FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Web Browser Fingerprinting

A framework for measuring the web browser entropy

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Vanja Culafic
Matrikelnummer 0426783

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Dr. tech. Martin Schmiedecker

Wien, 23.08.2016

(Vanja Culafic) (Edgar Weippl)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Web Browser Fingerprinting

A framework for measuring the web browser entropy

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Vanja Culafic
Registration Number 0426783

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Dr. tech. Martin Schmiedecker

Vienna, 23.08.2016

(Vanja Culafic) (Edgar Weippl)

Erklärung zur Verfassung der Arbeit

Vanja Culafic
Kandlgasse 44/25-26, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Vanja Culafic)

Acknowledgements

I would like to express my gratitude to Dr. Martin Schmiedecker for the continuous support, creative insights and comments for the whole duration of this project. My sincere gratitude also goes to my advisor Privatdoz. Dipl.-Ing. Mag. Dr. Edgar Weippl for helping me secure a research-internship position in Tokyo, Japan. I would also like to thank Prof. Isao Echizen of National Institute of Informatics Japan for his constructive criticism of my research during my stay in Japan.

My thanks also goes to my friend and colleague Dr. Thomas Paulin for his valuable input regarding data processing and visualization.

Furthermore, I want to thank my parents Nesiha and Ranko and my brother Vedran for supporting me throughout my academic and professional endeavors.

Elodie, sans ton soutien et ton aide précieuse, ce projet n'aurait pas pu aboutir. Merci.

Abstract

There are numerous ways in which websites track online users' behavior for various purposes like targeted advertising and price discrimination. Browser fingerprinting is a process of identifying a user by means of collecting and comparing the distinguishing features of the browser. This form of stateless tracking can aid third-party websites to track users across different domains without their knowledge. Fingerprinting methods exploit modern browser side technologies that provide interactive web experience (e.g. JavaScript and Flash) in order to measure different browser and operating system attributes.

In this thesis, we conduct an extensive survey of related work-field of browser tracking and fingerprinting. Furthermore, we implement an extensible fingerprinting framework that encompasses all fingerprinting methods from previous research and publish it online in order to collect browser fingerprints. We analyze 1,800 fingerprints collected in a time span between April 7, 2016 and August 21, 2016 and compare the findings to other research.

Kurzfassung

Es gibt verschiedene Möglichkeiten wodurch Webseiten das Verhalten von Internet-NutzerInnen verfolgen können, um zum Beispiel gezielte Werbung und Preisdiskriminierung durchzusetzen.

Browser Fingerprinting ist ein Prozess, der die NutzerInnen identifiziert, indem bestimmte, kennzeichnende Eigenschaften des Browsers gesammelt und verglichen werden. Diese Art vom zustandlosen Tracking ermöglicht fremden, externen Webseiten, NutzerInnen ohne ihr Wissen über verschiedene Domains zu verfolgen. Fingerprinting-Methoden verwenden moderne Browser-Technologien, die eine interaktive Web-Experience anbieten (u.a. JavaScript und Flash), um die Eigenschaften verschiedener Browser und Betriebssysteme zu messen.

In dieser Arbeit wird eine umfassende Studie zum Browser Tracking und Fringerprinting geführt. Des Weiteren wird ein erweitbares Framework, das alle bereits existierenden Fingerprinting-Methoden aus bisherigen Studien umfasst, implementiert und online gestellt. Die 1800 Fingerprints, die dadurch im Zeitraum 7. April-21. August 2016 gesammelt wurden, werden analysiert und zu den Ergebnissen der bisherigen Studien verglichen.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

1.1 Problem Definition

The web browser has been established as one of the dominant instruments for consuming and delivering information on the Internet. As a consequence of its popularity, it presents a large surface for misuse and exploitation. Many advertising and analytics parties developed their business around collecting users' personal information and tracking their online browsing habits [38] for the purpose of targeted advertising and price discrimination, among others [30].

The most widely used tracking mechanism across browsing sessions is based on the usage of browser (HTTP) cookies. Websites can store small files containing data such as unique identifier on the user's computer and retrieve this data upon each consequent visit to the website. Many websites comply to the Cookie Law [1], that was designed to protect user's online privacy by asking for their consent to store and read information on a computer when visiting the website. Some service providers even offer visitors the ability to opt out of interest based advertising. Furthermore, web browsers allow the users to delete unwanted cookies from their computer or disable them altogether in order to disassociate their browser from a profile created by a website and thus limit the tracking scope of the cookies.

There exists another form of tracking that, in contrast to cookies, does not rely on storing and reading data on a user's computer. *Browser fingerprinting* is a process of identifying a user by means of collecting and comparing the distinguishing features of the browser. Fingerprinting commonly exploits client-side technologies that provide interactive web experience (e.g. JavaScript and Flash) in order to measure different browser and operating system attributes like User-Agent string, HTTP-Accept headers, list of browser plugins, current timezone, screen resolution, installed system fonts, etc. A combination of these values leads to a *browser fingerprint* that can uniquely identify a user without their knowledge, since the fingerprinting occurs in the background. Another characteristic of the browser fingerprint is that it also works when using private or incognito browsing modes. While the cookies are separated from normal browsing sessions and deleted afterwards, browser fingerprint is still present, both in normal and private browsing sessions. Due to its stateless nature, a browser fingerprint is generally harder to block than other tracking techniques and can bypass the user tracking regulations imposed by law and as such poses a legitimate privacy threat on the web.

In recent years browser tracking and fingerprinting as a topic has gained interest in the privacy and security-related research community, with new fingerprinting techniques being discovered and methods described on a regular basis. In order to retain an overview and keep track of various browser fingerprinting methods we want to combine the practical part of research under one unified platform.

1.2 Goals

We define three main goals of the master's thesis:

- Extensive survey and summary of related work in the field of browser tracking and fingerprinting.
- Implementation of an extensible fingerprinting framework that encompasses all fingerprinting methods from the previous research in browser fingerprinting. The framework is published at `https://fingerprint.sba-research.org` for the purpose of collecting browser fingerprints. It also allows the visitors to review the fingerprinting information that their browser reveal along with statistics like uniqueness and similarity to other fingerprints.
- Analysis of collected fingerprint data in detail to gain more understanding about fingerprinting. The results presented here are analyzed and compared to findings in other research.

1.3 Structure of the Thesis

Chapter 2 gives an overview of related work regarding fingerprinting. It also describes other tracking mechanisms like tracking during one browsing session, persistent tracking and cache-based tracking. It finishes with the description of similar projects and a comparison table to other fingerprinting platforms.

Chapter 3 describes the design and architectural decisions for the framework. It also breaks down and depicts individual framework components and explains the fingerprinting process. Finally, it gives detailed description for all fingerprinting methods used in the framework.

Chapter 4 goes into implementation details for specific framework components and lists the implementation tools used during the development. It also describes the deployment process and briefly mentions privacy measures that have been taken into account when deploying.

Chapter 5 presents the results of two browser fingerprinting tests conducted from May to August 2016, first in general and then broken down on per-fingerprinting method basis. Finally, a comparison of data from the two tests is presented.

Chapter 6 shows a discussion and interpretation of data and also outlines the ideas for improvement of the framework and other future work.

Chapter 7 concludes the thesis with a short summary.

CHAPTER 2

Background

In this chapter we describe the development of online tracking mechanisms leading up to browser fingerprinting. These include tracking mechanisms in the scope of current session, persistent tracking and cache-exploiting mechanisms. Next, we present works related to browser fingerprinting upon which we base our research. Lastly, we mention projects with similar efforts and compare different fingerprinting platforms.

2.1 History of Online Tracking

In this section we briefly describe some of the tracking mechanisms through a historical context, while Bujlow et al. [17] and also Lerner et al. [40] give a more thorough overview.

2.1.1 Session-scoped tracking

The earlies known tracking mechanism was accomplished via HTTP protocol by adding an arbitrary unique identifier to HTTP's GET method or by passing it as a parameter in HTTP POST method. This way the web server would be able to track and assign the HTTP requests to a certain web client without relying on the IP address. The identifier could be a randomly generated number or a time stamp, for example. The scope of this technique is bound to one browsing session and would become useless once the user closed the web browser. It is, however, possible to pass the identifiers to a 3rd party domain by embedding a resource, like scripts or images, and creating a HTTP request to that domain via cross-origin resource sharing (CORS) [51]. Despite its limited effect, this tracking mechanism is still widespread today.

Another example of session-scoped tracking mechanism was the misuse of `window.name` property of Document Object Model (DOM) [37]. This property is capable of storing up to 2MB of text which can be used by websites to store more data than allowed via cookies.

2.1.2 Persistent tracking

Cookies were introduced in 1994 as a state management mechanism for the otherwise stateless HTTP protocol and allowed the web servers to store limited data (up to 4KB) on the web client in order to keep a stateful session with that client [11]. The server stores a cookie via `Set-Cookie` HTTP header followed by arbitrary data, which is then included upon each consequent HTTP GET request via `Cookie` header. Cookies are divided in two categories:

- *Session cookies* that expire once the web browser is closed and
- *Persistent cookies* that can have a discretionary expiration date, which is set upon cookie creation.

Research [38] [41] [43] shows that the use of cookies extends beyond the session state management, as originally intended and is extensively (mis)used for online tracking of users. Webpages are increasingly including content, be it multimedia or advertising scripts, hosted on external third-party websites that often act as information aggregators [38], linking the users' online behavior and browsing history across different websites. This type of online tracking is called *third-party tracking*. Li et al. [41] have shown that 46% of *home pages* of Alexa's[1] top 10 thousand websites have at least one third-party tracker. Although users do have an option to periodically remove unwanted cookies or to disable them altogether, research from 2007 [3] shows that some of 30% users delete cookies within a span of one month from the time of their creation. However, one can argue that this number could be higher today, considering the raised awareness towards cookies, ad-blocking and online tracking.

Cookie leaks and *cookie syncing* represent another big concern because the cookies from one domain are passed onto another. Adobe Flash and Java browser plugins also feature APIs for storing data on the client, via Local Storage Objects (LSO) [7] (or *flash cookies*) and JNLP Persistence Service [61], respectively. These methods allow for larger storage (LSO: 100KB) and are harder to remove than regular HTTP cookies, since they are not deleted when clearing browser data. This way, Flash cookies can facilitate cookie regeneration leading to *evercookies* or *supercookies* since they are very hard to get rid of [68]. Other noteworthy *persistent* tracking mechanisms, which will be described in Chapter 3, include:

- HTML5 `globalStorage`
- HTML5 `localStorage`
- HTML5 `sessionStorage`
- HTML5 `IndexedDB`
- IE `userData` storage
- Web SQL Database

One of the fingerprinting methods in our framework checks for the existence and browser support of the above-mentioned persistence facilities and uses it as part of the fingerprint.

[1]http://www.alexa.com

2.1.3 Cache-exploiting tracking

There exist many tracking mechanisms that exploit various caching facilities of the browser. In 2010, it was possible to read browsing history via JavaScript by inspecting the color property of URLs [75]. The technique, obsolete today, could inspect many thousands of predefined links and reconstruct a partial browsing history. However, a similar technique that exploits browser caching, is still utilized nowadays. It is possible for a website to check if a resource (image or script) has been downloaded and cached by a browser at some point in the past. This can be exploited in various ways [17]:

- by embedded identifiers in cached documents, for example a `<div>` element with a specific ID in a HTML document,
- by running loading performance tests with JavaScript, differences in timing could mean that the object is either cached or downloaded from server,
- *ETags* [10] and *Last-Modified* HTTP headers, and compare versions of local resources to their counterparts on server.

Besides browser caching facilities, a certain technique exploits the DNS cache by measuring the responsiveness of DNS lookup - if the website was visited before, then an entry must be present in the DNS cache and the lookup time should be shorter [26].

2.2 Related Work

This section provides a short overview of work related to browser fingerprinting. We divide the previous research in two categories:

1. statistical analysis of fingerprints collected over some period of time and
2. individual fingerprinting methods for exploiting web browser features.

For the former, we try to explain the pros and cons and to illustrate the differences to this very work, since it itself falls into the first category. For the latter, we briefly explain the technique and why we decided to include it into our framework.

In his seminal work on browser fingerprinting from 2010, Eckersley [22] shows that the distribution of the browser fingerprint holds at least 18.1 bits of entropy, meaning that by picking a browser at random, we could expect at best, that only one in 286.777 browsers shares its fingerprint ($2^{18.1} \approx 286.777$). As part of the *Panopticlick*[2] website, he developed an algorithm which runs in a browser and calculates the fingerprint by collecting and comparing multiple system and browser characteristics. Among the 470.161 collected fingerprints, 83,6% have shown to be unique. This number was even higher (94,2%) for browsers with Flash and Java plugins enabled [63]. He also observes the changes in fingerprints over time among the users who

[2]https://panopticlick.eff.org/

visited the website multiple times. He was able to anticipate a fingerprint change using a simple algorithm with a success rate of over 99.1%. According to the study, the most revealing characteristics of the browsers are:

- List of installed browser plug-ins with entropy of 15.4 bits
- List of installed system fonts with 13.9 bits of entropy
- User-Agent string with 10.0 bits of entropy
- HTTP Accept header with 6.09 bits of entropy, etc.

Furthermore, he argues that anti-fingerprinting techniques are unsuccessful and make the individual users stand out even more, if not embraced by an adequate amount of people surfing the Web. Although this study is dated, we use it as a starting point for our research and the framework. We build upon the browser and operating system characteristics collected in this study and combine it with recent fingerprinting techniques in order to understand their effect on the identifiability of the browser and their corresponding entropy values. Another downside to this project is that there is no source-code available.

In 2012, Mowery et al. [46] presented a fingerprinting method that utilizes WebGL API [42] to render 3D scenes in combination with text rendering on a HTML5 `<canvas>` element in a web page. The fingerprint is closely tied with operating system and underlying hardware - the GPU - because it relies on the system drivers and graphics card functionality to render a 3D scene in the browser. The fingerprinted value consists of pixel values of the rendered images that are passed through a cryptographic function and saved as a hash of values. This observed values among 294 collected fingerprints appear to be consistent on multiple fingerprinting attempts and has an observed entropy of 5.73 bits. The authors argue, however, that this value might be even higher if the tests were constructed in a more specialized way. Another important fact is that the WebGL fingerprint is orthogonal to other fingerprinting methods. The downside to this research is that the WebGL fingerprinting was conducted in isolation with regards to the browser characteristics used in the Panopticlick study. We use the ideas from this research in our framework to query the information about GPU and their respective driver version and capabilities.

In 2013, Acar et al. [5] implemented and deployed *FPDetective*, a framework for detection and analysis of browser fingerprints in the wild. They conducted a large-scale analysis by crawling one million most popular websites according to Alexa. FPDetective was able to detect 16 new fingerprinting scripts and expose new fingerprinting practices. 404 out of top million websites were using scripts from 3rd party fingerprinting providers for JavaScript based font probing, which they argue is a lower-bound figure constrained by the limitations of the FPDetective's automated crawler and that many fingerprinting techniques might not have been detected. Next, 97 out of the top 10 thousand websites were using Flash based font detection. We employ both the JavaScript font probing technique and Flash based font detection in our framework, in which the former is used as a side-channeling mechanism in cases where Flash is not installed or is disabled. Furthermore, Acar et al. report that many fingerprinting providers employ anti-debugging measurements associated with JavaScript malware, possibly to remove the evidence of fingerprinting actions and to evade detection from various privacy tools.

In another study from 2013, Nikiforakis et al [59] conducted a detailed analysis of three popular commercial fingerprinting service providers: Bluecava[3], Iovation[4] and ThreatMetrix[5]. They, too, found out that font detection was an important part of the fingerprint and that they utilized both JavaScript-based font probing and Flash font detection. The usage of Flash, however, goes beyond the sole purpose of detecting fonts - it is also used to detect screen resolution and presence of multiple monitors, identification of OS and kernel versions on Linux devices and the detection and circumvention of HTTP proxies in order to disclose the real IP address of the user. In the latter case, the Flash objects employed as a part of the fingerprinting script were ignoring the proxy preferences set in a browser and were communicating with the providers' server directly, while exchanging certain alphanumeric tokens, possibly an identifier to correlate different IP addresses. The authors also noticed that the aforementioned providers were probing for presence of a certain browser plugins. The plugins in questions were fingerprinting libraries shipped as Internet Explorer ActiveX plugins, which had full access to the Windows registry, hard drive identifiers, computer name, TCP/IP parameters, Windows Digital Product ID and system drivers - hence creating a much stronger identification link than JavaScript/Flash based fingerprinting techniques. Moreover, they found out that 40 from top 10 thousand Alexa websites were using the fingerprinting scripts from the three providers, some of which used the fingerprinting scripts to circumvent credit card fraud. They also made their own fingerprinting script that analyzes the structure of `navigator` and `screen` DOM [37] objects between different browsers and found that the differences in the order of properties in the objects suffices to distinguish different browser families. We employ this technique in our framework indirectly, through the use of 3rd party JavaScript libraries (more on that in Chapter 3). Lastly, they analyzed popular browser extension for User-Agent spoofing and found out, that they fall short against fingerprinting.

In 2014, Acar et al. [4] conducted a first large-scale study of canvas fingerprinting methods described in [46] and found out that 5% of the top 100 thousand Alexa websites employed canvas fingerprinting which makes it one of the most prevalent techniques in practice. Second, they analyzed the presence of *evercookies* and cookie respawning via Flash and `IndexedDB` storage and found out that 5% of top 200 websites used Flash cookies and 33 different Flash cookies were used to regenerate 175 'regular' HTTP cookies on 107 of top 10 thousand Alexa websites. Cookie syncing allows cookie regeneration by passing on cookie information from one domain to another. It allows trackers to link user's browsing history after the clearing of cookies. This study shows that browsing history of 1.4% of the users can be linked this way. Although the persistent tracking methods and cookie syncing are out of bounds of this work, this research brought the idea of checking for storage capabilities of the browser, like Web Storage [35], Web SQL Database [34], Indexed Database [60], File Access API [9], etc. and using it as part of our fingerprinting vector.

In 2013 Unger et al. [69] proposed a reliable browsers fingerprinting method based on CSS3 and HTML5 as part of a framework for HTTP(S) session management and prevent session hi-

[3]http://www.bluecava.com

[4]http://www.iovation.com

[5]http://www.threatmetrix.com

jacking. Since CSS3 and HTML5 standards' implementation status may differ across diverse browser vendors, it is possible to exploit these differences in order to detect the browser type and version. They identified three CSS-based methods for fingerprinting: CSS properties, CSS selectors and CSS filters. By applying CSS attributes to certain HTML elements and querying for the result via JavaScript they were able to observe differences between various browsers. Similarly to CSS, they applied the same methods to HTML5 and used both components as part of a fingerprint to identify the browser without the need to rely on UserAgent string. We use the methods described here in our framework by relying on a 3rd party JavaScript library for CSS3/HTML5 feature detection.

In another paper from 2013. Mulazzani et al. [57] presented a similar fingerprinting method based on the identification of browser's underlying JavaScript engine. They compared the test results of running a set of conformance tests that cover ECMAScript standard [24] in different browsers and browser versions on various operating systems including mobile platforms. Depending on the correctness of the implementation of JavaScript specifications some browsers would fail or successfully run certain tests. They used *test262*[6] suite for ECMAScript, which consists of thousands of tests. However, not all tests are necessary for a successful identification of a certain browser, hence they identified the set of failing tests and created a minimum fingerprint for each browser. The downside of this method is that a new test set or minimal fingerprint has to be created for every new browser version in order to compare new browser instances against it. The fact that all major browser vendors introduced rapid development and release cycles would mean that, in order to include this fingerprinting method in our framework, we would constantly have to update the database of minimal fingerprints in order for this method to bring consistent results. Because of this, we have decided to not include this method in our framework.

In 2015, Husák et al. [33] present a fingerprinting method for real-time identification of HTTPS clients based on network monitoring by analyzing the handshake of SSL/TLS protocol. Their experiment shows that it is possible to estimate the UserAgent of the client by analyzing SSL/TLS handshake packets. By monitoring live network traffic, they managed to collect almost 13 thousand unique combinations of UserAgent strings and SSL/TLS specific cipher-suite lists and filtered 316 unique cipher-suite lists. They were able to assign a UserAgent to almost all cipher-suite lists with a significant level of probability. We have included this fingerprinting method in our framework, which has had a significant impact on its design and implementation, since the fingerprinting is performed on the server side. The process is described in detail in section 3.3.3.

Fifield et al. [28] in 2015 presented another fingerprinting method that exploits the font attributes by measuring the on-screen dimensions of font glyphs. They thoroughly surveyed the layout of over 125 thousand Unicode code points by fingerprinting 1000 users and found out that the number of code points can be reduced to 43 in order for the fingerprint to have the same effect. The layout and font measurement is invisible to the users and takes only milliseconds to execute. Compared to the other font-related fingerprinting methods described above, this method does not output a list of installed fonts but rather a list of individual glyph dimensions of the font. Authors admit that this method is inferior to other methods like canvas fingerprinting, but argue

[6]https://github.com/tc39/test262

that it is still relevant because it is effective against Tor browsers, which is at the moment of writing the only browser capable of blocking the canvas fingerprint [66]. The authors have made the project source-code available[7] and we have included this method in our framework.

In a more recent study from 2016, Laperdrix et al. [39] perform a similar experiment to Panopticlick with the difference of including Canvas and WebGL fingerprinting. They published a website AmIUnique[8] and collected over 100 thousand fingerprint over a course of one year and compared the results to Eckersley [22]. They claim that innovations in HTML5, especially Canvas API, have had a large impact on fingerprinting. They also show that fingerprinting mobile users is as effective as for desktop browsers. The source code[9] for their website has been publushed as part of the DIVERSIFY project[10]. They have also stated that they removed those fingerprints from their study that had JavaScript disabled, which is a huge difference to our approach. With the inclusion of two fingerprinting methods that don't rely on JavaScript (HTTP headers and SSL/TLS handshake fingerpriting) we don't want to ignore the datasets with disabled JavaScript.

At the time of writing, another research related to browser fingerprinting has emerged. Englehardt et al. [25] conducted a similar experiment to [5]. They show the largest and most exhaustive measurement of online tracking by crawling the top 1 million websites with the help of OpenWPM[11], a web privacy measurement framework that in reality is an automated version of a commercial browser, developed at WebTAP Project[12] of Princeton University. They observe the use of fingerprinting methods and cookies, cookie syncing and the effect of browser privacy tools. During their investigations, they have discovered and analyzed a novel fingerprinting method utilizing Web Audio API [65] and `AudioContext` interface [48], an audio processing graph that can execute audio processing and decoding. The fingerprint checks for the availability of the API, but also calculated hash of values generated by the API. The visualization of the technique is available on the WebTAB Audio Fingerprint website[13]. We have retroactively included the `AudioContext` fingerprinting method to our framework.

2.3 Other Projects

In this section we mention a couple of more projects related to browser fingerprinting that have emerged at the time of writing.

Study on Browser Fingerprinting [14] is a project at Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) that aims to explore the diversity of browser fingerprints. They allowed the visitors of the website to register via and receive a link via email that would take them to the fingerprinting website. During the period of four weeks, they sent an email once per week with

[7] https://repo.eecs.berkeley.edu/git-anon/users/fifield/fontfp.git

[8] https://www.amiunique.org

[9] https://github.com/DIVERSIFY-project/amiunique

[10] http://diversify-project.eu

[11] https://github.com/citp/OpenWPM

[12] https://webtap.princeton.edu/

[13] https://webtap.princeton.edu/audio-fp

[14] https://browser-fingerprint.cs.fau.de/

the aim to analyze how the fingerprint changes over time. As of now they have not published any results in form of a research paper. They have, however, updated their website with the statistical analysis of the collected data.

Browserprint is a project developed at University of Adelaide whose website[15] resembles that of Panopticlick, but with updated fingerprinting methods. The project is open source[16].

BrowserLeaks [17] is a website that allows the visitors to test their browsers against various fingerprinting methods and inspect the results, but does not collect data or display statistics in any form. It also offers brief explanation and code samples for the fingerprinting methods.

The table 2.1 shows a side-by-side comparison of features implemented on the previously described fingerprinting platforms.

[15]https://browserprint.info/
[16]https://github.com/qqTYXn7/browserprint
[17]https://www.browserleaks.com/

Feature	**Panopticlick**	**AmIUnique**	**Browserprint**	**Our solution**
HTTP headers	✓	✓	✓	✓
Headers order				✓
SSL/TLS connection info				✓
SSL/TLS cipher suites				✓
SSL/TLS extensions				✓
Platform	✓*	✓	✓	✓*
Language	✓*	✓	✓	✓*
Screen resolution	✓*	✓	✓	✓*
Timezone	✓	✓	✓	✓
Cookies enabled	✓	✓	✓	✓
Supercookie test	✓	✓	✓	✓**
Use of AdBlockers	✓	✓	✓	✓
Flash presence	✓	✓	✓	✓
List of browser plugins	✓	✓	✓	✓
List of fonts (JS & Flash)	✓	✓	✓	✓
List of fonts (pure CSS)			✓	
Font metrics			✓	✓
HTML5 canvas	✓***	✓	✓	✓
WebGL info		✓	✓	✓
AudioContext			✓	✓
Touch events			✓	✓**
Social buttons			✓	
HTML5/CSS3 features				✓

*Only JS detection, no Flash

**Via HTML5/CSS features detection

***Since version 2.0

Table 2.1: Comparison of fingerprinting features

CHAPTER 3

Design

This chapter introduces key concepts from the framework that form the basis of this thesis. It starts with a list of requirements imposed on the framework. Next, it provides an overview of the framework architecture including architectural decisions. Furthermore, the fingerprinting process is described in detail. Last section provides detailed descriptions of the individual fingerprinting methods used in the framework, together with all properties and browser characteristics that we collect.

3.1 Requirements

This section briefly describes both functional and non-functional requirements imposed on our framework. The process of designing and implementing the framework was driven by the requirements listed here.

Extensibility In order for our framework to encompass as many fingerprinting methods as possible, the extensibility had to be taken into consideration. This has been a major driving force during the systems design and has enabled us to easily incorporate additional fingerprinting methods to the framework both during and after the development. By maintaining low coupling and well defined interfaces, the framework provides a way to add new fingerprinting methods without the need for changing the underlying architecture.

Robustness Modern web browsers are complex applications that support various web standards and protocols. Depending on the browser type and version, there can be slight differences in how the browser interprets these standards and respond to certain events when processing a web page. We have to make sure that our framework is robust and fault-tolerant to be able to account for various corner cases and exceptions in order to successfully perform the task.

Scalability Although the server part of the framework does not maintain a session state with the client, scalability in regards to processing of growing data set has to be taken into account. The design and implementation choices have enabled the framework to effectively accommodate that growth. Scalability was also influenced by deployment choices and configuration which will be discussed in next chapter.

Privacy Since the focus of this framework is a privacy-related topic, the anonymity of data collected during the research is of highest priority. In order to make data anonymous and disassociate it from actual participants in our study, certain measures in implementation and deployment of the framework had to be made. These will be discussed in the next chapter.

3.2 Framework Overview

3.2.1 Terminology

Before proceeding with presenting the architecture of the framework and going into detail outlining the fingerprinting process, we define some terms used throughout this and following chapters.

- **Fingerprinting script** is one or more JavaScript functions that run in a browser and exploit its functionality in order to return one or more values of some browser or operating system characteristics. Scripts can be contained in separate JavaScript files, directly embedded in the page or can be a combination of both.
- **Fingerprinting method** is *usually* a collection of fingerprinting scripts and other static files on which the scripts are dependent (e.g. flash SWF files, external font files, etc.), grouped into one fingerprinting method because they either pertain to a specific research or fall into the same logical topic. However, not all fingerprinting methods are executed directly in the browser, but are performed on server instead. Thus they don't consist of JavaScript files or similar, but of server-side code contained in one or more modules. Result of a fingerprinting method is one or more values or *attributes*.
- **Fingerprint** is a collection of results from multiple fingerprinting methods, transmitted from browser to the server.

3.2.2 Architecture

As visible in figure 3.1, our framework adheres to the client/server architecture and thus is composed of two main parts:

- **Client part** - which is presented in form of a web page running in a browser and consists of collection of JavaScript functions, both embedded in the web page and as standalone script files, as well as some additional files that are necessary for fingerprinting methods to function properly. The client part is also responsible for rendering the fingerprinting results received from the server.

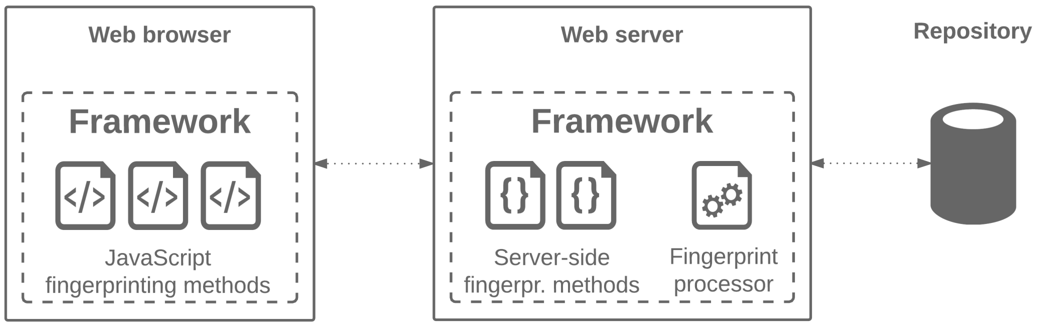


Figure 3.1: General overview of the framework

- **Server part** - which runs as a web application on a web server. This part of framework furthermore consists of a collection of non-JavaScript fingerprinting methods (more on this later in the chapter) and a business logic for processing the fingerprinting results. It also defines interfaces necessary for database connectivity and a templating mechanism for dynamically serving fingerprinting methods to clients.

Repository is not a direct part of the framework, however it is necessary for storing fingerprinting data and its evaluation.

3.2.3 Fingerprinting Process

This section describes the fingerprinting process and how the two framework parts interact with each other.

The fingerprint process can be roughly split into three steps:

1. Client (browser) runs fingerprinting scripts and submits the fingerprint to server
2. Server parses and persists the fingerprint
3. Server calculates statistics for fingerprint and returns result to client

Figure 3.2 illustrates the process in a more detailed manner. The fingerprinting process commences when the client (browser) requests a web page that contains the fingerprinting methods. On a more technical level, this corresponds to HTTP `GET` method. The server crafts a web page by referencing all fingerprinting scripts and additional resources needed for fingerprinting and serves it to the client. Once the requested page is loaded, the fingerprinting scripts are executed and the fingerprint is asynchronously sent to the server. This is achieved by issuing an HTTP `POST` request via AJAX [47]. The server then parses the fingerprint and persists it to the repository. Next, it calculates the statistics for each attribute and checks if the fingerprint is unique. The interaction ends when the server returns the result to the client. The **result** is a

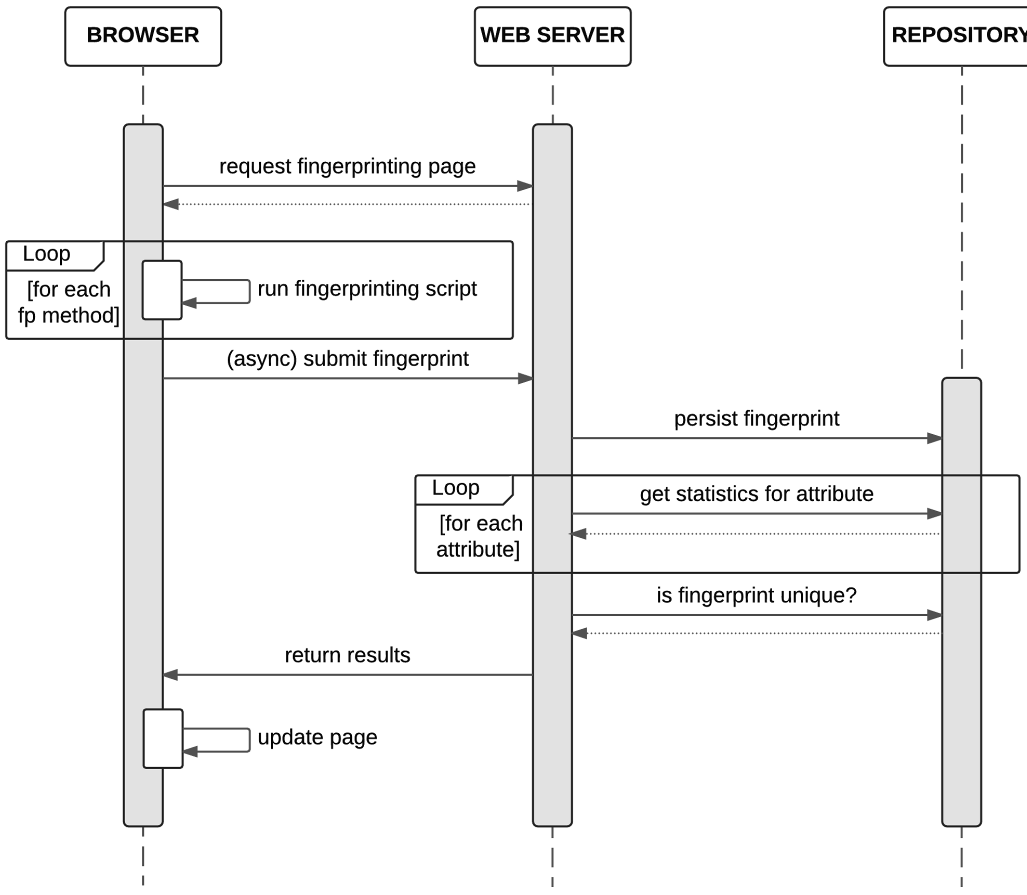


Figure 3.2: Fingerprinting process activity diagram

HTML table with all the fingerprint values and corresponding statistics, which is then (again, asynchronously) rendered by the client.

The notion of more technical terms like HTTP `GET` and `POST` methods and AJAX is necessary to make a distinction to the next fingerprinting scenario. Previous example illustrated the fingerprinting process for a client with enabled JavaScript. If, however, the client has disabled JavaScript or does not support it (bots, headless browsers, other non-browser HTTP clients etc.), the fingerprinting process is different, as shown in figure 3.3.

The most obvious difference is the absence of *run fingerprint script* loop and *submit fingerprint* activity. Because the client cannot run JavaScript code, there are no fingerprinting scripts that run on the client, nor is there an asynchronous AJAX `POST` to server. Another deviation from figure 3.2 is the missing response for the initial request of the fingerprinting page. When a client that does not support JavaScript requests the fingerprinting page (via HTTP `GET`), all fingerprint-relevant data is already contained in that request.

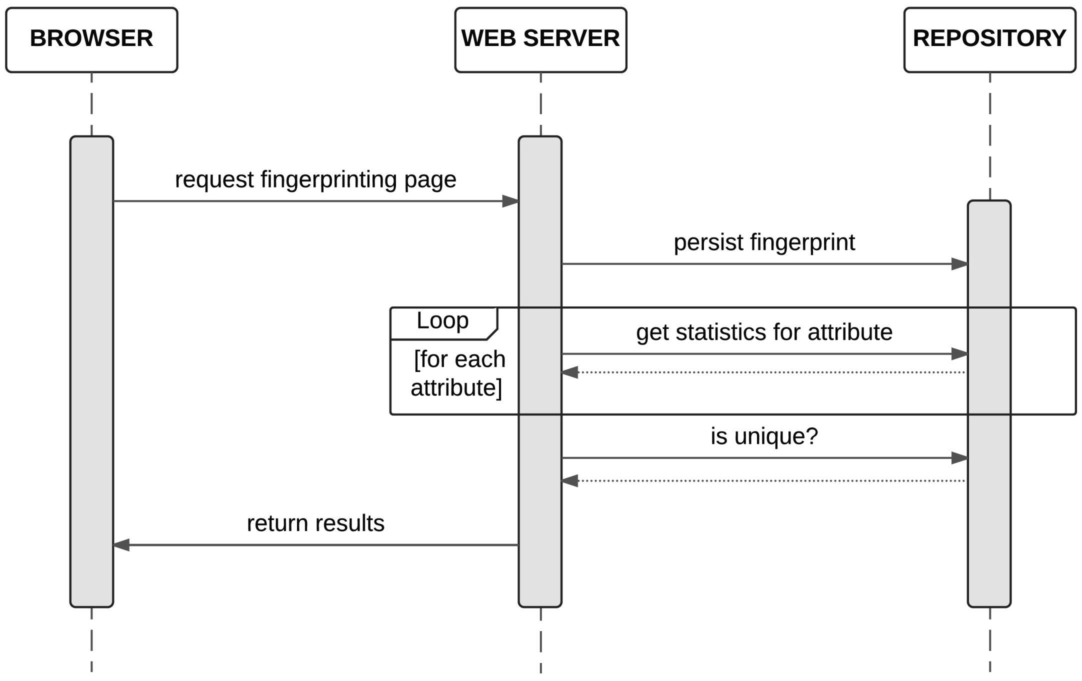


Figure 3.3: Fingerprinting process without JavaScript activity diagram

As mentioned before, not all fingerprinting methods are executed *in* the browser via JavaScript or other client-side technologies. Server-side fingerprinting methods implemented in the framework rely on analyzing the underlying HTTP(S) connection of client's request and in the next step framework parses and persists the fingerprint. Like in a previous scenario, the framework calculates statistics and the uniqueness of a fingerprint and *directly* returns the result to a client, without the need for an intermediate step and a HTTP `POST`.

3.2.4 Framework Components

Figure 3.4 illustrates a breakdown of the server part of framework into components. The figure represents final iteration of the framework design. This section describes individual components and the interaction between them.

Router This component can be regarded as the interface that communicates with the client part of the framework. When a client request comes in, the router serves fingerprinting scripts

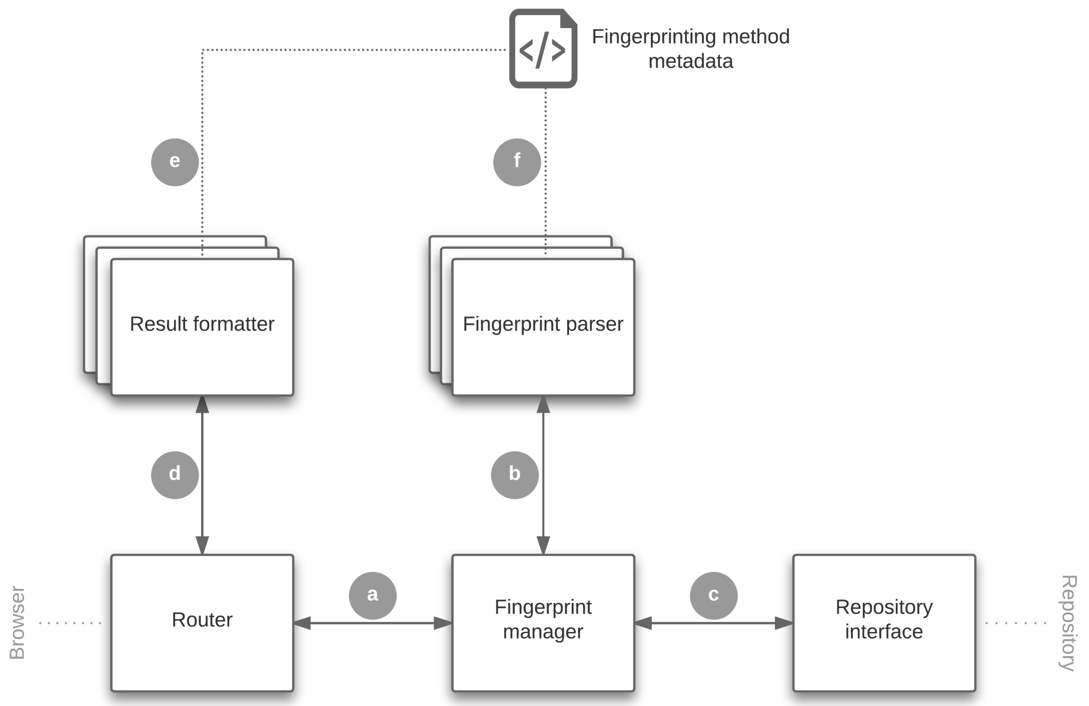


Figure 3.4: Server-side framework component diagram

and other static files necessary for fingerprinting. When a client response (or fingerprint) comes in, the router forwards it to fingerprint manager **(a)**, where the fingerprint is processed.

Fingerprint manager This component serves as a coordinator for other components of the framework. It was mentioned before that one of the main properties of the framework is its extensibility. To achieve this, we treat each fingerprinting method separately. That also means the fingerprint is composed of multiple parts, each pertaining to a single fingerprinting method. When the server receives a fingerprint from client, the fingerprint manager is responsible of deconstructing it and forwarding the right part of fingerprint to the right fingerprint parser **(b)**. Once the parsing is done for all parts of fingerprint, manager forwards the parsed fingerprint to repository interface **(c)**.

Fingerprint parser(s) This component processes the relevant part of the fingerprint and returns a partial result. For every fingerprinting method registered in the framework, there is a metadata file with a description of that method's attributes, parsing information etc. Fingerprint parser consults the metadata file **(f)** in order to correctly interpret and process the result. That way, each fingerprinting method can have its own parser and thus is independent from other methods.

Repository interface This component communicates with the back-end fingerprint repository. It is responsible for storing fingerprint information as well as calculating statistics for each fingerprint attribute. Once the fingerprint has been stored and the statistics have been evaluated, the component returns the information to fingerprint manager **(c)** who in turn forwards it to router **(a)**.

Result formatter(s) The fingerprint and statistics are forwarded to result formatter(s) in order to construct a result for client to review her fingerprinting information. In a similar fashion to fingerprint parser, each fingerprinting method has a dedicated parser that can render the result independently. It does so by referring to fingerprinting method's metadata file **(e)**.

Figure 3.5 shows the original, more simple architecture of framework's server-side part. The absence of fingerprint parsers and multiple result formatters shows that originally fingerprinting methods were not treated separately. In this design, all fingerprinting methods and their corresponding attributes were processed in the same fashion, but the addition of server side fingerprinting methods like cipher suites fingerprinting (described in the next section) has forced us to rethink the way how the fingerprinting data is processed and thus has influenced the framework design. This decision to treat all fingerprinting methods individually has allowed us to make the framework flexible and extensible.

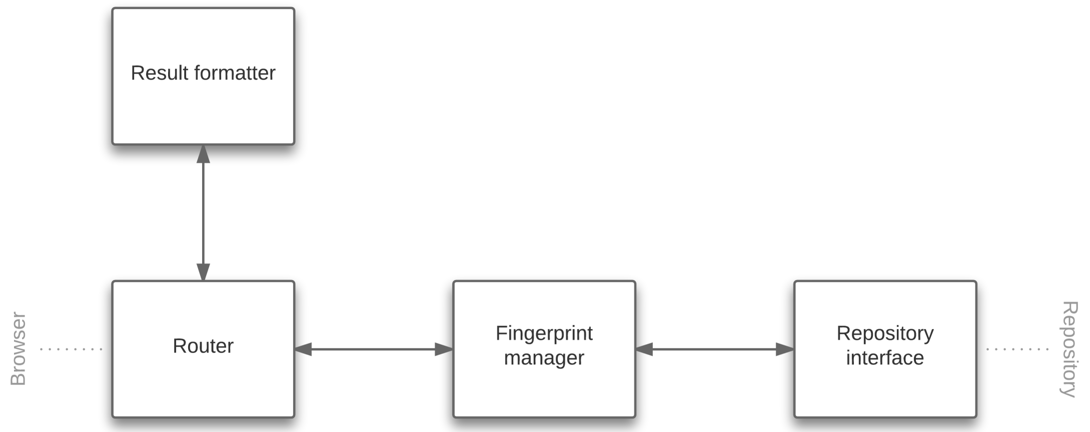


Figure 3.5: Server-side framework - original design component diagram

Figure 3.6 illustrates how the framework produces fingerprinting page in a generic way. By consulting fingerprinting method metadata it references all relevant fingerprinting scripts and additional static files required by scripts in a template. It also references some general-purpose JavaScripts that glue other scripts together and provide the execution mechanism on the client side. When the client requests the fingerprinting page the templating engine generates the fingerprinting page that contains finished HTML and JavaScript and serves it to the client.

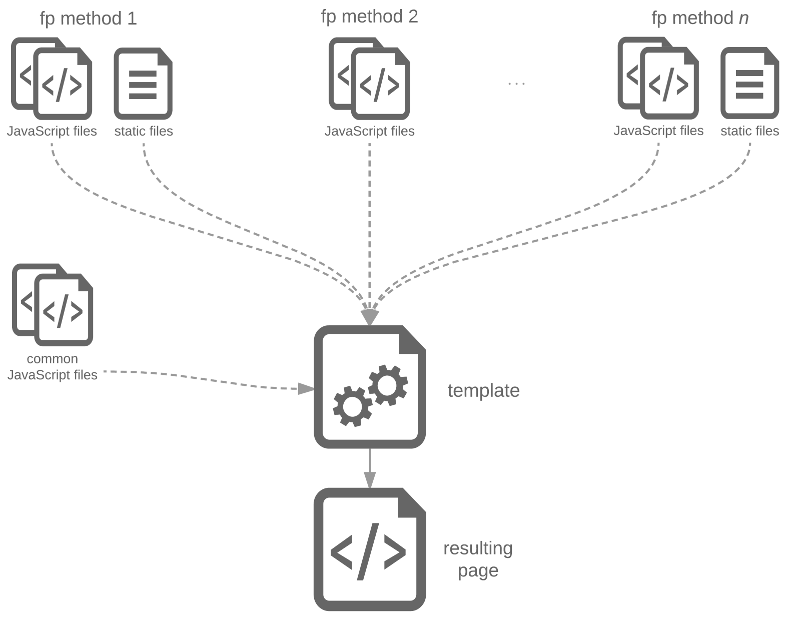


Figure 3.6: Generating the fingerprinting scripts

3.3 Fingerprinting Methods

This section describes each fingerprinting method in detail along with their corresponding properties. All the properties listed and described here are included in the fingerprinting algorithm and are collected and analyzed by the framework.

3.3.1 Navigator Features

This fingerprinting method encompasses the majority of fingerprinting properties that were used in the Panopticlick[1] survey [22]. Most properties described here are easily accessible via DOM Navigator object [71] which makes it logical to leave them in the same organizational group.

Platform This property of the Navigator DOM object [54] represents the platform on which the browser is executing, although earlier versions of JavaScript specification defined it as a machine type for which the browser was *compiled* [58]. Example values for this property are

[1]https://panopticlick.eff.org

- `Win32` for both 32-bit and 64-bit versions of Windows,
- `Linux x86_64` for 64-bit Linux distributions,
- `MacIntel` for MacOS and
- `Linux armv7l` for Android OS running on devices with ARM v7 processor.

Language This property represents the preferred language of the user and/or the browsers UI language. Example values are `en`, `en-US`, `de-AT`, `de-DE` etc. Since it is possible to define more than one preferred language in the browser, this property returns the first, or the language with highest priority. If this value differs from the first element of `navigator.languages` array, we can assume that the user has lied about the language and so we append an `*` to the value, e.g. `en-US*`.

Screen resolution This property is a combination of screen width, screen height and the color depth. It is accessible via the Screen DOM object [55]. Example value is `1280x720x24`. If the screen width or height is smaller than the available screen width or height, respectively, we assume the user has lied about the resolution and append an `*` to the value, analog to the language. For headless browsers, this property is unavailable. Interestingly, in a multi-monitor setup with different resolutions the Scren DOM object reports the dimensions of the screens on which the browser windows is currently open.

Timezone This property represents the time zone offset for the current locale from UTC, expressed in minutes. For example `-120` means UTC+02:00 - Central European Time (CET) with daylight saving time.

'Do Not Track' flag *Do Not Track* (or DNT for short) is a mechanism for expressing *tracking preference* of the user. It is defined by W3C[2] as both an HTTP header and an HTML DOM object [70]. Values for this property are `1` if DNT is enabled, `0` if the user opted-in for tracking and `unspecified` (default value) if user set no preferences. Older browsers used values `yes` and `no` for opting out and in, respectively. Websites that honor the DNT preference can set the `Tk` response header to specify the tracking status.

Cookies This property tells if the user has enabled cookies. It can take on values `true` and `false`. Besides that, we also use cookies to track returning users in order to measure fingerprinting change over time.

AdBlocker installed This property shows if an ad-blocker is in use. In order to detect the presence of an ad-blocker we create a JavaScript file `advert.js` (See code listing 3.2), define a `<div>` element called `ads` and append it to HTML document body. Then we go on and try to reference the element by id. If the ad-blocker is present it will block the element because its name matches the **ad** blocking rule.

[2]World Wide Web Consortium, https://www.w3c.org

Flash installed This property tells us if Adobe Flash is installed and/or blocked. We rely on Modernizr library [44] for detection.

Plugins This property enumerates all installed and enabled NPAPI plugins [53]. Plugins allow the user to display additional content types that are not natively supported in a browser, like different video formats and PDF documents. We collect plugin names, filenames alongside their descriptions and registered media (MIME) types. In case of Internet Explorer it is not possible to enumerate the plugins via `navigator.plugins` interface, but rather to reverse lookup the most popular plugins by name, e.g. `AcroPDF.PDF`, `QuickTime.QuickTime`, `Skype.Detection`, etc. We use Fingerprintjs2 library [29] for detection.

```
platform = navigator.platform;
language = navigator.language || navigator.userLanguage ||
           navigator.browserLanguage || navigator.systemLanguage;
if (language !== navigator.languages[0].substr(0, 2))
  language += '*';

screenres = screen.width + 'x' + screen.height + 'x' + screen.colorDepth;
if (screen.width < screen.availWidth ||
    screen.height < screen.availHeight)
  screenres += '*';

timezone = new Date().getTimezoneOffset();
dnt = navigator.doNotTrack || navigator.msDoNotTrack ||
      window.doNotTrack; // Safari and IE use different property

cookie = navigator.cookieEnabled;
adblock = document.getElementById('ads') ? 'False' : 'True'; // advert.js
flash = Modernizr.flash.blocked ? 'Blocked' :
       (Modernizr.flash ? 'True' : 'False');

var plugins = [];
for (var i = 0, l = navigator.plugins.length; i < l; i++)
  plugins.push(navigator.plugins[i]);
```

Listing 3.1: Navigator features detection

```
var ads = document.createElement('div');
ads.setAttribute('id', 'ads');
document.body.appendChild(ads);
```

Listing 3.2: Advert.js

3.3.2 HTTP Headers

Every time a browser or other HTTP client establishes a connection with server, it transmits certain HTTP headers that give detailed information about the client. This section describes HTTP request header fields used by our fingerprinting algorithm. It is worth noting that this fingerprinting method does not depend on JavaScript since all HTTP client, regardless of JavaScript support, transmit HTTP headers because they're part of the underlying HTTP protocol. All properties except for *headers order* were also used in Panopticlick [22] survey.

User Agent This header contains information about the browser (or any other client software) initiating the request. It is used by servers to tailor or customize the response in order to take advantage of clients' capabilities or to work around their limitations. The header contains one or more product tokens made up of name and version of the product and any necessary comments that are of value to the sever when processing the request [27]. For example, User-Agent header in code listing 3.3 (line 1) gives us following information:

- `Mozilla/5.0` Browser is compatible with Mozilla rendering engine.
- `Windows NT 6.3; WOW64; rv:47.0` It is running on 64-bit of Windows 8.1.
- `Gecko/20100101` It uses gecko layout engine, desktop version ('20100101').
- `Firefox/47.0` It is Firefox browser, version 47.0.

Accept This header specifies the response media types acceptable by the web browser (requester). It may contain multiple types and subtypes as well as an additional parameter *q* which indicates the relative weight or preference. If the weight is not specified, the most specific media type is preferred, than the first less specific and so on. Example `Accept` header in code listing 3.3 (line 2) has following precedence:

1. `text/html` and `application/xhtml+xml` are equally preferred
2. `application/xml`
3. `*/*`

Accept-Encoding This header indicates the preferred content-encoding of the response. `*` denotes any encoding is accepted. Analog to `Accept` header, browser can also specify `q` weight when multiple encodings are present. In code listing 3.3 (line 3) the browser accepts `gzip`, `deflate` and `br` encodings with equal preference.

Accept-Language This header specifies the language*s* that are preferred by the browser. `q` weight can be also specified for each language. In code listing 3.3 (line 4) the browser has US English as preferred language, but will accept other types of English.

Connection This header tells the server if the TCP connection should stay open and be reused for future requests by specifying `keep-value` in the header. Although this value is deprecated, it is used by all browser per default. However, there are some slight differences in capitalization (e.g. `Keep-Alive` or `Keep-alive`).

Headers order This property is not a header by itself, but rather a combination of previously mentioned headers which we collect while preserving the order in which they appear during the HTTP(S) connection.

```
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:47.0) Gecko/20100101
    Firefox/47.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
```

Listing 3.3: Example HTTP headers

3.3.3 SSL/TLS Cipher Suites Fingerprinting

Transport Layer Security (TLS), and its predecessor *Secure Sockets Layer* (SSL), are cryptographic protocols for secure communication over the Internet. TLS protocol offers cryptographic security, interoperability, relative efficiency and extensibility. The latter provides a way to incorporate new key and bulk encryption methods into the protocol without the need for creating a new protocol and implementing a new encryption library [20].

In order to establish a secure connection, the client and the server first have to agree upon an encryption method they both understand. This agreement takes place during the *TLS handshake phase* which is depicted in figure 3.7. The client begins by sending a `ClientHello` message to the server. This message contains the *TLS version* supported by the client, a list of *cipher suites*, a list of *supported extensions* and a *compression method*. It also contains a sequence of random 32 bytes, as well as a session ID, which are of little interest for fingerprinting purposes since they change for every session. The server then responds with a `ServerHello` message that contains the TLS version and supported ciphers, followed by the X.509 `Certificate`. Since the parties have not yet agreed upon a mutual encryption method, the initial packets of the communication are unencrypted and transmitted in plain text over the network. As it turns out, the packets, especially the list of cipher suites, differ from one client to another and as such are subject to fingerprinting.

Listing 3.6 shows a formatted `ClientHello` message. We use Fiddler[3] debugging proxy to intercept the HTTPS traffic by installing Fiddler's 3rd party certificate which enables us to extract the `ClientHello` message from *HTTP CONNECT tunnel*.

This is another fingerprinting method that does not depend on the availability of JavaScript on the client. In any case, client connecting to HTTPS server transmits the SSL/TLS attributes described below, that were also used in [33].

Handshake Version This property shows the SSL/TLS handshake version: either `2` for SSL v2 or (usually) `3` for SSL v3+.

[3]http://www.telerik.com/fiddler

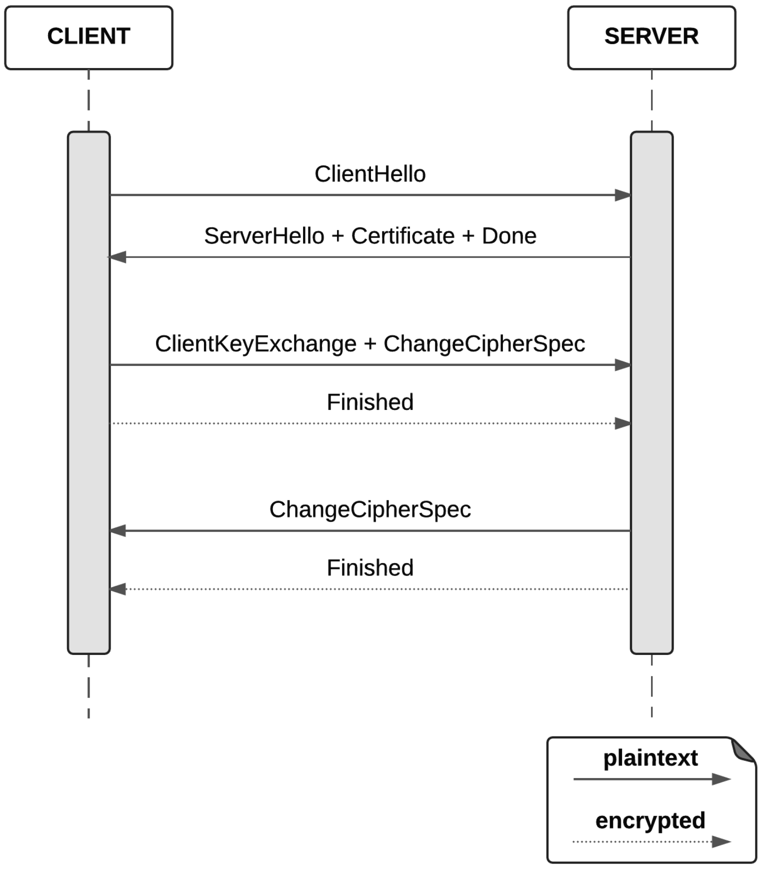


Figure 3.7: TLS handshake

Protocol Version This property tells us the preferred SSL/TLS version of the client during the session. The protocol version may take on values `3.0` that indicates the old and insecure SSL 3.0 protocol version, susceptible to POODLE attacks [45] to `3.1`, `3.2` and `3.3` indicating TLS protocol versions 1.0, 1.1, 1.2 and 1.3, respectively. According to the specifications, this *should* be the latest version supported by the client [20]. See listing 3.6 (line 9), for example.

Extensions This is the ordered list of TLS extensions which is browser specific, but also depends on the underlying operating system. RFC6066 [2] provides specification for existing TLS extensions, while [36] contains a comprehensive lists of all standardized extensions. Listing 3.6 (lines 14-23) show a list of extensions for Firefox 47.0 on Windows 8.1.

Compression This property indicates the compression methods supported by the client. All clients support `NULL` compression method, but some also support additional compression methods [32] like DEFLATE, a lossless compressed data format that uses LZ77 algorithm and Huffman coding [19]. See listing 3.6 (line 40).

Cipher Suites This is the list of cryptographic options supported by the client. The list is ordered from client's most to least preferable cipher suite. Each cipher suite consists of a key exchange algorithm, a bulk encryption algorithm, a MAC algorithm and a pseudo-random function. The server picks the preferred cipher and communicates it to the client. Listing 3.6 (lines 25-37) shows an example list of cipher suites.

3.3.4 Font detection

Detected fonts We use two different approaches to detect installed system fonts. First, we try enumerating the fonts via Flash by embedding a hidden SWF object [8] in a HTML page. The SWF object contains ActionScript code that can enumerate all installed system fonts [6] in the background:

```
var user_fonts = TextField.getFontList();
getURL("javascript:fontList(\"" + escape(user_fonts) + "\")","_self");
```

Listing 3.4: Decompiled SWF: ActionScript font detection

If Flash is enabled in the browser, we can retrieve the fonts from SWF via JavaScript:

```
if (!!Modernizr.flash && Modernizr.flash.blocked !== undefined) {
  var swfObj = document.getElementById("flh");
  if (swfObj && typeof(swfObj.GetVariable) != "undefined") {
    var available = swfObj.GetVariable("/:user_fonts");
    return available + ' (via Flash)';
  }
} // ...else detect via JS
```

Listing 3.5: JavaScript: Flash font detection

If, however, Flash is not installed or the Flash plugin is disabled, we rely on a fall-back mechanism by *probing a predefined list of fonts* with the help of JS/CSS Font Detector JavaScript library [64]. This approach is based on the assumption that text characters have different dimensions when rendered with different fonts (see figure 3.8). First we render three text strings with the browser default font faces serif, sans-serif and monospace and calculate the widths of the strings via CSS. Next, we render each font from our predefined list and compare the widths - if the width differs from all three default fonts we assume that the font is present. In contrast, if the width of our font matches one of the three default fonts, it can be assumed that the font is missing and one of the fallback default fonts is used.

This method, however, has some drawbacks. The biggest one is that it can't detected all system fonts, but rather can reverse lookup fonts from a predefined list. Next, it yields false positives, especially when the font names contain empty characters or numbers. This is probably due to the implementation bug in the JS library. It is important to note that the values of false positives remain stable. That is, when performing multiple fingerprints with this method, the values do not change. It is obvious that this fall-back font probing is not as powerful as Flash

```
CONNECT fingerprint.sba-research.org:443 HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:47.0) Gecko/20100101 Firefox/47.0
Connection: keep-alive
Connection: keep-alive
Host: fingerprint.sba-research.org:443

A SSLv3-compatible ClientHello handshake was found. Fiddler extracted the parameters below.

Version: 3.3 (TLS/1.2)
Random: CD 0F 37 92 2F 91 F9 10 7B 5C 9F 44 0A B3 AF 64 8C D6 9D 48 EA B2 2C 2C A1 95 68 3D 90 B5 64 C3
Time: 9/26/2047 5:41:01 AM
SessionID: D1 0A 00 00 A0 DD 3D 17 1D 4D 71 8B 74 37 01 7C 6A 88 D6 60 EE 20 A1 61 C0 BB 45 DC 55 8C 74 13
Extensions:
  server_name fingerprint.sba-research.org
  extended_master_secret  empty
  renegotiation_info  00
  elliptic_curves secp256r1 [0x17], secp384r1 [0x18], secp521r1 [0x19]
  ec_point_formats  uncompressed [0x0]
  SessionTicket empty
  NextProtocolNego  empty
  ALPN    h2, spdy/3.1, http/1.1
  status_request  OCSP - Implicit Responder
  signature_algs  sha256_rsa, sha384_rsa, sha512_rsa, sha1_rsa, sha256_ecdsa, sha384_ecdsa, sha512_ecdsa, sha1_ecdsa, sha256_dsa, sha1_dsa
Ciphers:
  [C02B]  TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
  [C02F]  TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
  [CCA9]  TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
  [CCA8]  TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
  [C00A]  TLS1_CK_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
  [C009]  TLS1_CK_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  [C013]  TLS1_CK_ECDHE_RSA_WITH_AES_128_CBC_SHA
  [C014]  TLS1_CK_ECDHE_RSA_WITH_AES_256_CBC_SHA
  [0033]  TLS_DHE_RSA_WITH_AES_128_SHA
  [0039]  TLS_DHE_RSA_WITH_AES_256_SHA
  [002F]  TLS_RSA_AES_128_SHA
  [0035]  TLS_RSA_AES_256_SHA
  [000A]  SSL_RSA_WITH_3DES_EDE_SHA

Compression:
  [00]  NO_COMPRESSION
```

Listing 3.6: SSL/TLS ClientHello message

font detection, because not only less popular fonts might not detected for they were not included in the predefined list of fonts, but also the font order is not a fingerprintable feature anymore.

One could argue that another drawback to this approach of font detection is slower than the Flash method. We have measured that the reverse lookup of 510 fonts in Firefox 47 takes under 300 ms on our test machine.[4], however we could not compare it to the Flash method since we found no easy way of measuring performance of initialization and execution of Flash without modifying and recompiling the SWF file.

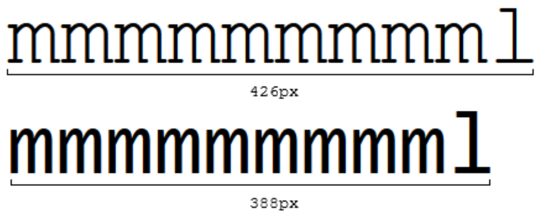


Figure 3.8: Font width: monospace (Courier New) vs Consolas

It is theoretically also possible to fingerprint fonts via Java and Silverlight plugins, but we have decided to exclude these method since browser vendors have started phasing out [14] support for NPAPI plugins [18] or disabling them for security reasons by default [15].

Font metrics This property is a checksum of measuring the on-screen dimension of individual font glyphs described in [28]. The glyphs are rendered in very large size to exaggerate the smallest differences in rendering dimensions. Figure 3.9[5] shows an example of a single font glyph rendered with default, serif, sans-serif, monospace, cursive and fantasy font families. The measurements of 43 code points are hashed and checksummed. The process is repeated three times:

- With all system fonts
- With standard fonts
- With only one standard font - also effective against Tor Browser hat limits font enumeration

3.3.5 HTML5 Canvas Fingerprinting

HTML5 Canvas API [49] allows web applications to draw 2D graphics on a HTML `<canvas>` element using JavaScript. This fingerprinting method utilizes canvas API to draw shapes, arcs and text to the canvas element and measure the differences in rendering and anti-aliasing on a pixel level. This fingerprinting method was first reported in [46].

[4]Windows 8.1; Intel Core i5-4200 CPU @ 2x1.60 GHz; 8GB RAM

[5]Image generated in Firefox 47 on Widows 10 machine, via https://www.bamsoftware.com/talks/fc15-fontfp/fontfp.html

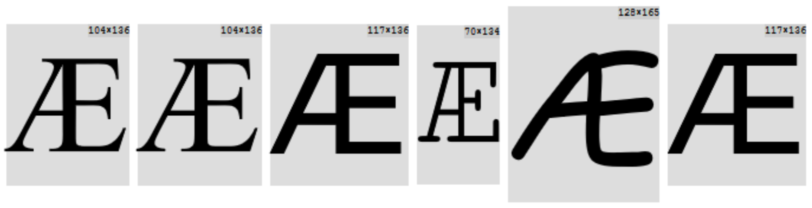


Figure 3.9: Font metrics: single character rendered with different font families

Canvas Image This property represents the image generated via canvas API. Using JavaScript 3.7, we draw two text strings with different size and color followed by a couple of special characters and a smiley face (Unicode characters). Additionally, we use an invalid font name for one of the two strings to force the browser to fall back to default font. The actual results of the script is a `base64` encoded image, that we embed in the HTML page via `<image>` element. Figure 3.10 shows the results of the same canvas script rendered on different browsers - Firefox 47/Windows 8.1, Chrome 51/Windows 8.1 and Chrome 52/Android 6.0.1, from top to bottom.

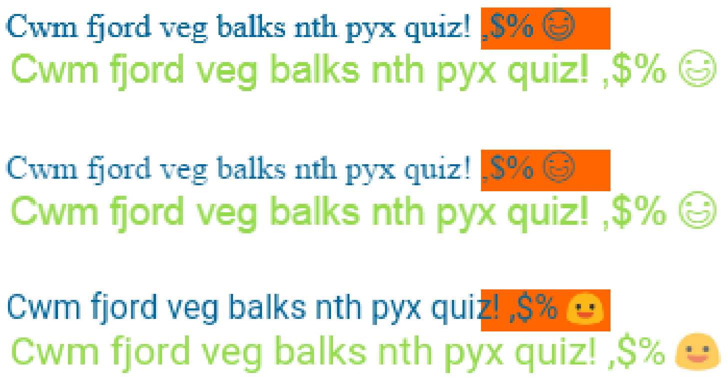

Cwm fjord veg balks nth pyx quiz! ,$%
Cwm fjord veg balks nth pyx quiz! ,$%
Cwm fjord veg balks nth pyx quiz! ,$%
Cwm fjord veg balks nth pyx quiz! ,$%
Cwm fjord veg balks nth pyx quiz! ,$%
Cwm fjord veg balks nth pyx quiz! ,$%

Figure 3.10: Image generated via canvas API

3.3.6 WebGL Fingerprinting

WebGL [42] is a low level 3D graphics API for the browsers that allows drawing of interactive 3D and 2D graphics on the `<canvas>` element, similarly to Cavas API. WebGL is based on OpenGL ES 2.0 and is integrated into the browser, removing the need for 3rd party plug-ins.

WebGL Support In order for a browser to support WebGL, it has to be supported by the underlying hardware - the GPU. This property shows if the browser supports WebGL and can take values `true` and `false`.

```javascript
function drawOnCanvas() {
  var cEl = document.createElement("canvas");
  cEl.setAttribute("height", 50);
  cEl.setAttribute("width", 400);
  cEl.style.display = "inline";

  var cCtx = cEl.getContext("2d");
  cCtx.textBaseline = "alphabetic";
  cCtx.fillStyle = "#f60";
  cCtx.fillRect(230, 1, 62, 20);
  cCtx.fillStyle = "#069";
  cCtx.font = "12pt invalid-font-42";
  var txt = "Cwm fjord veg balks nth pyx quiz! ,$% \ud83d\ude03";
  cCtx.fillText(txt, 2, 15);
  cCtx.fillStyle = "rgba(102, 204, 0, 0.7)";
  cCtx.font = "14pt Arial";
  cCtx.fillText(txt, 4, 37);

  return cEl.toDataURL();
}
```

Listing 3.7: Drawing on canvas via JavaScript

```javascript
if (window.WebGLRenderingContext) {
  var names = ["webgl", "experimental-webgl", "moz-webgl"];
  var supported = [];
  for (var i in names) {
    try {
      var gl = document.createElement('canvas').getContext(names[i]);
      if (gl && typeof gl.getParameter == "function")
        supported.push(names[i]);
    } catch(e) {}
  }
}
```

Listing 3.8: WebGL initialization

WebGL Enabled It is possible, although not straightforward, to disable WebGL in the browser. It usually involves setting the right flags on the advanced settings page (`about:config` in Firefox or `chrome://flags` in Chrome). This property shows if the WebGL is enabled in browser and can take values `true` and `false`.

Supported Context Names In order to draw via WebGL onto the `canvas` element, a WebGL context needs to be initialized. The context is used to render to the drawing buffer and to manage the state of WebGL. It is initialized via canvas by supplying the context name (see listing 3.8). Standard context names are `webgl`, `experimental-webgl` and `moz-webgl`. This property shows all available context names.

Version This property indicates the version or the release number of the WebGL implementation. For example:

- `WebGL 0.94` on Internet Explorer 11,
- `WebGL 1.0 (OpenGL ES 2.0 Chromium)` on Chrome on Android

Shading Language Version Similar to *version*, but more specific:

- `WebGL 0.94 GLSL ES 0.94` on Internet Explorer 11,
- `WebGL GLSL ES 1.0 (OpenGL ES GLSL ES 1.0 Chromium)` on Chrome on Android

Vendor This property shows the vendor of the WebGL implementation and in many cases it is equivalent to the browser platform (e.g. `Mozilla`, `WebKit`, etc.).

Renderer This property shows the WebGL renderer which is typically equivalent to the browser platform (e.g. `Mozilla`, `Webkit`, `WebKit WebGL`, etc.).

Unmasked Vendor This property shows the real vendor of the WebGL implementation (e.g. `Google Inc.`, `Microsoft`, `Qualcomm`, etc.). Some browsers, like newer versions of Firefox are hiding the real vendor and renderer with the values `n/a`. Retreiving the unmasked vendor and renderer is straighforward, as seen in listing 3.9.

```
var ctx = document.createElement("canvas").getContext('webgl');
var dbgRend = ctx.getExtension('WEBGL_debug_renderer_info')
if (dbgRend != null) {
  vendorReal = ctx.getParameter(dbgrnd.UNMASKED_VENDOR_WEBGL);
  rendererReal = ctx.getParameter(dbgrnd.UNMASKED_RENDERER_WEBGL);
}
```

Listing 3.9: WebGL: unmasked vendor and renderer

Unmasked Renderer This property indicates the real renderer and may contain information about the GPU and the version of the installed drivers. For example, three browsers reveal different amount of information on the same machine:

- Mozilla Firefox 47: `n/a`
- Internet Explorer 11: `Intel(R) HD Graphics Family`
- Google Chrome 51: `ANGLE (Intel(R) HD Graphics Family Direct3D11 vs_5_0 ps_5_0)`

Supported Extensions WebGL API, like OpenGL API, supports extensions that offer additional functionalities. Extensions are prefixed with `ANGLE_`, `OES_`, `EXT` and. `WEBGL`. There are also vendor specific extensions. These are prefixed with `MOZ_`, `WEBKIT_` etc. This property list all supported extensions in this particular WebGL implementation.

Max Anisotropy The extension `EXT_texture_filter_anisotropic` enables support for anisotropic filtering in WebGL. This property shows the maximum anisotropy level supported by the extension. Example values are `2`, `16`, etc.

3.3.7 HTML5/CSS3 Features Detection

Modern web browsers support a vast number of Internet technologies and implement various W3C standards like HTML, CSS, JavaScript etc. Many of these standard have not yet reached the final status and are in an ongoing process of revision and rewriting. Some browser vendors like Google and Mozilla have shortened the development and release cycles of their browsers to be able to keep up with the growing number of features defined by aforementioned standards, in addition to fixing bugs and security updates [12] [16].

This race to implement the best and newest features in the next browser release has led to fragmentation of implementation status of various features across different browsers. Websites like *HTML5 Test*[6] and *CSS3 Test*[7] rate the browsers with score points based on the implementation status of HTML5 and CSS3 standards, respectively. Another website *Can I Use*[8] helps web application developers to keep track of the features which they might or might not use in order to accomplish an unique cross-browser experience for their web application. Modernizr[9] is a JavaScript library for HTML5/CSS3 feature detection that enables web developers to programmatically check if the browser supports a certain feature. Code listing 3.10 shows a check for a Web Animation API.

```
if (Modernizr.animation) {
  // show animation
} else {
  // display static content
}
```

Listing 3.10: Feature detection with Modernizr

We utilize Modernizr to detect almost 300 features and use it as part of our fingerprinting algorithm as proposed in [69]. The detections range from various CSS properties, over HTML5 features from different APIs, to storage capabilities of the browser, like *local storage*, *session storage* and *Web SQL database*.

HTML5 APIs Battery, Emojis, Fullscreen API, Gamepad, Geolocation, History, Internationalization, Page Visibility, MathML, Navigation Timing, Notifications, DOM Pointer Events,

[6]https://html5test.com/
[7]http://css3test.com/
[8]http://caniuse.com/
[9]https://modernizr.com

Pointer Lock, Proximity, Quota Storage Management, ServiceWorker, IE User Data, Vibration, Web Animation, Web Audio, Dataset, File, Filesystem, Beacon, Fetch, Speech Recognition, Speech Synthesis, Timed Text Track, Web Cryptography, Web Intents, WebSockets, Binary WebSockets, Touch Events, RTC Data Channel, RTC Peer Connection, Web Workers, Shared Workers.

ECMAScript 5 and 6 ES5 Array, ES5 Date, ES5 Function, ES5 Object, ES5, ES5 Strict Mode, ES5 String, ES5 Syntax, ES5 Immutable Undefined, ES6 Array, ES6 Collections, ES5 String.prototype.contains, ES6 Generators, ES6 Math, ES6 Number, ES6 Object, ES6 Promises, ES6 String.

CSS Properties and Attributes Animations, Background Blend Mode, Clip Text, Calc, Filters, `:checked` `:nth-child` and `:last-child` Pseudo-Selectors, `:target` `:valid` and `:invalid` Pseudo-Classes, Font `ch` `ex` and `rem` Units, Columns attributes, Cubic Bezier Range, Display Table, `text-overflow` Ellipsis, `CSS.escape()`, Gradients, HSLA Colors, Hyphens, Mask, Media Queries, Multiple Backgrounds, Object Fit, Opacity, Overflow Scrolling, Pointer Events, position: sticky, Generated Content Animations and Transitions, Reflections, Regions, UI Resize, rgba, Stylable Scrollbars, Shapes, general sibling selector, Supports, `text-align-last`, `textshadow`, Transforms, Transforms 3D, Transform Style preserve-3d, Transitions, `user-select`, `vh` `vmax` and `vmin` unit, `cssall`, `wrap-flow`.

Multimedia HTML5 Audio Element (ogg, mp3, opus, wav, m4a), HTML5 Video (ogg, h264, webm, vp9, hls), Audio Loop Attribute, Audio Preload, JPEG 2000, JPEG XR, EXIF Orientation, WebP, WebP Lossless, Animated WebP, Animated PNG, SVG clip paths, SVG filters, SVG foreignObject, Inline SVG, SVG SMIL animation, Video Loop Attribute, Video Preload Attribute.

Storage Application Cache, Local Storage, Session Storage, Web SQL Database, IndexedDB, IndexedDB Blob, `deleteDatabase()`.

Other Ambient Light Events, Blob constructor, Canvas, Canvas text, Content Editable, Context menus, Cross-Origin Resource Sharing, Custom protocol handler, CustomEvent, Dart, DataView, Emoji, Event Listener, Hashchange event, Hidden Scrollbar, HTML Imports, IE8 compat mode, input element attributes, `input[search]` search event, input type attributes, JSON, Font Ligatures, Reverse Ordered Lists, postMessage, QuerySelector, requestAnimationFrame, Template strings, Touch Events, Typed arrays, Unicode characters, VML, XDomainRequest, `a[download]` Attribute, Low Battery Level, canvas blending support, `canvas.toDataURL` type support, getRandomValues, Appearance, Backdrop Filter, Background Position Shorthand and XY, Background Repeat (`bgrepeatspace` and `bgrepeatround`), Background Size and Cover, Border Image, Border Radius, Box Shadow, Box Sizing, Flexbox, Flex Line Wrapping, `@font-face`, will-change, classList, Document Fragment, `[hidden]` Attribute, microdata, DOM4 MutationObserver, bdi Element, datalist Element, details Element, output Element, picture Element, progress Element (progressbar), progress Element (meter), ruby, rp, rt Elements, Template Tag, time Element, Track element, Unknown Elements, Motion Event, Orientation Event, onInput Event, `input[capture]` Attribute, `input[file]` Attribute, `input[directory]` Attribute, `input[form]` Attribute, `input[type="number"]` Localization, placeholder attribute, form `#requestAutocomplete()`, Form Validation, Server SentEvents, `iframe[sandbox]` Attribute, `iframe[seamless]\verb` Attribute, Image crossOrigin, sizes attribute, srcset attribute, input formaction, input formenctype, input formmethod, input formtarget, Low Bandwidth Connection, Server SentEvents, XML HTTP Request Level 2 XHR2, `script[async]`, `script[defer]`, `style[scoped]`, SVG as an `<img>` tag source, textarea maxlength, Blob URLs, Data URI, URL parser, getUserMedia, Framed window, Workers from Data URIs, Transferables Objects, `iframe[srcdoc]` Attribute.

Modernizr documentation [44] contains brief descriptions for each property.

3.3.8 AudioContext Fingerprinting

This fingerprinting method exploits the WebAudio API [65] by generating audio signal which is then hashed and used as an identifier. This method does not generate audio playable on the speakers nor does it use the microphone. Instead it analyzes the differences in how the audio signal is processed. There are two ways of analyzing audio signal: via Oscillator [52] and Dynamics Compressor [50] nodes. The fingerprinting method has first been analyzed in [25].

AudioContext properties This is a collection of general browser/system audio properties available to AudioContext API. An example value might look like listing 3.11:

```json
{
  "ac-sampleRate": 44100,
  "ac-state": "suspended",
  "ac-maxChannelCount": 2,
  "ac-numberOfInputs": 1,
  "ac-numberOfOutputs": 0,
  "ac-channelCount": 2,
  "ac-channelCountMode": "explicit",
  "ac-channelInterpretation": "speakers",
  "an-fftSize": 2048,
  "an-frequencyBinCount": 1024,
  "an-minDecibels": -100,
  "an-maxDecibels": -30,
  "an-smoothingTimeConstant": 0.8,
  "an-numberOfInputs": 1,
  "an-numberOfOutputs": 1,
  "an-channelCount": 1,
  "an-channelCountMode": "max",
  "an-channelInterpretation": "speakers"
}
```

Listing 3.11: AudioContext properties

DynamicsCompressor values By using Dynamics Compressor API the fingerprinting scripts generates audio signal, whose resulting values which are utilized in two ways to add to the fingerprinting algorithm:

- Sum of buffer values
- Hash of full buffer

OscillatorNode values This is a result of generating audio signal via OscillatorNode API. The results is a list of node values which when visualized look like figure 3.11[10].

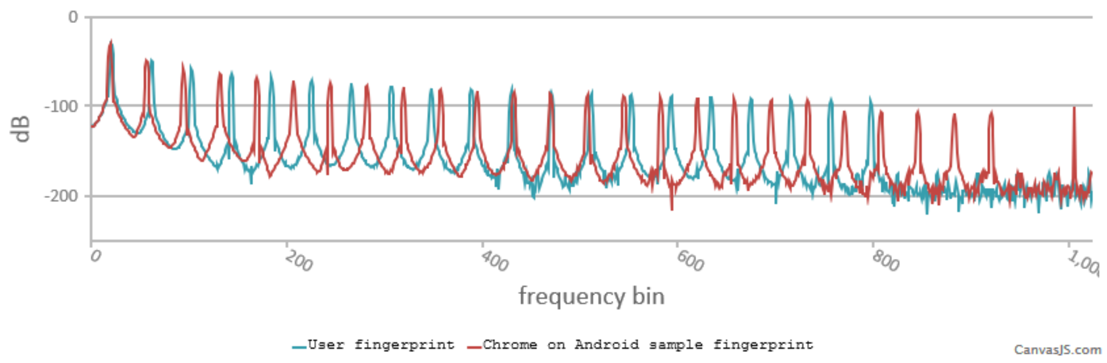


Figure 3.11: Audio fingerprint

[10]Values generated on Firefox 47 on Windows 10 machine via https://audiofingerprint.openwpm.com/

CHAPTER 4

Implementation

This chapter lists the tools and technologies used for implementation and shows how they are combined in order to create the framework in its final form. Next, it outlines details about the deployment in form of a website. Finally, some privacy considerations are mentioned.

4.1 Tools

This section lists the tools and libraries used to develop and deploy the framework.

Python 2.7 [1] This is the programming language used for the development of the server part of our framework. Python was chosen for following reasons:

- it allows for fast application prototyping and testing
- it has good tooling support and an extensive ecosystem of libraries and web frameworks

Bottle [2] This is a web framework of choice. Bottle describes itself as a *fast, simple and lightweight WSGI micro web-framework for Python* [31]. This open-source framework is contained in a single Python module and has no external dependencies. It supports function binding to URL routes, has a built-in template engine for server-side content generation and a built-in (and interchangeable) WSGI [21] HTTP server.

MongoDB [3] This is an open-source document-oriented (NoSQL) database where browser fingerprints are stored. The choice of NoSQL over more traditional relational database is further discussed in the next section.

[1] https://www.python.org/
[2] http://bottlepy.org
[3] https://www.mongodb.com/

PyMongo [4] This is a Python library/driver for MongoDB that allows interaction with the said database. We use it to store fingerprints and derive statistics from the data.

Virtualenv [5] This tool is used to create isolated Python environment for our web application, during both development and production.

pip [6] This is a package management system for Python ecosystem and is used to install dependencies for the web application when deploying on the server.

Apache Web Server [7] This is the web server used for hosting our framework as a web application.

mod_ssl [8] This is an Apache module that provides SSL/TLS support for the Apache Web Server.

ssl_haf [9] This is an Apache module for SSL/TLS handshake analysis used by Cipher Suite fingerprinting method in Section 3.3.3.

mod_wsgi [10] This is an Apache module that provides an WSGI-compliant interface for hosting Python based web applications.

jQuery [11] This is the most popular JavaScript library intended on simplifying the client-scripting with cross-browser compatibility in mind. We use it to execute fingerprinting scripts and arrange results in the browser.

Twitter Bootstrap [12] This is a front-end framework for developing responsive web sites which is used to design a user-friendly website for our framework.

4.2 Technical Realization

Figure 4.1 shows how the above-mentioned tools fit together to form the development and deployment stack for the framework. Since the fingerprinting framework was developed with Python, we had a choice of developing it on top of an existing web application framework.

[4] https://api.mongodb.com/python/current/
[5] https://virtualenv.pypa.io/en/stable/
[6] https://pip.pypa.io/en/stable/
[7] https://httpd.apache.org/
[8] https://httpd.apache.org/docs/current/mod/mod_ssl.html
[9] https://github.com/ssllabs/sslhaf
[10] https://modwsgi.readthedocs.io/en/develop/
[11] https://jquery.com
[12] http://getbootstrap.com

Bottle framework was chosen because it supported URL routing, templating and had an interchangeable WSGI component. WSGI is a Python standard that specifies the interface between web servers and Python applications and defines how they should interact [21]. That way Bottle's own primitive WSGI implementation could be replaced with another, more robust WSGI server.

Initially, the plan was to deploy the website on Microsoft Azure[13] cloud platform, because it offered 'one-click deployment' directly from Visual Studio IDE. However, the decision to include SSL/TLS fingerprinting has forced us to switch to Apache Web Server and use `ssl_haf` module to analyze the SSL/TLS handshake. Additionally, `mod_ssl` was employed in order to enable HTTPS communication in Apache in the first place.

In order for Bottle to be able to communicate with Apache Web Server, we had to use `mod_wsgi` module, since Apache does not support WSGI natively. We utilize `virtualenv` to create an isolated Python environment for Bottle framework and our application, and `pip` to install Python library dependencies. For communication with MongoDB, we use PyMongo.

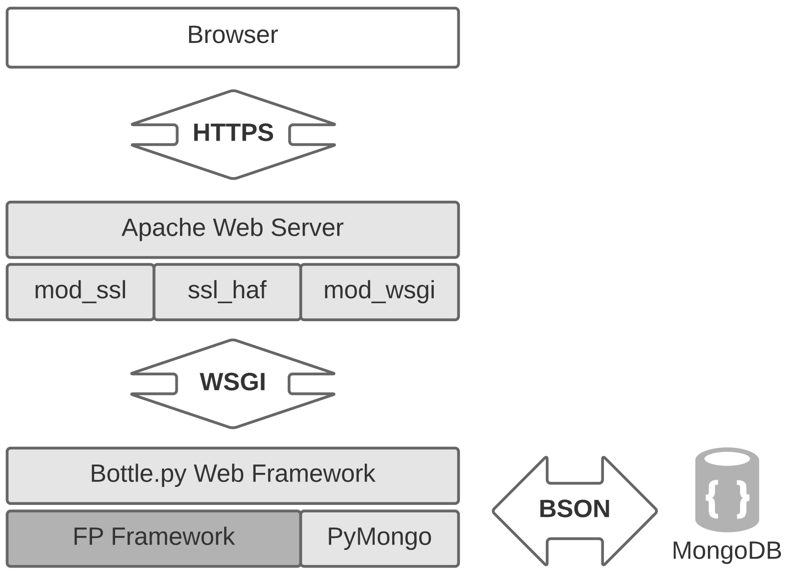


Figure 4.1: Development stack

4.2.1 Database choice

For persisting fingerprints we have opted for a schema-less database, more specifically MongoDB. There are 3 main reasons we picked document-oriented over relational database:

[13]https://azure.microsoft.com/

1. Our fingerprinting data has no relations to other entities, hence we would theoretically only need one table for fingerprints in a relational database. One possibility would be to have a table per fingerprinting method and then joining all tables to form a fingerprint, however it would add unnecessary complexity to the DB-model. In our case, one fingerprint equals one document (or data entry) in the database.

2. There is no purpose in defining a schema for our fingerprint model. Fingerprints can have varying number of fields depending on the result of a fingerprinting scripts. If a browser has disabled JavaScript, then only server-side fingerprinting methods will run and the rest of results is not present. Since the fingerprinting metadata file already contains the list of properties for that method, we can check for presence of property values in the database via simple business logic. Second argument for having a schema-less database is the extensibility aspect of framework. When adding new fingerprinting methods, there is no need for schema adaptation or update scripts.

3. Fingerprinting result and MongoDB database have a matching document format. Fingerprints are sent to server as JSON (JavaScript Object Notation) [23], whereas MongoDB uses BSON (or Binary JSON) as a format for storing documents. This way we can store verbatim fingerprinting results in the database without prior modification.

4.2.2 Extendibility

To make the framework extensible, we had to define fingerprinting methods in an abstract way. We achieved this by introducing a metadata file for each and every fingerprinting method. The file is named `method_info.js` and its content is shown in listing 4.1.

```
{
  "id": "navigator",
  "name": "Navigator",
  "js": "navfp()",
  "scripts": [
    "/js/advert.js",
    "/js/fingerprint2.js",
    "/js/navigator.js"
  ],
  "properties": {
    "plugins": "Plugins",
    "lang": "Language",
    "video": "Screen Resolution",
    "tz": "Timezone",
    ...
  },
  "hashed_properties" : {
    "plugins" : "Browser Plugins"
  },
  "result_handler": "PluginsResultHandler"
}
```

Listing 4.1: Navigator fingerprinting method metadata

Fingerprint method metadata specifies the `id` and names of fingerprinting properties for that method (keys in the *properties* element). The naming schema is also used when constructing fingerprinting result and persisting a fingerprint to database. Name and property descriptions (values in the *properties* element) are used for displaying full property names on the result page. Hashed properties are properties with long values, for which the framework creates database indexes during startup, for faster value comparison. Scripts represents the list of JavaScript files to be embedded in the web page which are necessary for performing fingerprinting. `js` is the main function for that fingerprinting method, from which the embedded scripts are called. `result_handler` is the implementation of fingerprint parser and result manager (see section 3.2.4). It is a python module where one can define how the results should be parsed and displayed. Listing 4.2 shows an excerpt from `result_handler` for Navigator fingerprinting method, which checks the value of `plugins` property. If it's empty, it will display 'No plugins detected' on the display page.

```python
def format_property(self, property_name, property_value):
  if property_name == 'plugins':
    if not str(property_value).strip():
      return 'No plugins detected'
```

Listing 4.2: Navigator result handler

This way we have decoupled the business logic of individual fingerprinting methods from the core part of the framework.

4.2.3 Script Execution

Listing 4.3 shows the JavaScript code that performs the execution of scripts and sends the result to server once the fingerprinting is done. `names` and `requests` are dynamically generated server-side before serving a page to the client. Server does this by enumerating the registered fingerprinting methods and consulting metadata. When the fingerprinting is finished, the script packs the results into a JSON object and sends it to server.

```javascript
$(document).ready(function () {
  setTimeout(function() {
    var names = ... // array of fp ids, for nesting results
    var requests = ... // array of js functions to execute

    $.when.apply($, requests).done(function () {
      var result = new Object();
      $.each(arguments, function (index, responseData) {
        result[names[index]] = responseData;
      });

      $.ajax({
        type: "POST",
        url: "/fp",
        data: JSON.stringify({ 'fp': result }),
        contentType: "application/json; charset=utf-8",
        dataType: "json",
        converters: { 'text json': true },
        success: // wait for results
        failure: // display error message
      });
    });
  }, 2000);
});
```

Listing 4.3: Script runner

4.3 Deployment

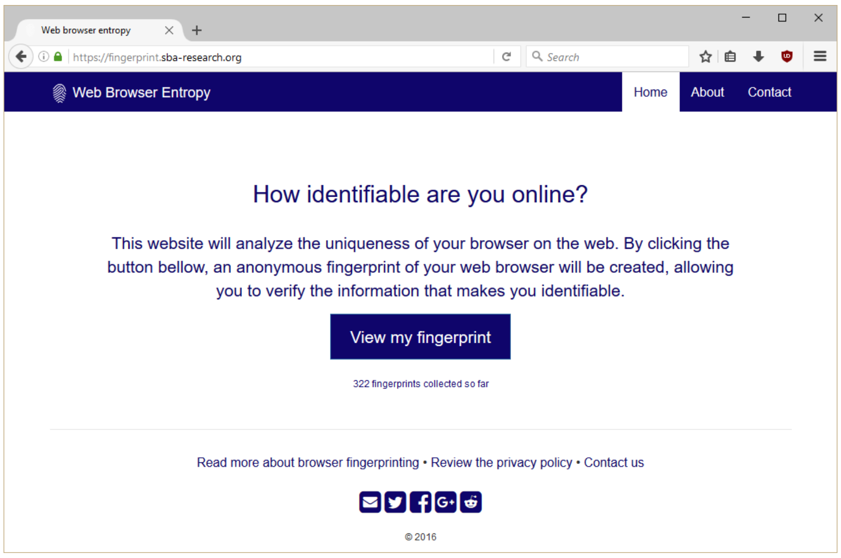
The framework was deployed at DigitalOcean[14], where a virtual private server (VPS or *droplet* in DigitalOcean jargon) was created. The VPS had 1 CPU and 30GB SSD at disposal. In order to support HTTPS communication, installation of a SSL/TLS certificate on the server was necessary. We have used Let's Encrypt[15] which is a free and automatic open certificate authority.

Figure 4.2 shows the front page of the website[16] where framework is deployed. When the user clicks on *View my fingerprint button* the fingerprinting is performed and once finished the fingerprinting result page is shown 4.3.

4.3.1 Privacy Considerations

In order to respect the privacy of website visitors some measures had to be taken into consideration. First, when saving fingerprints into the database we SHA256 encrypted the IP addresses in combination with a secret key. Furthermore, we periodically deleted the web server logs with access times and IPs. Lastly, we have fuzzed the fingerprinting times by rounding it to the next full hour.

[14] https://www.digitalocean.com/
[15] https://letsencrypt.org/about/
[16] https://fingerprint.sba-research.org



How identifiable are you online?

This website will analyze the uniqueness of your browser on the web. By clicking the button bellow, an anonymous fingerprint of your web browser will be created, allowing you to verify the information that makes you identifiable.

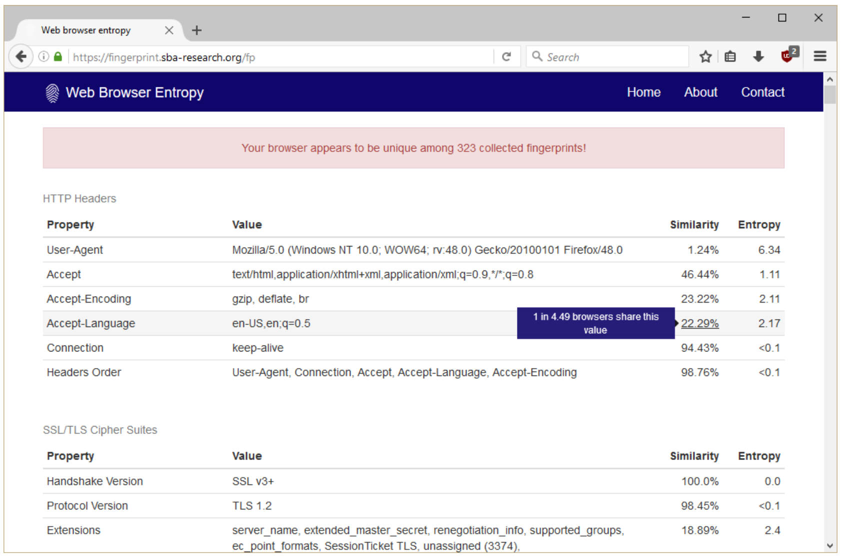
View my fingerprint

322 fingerprints collected so far

Read more about browser fingerprinting • Review the privacy policy • Contact us

© 2016

Figure 4.2: Front page of the website


Web browser entropy
https://fingerprint.sba-research.org/fp
Search
Web Browser Entropy
Home
About
Contact
1 in 4.49 browsers share this value

Your browser appears to be unique among 323 collected fingerprints!

HTTP Headers

Property	Value	Similarity	Entropy
User-Agent	Mozilla/5.0 (Windows NT 10.0; WOW64; rv:48.0) Gecko/20100101 Firefox/48.0	1.24%	6.34
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8	46.44%	1.11
Accept-Encoding	gzip, deflate, br	23.22%	2.11
Accept-Language	en-US,en;q=0.5	22.29%	2.17
Connection	keep-alive	94.43%	<0.1
Headers Order	User-Agent, Connection, Accept, Accept-Language, Accept-Encoding	98.76%	<0.1

SSL/TLS Cipher Suites

Property	Value	Similarity	Entropy
Handshake Version	SSL v3+	100.0%	0.0
Protocol Version	TLS 1.2	98.45%	<0.1
Extensions	server_name, extended_master_secret, renegotiation_info, supported_groups, ec_point_formats, SessionTicket TLS, unassigned (3374),	18.89%	2.4

Figure 4.3: Fingerprinting result web page

CHAPTER 5

Results

This chapter presents the dataset of collected fingerprints in detail. First section introduces the mathematical foundation of the work - entropy - and how it is used in the context of fingerprinting. Next, the results are presented on a per-fingerprint method basis. After that, a comparison of desktop and mobile fingerprints is given. Finally, a comparison of fingerprint data from two different datasets is presented.

5.1 Entropy

In information theory, *entropy* tells us how much information is contained in a certain event or variable by quantifying the amount of randomness of a variable. We use Shannon's entropy to measure the *expected* value of *self-information* or *surprisal* of fingerprints. The equation

$$H(X) = -\sum_{i=1}^{n} P(x_i) log_2 P(x_i) \tag{5.1}$$

represents the entropy of a discrete random variable x with possible values $\{x_1, ..., x_n\}$, where $P(x_i)$ is the size of data sample with $x = i$, divided by the size of dataset N:

$$P(x_i) = \frac{|x_i|}{|N|}. \tag{5.2}$$

Shannon's entropy is measured in bits, since it uses base 2 logarithm for calculation. If we had three fingerprints with two of them having JavaScript enabled ($P(_{JS=1}) = 0.667$) and one JavaScript disabled ($P(_{JS=0}) = 0.333$), the entropy of that fingerprint property would be

$$H(JS) = -[(0.667 log_2 0.667) + (0.333 log_2 0.333)] = 0.9183. \tag{5.3}$$

This simple example illustrates how the distribution of values of a discrete variable influences the entropy. We apply this method for all fingerprinting properties that we collect in order to understand which of them are most revealing.

5.2 Descriptive Results

5.2.1 Data Overview

This section describes the fingerprinting data that was collected over two periods of time. We will analyze the two sets separately, concentrating on the first and then making a comparison to the second, later on in this chapter.

During the first collection phase, that occurred from *April 7, 2016* to *July 29, 2016* we have managed to collect exactly **1,500** fingerprints. The framework website was promoted to friends and colleagues via email and through social network channels, like Facebook and Twitter.

In the second phase, which took place from *July 29, 2016* to *August 21, 2016* we collected **310** fingerprints. Before collecting in the second phase, a few changes have been made to the framework. First, an additional property *Headers order* has been added to *HTTP Headers* fingerprinting method. Second, *AudioContext* fingerprinting method has been implemented in the framework. Besides that, no other changes except for a couple of smaller bug fixes in the framework were made. The website was promoted in the same fashion as in the first phase.

For the first data set we had to remove 119 fingerprints that belonged to various crawlers and bots, since we did not deploy Robots exclusion standard or `robots.txt` [62] on the server to explicitly tell bots and crawlers like Googlebot [13]to refrain from accessing URLs used for fingerprinting. That reduced the number of *valid* fingerprints from the first set to **1,381**.

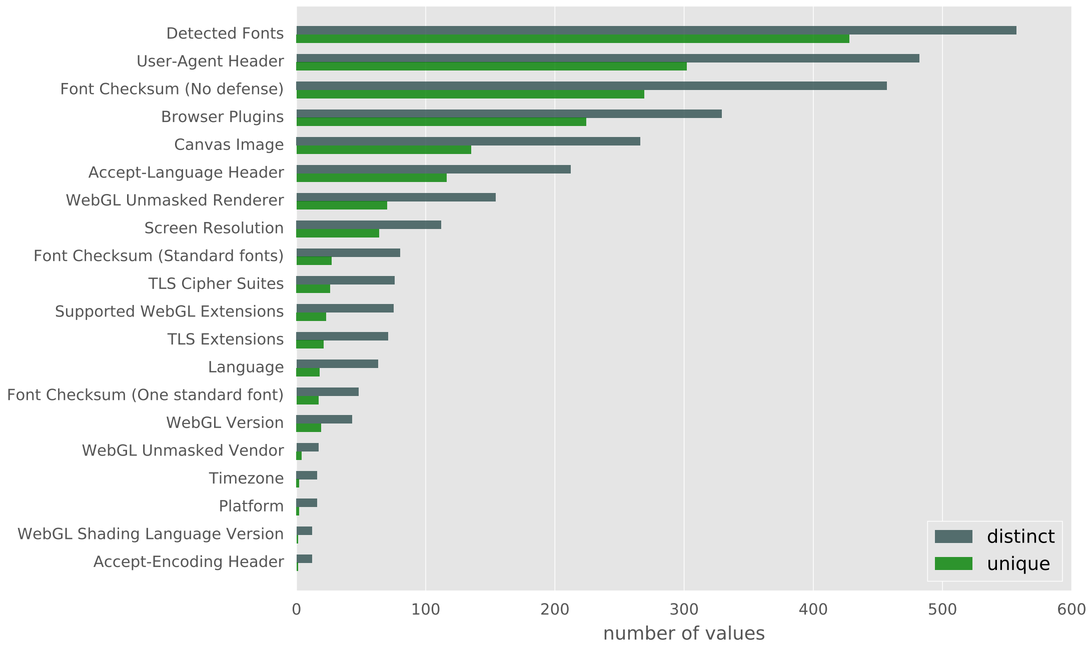


Figure 5.1: Distinct vs. unique values per property

Figure 5.1 shows the properties with most distinct and unique values, with detected fonts having highest number of distinct (557) and unique (428) values. It is worth noting that this amounts to number of different *sets* or combinations of installed fonts and not individual fonts. The next property is user agent HTTP header with 482 distinct and 302 unique values, followed by font-metrics checksum with 457 distinct and 269 unique values. The order of properties in table 5.1 is different, since they are sorted by their respective entropy values, which is of more significance to our research.

Property	**Distinct Values**	**Unique Values**	**Entropy**
User-Agent	482	302	7.952
Font-metrics (No defense)	457	269	7.781
Detected Fonts	557	428	7.548
Canvas Image	266	135	6.648
Browser Plugins	329	224	5.461
Accept-Language	212	116	5.236
Unmasked WebGL Renderer	154	70	4.691
Supported WebGL Extensions	75	23	4.677
Screen Resolution	112	64	4.582
Font-metrics (Standard fonts)	80	27	4.241

Table 5.1: Properties with highest entropy values

Detected fonts are now on the third place with entropy of 7.548 bits after user agent HTTP header, being the property with highest entropy of 7.952 bits and font-metrics of 7.781 bits. This again illustrates how entropy takes the distribution of values of the variable into account.

5.2.2 HTTP headers

The table 5.2 shows the entropy values for all HTTP headers that we collect. Each header is discussed separately.

Property	**Distinct Values**	**Unique Values**	**Entropy**
User-Agent	482	302	7.952
Accept-Language	212	116	5.236
Accept-Encoding	12	1	2.094
Accept	12	3	1.42
Connection	4	1	0.379

Table 5.2: Distribution of HTTP header values

Connection header had only four values: `keep-alive`, `Keep-Alive`, `n/a` and `close` with the first one being predominant (in 93.34% browsers).

Accept header had two dominant, almost equally distributed groups of values. It was possible to correlate the accept headers to the user-agent string, or rather browser family derived from user-agent. For example, first value from table 5.3 was found within Firefox and Safari browsers

(50.11% of all browsers), where the second value exclusively matched Chrome's user agent strings, both desktop and mobile (43.01%). The third value corresponded to IE and IE mobile (3.77%) and the last entry to Microsoft Edge browser included in Windows 10 (1.96

Freq.	Accept header
692	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
594	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
52	text/html, application/xhtml+xml, */*
27	text/html, application/xhtml+xml, image/jxr, */*

Table 5.3: HTTP Accept headers

Accept-Encoding header had seven different individual encoding values: deflate, gzip, sdch, lzma, br, peerdist, identity and an additional empty value. The largest combinations of the individual values are shown in table 5.4. Similarly to Accept header, there were some correlations to browser family. The first group correlates to Chrome (31.14%), second group to Firefox (29.62%) and the third to Safari, IE and older versions of Firefox (28.96%).

Accept-Encoding header	Frequency	%
gzip, deflate, sdch	430	31.14
gzip, deflate, br	409	29.62
gzip, deflate	400	29.96

Table 5.4: HTTP Accept-Encoding headers

Accept-Language header	Frequency	%
en-US,en;q=0.5	318	23.02
en-US,en;q=0.8	147	10.64
de,en-US;q=0.7,en;q=0.3	93	6.73
de-DE,de;q=0.8,en-US;q=0.6,en;q=0.4	92	6.66
en-us	72	5.21

Table 5.5: HTTP Accept-Language headers

Accept-Language has five large sets of values, as shown in table 5.5 with a lot of distinct values which are a product of multiple individual language tokens and their corresponding weights/priorities, for example:

- nl-NL,nl;q=0.8,en-US;q=0.6,en;q=0.4,de;q=0.2,fr;q=0.2,it;q=0.2 or
- de-DE,de;q=0.8,en-US;q=0.6,en;q=0.4,fr;q=0.2,zh-TW;q=0.2,zh;q=0.2,it;q=0.2 .

The *User-Agent* string is one of the more interesting fingerprint properties, not only from the statistical point of view since it has a high entropy value, but also from the semantical standpoint. User agent shows the browser version, vendor and the platform the browser is running on. Besides that, it can also contain information about the device it is running on and in case of mobile clients, the network operator. Table 5.6 shows the most frequent user agents while table 5.7 lists the browser family classifications. Most prevalent browser is Firefox with 32.44% followed by Chrome (26.36%) and Chrome Mobile (10.86%).

Freq.	User-Agent String
40	Mozilla/5.0 (Windows NT 6.1; rv:38.0) Gecko/20100101 Firefox/38.0
35	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101 Firefox/45.0
34	Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:46.0) Gecko/20100101 Firefox/46.0
32	Mozilla/5.0 (X11; Linux x86_64; rv:45.0) Gecko/20100101 Firefox/45.0
30	Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.2623.112 Safari/537.36
26	Mozilla/5.0 (iPhone; CPU iPhone OS 9_3_1 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko) Version/9.0 Mobile/13E238 Safari/601.1
24	Mozilla/5.0 (Windows NT 10.0; WOW64; rv:46.0) Gecko/20100101 Firefox/46.0
21	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.2623.112 Safari/537.36
20	Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
19	Mozilla/5.0 (Windows NT 6.3; WOW64; rv:47.0) Gecko/20100101 Firefox/47.0

Table 5.6: Most frequent User-Agent strings

Browser Family	Frequency	%
Firefox	448	32.44
Chrome	364	26.36
Chrome Mobile	150	10.86
IE	62	4.49
Mobile Safari	58	4.20
Safari	50	3.62
Firefox Mobile	44	3.18
Facebook	40	2.89
Opera	35	2.53
Mobile Safari UI/WKWebView	29	2.01
Other	100	7.24

Table 5.7: Browser families

In table 5.7 one can see the *Facebook* browser family, which corresponds to Facebook mobile application that uses the rendering engine of the mobile platform's browser. The application appends additional headers, in some cases even the name of mobile network provider like *T-MobileA*, *Telekom.de* or *OrangeFrance*.

5.2.3 Cipher Suites

Table 5.8 shows all cipher suite properties with related statistics. All fingerprinted browsers support the *TLS 1.2* version of SSL/TLS protocol and *SSL V3+* version of handshake protocol, while 99.71% have the same compression value `NULL` with 4 browsers having the value `DEFLATE, NULL` instead. There are 76 different cipher suites composed of 115 individual ciphers, two of which were present in all cipher suites. In contrast, there are 71 distinct extension method values that are composed of only 17 individual extensions. However, the order in which extensions and ciphers appear is preserved which leads to a large number of combinations.

Property	**Distinct Values**	**Unique Values**	**Entropy**
Extensions	71	21	4.062
Supported Cipher Suites	76	26	3.665
Protocol Version	3	0	0.193
Compression	2	0	0.029
Handshake Version	1	0	0

Table 5.8: Distribution of cipher suite values

Table 5.9 shows the distribution of individual values with relation to the number of fingerprints in which they appear. The columns show amount and percentage of individual values that are present in 95%, 50%, 10% and 1% of all fingerprints.

Property	**Total**	**<95% FP**	**<50% FP**	**<10% FP**	**<1% FP**
TLS Cipher Suites	115	107 (93%)	105 (91%)	81 (70%)	51 (44%)
TLS Extensions	17	13 (76%)	5 (29%)	4 (24%)	2 (12%)

Table 5.9: Distribution of individual cipher suites and extensions

5.2.4 Navigator features

Table 5.10 shows all navigator properties with the related entropy values. There are 329 different plugin sets of which 224 are unique. The sets are made up of 538 individual plugins. The majority of plugins are used for playback of 3rd party media types or reading particular file formats in the browsers, but many plugins can also reveal information about installed software on the machine that perform drive-by browser plugin installations without user's knowledge. Table 5.14 illustrates the distribution of plugins with relation to percent of all fingerprints containing the individual plugins. It shows that 39% of all individual plugins are contained in less than 1% fingerprints.

Property	Distinct Values	Unique Values	Entropy
Browser Plugins	329	224	5.461
Screen Resolution	112	64	4.582
Language	63	18	3.242
Platform	16	2	2.609
'Do Not Track' Flag	5	0	1.626
Timezone	16	2	1.49
Flash Installed	2	0	0.999
AdBlocker Installed	2	0	0.945
Cookies enabled	2	0	0.078

Table 5.10: Distribution of navigator values

Screen resolution has 112 distinct and 64 unique values. Many of these values seem like fake or random resolutions like `1829x1143x24`, `1512x829x24`, `1371x771x24`. Such values could represent Tor browsers or an attempt to mask the real resolution with the available screen width and height. Most frequent values are show in table 5.11. The highest reported 3840x1080x24 and the smallest is 480x320x32.

Screen resolution	Frequency	%
1920x1080x24	273	19.77
640x360x32	125	9.05
1920x1200x24	100	7.24
1600x900x24	89	6.44
1366x768x24	79	5.72
1440x900x24	62	4.49
667x375x32	52	3.77

Table 5.11: Screen resolutions

Navigator language almost always correlates to the Accept-Language header except in a few cases where there's a mismatch. For example navigator language is set to `fr`, but the Accept-Language header is `en-US,en;q=0.5`.

Navigator platform also correlated with User-Agent string, which also contains operating system information. However, navigator platform is less significant when compared to user agent, especially in Windows' case where it does not distinguish between x86 and 64-bit windows, since both platforms are reported as `Win32`. The table 5.14 lists the most frequent platforms.

The 'Do Not Track' flag has values as shown in table 5.13. If we assume that *unknown* and *unspecified* are the same, we derive that around 22% browsers have explicitly stated not to be tracked by activating the DNT setting.

Navigator platform	Frequency	%
Win32	524	37.94
Linux x86_64	187	13.54
MacIntel	154	11.15
Linux armv7l	147	10.64
iPhone	92	6.67
Linux armv8l	66	4.78
iPad	26	1.81

Table 5.12: Navigator platforms

Do Not Track	Frequency	%
unknown	712	56.87
unspecified	246	19.65
1	225	17.97
yes	62	4.95
0	7	0.56

Table 5.13: 'Do Not Track' Flag

Property	Total	<50% FP	<10% FP	<1% FP	<0.1% FP
Plugins	538	538 (100%)	531 (99%)	473 (88%)	212 (39%)
Fonts	10,197	10,165 (99.7%)	9,822 (96%)	8,280 (81%)	4,789 (47%)

Table 5.14: Distribution of fonts and plugins

5.2.5 Fonts

Property	Distinct Values	Unique Values	Entropy
Font-metric Checksum (No defense)	457	269	7.781
Detected Fonts	557	428	7.548
Font-metric Checksum (Standard fonts only)	80	27	4.241
Font-metric Checksum (One standard font only)	48	17	2.377

Table 5.15: Distribution of font-related values

Detected fonts have 557 distinct values, of which 428 are unique. The number of individual fonts is 10,197. Table 5.14 shows the distribution of number of individual fonts with relation to fingerprints count. The highest count of fonts in one fingerprint is 2,217.

In 65% of cases, font detection was done via JavaScript, and in 35% (or 440 fingerprints) via Flash. However, if we look at the Flash statistics, we can see that Flash was detected in almost half of all fingerprints (600 to be specific). From this we can deduce that 160 fingerprints had the Flash installed but blocked, hence the fall-back detection via JavaScript. This check was implemented in the framework before the second fingerprint collecting period.

The largest anonymity set sizes with relation to font-metric checksum without defense are 93 (7.42%), 52 (4.15%) and 37 (2.15%). The rest of anonymity sets are lower than 28.

5.2.6 WebGL

Property	Distinct Values	Unique Values	Entropy
Unmasked Renderer	154	70	4.691
Supported Extensions	75	23	4.677
Unmasked Vendor	17	4	2.751
Version	43	19	2.574
Shading Language Version	12	1	2.194
Renderer	10	4	1.619
Vendor	8	2	1.613

Table 5.16: Distribution of WebGL values

The largest group of unmasked renderer is 'n/a' or unknown (39,6%), the rest is visible in table 5.17 that shows largest sets in regards to combination of regular and unmasked renderer values and the prevalence of `WebKit WebGL` renderer, which can be correlated to Chrome browsers. WebGL extensions have 75 distinct and 23 unique values. They are composed of 39 different individual extensions, with 50% of extensions found in 50% of fingerprints.

Unmasked Renderer	Renderer	Freq.
ANGLE (Intel(R) HD Graphics Family Direct3D11 vs_5_0 ps_5_0)	WebKit WebGL	60
Adreno (TM) 330	WebKit WebGL	49
Adreno (TM) 418	WebKit WebGL	37
Apple A8 GPU	WebKit WebGL	37
Apple A7 GPU	WebKit WebGL	24
Apple A9 GPU	WebKit WebGL	23
Intel(R) HD Graphics Family	Internet Explorer	21
Adreno (TM) 305	WebKit WebGL	21
ANGLE (Intel(R) HD Graphics 4000 Direct3D11 vs_5_0 ps_5_0)	WebKit WebGL	20

Table 5.17: WebGL unmasked renderer x renderer combinations

5.2.7 Canvas

Canvas generated image has 266 distinct and 135 unique values as seen in table 5.18. The top 15 anonymity sets with relation to canvas image are of sizes 75 to 25 which encompasses 50% of the fingerprints. Figure 5.2 illustrates how the image looks on different devices.

Property	Distinct Values	Unique Values	Entropy
Image	266	135	6.648

Table 5.18: Distribution of canvas values

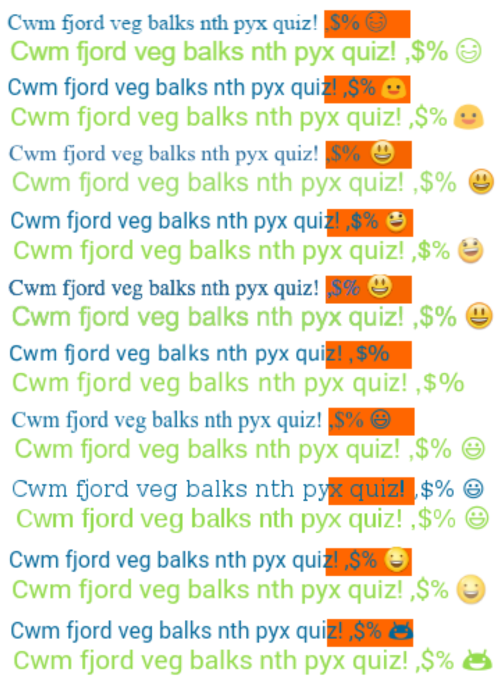


Figure 5.2: Canvas images

5.3 Comparison of mobile and desktop clients

By analyzing the User-Agent string it is possible to classify fingerprints into mobile and desktop browsers. As seen in figure 5.4, around three quarters of fingerprints are classified as desktop browsers (1,007, or 74%) and the rest is mobile (364 fingerprints).

Figure 5.4 shows the fingerprinting properties with highest distinct and unique value count, analog to 5.1. The most obvious difference to previous figure is the absence of fonts and plugins. The reason behind this is the lack of support for NPAPI plugins on mobile browsers and inabil-

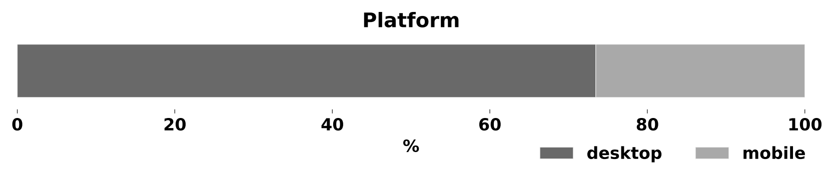


Figure 5.3: Platforms: desktop vs mobile

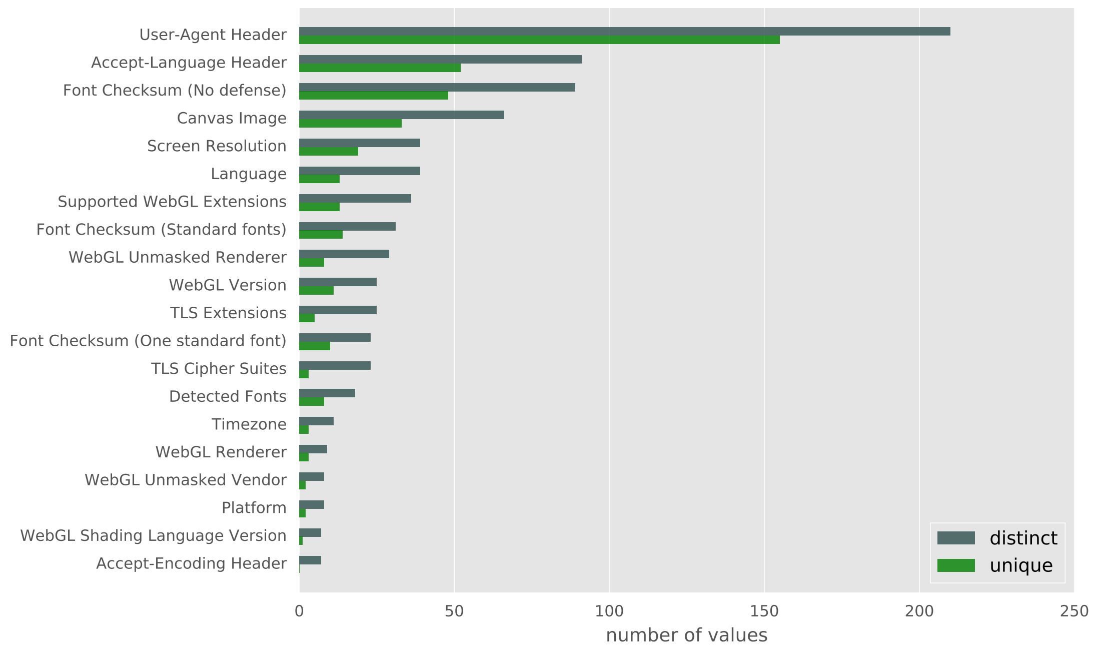


Figure 5.4: Distinct vs. unique values per property for mobile clients

ity to install additional fonts. However, mobile browsers *compensate* this by having very rich User-Agent strings which usually include version and build number of the operating system and in some cases even the name of mobile operator. Table 5.19 shows the top ten properties ordered by entropy, analog to table 5.1. From the set of 364 mobile fingerprints, it is possible to identify almost one in two fingerprints by checking only User-Agent string. Font-metric checksum without defense shows high entropy value also for mobile browsers. Another interesting value is canvas generated image. Although the number of WebGL vendors and renderer is lower, correlated to the number of graphics chip manufacturers for the mobile platform, canvas image still produces high entropy values since the majority of mobile vendors include custom emoticons in their own flavor of the mobile operating system, e.g. Android. This is visible in the figure 5.2 where, beside the slight differences in font rendering, the 'smiley face' looks clearly different from one to next image.

Another noticeable difference between desktop and mobile clients is the percentage of browsers

Property	Distinct Values	Unique Values	Entropy
User-Agent	210	155	7.124
Accept-Language	91	52	5.331
Font-metrics (No defense)	89	48	4.831
Canvas Image	66	33	4.689
Language	39	13	3.847
Unmasked WebGL Renderer	29	8	3.823
Supported WebGL Extensions	36	13	3.53
Screen Resolution	39	19	3.503
SSL/TLS Extensions	25	5	3.477
SSLCipher Suites	23	3	2.963

Table 5.19: Properties with highest entropy values for mobile clients

with disabled JavaScript. Figure 5.5 shows that around 11% of desktop clients had JavaScript disabled or blocked in contrast to around 3% for mobile clients.

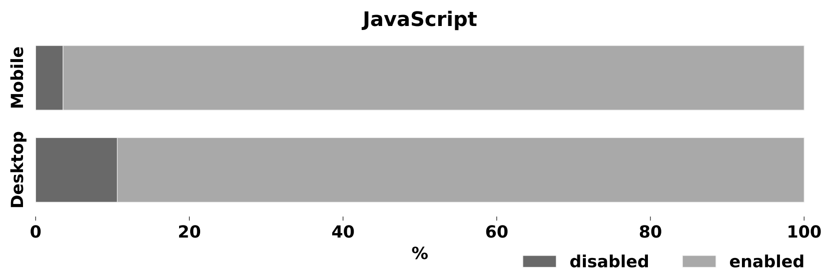


Figure 5.5: JavaScript: desktop vs mobile

5.4 Comparison of two fingerprinting datasets

Like previously stated, the second fingerprinting dataset contains 310 fingerprints. This time, bots and crawlers were not present in the database so no preprocessing step was necessary before analyzing data. Table 5.20 shows the fingerprint properties with highest entropy values, analog to table 5.19. The properties are same as in first dataset with a slightly different order, namely browser plugins have higher entropy value than Accept-Language header and unmasked WebGL renderer and supported WebGL extensions are higher compared to screen resolution and standard font metrics.

However, in order to compare two datasets that obviously differ in size, it is necessary to calculate *normalized Shannon's entropy*:

$$H_N(X) = \frac{H(X)}{log_2 N}, \tag{5.4}$$

Property	Distinct Values	Unique Values	Entropy
User-Agent	113	53	6.295
Font-metrics (No defense)	119	69	6.226
Detected Fonts	135	101	6.188
Canvas Image	94	54	5.712
Accept-Language	69	39	4.617
Unmasked WebGL Renderer	65	31	4.555
Browser Plugins	85	59	4.483
Font-metrics (Standard fonts)	43	18	4.345
Supported WebGL Extensions	33	13	3.977
Screen Resolution	37	12	3.831

Table 5.20: Dataset 2: Properties with highest entropy values

where N is the size of dataset. This method does not depend on the size of the dataset, however it does depend on the distribution of data. Table 5.21 compares the normalized entropy values of the first and second dataset.

Attribute	Entropy (Dataset 1)	Entropy (Dataset 2)
User-Agent	0.762	0.762
Font-metrics (No defense)	0.756	0.762
Detected Fonts	0.734	0.757
Canvas Image	0.646	0.699
Accept-Language	0.502	0.558
Unmasked WebGL Renderer	0.456	0.558
Browser Plugins	0.531	0.549
Font-metrics (Standard fonts)	0.412	0.532
Supported WebGL Extensions	0.455	0.487
Screen Resolution	0.445	0.469

Table 5.21: Dataset 1 vs Dataset 2: Comparison of entropy values

From the table 5.21 we observe that the entropy values for both dataset are almost identical, except for two properties with a difference of >0.1 bits: Unmasked WebGL Renderer and Font-metrics with standard font. The explanation could be that the distribution of values for the two properties in the second dataset is not in line with other values. As previously mentioned, normalized entropy does not depend on the dataset size, but is influenced by the distribution of anonymity sets.

CHAPTER 6

Discussion

This chapter shows the interpretation of collected data and comparison to other research. Furthermore, drawbacks of the dataset are pointed out. Next, fingerprinting countermeasures are discussed. Finally, future work is presented.

6.1 Data Interpretation

After detailed data analysis in the previous chapter, we now compare our findings to other similar fingerprinting projects, namely Panopticlick and AmIUnique. Table 6.1 shows normalized entropy values for fingerprinting properties from both project compared to our findings.

For Panopticlick only 6 of 15 values are available, some were not collected at all and others were omitted from statistics without explanation. Panopticlick' dataset size is 470,161 fingerprints, while IAmUnique have collected 118,934 fingerprints. Our combined dataset size from both fingerprinting phases consists of 1,691 fingerprints. In order to compare the datasets of different sizes, we use normalized entropy like in previous chapter.

We can see that the majority of entropy values in our dataset is higher than in AmIUnique. That is probably the side-effect of different distribution of anonymity sets with relation to the whole fingerprint dataset, analog to comparison of fingerprinting dataset 1 and 2 in previous chapter. If we use only dataset 2, the entropy values are even higher, despite smaller dataset and normalization of entropy. One thing to note is lower value for browser plugins and detected fonts which indicates a drop in NPAPI plugins and disappearance of Flash font fingerprinting over time, especially when AmIUnique values are taken into consideration. Font entropy in our case is higher than AmIUnique, since we revert to JavaScript font probing in case Flash is not installed or disabled. The differences in AdBlock and Cookies can hint at bias of fingerprinted users towards privacy and security.

Attribute	Panopticlick	AmIUnique	Normalized entropy
User-Agent	0.531	0.580	0.766
Browser Plugins	0.817	0.656	0.534
Detected Fonts	0.738	0.497	0.724
Screen Resolution	0.256	0.290	0.427
Timezone	0.161	0.198	0.136
Cookies enabled	0.019	0.015	0.008
Accept Header		0.082	0.126
Accept-Encoding		0.091	0.207
Accept-Language		0.351	0.504
Platform		0.137	0.247
DNT		0.056	0.156
Canvas		0.491	0.635
WebGL Vendor		0.127	0.266
WebGL Renderer		0.202	0.453
AdBlock		0.059	0.091

Table 6.1: Comparison of entropy values to Panopticlick and AmIUnique

6.2 Limitations

From the previous section we can understand that our dataset, and thus the interpretation thereof, has its limitations. There are two reasons why the fingerprints sample we collected might not be representative of the whole population.

First, the dataset size of two other projects are 70- and 280-fold compared to the size of our dataset, for AmIUnique and Panopticlick, respectively. Our attempts to promote the fingerprinting website on news aggregate websites like Slashdot[1] and Reddit[2], in order to collect more data, have been met with limited success. We believe this has significantly skewed the distribution of our dataset.

Second, our dataset was also affected by the running time of the experiment, which was just under four months for the first and almost one month for the second dataset. In contrast, other two projects were active over a year before presenting the results. However, even if he *have* managed to collect enough fingerprints in that short amount of time, we would have been able to capture only a snapshot of the fingerprint distribution and could still not measure the *change* of fingerprints distribution over time and the change in fingerprint properties of returning visitors.

6.3 Fingerprinting Countermeasures

The main idea when combating fingerprinting is to reduce the differences in resulting fingerprint across browsers. Browser vendors have acknowledged the problem of fingerprinting [56] [67]

[1]https://www.slashdot.org

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

[72], but the concrete measures against it are very limited. The phasing-out of NPAPI Plugins could be seen as a move in a right direction, only to be replaced by numerous HTML5 APIs that reveal lots of information about the browser and the underlying operating system. Flash is slowly disappearing and so is font fingerprinting becoming more difficult, but there are already other fingerprinting methods for font probing.

Tor Browser[3] has a more active stance on preventing fingerprinting since privacy is an integral part of its design [66]. Tor reduces a fingerprint by disabling enumeration of plugins by default, faking the screen resolution, preventing canvas fingerprinting [74], limiting the font list [73] etc. However, even Tor is susceptible to font metrics checksum [28] which was part of our framework.

There are also multiple browser add-ons and extensions, designed to prevent or limit the extent of fingerprinting. Ghostery[4] and Privacy Badger[5] are browser add-ons that block ads and scripts by referring to a predefined list of blacklisted third-party domains. NoScript[6] is a Firefox add-on that prevents JavaScript execution and gives the user ability to selectively enable or disable execution for every script in the web page.

Besides installing third-party add-ons, regular user could also turn off automatic Flash execution in the browser' plugins settings and disable WebRTC API to prevent private network IP leaks.

6.4 Future Work

Our browser fingerprinting project with help of the framework is **not** concluded with the finalization of this thesis. As previously mentioned, a plan for the framework is for it to be an ongoing project in which additional fingerprinting features - either discovered in the wild via analysis of the advertising networks or fingerprinting service providers or presented as proof of concept by researchers - will be included and deployed on the website. A goal is to have a long running fingerprinting project in order to gather more data and understand how the fingerprinting changes over time, something we have sadly not managed to do during the duration of this thesis.

The framework will be published as an open source project and will be available for free. The next concrete step in framework development is creating an additional page where the visitors can interactively review their fingerprinting data, but also compare it to other data entries. In general, it would be in the best interest to research community if other projects were to publish their collected fingerprinting data for researchers to analyze. We believe that such a project is feasible if correct measures are taken to make data anonymous and disassociate it from the real person.

[3]https://www.torproject.org/projects/torbrowser.html.en
[4]https://www.ghostery.com/
[5]https://www.eff.org/privacybadger
[6]https://addons.mozilla.org/en-US/firefox/addon/noscript/

CHAPTER 7

Conclusion

In this thesis we have conducted an extensive survey of research related to browser tracking and fingerprinting. We have described the evolution of tracking methods in a historical context, starting with session-scoped and persistent tracking, over cache-exploiting mechanisms and on to fingerprinting.

We have developed a fingerprinting framework based on an extensible client-server architecture, where we join all previous fingerprinting methods under one roof. Furthermore, we deployed the framework as a website under `https://fingerprint.sba-research.org` in order to collect fingerprints.

In a period from April 7, 2016 to August 21, 2016 we have managed to collect 1,800 browser fingerprints. By examining the result for each fingerprinting method separately, we analyzed the distinct and unique values for each property and calculated the entropy. Results show that the most revealing property of a browser from our dataset is list of installed fonts with 7.952 bits of entropy, followed by HTTP User-Agent header, font-metric checksum, list of installed browser plugins and so on. We analyzed mobile browsers separately, which represented 26% of our dataset. The most identifying property was HTTP User-Agent header with 7.124 bits of entropy, followed by HTTP Accept-Language header, font-metric checksum and HTML5 Canvas generated image.

Despite the limitations of our dataset, both in size and data distribution, we have managed to identify some trends in fingerprinting by comparing our results to previous research. The disappearance of NPAPI plugins have displaced browser plugins from being most identifiable feature further down the line. List of fonts have retained high entropy values, in spite of decrease in Flash presence and due to new JavaScript/CSS font probing methods. More recent methods like WebGL, Canvas and AudioContext fingerprinting can generate high entropy results, for both desktop and mobile browsers, by relying on rich and expressive HTML5 API - a trend that will very likely extend into future.

Bibliography

[1] The Cookie Law Explained. https://www.cookielaw.org/the-cookie-law/. [Online; accessed: 28-March-2016].

[2] Donald E. Eastlake 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, October 2015.

[3] M Abraham, CAMERON Meierhoefer, and ANDREW Lipsman. The impact of cookie deletion on the accuracy of site-server and ad-server metrics: An empirical comscore study. *Retrieved October*, 14:2009, 2007.

[4] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 674–689, New York, NY, USA, 2014. ACM.

[5] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1129–1140. ACM, 2013.

[6] Adobe. ActionScript 2.0 Language Reference. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/2/help.html?content=00001593.html#505940. [Online; accessed: 16-June-2016].

[7] Adobe. Adobe Flash Platform * Shared Objects. http://help.adobe.com/en_US/as3/dev/WS5b3ccc516d4fbf351e63e3d118a9b90204-7d80.html. [Online; accessed: 5-August-2016].

[8] Adobe. SWF and AMF Technology Center, SWF File Format Specification (version 19). http://www.adobe.com/devnet/swf.html. [Online; accessed: 16-June-2016].

[9] Arun Ranganathan, Jonas Sicking. Web storage (Second Edition), W3C Working Draft 21 April 2015. https://www.w3.org/TR/FileAPI/. [Online; accessed: 02-August-2016].

[10] Mika D Ayenson, Dietrich James Wambach, Ashkan Soltani, Nathan Good, and Chris Jay Hoofnagle. Flash cookies and privacy ii: Now with html5 and etag respawning. *Available at SSRN 1898390*, 2011.

[11] Adam Barth. Rfc 6265-http state management mechanism. *Internet Engineering Task Force (IETF)*, pages 2070–1721, 2011.

[12] Anthony Laforge. Chromium Blog. Release Early, Release Often. `http://blog.chromium.org/2010/07/release-early-release-often.html`. [Online; accessed: 19-June-2016].

[13] Google Webmaster Central Blog. Updating the smartphone user-agent of Googlebot. `https://webmasters.googleblog.com/2016/03/updating-smartphone-user-agent-of.html`. [Online; accessed: 11-August-2016].

[14] Justin Schuh. Chromium Blog. Saying Goodbye to Our Old Friend NPAPI. `http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html`. [Online; accessed: 17-August-2016].

[15] Michael Coates. Mozilla Security Blog. Protecting Users Against Java Vulnerability. `https://blog.mozilla.org/security/2013/01/11/protecting-users-against-java-vulnerability/`. [Online; accessed: 17-August-2016].

[16] The Mozilla Blog. New Channels for Firefox Rapid Releases. `https://blog.mozilla.org/blog/2011/04/13/new-channels-for-firefox-rapid-releases/`. [Online; accessed: 19-June-2016].

[17] Tomasz Bujlow, Valentín Carela-Español, Josep Solé-Pareta, and Pere Barlet-Ros. Web tracking: Mechanisms, implications, and defenses. *CoRR*, abs/1507.07872, 2015.

[18] Chrome Developer. NPAPI Plugins. `https://developer.chrome.com/extensions/npapi`. [Online; accessed: 17-August-2016.

[19] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996.

[20] Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, October 2015.

[21] Phillip J. Eby. PEP 333 – Python Web Server Gateway Interface v1.0. `https://www.python.org/dev/peps/pep-0333/`. [Online; accessed: 03-May-2016].

[22] Peter Eckersley. *How Unique Is Your Web Browser?*, pages 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[23] E.C.M. Association. ECMA-404: The JSON Data Interchange Format, 1st Edition (October 2013). http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. [Online; accessed: 22-August-2016].

[24] E.C.M. Association. ECMAScript 2016 Language Specification, 7th Edition (June 2016). http://www.ecma-international.org/ecma-262/7.0/index.html. [Online; accessed: 23-July-2016].

[25] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. [Technical Report], May 2016.

[26] Edward W Felten and Michael A Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 25–32. ACM, 2000.

[27] Roy T. Fielding and Julian F. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, October 2015.

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

[29] Fingerprintjs2. Modern & flexible browser fingerprinting library. https://github.com/Valve/fingerprintjs2. [Online; accessed: 15-May-2016].

[30] Avi Goldfarb and Catherine E. Tucker. Online advertising, behavioral targeting, and privacy. *Commun. ACM*, 54(5):25–27, May 2011.

[31] Marcel Hellkamp. Bottle: Python Web Framework. http://bottlepy.org/docs/dev/index.html. [Online; accessed: 03-May-2016].

[32] Scott Hollenbeck. Transport Layer Security Protocol Compression Methods. RFC 3749, March 2013.

[33] M. Husák, M. Cermák, T. Jirsík, and P. Celeda. Network-based https client identification using ssl/tls fingerprinting. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 389–396, Aug 2015.

[34] Ian Hickson. Web SQL Database, W3C Working Group Note 18 November 2010. https://dev.w3.org/html5/webdatabase/. [Online; accessed: 02-August-2016].

[35] Ian Hickson. Web storage (Second Edition), W3C Recommendation 19 April 2016. https://www.w3.org/TR/webstorage/. [Online; accessed: 02-August-2016].

[36] Internet Assigned Numbers Authority (IANA). Transport Layer Security (TLS) Extensions. http://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xml. [Online; accessed: 12-June-2016].

[37] Jonathan Robie. What is the Document Object Model? `https://www.w3.org/TR/WD-DOM/introduction.html`. [Online; accessed: 31-July-2016].

[38] Balachander Krishnamurthy and Craig E. Wills. Generating a privacy footprint on the internet. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 65–70, New York, NY, USA, 2006. ACM.

[39] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *37th IEEE Symposium on Security and Privacy (S&P 2016)*, 2016.

[40] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, August 2016. USENIX Association.

[41] Tai-Ching Li, Huy Hang, Michalis Faloutsos, and Petros Efstathopoulos. Trackadvisor: Taking back browsing privacy from third-party trackers. In *International Conference on Passive and Active Network Measurement*, pages 277–289. Springer, 2015.

[42] Marrin, Chris. WebGL Specification; Version 1.0.3, 27 October 2014. *Khronos WebGL working Group*. [Online; accessed: 18-June-2016].

[43] J. R. Mayer and J. C. Mitchell. Third-party web tracking: Policy and technology. In *2012 IEEE Symposium on Security and Privacy*, pages 413–427, May 2012.

[44] Modernizr. The feature detection library for HTML5/CSS3 - documentation. `https://modernizr.com/`. [Online; accessed: 15-May-2016].

[45] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: exploiting the ssl 3.0 fallback. *PDF online*, 2014.

[46] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, 2012.

[47] Mozilla Developer Network. AJAX: Asynchronous JavaScript + XML. `https://developer.mozilla.org/en-US/docs/AJAX`. [Online; accessed: 11-August-2016.

[48] Mozilla Developer Network. AudioContext. `https://developer.mozilla.org/en-US/docs/Web/API/AudioContext`. [Online; accessed: 02-August-2016.

[49] Mozilla Developer Network. Canvas API. `https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API`. [Online; accessed: 16-June-2016.

[50] Mozilla Developer Network. DynamicsCompressorNode. `https://developer.mozilla.org/en-US/docs/Web/API/DynamicsCompressorNode`. [Online; accessed: 02-August-2016.

[51] Mozilla Developer Network. HTTP access control (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS. [Online; accessed: 05-August-2016.

[52] Mozilla Developer Network. OscillatorNode. https://developer.mozilla.org/en-US/docs/Web/API/OscillatorNode. [Online; accessed: 02-August-2016.

[53] Mozilla Developer Network. Plugins. https://developer.mozilla.org/en-US/Add-ons/Plugins. [Online; accessed: 15-May-2016.

[54] Mozilla Developer Network. Web APIs - Navigator. https://developer.mozilla.org/en/docs/Web/API/Navigator. [Online; accessed: 15-May-2016].

[55] Mozilla Developer Network. Web APIs - Screen. https://developer.mozilla.org/en/docs/Web/API/Screen. [Online; accessed: 15-May-2016.

[56] MozillaWiki. Fingerprinting. https://wiki.mozilla.org/Fingerprinting. [Online; accessed: 22-August-2016].

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

[58] Netscape DevEdge (Internet Archive WayBack Machine). What's New In JavaScript 1.2. https://web.archive.org/web/19971015223714/http://developer.netscape.com/library/documentation/communicator/jsguide/js1_2.htm. [Online; accessed: 15-May-2016].

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

[60] Nikunj Mehta, Jonas Sicking, Eliot Graff, Andrei Popescu, Jeremy Orlow, Joshua Bell. Indexed Database API, W3C Recommendation 08 January 2015. https://www.w3.org/TR/IndexedDB/. [Online; accessed: 02-August-2016].

[61] Oracle. PersistenceService, JNLP API 1.7.0_95. http://docs.oracle.com/javase/7/docs/jre/api/javaws/jnlp/javax/jnlp/PersistenceService.html. [Online; accessed: 5-August-2016].

[62] The Web Robots Pages. Frequently Asked Questions. http://www.robotstxt.org/faq.html. [Online; accessed: 11-August-2016].

[63] Panopticlick. How unique, and trackable, is your browser? `https://panopticlick.eff.org/`. [Online; accessed: 17-August-2015].

[64] Latit Patel. JavaScript/CSS Font Detector. `http://www.lalit.org/lab/javascript-css-font-detect/`. [Online; accessed: 15-May-2016].

[65] Paul Adenot, Chris Wilson, Chris Rogers. Web Audio API, W3C Working Draft 08 December 2015. `https://www.w3.org/TR/webaudio/`. [Online; accessed: 02-August-2016].

[66] Clark E. Murdoch S. Perry, M. The Design and Implementation of the Tor Browser [DRAFT]. `https://www.torproject.org/projects/torbrowser/design/#fingerprinting-linkability`, May 2016. [Online; accessed: 16-June-2016].

[67] The Chromium Project. Technical analysis of client identification mechanisms. `https://www.chromium.org/Home/chromium-security/client-identification-mechanisms#TOC-Browser-level-fingerprints`. [Online; accessed: 22-August-2016].

[68] Ashkan Soltani, Shannon Canty, Quentin Mayo, Lauren Thomas, and Chris Jay Hoofnagle. Flash cookies and privacy. In *AAAI spring symposium: intelligent information privacy management*, volume 2010, pages 158–163, 2010.

[69] T. Unger, M. Mulazzani, D. Frühwirt, M. Huber, S. Schrittwieser, and E. Weippl. SHPF: Enhancing HTTP(S) session security with browser fingerprinting. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 255–261, Sept 2013.

[70] W3C. Tracking Preference Expression (DNT). `https://www.w3.org/2011/tracking-protection/drafts/tracking-dnt.html`. [Online; accessed: 15-May-2016].

[71] W3C. Web application APIs, W3C Candidate Recommendation 20 August 2015. `https://www.w3.org/TR/html5/webappapis.html#the-navigator-object`. [Online; accessed: 15-May-2016].

[72] WebKit. Fingerprinting. `https://trac.webkit.org/wiki/Fingerprinting`. [Online; accessed: 22-August-2016].

[73] Tor Bug Tracker & Wiki. Limit the fonts available in Tor Browser. `https://trac.torproject.org/projects/tor/ticket/2872`. [Online; accessed: 16-June-2016].

[74] Tor Bug Tracker & Wiki. Prompt before allowing HTML5 Canvas image extraction. `https://trac.torproject.org/projects/tor/ticket/6253`. [Online; accessed: 22-August-2016].

[75] Gilbert Wondracek, Thorsten Holz, Engin Kirda, and Christopher Kruegel. A practical attack to de-anonymize social network users. In *2010 IEEE Symposium on Security and Privacy*, pages 223–238. IEEE, 2010.