it today, browsers have to report what technologies are or are not accessible in order for websites to display correctly. With the diversity of operating systems, browsers, and their implementations, websites will always be able to abuse that information to help them identify individuals.

While it is not possible to avoid being fingerprinted altogether, there are, nevertheless, ways to prevent getting identified. As explained above, browser fingerprint identification works by collecting a predefined set of features from a browser, and comparing these values to the values it had previously collected. If it finds a match or an algorithm identifies a fingerprint as belonging to a specific user, it will assume that these two fingerprints came from the same browser.

5.1 Fingerprint with common values

We observed that 54.88% of the browser fingerprints we had collected were unique. The rest was observed at least twice. If the fingerprint of a user is not a unique one, meaning that a number of browsers have the same fingerprint, it will be harder for a website to identify this browser. Websites may use additional data, such as pages visited, to distinguish multiple users or browsers with the same fingerprint. A further discussion about the latter is, however, out of the scope of this thesis.

Changing the parameters and values of the browser that are commonly abused for

browser fingerprinting purposes to their most common values will increase the chance of the fingerprint not being a unique one, and the user staying anonymous. For the most common fingerprint observed in our data, see Appendix A. The same browser fingerprint was collected from 493 distinct browsers.

In our sample, Chrome and Firefox were the most popular browsers. Both offer built-in options to change browser parameters that are often used in browser fingerprints. However, Firefox and its community is ahead of Chrome when it comes to privacy. For example, whilst an external extension is necessary to spoof the user-agent string in Chrome, Firefox allows this to be done directly in its settings. Firefox settings can also be imported and exported using a single file. This possibility encouraged its users to create a comprehensive open-source template for Firefox settings called ghacks-user.js [4]. The latter contains a predefined configuration that strengthens Firefox's security, privacy, and anti-fingerprint character. Browsers with these settings are less likely to be unique. Moreover, all browsers using this settings template will, in most cases, have the same fingerprint.

5.2 Randomizing browser values

Another way to prevent being identified is by randomizing the values that browser fingerprint scripts usually collect. As Englehardt [19] pointed out, many websites only use canvas fingerprint for browser identification. In that case, adding random noise to all HTML5 canvases that the browser renders would make each canvas fingerprint of that browser unique. The noise does not have to be significant. Since the result of the canvas fingerprint is usually a hash of the bitmap from this canvas, it can be changed by changing the color value of a single pixel.

The same can be done for audio fingerprints. It is possible to add random noise to the Oscillator node in the AudioContext API. This will result in a difference indistinguishable by the human ear, yet sufficient for changing the browser's audio fingerprint, making it unique every time.

In another study, Pierre Laperdrix et al. [23] were able to achieve this with their modified version of Firefox. As for the canvas fingerprint, rather than adding random noise to the canvas, they slightly changed the shades of the colors of every object, and used a random fallback font. To randomize the audio fingerprint, they modified the volume of the audio signal while it is being processed by a factor ranging from $0{:}000$ to $0{:}001$. The researchers then tested their modified version of Firefox against

two well-known fingerprinting scripts, namely Fingerprintjs2 [2] and MaxMind's fraud detection device tracking add-on, and were able to get a di erent fingerprint for each out of their 100 attempts.

5.3 Blocking fingerprinting scripts

It is also possible to block fingerprinting scripts completely, using privacy extensions, such as Ghostery [5], Privacy Badger [12], and others. These extensions use a list of unwanted scripts that will get blocked upon detection. However, since websites are capable of detecting whether a script was executed or not, blocking fingerprinting scripts is only useful if more users do it on the same site. If a script on a website is blocked by too few browsers, the latter will be easy to identify.

5.4 Response of browser developers

Developers have a significant amount of power in protecting the privacy of their users. User privacy on the web depends mainly on browser developers, and on the specifications of web technologies. This section covers a few examples to demonstrate why.

Firefox

Firefox is a really good example of how browser developers can fight for the privacy of browser users. It has a built-in setting called "Resist fingerprinting". Provided that it is enabled, the following is an example of what privacy measures will be applied:

- User is notified when a script is trying to extract bitmap from HTML5 canvas, and the latter will not be able to do so unless the user agrees.
- Both `navigator.plugins` and `navigator.mimeTypes` are hidden. They cannot be accessed as iterable lists. Instead, they have to be queried for when a script wants to check if a certain plugin or mimeType is supported.
- Third-party cookies are disabled.
- Time precision is reduced.

More information on fingerprint resistance in Firefox can be found in the documentation of this feature [10]. For now, this option is disabled by default, and can only be accessed via advanced settings.

Specification with privacy in mind

Two examples of how better specification could have no negative impact on the user, while preventing a feature from being misused for identification purposes, are briefly discussed below.

The first example, mentioned in the previous subsection, relates to the way Firefox treats access to plugins. In all other browsers, `navigator.plugins` will always return a full list of plugins, with full details including the precise version. There is rarely a reason for reading this list, other than checking whether a plugin one needs to use on their website is available in the browser or not. One almost never actually needs the full list of plugins. Firefox thus does not return a list of plugins. Instead, each plugin of interest must be queried separately. `navigator.plugins.namedItem('Shockwave Flash');`, for example, will return an object if Shockwave Flash is available, or a null value if it is not.

No specification requires browsers to implement plugin availability to be queried for. Firefox decided to do so in order to protect the privacy of their users.

The second example is Battery Status API, a browser feature that is now deprecated (for security reasons), that allowed websites to read the current state, maximum capacity, or charging time of the battery of the device, provided that the latter was a handheld device or a laptop. Lukasz Olejnik et al. [27] reported how this API could be abused for fingerprinting, and why it was important to remove this API from browsers. Their work highlights how privacy research can help influence standards and improve privacy on the web.

5.5 GDPR in context of browser fingerprinting

Another way of fighting for online privacy is enforcing the use of rules and standards by law. One recent example of such approach is the release of the General Data Privacy Regulation (GDPR) [3], adopted by the EU on 27th of April 2016. This regulation aims to give privacy back to the citizens by regulating how personal data can be processed and collected. GDPR defines personal data as follows:

> Personal data is any information relating to an individual, whether it relates to his or her private, professional or public life. It can be anything from a name, a home address, a photo, an email address, bank details, posts on social networking websites, medical information, or a computer's IP address.

Since the IP address is basically a subset of the fingerprint, browser fingerprints can be considered personal data, and GDPR also applies to them. This means, in short, that websites have to ask for explicit user consent prior to collecting and storing their browser fingerprint. If this data can be connected to a user directly, this user has a right for this data to be forgotten.

Whilst it can be argued that the majority of users tend to overlook the terms of service when using a service, the existence of laws designed to protect these users is certainly a step forward.

Chapter 6

Results and discussion

This chapter presents and discusses the most notable findings of our work. We considered the entropy of the entire dataset, as well as the entropy of each feature, on di erent device types. We also examined how fingerprints change in time, what fingerprint is the most typical, and what is the smallest subset of features we can use to make our script faster, while maintaining its accuracy.

6.1 Dataset description

Between the April 5th and 25th, 2018, we were able to collect 566,703 browser fingerprints with 323,746 distinct browser IDs, out of which 177,677 (54.88%) were unique. This is more than the amount Panopticlick (470,161) or AmIUnique (118,934) were able to collect. In addition, almost 370,000 of the fingerprints in our dataset were collected from smartphones, such as Android and iPhone, which is significantly more than in previous studies.

The only way we can determine whether two fingerprints come from the same browser is by storing a unique ID in the user's browser. Whenever a browser without such an ID visited our web app, we generated and stored it in its cookies, as well as in local storage, in order to make it more robust. However, this browser ID is deleted every time a user decides to delete their cookies and local storage, and is hidden if a user uses the privacy browser mode (also known as "incognito mode"). This means that an unknown error will be present in our results. However, it also means that the real entropy of our dataset can only be higher.

Figure 6.1 illustrates the distribution of the most frequent device types within our

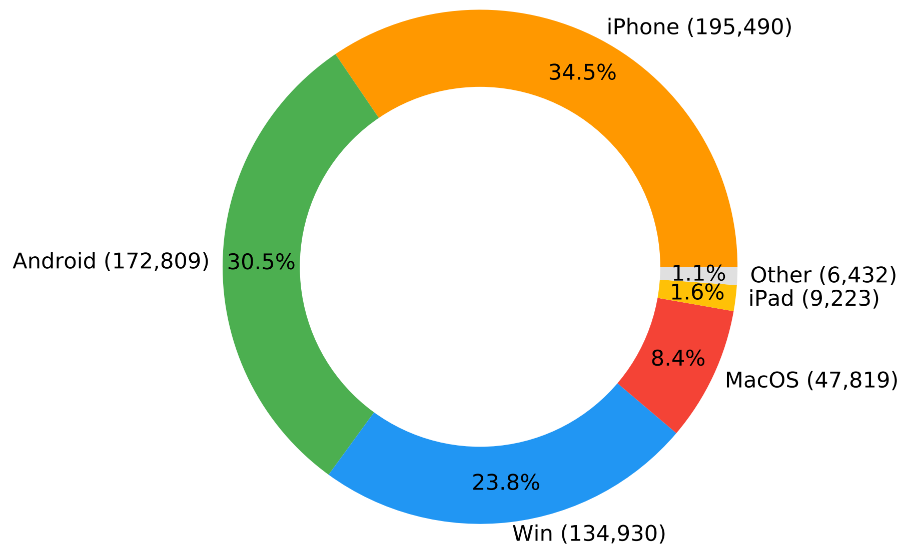

Figure 6.1: Distribution of devices in our dataset

dataset.

6.2 Entropy

The accuracy with which a browser fingerprinting method can identify browsers when no other method (such as login or cookie information) is available, is its most important characteristic. In general, the fewer distinct browsers with the same fingerprint there are, the more accurate this method is. We use entropy (see 1.4) to quantify this quality. In our dataset, with all the browser features combined, we observed a total of 16.685 bits of entropy. This means that if we pick a browser at random, at most one in 105,362 browsers will share the exact same browser fingerprint.

Isolating fingerprints based on the device they were collected from allows us to compare the accuracy with which we can identify browsers on that particular device type. Since we are comparing the entropy of datasets of di erent sizes, the latter must first be normalized. Normalized Shannon entropy is defined as:

$$\frac{H(X)}{H_M}$$

H_M is the maximum attainable entropy of the variable, which in our case is $H_M = log_2(N)$, with N being the number of fingerprints in the dataset. The normalized entropy of all features for each device type in isolation is shown in Table 6.1. When speaking of an isolated device type or an isolated feature, we mean respectively, the entropy of a fingerprint given data consisting solely of fingerprints from a specific device, or only information about a certain browser feature.

Our most notable observations include:

- The entropy of fingerprints on mobile phones is lower than their entropy on desktop devices.
- Android smartphones have higher entropy than iPhones.
- Isolated browser features allowed us to make the following observations:
  - We have confirmed that mobile phones have very rich user-agent strings, as previously observed by Laperdrix et al. [24].
  - System languages and canvas fingerprints, too, work vastly better on mobile phones than on desktop devices.
  - Features that work significantly better on mobile phones also include the color depth, the date format, the platform, the time zone, and the HTTP encoding.
  - The results of Math.tanh and the list of system fonts reveal much less information on mobile phones than on desktop devices.
  - Information about CPU class and list of plugins practically does not contribute any useful data to the fingerprint on mobile devices.

To be able to compare the entropy found in our dataset to those found in other studies, the entropies first have to be normalized. The normalized entropy of the Panopticlick study by Eckersley [17] is 0.96, while the normalized entropy of our dataset is 0.87. The two entropies di er for several reasons. Firstly, Eckersley has demonstrated that the use of Flash can greatly improve the accuracy of browser fingerprinting. However, since Flash technology is soon to be deprecated, we decided not to include it in our browser fingerprint implementation. Secondly, the visitors of the Panopticlick website were encouraged to try changing their browser fingerprint by manipulating their browser and system settings, as well as deleting their cookies and session information. This user behavior results in data bias, where two fingerprints that originate from the same browser but di er in their feature values might be considered to present two fingerprints collected from two distinct browsers. The possibility that the Panopticlick dataset is biased is also supported by the results of our analysis of fingerprint change rate (see Section 6.4). Lastly, most of the fingerprints collected by Panopticlick

Table 6.1: Normalized entropy of all features for each device type

	Desktop	Windows	MacOS	Mobile	Android	iPhone
All features	0.7818	0.7732	0.7627	0.7589	0.7662	0.7225
JS features	0.7817	0.7731	0.7627	0.7578	0.7638	0.7224
Headers	0.5131	0.4715	0.5151	0.5898	0.7279	0.4274
AdBlock	0.0527	0.0467	0.0294	0.0461	0.0545	0.0548
Audio FP	0.1144	0.0808	0.0398	0.1480	0.0879	0.0890
Available size	0.1854	0.1255	0.0688	0.1710	0.0993	0.1010
Canvas FP	0.1669	0.1196	0.0522	0.2946	0.1826	0.1478
Color depth	0.0534	0.0469	0.0294	0.0824	0.0726	0.0562
Cookies	0.0532	0.0470	0.0294	0.0474	0.0556	0.0553
CPU class	0.0222	0.0228	N/A	0.0015	N/A	0.0016
Date format	0.1591	0.1260	0.0558	0.2684	0.1714	0.1722
DNT	0.0261	0.0221	0.0082	0.0333	0.0097	0.0282
Hardware conc.	0.0628	0.0486	0.0298	0.0796	0.0769	0.0099
IE plugins	0.0793	0.0722	0.0294	0.0474	0.0545	0.0561
Indexed DB	0.0538	0.0477	0.0294	0.0471	0.0548	0.0557
Installed fonts	0.1443	0.1070	0.0449	0.0964	0.0595	0.0612
Languages	0.1596	0.1217	0.0558	0.2899	0.2031	0.1401
Local storage	0.0539	0.0477	0.0296	0.0506	0.0561	0.0575
Math tanh	0.0667	0.0589	0.0294	0.0472	0.0553	0.0548
Pixel ratio	0.0964	0.0782	0.0366	0.1215	0.1068	0.0752
Platform	0.0782	0.0528	0.0294	0.1136	0.0796	0.0567
Plugins	0.1466	0.1067	0.0455	0.0478	0.0558	0.0553
Screen size	0.1350	0.1053	0.0449	0.1638	0.0993	0.0931
Session storage	0.0532	0.0470	0.0296	0.0495	0.0550	0.0575
Timezone	0.1156	0.0928	0.0482	0.1750	0.1226	0.1272
Touch	0.0692	0.0607	0.0297	0.0886	0.0606	0.0562
User-agent	0.1746	0.1239	0.0567	0.3653	0.2631	0.1488
User data	0.0527	0.0467	0.0294	0.0461	0.0545	0.0548
WebGL renderer	0.1668	0.1209	0.0545	0.2164	0.1458	0.1056
WebGL vendor	0.0941	0.0626	0.0364	0.1123	0.0799	0.0600
HTTP accept	0.0527	0.0467	0.0294	0.0464	0.0547	0.0551
HTTP encoding	0.0729	0.0597	0.0361	0.1028	0.0666	0.0667
HTTP language	0.1633	0.1234	0.0570	0.2949	0.2070	0.1349
HTTP user-agent	0.1597	0.1086	0.0568	0.3654	0.2631	0.1489

originate from desktop devices, while 65% of fingerprints in our dataset come from a mobile device. Section 6.3 shows that the entropy of fingerprints from mobile devices is, indeed, lower than the entropy of desktop browser fingerprints.

All of these reasons contribute to the fact that the overall entropy of our dataset is lower than the one found by Eckersley. We argue that the first two reasons are also the reasons why our dataset is better at reflecting how accurate browser fingerprinting techniques are at identifying distinct browsers. However, we are aware that the goal of Eckersley was to raise awareness about online privacy, and to estimate how accurate browser fingerprinting techniques are, and acknowledge that they were very successful at achieving these goals.

6.3 Anonymity set sizes

The degree to which a browser fingerprinting method can successfully identify browsers can also be judged by considering what portion of the collected fingerprints was unique, and how big the groups of browsers that share the exact same fingerprint are. We call these groups anonymity sets. An anonymity set of size 1 represents the number of browsers that had a unique browser fingerprint. An anonymity set of size 2 is the total number of browsers that shared their browser fingerprint with exactly one other browser.

The distribution of anonymity set sizes in our dataset is shown in Figure 6.2. Since this distribution is severely skewed, we used logarithmic scales on both axes of this graph. In order to understand our data better, we split the dataset into smaller fractions, with each containing fingerprints from one device type exclusively. Figure 6.3 illustrates the anonymity set sizes found in these fractions.

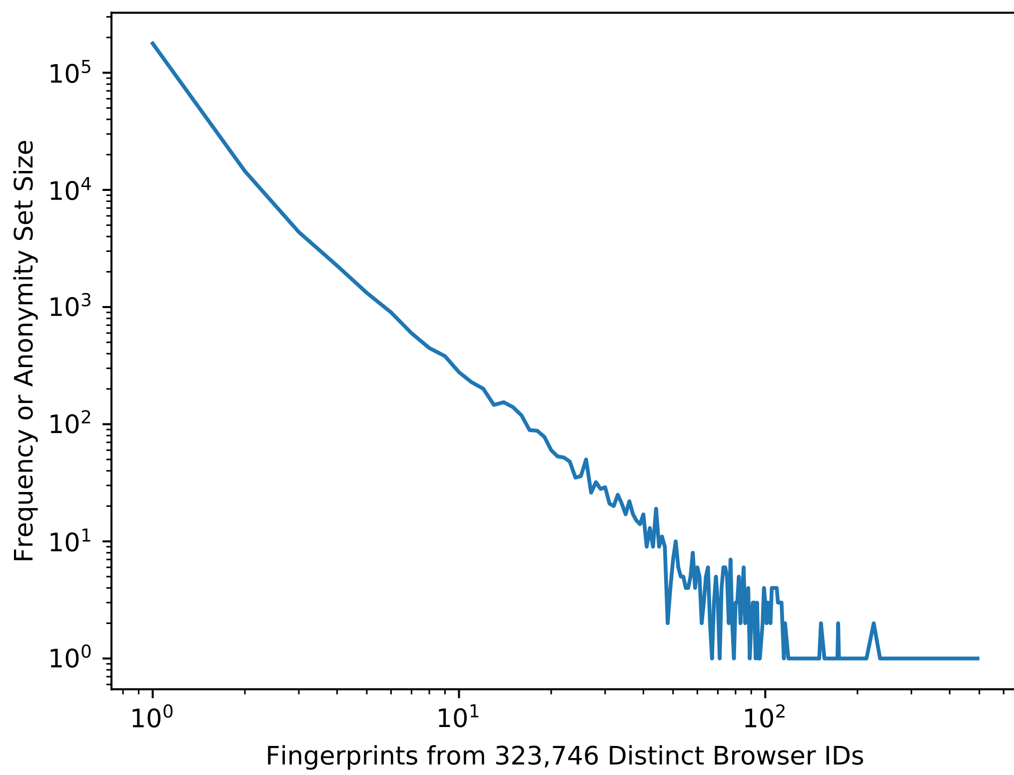

Figure 6.2: Distribution of fingerprints as observed in our dataset

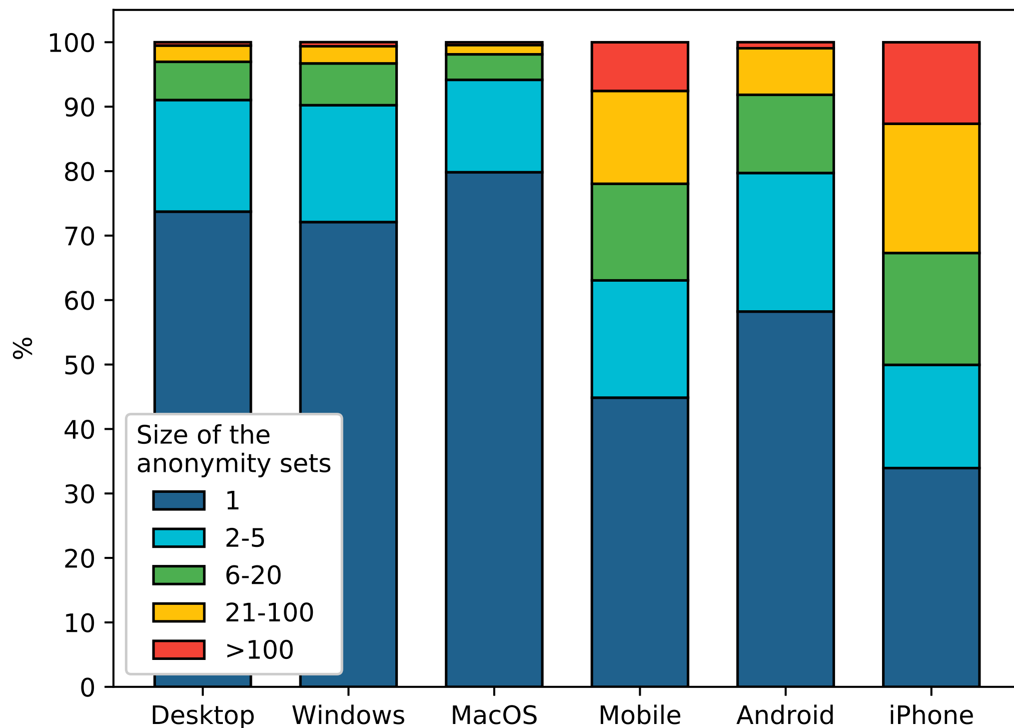

Figure 6.3: Anonymity set sizes of various devices

The height of the lowermost bars in this graph is important because it represents the portion of unique fingerprints in the given dataset. If we were to identify browsers based on their fingerprints, these would be the only ones we would be able to distinguish reliably. All others would fall under a group of browsers, and we would not be able to distinguish them based on the fingerprint alone.

We observed a considerable di erence in the ratio of unique fingerprints from desktop and mobile devices. 73.72% of desktop fingerprints were unique, while only 44.86% of fingerprints collected from mobile devices were unique. The ratio of unique fingerprints is even lower for iPhone devices, with only 33.94% of them being unique, and 49.95% of them in an anonymity set of size 6 or more. In contrast, 58.22% of Android browser fingerprints were unique, and only 20.29% of them in an anonymity set of size 6 or more. These results suggest that identifying distinct iPhone browser instances is significantly more di cult than identifying distinct Android browsers, and that desktop device browsers are considerably easier to distinguish than mobile phone browsers.

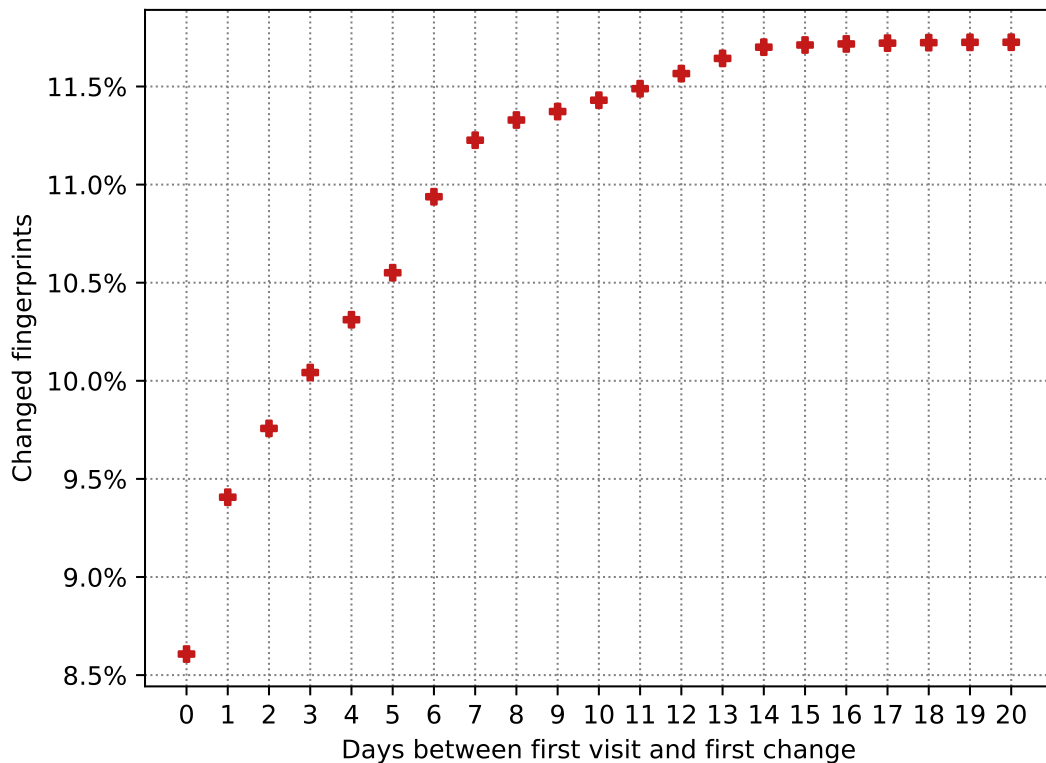

Figure 6.4: Fingerprint change as a function of time

6.4 Change of fingerprints in time

The rate of fingerprint change for returning visitors on the Panopticlick website was 37.4%, as reported by Eckersley [17]. The fact that this rate was as low as 11.7% in our dataset appears to confirm our suspicion that the data collected through Panopticlick does not represent a real-world scenario. The main purpose of Panopticlick was to display browser fingerprints to its visitors, and to inform them how unique their fingerprint was within this dataset. This encouraged visitors to try changing their browser fingerprints, thus introducing a bias to the dataset.

In contrast, our dataset was collected from a real-world web application, which brings it as close to the real world as technically possible. Figure 6.4 shows what fingerprint change looks like as a function of time. Figure 6.5 shows the latter for di erent device types.

Having analyzed these graphs, and the features that were the most common subject of change for individual device types, we can conclude that:

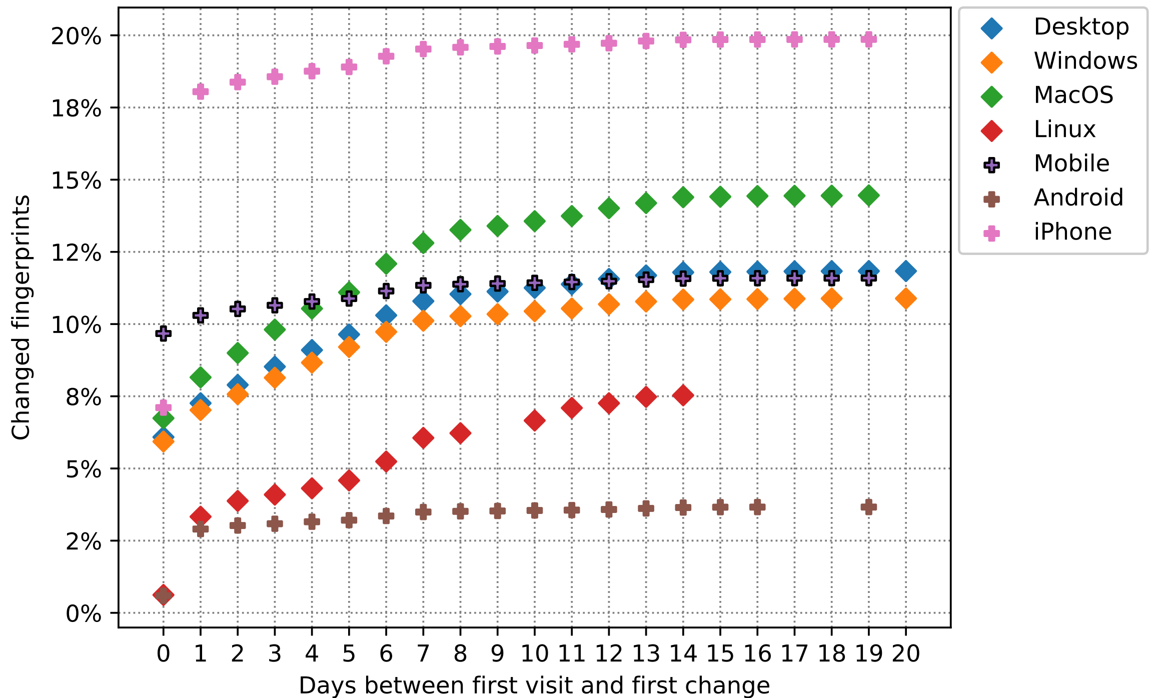

Figure 6.5: Fingerprint change for each device type separately

Of the device types we have observed, iPhones, with a fingerprint change rate of 19%, are the most prone to browser fingerprint changes. The canvas fingerprint, the audio fingerprint, and the user-agent string were usually the features that changed between visits. We assume that canvas and audio fingerprints are unstable on iPhones due to software and hardware optimization that prefers performance over accuracy.

System languages and the canvas fingerprint were the least stable browser features on Android devices. However, the fingerprint change rate on this operating system was just 3%.

On Windows and MacOS devices, the features that tended to change the most include screen properties such as the screen size, the available size, and the pixel ratio, as well as plugin information and user-agent strings. Screen properties will change every time the browser is open on a di erent display, whenever the browser window is resized, or when the user zooms in or out on the content of the browser window.

Changes of the user-agent string between two visits were usually caused by a browser update. The latter are usually automatic.

All of the above observations should be taken into consideration when attempting to identify browsers over extended periods of time.

Table 6.2: Entropy with and without error description

Feature name	Without	With	Improvement
localStorage	0.1164	0.1574	35.14%
sessionStorage	0.0794	0.0968	21.91%
webGLVendor	2.6577	2.7223	2.43%
webGLRenderer	5.7542	5.8189	1.12%

6.5 Entropy in error descriptions

As explained in Section 4.2, we store error descriptions rather than null values whenever our browser fingerprinting script fails to retrieve a value of a feature. To quantify the improvement brought by this change, we have simulated both scenarios using our dataset.

We noticed an increase of entropy in 7 di erent features, namely: the indexed DB, the date format, the canvas fingerprint, the WebGL renderer, the WebGL vendor, the session storage, and the local storage. This improvement was the most striking for the latter 4 features. The extraction of a feature value can fail for a number of reasons, and the error description holds this information. Moreover, error messages are usually written in the system language of the browser. For these reasons, storing error descriptions will result in more entropy than simply using a null value. Table 6.2 presents the entropy of these four features with and without error descriptions stored, and the improvement of entropy in percentage. However, the entropy of the entire fingerprint improved by mere 0:000008 bits, i.e. not significantly.

6.6 Minimal fingerprint

Each browser fingerprint we collect contains 31 di erent feature values. The execution of the script that extracts these values takes 50-150 ms on desktop devices, and 250-1000 ms on mobile devices. Extracting the list of available fonts, and generating the canvas and audio fingerprints is especially time demanding. However, while removing any of these features from our script would reduce its run time, it would also reduce the entropy in our dataset. We thus decided to find a subset of our fingerprint that is small in its number of features, while maintaining as much entropy as possible.

The distribution of our fingerprint with all of the features contains 16.685 bits of

Table 6.3: Highest achievable entropy for the given number of features

Subset size	Features	Entropy
3	Date format, User-agent, Available size	14.2218
4	All of the above + Canvas FP	15.2366
5	All of the above + WebGL renderer	15.7247
7	All of the above + DNT + HTTP language	16.3192
9	All of the above + Audio FP + Installed fonts	16.5168

entropy. We decided to find the smallest subset of browser features that contains at least 16.5 bits of entropy. There are 231 possible subsets of our set of browser features. We did not have access to the computing power necessary to test all of these combinations. We were, however, able to find the subset by approaching this issue in the opposite way. We expected the number of features necessary to achieve 16.5 bits of entropy to be lower than 15, meaning that we most likely do not need to check any combinations containing more than 15 features. To save even more computing power, we started by trying all possible combinations of 3 di erent features, determined what the highest achievable entropy of such a subset was, and increased the size of this subset by 1 until we found the minimum number of features that yield at least 16.5 bits of entropy. The results of these tests are shown in Table 6.3.

Firstly, we want to note that every time we increased the subset size, all of the previous features remained in the ideal subset of features, and one new feature was added. This might not always be the case. Due to correlations between browser features, adding a feature to the subset might render one of the previous features redundant, and change the ideal subset of features completely. However, it was not the case this time and we discovered that, by using just the 9 browser features listed in Table 6.3 an entropy of 16.5 bits can be achieved in our dataset. The following is a summary of our learnings from the experiment:

- Both canvas and audio fingerprint are present in the subset of 9 features necessary for an entropy of at least 16.5 bits.
- By using just 3 browser features - the date format, the user-agent, and the available size - we can achieve an entropy of 14.2 bits.
- Available size works better as a browser identifier than screen size. As explained in Section 3.1, available size is equal to screen size with widths and heights of various taskbars and scrollbars excluded. This means that available size holds information about the dimensions of the UI elements of the browser and the operating system.

WebGL renderer (information about the graphic chip version) is a better identifier than WebGL vendor (the name of the vendor that manufactured the graphic chip).

As seen in Table 6.1, the Do Not Track (DNT) header only contains a small amount of information on its own. Nevertheless, because it has to be turned on by the user explicitly, it does not correlate with any other browser feature, which makes it a useful addition, even to a small subset of other browser features.

Removing the canvas fingerprint from the subset of 4 features lowers the entropy by 1 bit.

While looking for a minimal fingerprint, we also examined how the entropy would decrease if only one of the features was removed. Based on the results of this experiment, we then removed all browser features that had an insignificant effect on the entropy when absent from the browser fingerprint. We managed to remove 12 browser features with the entropy dropping by mere 0.003 bits. The features we removed in this experiment were: the indexed DB, the local storage, the session storage, the screen size, one of the user-agents, the color depth, the platform, the CPU class, Math.tanh, the user data, the WebGL vendor, and the HTTP accept header.

Conclusions

In this work, we have implemented a browser fingerprinting script that contains the most advanced browser fingerprinting features, including the audio fingerprint and the canvas fingerprint. In our implementation, we improved the extraction of separate browser features whenever possible. In order to ensure that our dataset is as close to the real world as possible, we collected 566,703 browser fingerprints from a real-world web application,. The fact that 65% of the fingerprints in our dataset originate from a mobile device enabled us to compile the first large-scale analysis of the use of browser fingerprinting techniques on mobile devices.

We have observed that the distribution of fingerprints in our dataset contains 16:685 bits of entropy -slightly less than the entropy found by Eckersley [17] or Laperdrix et al. [24]. However, in Section 6.2 we argue that our results might be better at reflecting the accuracy of identification by browser fingerprinting in the real world.

We have found that mobile devices are significantly harder to fingerprint than desktop devices. The distribution of fingerprints collected from mobile devices contains less entropy, and it is less common for browsers on mobile devices to have a unique fingerprint. In our dataset, only one third of the browser fingerprints collected from iPhone devices were unique.

To our knowledge, no other study had implemented or reported how e cient the audio fingerprinting technique is at identifying browsers. We have demonstrated that this feature is, in fact,one of the most powerful features within our dataset. Its entropy in isolation is similar to the entropy of the list of installed fonts, the user-agent string or the available screen size. It is slightly better at identifying mobile browsers than browsers found on desktop devices.

While searching for the smallest subset of browser features capable of replacing our set of features with no loss in entropy, and improved run time, we identified 12 out of 31 browser features that could be removed with almost no loss in entropy. To achieve an entropy of fingerprint distribution of 16.5 bits in our dataset, we would only need

to collect the date format, the user-agent string, the available screen size, the audio fingerprint, the WebGL renderer information, the DNT header, the HTTP language header, the audio fingerprint, and the list of installed fonts. By only using the first 3 browser features (the date format, the user-agent string, and the available screen size), we can achieve an entropy of 14.2 bits. These results indicate how powerful each of these features is. More detailed results can be found in Section 6.6.

When analysing the change rate of the fingerprints in our dataset, we found that the fingerprints of browsers on Android devices are the most stable, while the fingerprints of iPhone browsers change the most. The change rate of browser fingerprints on other systems is close to average. Although we are unsure about the cause of this behaviour, we also noticed that both the canvas and the audio fingerprint values are rather unstable on iPhone devices.

Lastly, we have provided an overview of how users can prevent getting identified by browser fingerprinting techniques. Their identity can be hidden by using common browsers with common settings, or by using extensions that detect browser-fingerprinting scripts and prevent their execution. Of all browsers, Firefox seems to care about the privacy of its users the most. It can, for example, warn the user if a website is trying to extract a bitmap from any HTML5 canvas element, necessary for collecting the canvas fingerprint of a web browser.

In future work, it might be worth trying to detect intentional and unintentional changes in browser fingerprints. Detecting an intentional change of the browser fingerprint would potentially be helpful for purposes of fraud detection. However, collecting a dataset with browser ID labels for these purposes is a great challenge because, in most cases, a user that attempts to change their fingerprint will also delete their cookies and local storage data, in order to remain anonymous. Other interesting topics to explore include correlations between browser features, and browser comparison in terms of browser fingerprinting. The results of a correlation analysis might be useful for detecting anomalies in fingerprints when a user tries to change their browser fingerprint.

Bibliography

[1] Browserprint - is your browser safe against tracking? *https://browserprint.info/*.

[2] Fingerprintjs2 - modern flexible open-source browser fingerprinting library. *http://valve.github.io/fingerprintjs2/*.

[3] Gdpr - rules for the protection of personal data inside and outside the eu. *https://ec.europa.eu/info/law/law-topic/data-protection$_e$n:*

[4] Ghacks-user.js - an ongoing comprehensive user.js template for configuring and hardening firefox privacy, security and anti-fingerprinting. *https://github.com/ghacksuserjs/ghacks-user.js*.

[5] Ghostery, a browser extension that helps you manage website trackers for a cleaner, faster, safer experience.

[6] Github octoverse 2017 - a look back at the projects, people, and teams of 2017. *https://octoverse.github.com/*.

[7] Maxmind - minfraud device tracking add-on. *https://www.maxmind.com/en/minfraud-device-tracking*.

[8] Mdn web docs - resources for developers, by developers. *https://developer.mozilla.org/*.

[9] Modernizr - respond to your user's browser features. - https://modernizr.com/.

[10] Mozilla firefox - fingerprinting security documentation. *https://wiki.mozilla.org/Security/Fingerprinting*.

[11] Plugindetect.js - javascript library for browser plugins detection. *http://www.pinlady.net/PluginDetect/All/*.

[12] Privacy badger - block spying ads and invisible trackers. it's here to ensure that companies can't track your browsing without your consent. *https://www.e .org/privacybadger*.

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

[14] Harald Tveit Alvestrand. Tags for the identification of languages. 2001.

[15] Hristo Bojinov, Yan Michalevsky, Gabi Nakibly, and Dan Boneh. Mobile device identification via sensor fingerprinting. *arXiv preprint arXiv:1408.1416*, 2014.

[16] Matthew Cortland. 2017 adblock report, Jul 2017.

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

[18] Eric Enge. Mobile vs desktop usage in 2018: Mobile takes the lead. *https://www.stonetemple.com/mobile-vs-desktop-usage-study/*.

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

[20] Erik Flood and Joel Karlsson. Browser fingerprinting. 2012.

[21] Markus Jakobsson, Elaine Shi, Philippe Golle, and Richard Chow. Implicit authentication for mobile devices. In *Proceedings of the 4th USENIX conference on Hot topics in security*, pages 9–9, 2009.

[22] Samy Kamkar. Evercookie. *http://samy.pl/evercookie*, 2010.

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

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

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

[26] Gabi Nakibly, Gilad Shelef, and Shiran Yudilevich. Hardware fingerprinting using html5. *arXiv preprint arXiv:1503.01408*, 2015.

[27] Lukasz Olejnik, Steven Englehardt, and Arvind Narayanan. Battery status not included: Assessing privacy in web standards.

[28] Arvind Narayanan Steven Englehardt. Audiocontext fingerprint test page. *https://audiofingerprint.openwpm.com/*.

Appendix A

Table A1 shows the values of browser features found in the most typical fingerprint of our dataset. We collected the very same fingerprint from 493 distinct browsers.

Table A1: Most typical fingerprint - observed 493 times

Feature	Value
AdBlock	False
Audio FP	35.9802674
Available size	667; 375
Canvas FP	ded0f60083aee3cc8920f46a3152afcb
Color depth	32
Cookies	True
Date format	01/01/1970, 01:00:00
DNT	False
Hardware conc.	1
IE plugins	empty
Indexed DB	True
Installed fonts	Arial; Arial Hebrew; Arial Rounded MT Bold; Courier; Courier ...
Languages	en-GB′; ′en-GB′; empty; empty; empty
Local storage	True
Math tanh	-1.421448824
Pixel ratio	2
Platform	iPhone
Plugins	empty
Screen size	667; 375
Session storage	True
Timezone	-60
Touch	0; True; True
User-agent	Mozilla/5.0 (iPhone; CPU iPhone OS 11_3 like Mac OS X) ...
User data	FALSE
WebGL renderer	Apple A10 GPU

Continuation of Table ??	
Feature	Value
WebGL vendor	Apple Inc.
HTTP accept	application/json, text/plain, */*
HTTP encoding	br, gzip, deflate
HTTP language	en-gb
HTTP user-agent	Mozilla/5.0 (iPhone; CPU iPhone OS 11_3 like Mac OS X) ...

Appendix B

On the CD attached to this thesis, you will find:

- getBrowserFingerprint.ts - Our implementation of browser feature extraction. We collected our data using this script, written in TypeScript.
- package.json - An NPM configuration file that holds information about the dependencies of our browser fingerprint script implementation. Executing `npm install` in the folder containing this file will install the MD5 NPM package which contains a TypeScript function for hashing messages with MD5. We used the latter for hashing canvas fingerprints.