Browser Fingerprinting: Analysis, Detection, and Prevention at Runtime

by

Amin Faiz Khademi

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
October 2014

Abstract

Most Web users are unaware of being identified or followed by web agents which leverage techniques such as browser fingerprinting (or fingerprinting). Data obtained through such fingerprinting techniques can be utilized for various purposes ranging from understanding the types and properties of the user's browser to learning the user Web experience (e.g., through the browsing history). For enterprises, this can be a useful means to personalize services for their end-users or prevent online fraudulent activities. Similarly, a good fingerprinting technique can provide a rich set of data for various adversary purposes such as for compromising the security and privacy of Web users.

Careful or attentive Web users might configure privacy enhancing tools (e.g., pop-up and cookie blockers) or operate in the private mode of the browser in order to block or prevent fingerprinters. However, recently we have observed that new fingerprinting methods can easily bypass the existing fingerprinting detection and prevention mechanisms. Moreover, while the topic of browser fingerprinting has been well studied, little attention was given to their detection and prevention.

To address this challenge, we first analyze and reverse engineer the most widely used fingerprinting methods on the Web and unify these methods for developing a hybrid fingerprinting tool, called `Fybrid`. Furthermore, we integrate `Fybrid` with a

social networking service and develop an integrated Web application, called `iFybrid`. Using `iFybrid`, we show the possibility of performing individual identification on top of browser identification using fingerprinting. We also identify metrics related to each method which are the indicators for performing fingerprinting attempts. Then, we use the identified metrics and propose a novel runtime fingerprinting detection and prevention approach, called `FPGuard`. `FPGuard` monitors activities of the running websites on the user's browser. While the detection capability of `FPGuard` is evaluated using the top 10,000 Alexa websites, its prevention mechanism is evaluated against four fingerprinting providers. Our evaluation results show that `FPGuard` can effectively detect and mitigate fingerprinting at runtime without interfering the user's browsing experience.

Acknowledgments

I am grateful to a number of people without whose help this thesis would have not been possible:

My supervisor, Prof. Mohammad Zulkernine, for his great support, guidance, persistent help, and patience over the last two years;

Dr. Komminist Weldemariam, for providing valuable comments, indispensable advice, and the time he spent on reviewing my thesis;

My family who has been a source of encouragement, confidence, unconditional love, inspiration, and wisdom through my entire life;

My colleagues at Queen's Reliable Software Technology Group (QRST) who have been more than supportive during this effort.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Web has become a popular medium for delivering a handful of static, dynamic, and interactive Web contents such as images, audios, and videos. Such contents no longer originate only from the website that a user voluntarily interacts with as they mostly rely on third-party services for various purposes such as static or dynamic contents fetching (e.g., news feed widgets), extending functionalities (e.g., integrating with social networking services), and monetizing (e.g., through embedding advertisements). Third-party service providers often deploy techniques to monitor interactions and consumption patterns or behaviors of the Web users to personalize services accordingly. This has transformed the Web negatively from communication, trading, and collaboration to tracking users for various purposes.

User tracking involves monitoring and recording user's Web/browsing experiences (e.g., between multiple visits or across different websites) [1]. Based on the recorded information, a user profile can be created for delivering personalized and targeted services (e.g., advertisements) [2]. The collected browsing information, or profile, can be categorized as Personally Identifiable Information (PII) and not Personally Identifiable Information (non-PII) [1]. The first category includes personal information

such as name, and email address about individuals while the second category includes information about the user's interests, location, browsing habit, etc. Many advertisers collect non-PII about users without asking for their consent. More fine-grained information about the user would be helpful in developing and delivering more accurate personalized services. However, analyzing a user's profile might reveal private or personal information about the user such as marital status, health condition, employment status, gender, and sexual orientation, since the user profile is strongly linked to the user's personal information. In the following, while we use the advertisement scenario for the purpose of explanation, the ideas embodied here can be generalized and adapted to any other Web data collection scenarios that the user is unaware of.

According to the existing studies [3, 4, 5, 6], Web users prefer to have control over their privacy and thus prefer to stay anonymous to unknown or hidden tracking service providers without their intent. More specifically (as reported in a study [6]), 68% of Americans prefer to use blocking features of their browser (e.g., private-mode) so that they can stay away from advertisements and other tracking Web objects which are not originally from the website they are visiting directly (i.e., third-party trackers).

To identify users on their Web presence, there are two types of tracking approaches employed by advertising companies — namely, active tracking and passive tracking [7]. Active trackers mainly focus on exploiting client-side technologies such as browser and Flash cookies [8]. Typically, they store an identifier (called a client-side identifier) in the user's browser or computer for the purpose of identifying users in future visits. This type of tracking is becoming more prevalent and popular among Web trackers or more technical Web-enabled fingerprinting tools. For instance, in

a study on the top Alexa[1] websites, 91% of the domains (out of 500 domains) embedded at least one tracker, while 50% of them embedded 4 or 5 trackers [9]. To protect users against active tracking methods, various approaches, techniques, and tools have been developed (e.g., `Ghostery` [10] and `AdblockPlus` [11]). The goal of these tools is to make users aware that they are being tracked by third-party services and to stop the tracking attempts. Additionally, most modern browsers are equipped with privacy enhancement tools to improve their users browsing experiences with respect to their privacy. A good example for this is running a browser instance with a private-mode [12]. It is noteworthy that `evercookie` [13] is a highly persistent type of client-side identifier that uses various client-side storage technologies (e.g., browser cookies, Flash cookies, and HTML5-based storages) to make it hard to be deleted by the user. It is possible to delete `evercookie` from most browsers; however it might be inconvenient (not simple). In fact, an attentive user can prevent or otherwise permanently stop active trackers by just deleting the client-side identifiers. Unfortunately, removing client-side identifiers is no longer sufficient to prevent the user from being identified and tracked.

The second tracking approach (i.e., passive tracking) covertly collects information about the user's browser and device to create a unique identifier for the purpose of identifying the user. For example, a browser or computer can be identified by using information about the configurations of the underlying software and hardware as these configurations could be customized by each individual user. For instance, individual customization can be made by installing a particular operating system, setting the screen resolution, installing a favorite set of fonts, etc. Such information about the system settings and configurations combined would be helpful in creating a unique

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

identifier for the browser or computer and can be used by fingerprinters to create an association between the user and the underlying browser or computer.

Passive tracking approaches such as browser fingerprinting [14] and history stealing [15] can easily evade the existing anti-tracking tools. Browser fingerprinting (or fingerprinting) is one of the passive tracking approaches. Browser fingerprinters systematically send a number of queries to the underlying browser to extract various properties of the underlying browser and device such as screen resolution, supported plugins, system fonts, time-zone, etc. Then, they use the collected data as an identifier (or fingerprint) for the browser or device. For example, in a 2010 study [14], authors conducted a large scale experiment on browser fingerprinting by extracting the data of nearly half a million browsers of which 94.2% of the browsers had unique fingerprints. The authors concluded that the browser fingerprint could be used for browser identification.

1.1 Research Problems

Browser fingerprinting can be persistent, invisible, and stateless. These features make the browser fingerprinting a reliable source of identity. With respect to persistence, the browser fingerprint depends on the properties of the browser and the underlying device, and hence various events such as updating operating system, updating browser plugins, installing new fonts, etc. could change the browser fingerprint. It is true that the fingerprint might frequently change over time. However, a study [14] showed that the changes of a browser fingerprint are detectable using a simple heuristic algorithm with over 99% precision. With respect to invisibility, fingerprinting is hidden to users and it is stateless, which means that it usually does not leave any footprint on the

browser. As noted before, an attentive user can see the list of client-side identifiers and delete them. As a result, many tracking companies are exploiting fingerprinting as an identification method and hence it is becoming more prevalent than before [16, 17].

On the bright side, fingerprinting is useful for preventing fraud and account hijacking [18, 19]. For example, anti-fraud companies that are offering this service, exploit fingerprinting with the purpose of creating a rich repository of browsers that transact online. In this case, whenever a fraud is suspected or confirmed for a particular browser, they store the evidence and the browser's fingerprint in a blacklist. Upon subsequent visits, such blacklist can be consulted to detect and prevent further fraudulent online transactions. On the other hand, fingerprinting can enable trackers and advertisers to invisibly follow users without relying on client-side identifiers (e.g., cookies) across different websites. Moreover, the collected fingerprinting data can be a basis for developing and launching various attacks for under-ground economic gains (e.g., Web-based malware that is designed for a particular browser plugin or specific versions of the installed plugins [20]) through privacy and security breaches. Additionally, the stateless nature of fingerprinting further challenges the existing detection tools.

Attempting to address the abovementioned challenges, a number of fingerprinting providers offer custom solutions (e.g., `bluecava` [21] and `AddThis` [22]) to their users who want to stay anonymous and would prefer not to share their browser's fingerprint [23]. These solutions rely on server side for collecting the browser's fingerprint, without sharing it with third-parties (e.g., advertising companies). However, the fingerprinting provider might still share the browser's fingerprint with clients who are exploiting fingerprinting for fraud detection purposes. Furthermore, despite the

alarming prevalence of malicious activities on the Web and the constantly evolving tactics of fingerprinting techniques to harvest Web users' personal information so as to breach their privacy and security, the existing approaches are limited to partial and coarse-grained analysis and characterization of fingerprinting activities. As a result, there is the need for a client-side solution where users are able to protect their privacy against fingerprinting providers based on their intent.

1.2 Overview: Proposed Approach

To address the abovementioned challenges, in this thesis, we first reverse engineer the four most common fingerprinting methods exploited by popular fingerprinters and identify metrics related to each method. The metrics are indicators for performing fingerprinting attempts. Next, we unify all the four fingerprinting methods and develop a hybrid fingerprinting tool, named `Fybrid` to compare the effectiveness of each method in browser identification. We further integrate `Fybrid` with a social networking service and build a proof-of-concept integrated fingerprinting tool, named `iFybrid`. The objective of `iFybrid` is to show the possibility of performing individual identification rather than browser identification using fingerprinting. Then, we present a novel approach for runtime detection and prevention of browser fingerprinting, named `FPGuard`. With respect to detection, `FPGuard` monitors the running Web objects on the user's browser and collects nine metrics about fingerprinting-related activities. We then propose a number of algorithms to analyze the metrics at runtime and determine whether the webpage is performing fingerprinting or not. In case of fingerprinting, `FPGuard` stores the URL of the webpage in a blacklist and applies our prevention approach whenever the webpage is visited in the future. With respect to

prevention, `FPGuard` frustrates fingerprinting attempts by combining randomization and filtering techniques. In particular, we employ four different randomization policies and two filtering techniques to keep the browser's functionality and protect users from fingerprinting attempts.

FPGuard is implemented as a Google Chrome extension service by instrumenting the Chromium browser. We evaluate its detection performance using the top 10,000 Alexa websites. The prevention mechanism of FPGuard is also evaluated against four fingerprinting providers: two popular commercial fingerprinters and two proof-of-concept fingerprinters. We also evaluate the performance of FPGuard when deployed as an extension and compare it with the existing approaches. As compared to the existing approaches, our evaluation results show that FPGuard can effectively detect and prevent fingerprinting attempts without affecting the user's browsing experience.

1.3 Contributions

The main contributions of this thesis are as follows:

- We reverse engineer the workflows of the most widely used fingerprinting methods on the Web to understand and define a common architecture and the characteristics of fingerprinting process. Moreover, we introduce nine metrics which are the indicators for fingerprinting attempts.

- We develop a hybrid fingerprinting tool by combining the existing methods and evaluate the effectiveness of each method in fingerprinting. We also integrate our fingerprinting tool with a popular social networking service to derive scenarios in which the user's personal information can be revealed using the browser's fingerprint.

- We propose a novel approach for detecting browser fingerprinting at runtime based on the derived fingerprinting metrics. We then propose a prevention mechanism by extending and completing the existing solutions. Our prevention mechanism keeps the browser functionality and reduces the uniqueness of the browser fingerprint. More specifically, we combine randomization policies with filtering methods to prevent fingerprinting.

1.4 Thesis Organization

The remainder of this thesis is structured as follows. In Chapter 2, we present background information on browser fingerprinting along with a categorization of fingerprinting and anti-fingerprinting methods. In Chapter 3, we present our hybrid browser fingerprinting tool which combines the currently present fingerprinting methods on the Web. Here, we also describe the reverse engineering process followed to extract the metrics for each method. We then present our proposed approach for runtime detection and prevention of fingerprinting attempts based on the extracted metrics in Chapter 4. The details of our experimental analysis and results are discussed in Chapter 5. We review the related work on browser fingerprinting in Chapter 6. Here, we also present the existing tools that are developed with the purpose of detecting and preventing fingerprinting attempts and highlight their strengths and weaknesses. In addition, we compare our approach with the existing anti-fingerprinting tools. We conclude our work, point out its limitations, and suggest directions for future work in Chapter 7.

Chapter 2

Background

In this chapter, we present relevant background information on browser fingerprinting. In addition to providing terminologies and definitions, we describe the methods that are used for obtaining fingerprinting data. We also describe the fingerprinting process and a categorization of the existing fingerprinting methods in Section 2.1. We explain a tracking scenario which leverages fingerprinting to reveal the user's browsing history in Section 2.2. In Section 2.3, we explain the concerns about fingerprinting from the user's perspective. Moreover, we provide a categorization of the existing anti-fingerprinting solutions based on the approaches they use.

2.1 Browser Fingerprinting

Web browser (or browser) is a stand-alone application for retrieving information resources from the Web and presenting them on the client-side. *Browser fingerprinting* (or *fingerprinting* in short) is the process in which the device and browser-related properties (or attributes) are collected through the browser for various reasons, especially, for user identification. Examples of these properties include screen dimensions (i.e., `width` × `height`), system fonts, supported browser plugins, time-zone, browser

version, etc. A combination of these properties, which is called *browser fingerprint*, could serve as a nearly unique identifier for the browser and thus could be used for browser identification.

Most existing work on browser fingerprinting assume that once a browser instance is identified using fingerprinting, it remains stable (i.e., it does not change over time). In fact, properties with values that are more stable over time or change gradually over time (e.g., operating system version) are utilized to build the fingerprinting mechanisms used in practice. Moreover, properties such as those with more variant values (e.g., system fonts) can make the fingerprints more unique than properties with less variable values (e.g., operating system's name). This shows the diversity of browser identification using fingerprinting.

A website is able to extract most of the properties used in fingerprinting techniques and tools through the browser that runs the website, the underlying communication mechanism as well as from the running operating system itself. One might wonder how such fine-grained information about the underlying browser and device are observable to the websites. The reason is that many websites or applications need to know about the underlying browser and device to run properly. For instance, a Web application might need to customize its appearance based on the dimensions of the user's screen, and may need to ask for the presence of a particular browser plugin (e.g., Adobe Flash Player) to load further resources (e.g., Flash movies and games). To provide developers with such information, various high level APIs have been exposed by browser vendors. For example, `navigator` and `screen` are browser's built-in JavaScript objects which expose APIs to Web applications for accessing device and browser-related properties such as browser name, version, platform, and

the resolution of the screen displaying the Web application. Furthermore, browser plugins — software components which add features such as supporting a particular file type (e.g., files with `.swf` extension) to the browser — are equipped with APIs for accessing the underlying system properties such as the operating system's name and the resolution of the screen, etc. While the above APIs are useful in developing customized Web applications, at the same time, they expose valuable information and data to fingerprinting providers allowing them to build unique browser-specific identifiers.

As presented by Acar et al. [16], browser fingerprinting methods can be categorized as follows with respect to the approach they use: (i) JavaScript-based, (ii) Plugin-based, (iii) Extension-based, and (iv) Header-based and Server-side.

2.1.1 JavaScript-based

JavaScript is an object-oriented and cross-platform scripting language that has been used for different purposes such as interactive Web contents, asynchronous communications (e.g., validating input fields of forms without refreshing the entire page), creating animations on the fly, altering the contents of documents dynamically, etc. [24]. JavaScript also provides APIs for accessing device and browser-related properties, a feature in favor of fingerprinters. JavaScript APIs that have been mainly exploited for the purpose of fingerprinting are `navigator` and `screen` (or *informative* objects). It should be noted that the `navigator` object represents the device and browser environment properties (e.g., browser name, version, and operating system's platform). The `screen` object, however contains information about the settings of the screen displaying the browser (e.g., screen resolution and color depth).

In addition to the informative JavaScript objects (i.e., `navigator` and `screen`), JavaScript is exploited in different ways for fingerprinting a browser instance with the purpose of identification. These methods mainly use HTML5 canvas element [25], browsing history [26], performance [27], and design difference of JavaScript engines of Web browsers [28] (see Chapter 6 for more details). It is noteworthy that not all of the mentioned methods are present in real-life fingerprinting tools [23]. However, they can increase the accuracy of the identification task.

2.1.2 Plugin-based

As noted before, a browser plugin is a software component which adds additional features to the browser that the browser does not natively include (e.g., supporting Flash movies). Plugins are equipped with APIs for accessing software and hardware-related properties on the underlying system. These APIs enable developers to gain knowledge about the capabilities of the running system. Adobe Flash Player (Flash) and Java are examples of popular plugins which provide such APIs. Flash is a cross-platform plugin that is widely available on almost all popular Web browsers [29]. Typically, Flash is used for displaying videos and games in `.swf` format on the browser. At the same time, Flash has APIs for obtaining information about the underlying system (e.g., operating system's version) and the running application (e.g., Flash plugin version). In addition to system information, Flash has an API for enumerating the system fonts (i.e., the `enumerateFonts` method of the `Font` class). These APIs are in favor of fingerprinters who want to collect as much information as possible about the user's system.

Java plugin is used by webpages for displaying interactive Web contents such as

online games and online chat programs through Java applets. A Java applet is a program written in Java that can run on the browser. Java plugin is becoming less popular in Web market. Indeed, it is advised to disable this plugin from the browser, due to the security bugs that have been exploited in the last few years [30]. While Java can be exploited for collecting system information (e.g., operating system's name and system fonts), it has not been used by popular fingerprinters. The reason is that Java contents (i.e., Java applets) need the user's permission to run on the browser (Figure 2.1).

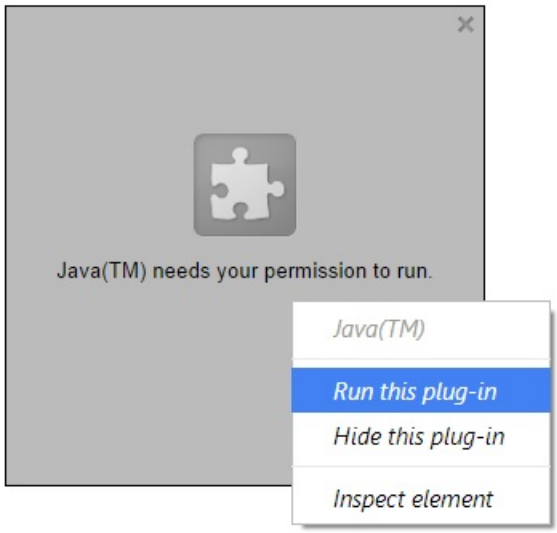

Figure 2.1: Java plugin security prompt for a Java applet

2.1.3 Extension-based

A browser extension is a software component which adds additional functionalities to the browser (e.g., Google dictionary and password managers). It should be noted that browser plugins are software components which enable the browser to display contents such as Flash movies, while browser extensions are programs written mainly

in JavaScript and are used for adding new functionalities to the browser. A number of browser extensions can be exploited by fingerprinters for collecting additional information from the user's browser. For example, `NoScript` [31] which is a popular browser extension for enhancing the user's privacy and the browser's security can be exploited for fingerprinting purposes. This extension only allows the execution of Web objects (e.g., JavaScript programs and Flash objects) from the URLs whitelisted (i.e., marked as trusted) by the user. As explained by Mowery et al. [27], a fingerprinter is able to collect the whitelisted URLs by simply creating a list of various scripts or Flash objects from different domains and requesting them on the browser. To check whether a URL is present in the whitelist or not, the fingerprinter only checks if the requested script is executed (which means the URL is present in the `NoScript`'s whitelist) or not. In this way, the fingerprinter can create a list of URLs and check if they are whitelisted by the user or not.

2.1.4 Header-based and Server-side

Web browsers provide information about the browser's environment (e.g., IP address, HTTP Accept headers, and `userAgent` string) to the Web servers with every request. For example, the `userAgent` property includes information such as the name, version, and platform of the browser. Therefore, a fingerprinter is able to passively analyze the request from the server-side and identify the user's Web browser [32].

2.2 Example: A Browser Fingerprinting Usecase

An example of a typical user tracking process across multiple websites is shown in Figure 2.2. In this example, the tracker employs fingerprinting for the purpose of

browser identification. The details of the tracking process are as follows:

1. A user visits website `example-1.com` (E_1).
2. E_1 includes and loads $ad.1$ from third-party service `tracker-1.com` (T).
3. T employs fingerprinting methods and collects the browser information. Then, it applies an encryption algorithm on the data obtained. Finally, it assigns a unique identifier to the user's browser (here, the assigned id is 10001).
4. T sends back the generated id and the URL of the visited website (i.e., E_1) along with other information (e.g., user's activity on E_1 such as search history) to its server through different ways (e.g., cookie and Web beacon).
5. T creates a profile for the received id and associates the collected information with the profile.
6. The user directly visits another website `example-2.com` (or E_2) which includes the same advertisement (i.e., $ad.1$) from T.
7. T again generates the same id for the user's browser.
8. T sends the id, the URL of the visited website (i.e., E_2), and the recorded information to its server.
9. T updates the profile for the received id and adds the collected information to the profile.

As shown in Figure 2.2, the advertising company T tracks the user across different websites (i.e., `example-1.com` and `example-2.com`) and records the user's browsing

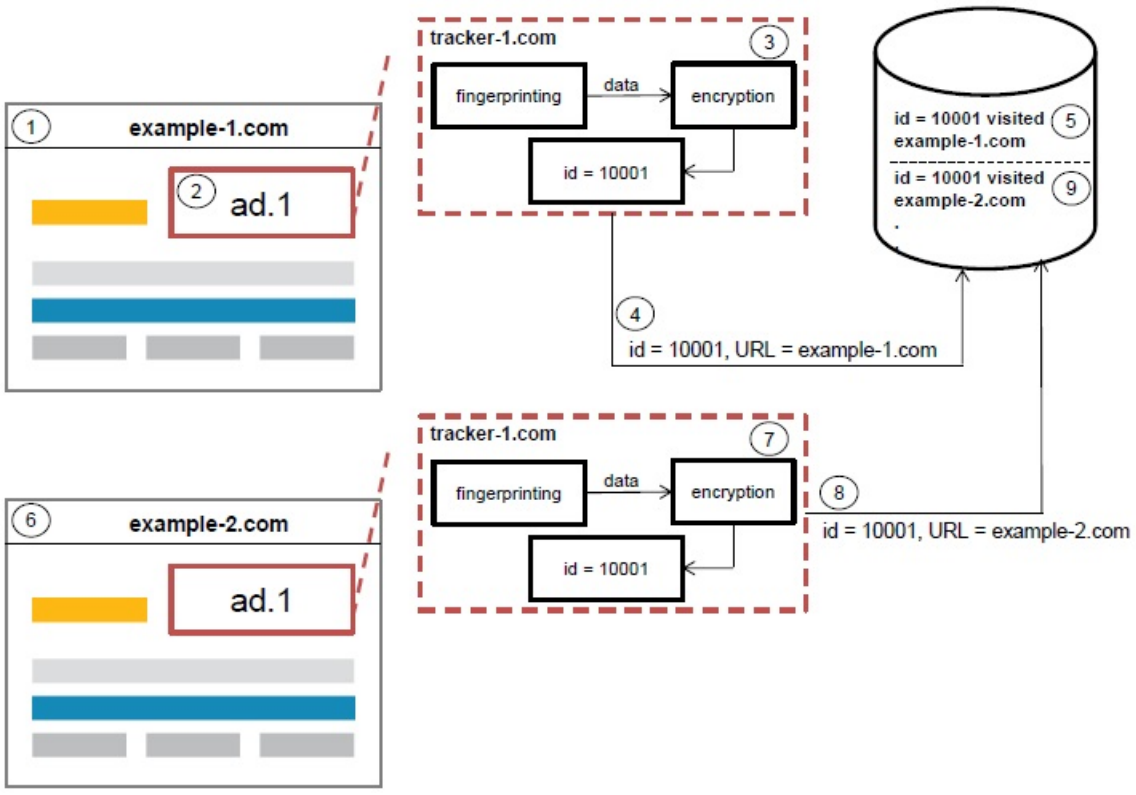

Figure 2.2: An overview of user tracking across different websites using fingerprinting

history. Analyzing the user's profile might reveal personal information about her. For example, a user who frequently visits dating websites might be single or a user who visits websites for getting information about HIV disease might be infected with the disease. In this way, advertising company T can infer personal information of the user based on the history of visited websites.

2.3 Anti-Fingerprinting Solutions

As explained by Acar [33], collecting data through fingerprinting methods introduces two sets of concerns. First, extracting data through fingerprinting takes advantage of the user's Web or browsing experiences. It should be noted that, users have limited

otherwise no knowledge about the usage of the collected data. Second, fingerprinters (at least all those we investigated) are invisible to users and hence the users are unaware of being fingerprinted. Additionally, some attentive users might observe fingerprinting activities, but they have no control to stop or reduce the disclosed information.

When browser fingerprints are associated with individuals (i.e., real names) and are collected without their consent, privacy is the most pressing concern. In fact, a common practice in today's social networking platforms (e.g., Facebook) is that most websites integrate with such platforms for popularity and traffic attractions. This opens a gateway for fingerprinters to easily reveal the privacy of the users. As mentioned earlier, fingerprinting providers often provide an opt-out feature for the users who want to stop sharing their browser's fingerprint. However, such solutions are not adequate as most users simply do not trust fingerprinting providers. As reported in a recent study on fingerprinting [17], `AddThis` [22] which is a popular content sharing platform was exploiting a fingerprinting method called canvas fingerprinting (see Section 3.1.3)[1]. `AddThis` offers an opt-out feature which can be used by the users to stop being fingerprinted. However, authors in the study investigated the fingerprinter and found that it does not stop data collection if the user chooses to opt out of being fingerprinted. As a result, users need a solution for detecting fingerprinting and controlling their privacy rather than trusting the fingerprinting company.

In general, the existing anti-fingerprinting solutions can be categorized as follows based on the approach they use: (i) randomization-based, (ii) filtering-based, and (iii) invalid representation.

Randomization-based [34]: The key idea behind randomization techniques is to

[1]AddThis stopped using canvas fingerprinting as of mid July, 2014.

use randomization policies to change the browser fingerprint. Browser fingerprint is highly dependent on the browser and device properties. As a result, changing the properties randomly causes fingerprints to change randomly. A randomized browser fingerprint cannot be used for tracking purposes, because the generated fingerprint is not constant once a randomization algorithm is applied. Moreover, a poor randomization policy could limit the browser's functionality. For example, a randomization policy might randomly hide a number of plugins from the current browser's plugin list to deceive fingerprinters. This approach might hide a particular plugin which a website needs for loading its resources. For example, `Facebook`, one of the most popular social networking services, does not load Flash files if the Flash plugin is hidden from the browser's plugin list.

Filtering-based [35, 36]: Filtering approaches either disable the technologies used for fingerprinting (e.g., HTML5 canvas element and Flash plugin) or return an empty value for the browser's properties. While these approaches reduce the fingerprinting surface, users can still get fingerprinted and tracked. The reason is that the browser fingerprint remains constant and is not randomized. Furthermore, disabling Web technologies or returning empty values for the properties that are queried by websites can cause functionality loss of the browser.

Invalid Representation [37]: Invalid representation techniques return an incorrect (spoofed or mimicked) values for the browser's properties. For example, `userAgent` switchers are browser extensions that have been widely used to change the `userAgent` property of the browsers. The `userAgent` property is accessible through the browser and contains information such as the name, version, and platform of the browser. Therefore, changing the `userAgent` string changes the user's browser name, version

and, platform and thus can change the browser's fingerprint.

The abovementioned anti-fingerprinting solutions are not only ineffective to prevent fingerprinting attempts but can cause loss of functionality of the browser (e.g., denial-of-service). This highlights the need for a more effective approach which detect and prevent fingerprinting attempts at runtime without causing loss of browser functionality.

2.4 Summary

In this chapter, we described relevant background information on browser fingerprinting. We described the terminologies and definitions used in the fingerprinting area. We also presented a categorization of the methods that are exploited for obtaining fingerprinting data. Furthermore, we brought attention to the privacy concerns from the user's perspective by explaining a tracking scenario leveraging fingerprinting. As discussed in this chapter, tracking the user's browsing history across multiple websites might reveal the personal information of the user. In addition, we described a solution, offered by fingerprinting companies and pointed out its shortcoming and the reason for a better solution where the user does not need to trust the fingerprinting company. Moreover, we presented a categorization of the existing solutions for overcoming fingerprinting (see Chapter 6 for more details). In the next chapter, we present our hybrid browser fingerprinting tool which employs the most widely used fingerprinting methods on the Web. In addition, we extract the metrics related to each method which are indicators for performing fingerprinting attempts.

Chapter 3

Fingerprinting: Workflows and Metrics

In this chapter, we present our hybrid fingerprinting tool which employs the existing fingerprinting methods on the Web. We show how these methods work (their workflows) and how they hide themselves from fingerprinting scanners. We call this process reverse engineering of fingerprinting. In particular, Section 3.1 describes an overview of the reverse engineering process we followed for identifying the fingerprinting indicators, which in turn would help us detect fingerprinting attempts. In Section 3.2, we present the design and implementation of our hybrid fingerprinting tool. We present the data collection process and our experiments on fingerprinting methods and compare them in terms of effectiveness in fingerprinting in Section 3.3. We separately present the extracted metrics related to each fingerprinting method in Section 3.4. Section 3.5 discusses how we integrated our hybrid fingerprinting tool with a social networking service. We also present a scenario in which a user's personal information can be linked to the browser's fingerprint, causing the privacy leakage of the user.

3.1 From Fingerprinting Methods to Features

As mentioned in the previous chapter (Chapter 2), the fingerprinting process obtains the device information through the browser. In particular, it uses different APIs or exploits various Web technologies for acquiring such information. The goal of this chapter is to analyze the most widely used fingerprinting methods on the Web, identify their building blocks (i.e., the workflow which is followed to fingerprint the user's browser), and define fine-grained and consolidated metrics. We first describe the workflow for each of the fingerprinting methods along with the details of their characteristics. By reverse engineering their workflow, we identify metrics associated with each method. These metrics are the basis for designing our detection approach. We then develop a fingerprinting service by unifying the properties of each of the studied fingerprinting methods and deploy it for data collection. Our unified proof-of-concept fingerprinting service is available for public use[1].

To understand the effectiveness of fingerprinting methods and reverse engineer them, we first implemented a hybrid proof-of-concept fingerprinting tool and developed it as a Web application, named `Fybrid`. `Fybrid` implements the features of popular fingerprinting methods currently existing on the Web as presented by previous studies [23, 16, 17]. In what follows, we present the four major fingerprinting methods present on the Web.

3.1.1 JavaScript Objects Fingerprinting

Web browsers have built-in JavaScript objects which provide APIs for accessing the browser and underlying device-related properties. The JavaScript objects found to

[1] http://www.fademi.com/fp

be effective for fingerprinting are `navigator` and `screen` [38, 14]. We analyzed these objects and collected the following properties that are programatically accessible using JavaScript (it should be noted that the following properties are not supported by all browsers):

- **navigator (20 properties):** *userAgent, appCodeName, appVersion, buildID, platform, cpuClass, oscpu, product, productSub, vendor, language, browserLanguage, systemLanguage, doNotTrack, javaEnabled, geolocation, onLin, plugins, mimeTypes, cookieEnabled.*

- **screen (12 properties):** *width, height, colorDepth, pixelDepth, bufferDepth, deviceXDPI, deviceYDPI, logicalXDPI, logicalYDPI, systemXDPI, systemYDPI, updateInterval.*

Among the abovementioned properties, `plugins` and `mimeTypes` are arrays of objects. The first array contains `Plugin` objects and the second array contains `MimeType` objects. As mentioned earlier, a plugin is a piece of software for equipping the browser with additional features. `Plugin` object has some properties such as the name, description, and file name of the plugin. MIME type indicates the type of a content on the Web. For example, `text/html` is used for normal webpages or `application/pdf` is used for Adobe PDF documents. `MimeType` object has some properties such as the type, description, and the `Plugin` object for the MIME type.

In order to collect the extracted properties, the fingerprinter must first access these properties by calling them. Consequently, we consider having access to the properties of the aforementioned objects as an indicator for JavaScript objects fingerprinting.

3.1.2 JavaScript-based Font Detection

An HTML document consists of a number of HTML elements where each of them can have a number of properties for styling purposes. Font-family is one of the properties that is used for an HTML element to specify the element's font. This property has a fallback system indicating that the property can hold several font names. If the first font is not available on the user's system, it tries the next font and this process continues for all fonts in the fallback list. If none of the fonts are available on the user's system, the browser renders the element with a particular default font. It should be noted that each character of a text has different style and size in different fonts. For example, Table 3.1 shows the offset properties (i.e., `offsetWidth` and `offsetHeight`) for a `<DIV>` HTML element containing the sentence "The quick brown fox jumps over the lazy dog", which is an English-language pangram[2], for four different font names with the same font size (`72px`). `offsetWidth` and `offsetHeight` are read-only properties showing the width and height of an HTML element. `<DIV>` is an HTML element which defines a container in an HTML document.

Table 3.1: Offset properties for an example pangram

Font Name	offsetWidth (pixels)	offsetHeight (pixels)	Available on System
Default	**273**	**19**	**yes**
monospace	344	16	yes
cursive	327	23	yes
new-font	**273**	**19**	**no**

Table 3.1 shows the offset properties for the system's default font, two fonts (i.e., "monospace" and "cursive") that are available on the system, and a font which is not

[2]A pangram is a sentence which contains all alphabet letters of a language (e.g., English).

present on the system. The offset properties for available fonts differ from the default font. However, the offset properties for an unavailable font is equal to the default font because of the fallback system. Therefore, it is possible to check the presence of a particular font on the user's system by loading the font and comparing the offset properties with the default font. In order to check for the availability of a list of fonts on the user's system, firngerprinters first apply an unknown font (e.g., "new-font") which is unavailable on the user's system on a sample text (usually a pangram) in an HTML element and record the value of the properties for the HTML element. The recorded values are default values. Fingerprinters then apply the fonts in the list on the text and measure the current values for the offset properties of the element (new values). Next, they compare the new values with the default values. An unavailable font has new values equal to the default values, while the new values are different for an available font [23].

The process of JavaScript-based font detection consists of loading fonts for a number of HTML elements and measuring the offset properties of the elements. Therefore, we consider these attempts as indicators for fingerprinting using JavaScript-based font detection.

3.1.3 Canvas Fingerprinting

Canvas is an HTML5 element for drawing graphical contents on the fly using JavaScript. It has methods for drawing images as well as retrieving the image data (or pixels) of the canvas content. In order to fingerprint a browser, first the fingerprinter draws an arbitrary context (usually a pangram text) on a canvas element. Then, it collects the image data of the canvas element. The extracted image data (pixels) is dependent on

the underlying browser and the running operating system. Therefore, the image data is different for various combinations of browsers and operating systems and thus could be used for browser identification [25]. Therefore, we consider the above steps (writes and reads to a canvas element) as indicators for detecting canvas fingerprinting.

3.1.4 Flash-based Fingerprinting

Flash has classes for accessing system-related properties. Two classes of Flash that are mainly accessible through APIs for acquiring system information are `Capabilities` and `Font` [14]. `Capabilities` is a Flash class which provides information (Listing 3.1 [39]) about the underlying system (e.g., operating system's name) and the running application (e.g., Flash plugin version). The `Font` class provides APIs for managing embedded fonts in Flash files. For example, this class has a method (i.e., `Font.enumerateFonts`) for getting all available embedded and system fonts. As a result, Flash can be used for enumerating system fonts and collecting system information. We consider the presence of methods for collecting system fonts and getting system-related properties in the source code of Flash files as indicators for performing fingerprinting.

Listing 3.1 System properties that are accessible through the Capabilities class

```
avHardwareDisable, hasAccessibility, hasAudio, hasAudioEncoder,
hasEmbeddedVideo, hasIME, hasMP3, hasPrinting, hasScreenBroadcast,
hasScreenPlayback, hasStreamingAudio, hasStreamingVideo, hasTLS,
hasVideoEncoder, isDebugger, isEmbeddedInAcrobat, language,
localFileReadDisable, manufacturer, maxLevelIDC, os, pixelAspectRatio,
playerType, screenColor, screenDPI, screen Resolution, touchscreenType,
cpuArchitecture, languages, supports32BitProcesses, supports64BitProcesses,
version, System.ime, System.useCodePage, System.vmVersion, Mouse.cursor,
Keyboard.physicalKeyboardType, flash.ui.Keyboard.hasVirtualKeyboard,
Mouse.supportsCursor
```

3.2 Fybrid Overview

As described in the previous section, `Fybrid` unifies the fingerprinting methods that are widely used by popular fingerprinters to create a unique identifier for a browser instance. Figure 3.1 shows an overview of our hybrid approach for performing browser fingerprinting. As shown in the figure, the main components of `Fybrid` are: *Collector* and *ID Builder*. Collector is the core component of `Fybrid`. This component is responsible for collecting the browser's properties by exploiting various APIs such as *JavaScript objects*, *Flash plugin*, and *HTML5 canvas element*. The ID Builder component transforms the collected browser properties into an ID.

Let us assume that a user visits `Fybrid`. After the user clicks on the "submit fingerprint" link, Collector first fetches the device and browser-related properties. Then, ID Builder applies an encryption function (e.g., MD5) on the collected information and generates an ID for the browser. Finally, `Fybrid` asynchronously sends the generated ID along with the collected information to a remote database. The remote database accumulates the produced IDs and the retrieved fingerprinting data

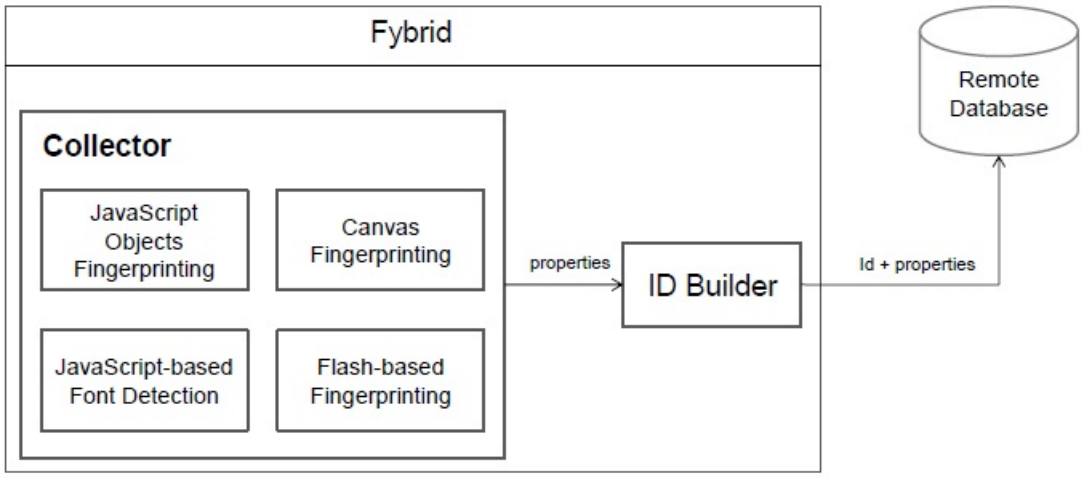

Figure 3.1: Overview of Fybrid — a hybrid browser fingerprinting tool

for each user.

Collector implements the four fingerprinting methods described in the previous section (Section 3.1). In what comes after, we discuss the properties that are collected using JavaScript objects, JavaScript-based font detection, canvas fingerprinting, and Flash-based fingerprinting.

JavaScript Objects Fingerprinting: As noted before, 20 properties are observable through the `navigator` and `screen` JavaScript objects (see Section 3.1.1). As a result, Collector queries for all properties of these objects. Additionally, for the `plugins` and `mimeType` properties which are arrays of `Plugin` and `MimeType` objects, Collector concatenates all the properties for each object in the arrays and considers it as one property. Therefore, we consider `plugins` array as one property and the `mimeType` array as one property. In total, Collector extracts 32 fingerprintable properties from the mentioned objects and includes them in our fingerprinting tool.

JavaScript-based Font Detection: Collector executes the same procedure as described in Section 3.1.2 for detecting fonts on the user's system. In order to create a

list of fonts for checking their presence on the user's system, we investigated the code of a popular fingerprinting provider (i.e., `bluecava` [21]) and found 460 fonts in the code of this fingerprinter. Collector checks for the presence of these 460 fonts which is divided into three groups targeting different operating systems (Windows: 231 fonts, Mac OS: 167 fonts, and other operating systems: 62 fonts).

Canvas Fingerprinting: Collector implements this method by first drawing a pangram (i.e., "The quick brown fox jumps over the lazy dog") on a canvas element ($300px \times 177px$) and then extracting the generated image data as a `base64` encoded string.

Flash-based Fingerprinting: Collector employs two APIs of the Flash plugin for collecting information as described in Section 3.1.4: `Font` and `Capabilities`. Collector uses the `Font` class for extracting system fonts by calling the `enumerateFonts` method. `Capabilities` class is used for collecting system-related properties such as operating system's name, Flash plugin version, etc. Collector extracts 39 system properties using the `Capabilities` class (see Listing 3.1).

3.3 Data Collection and Experimental Evaluation

In this section, we present our experimental results on fingerprinting methods over a dataset of browsers' fingerprints. We first describe our database and the process of collecting fingerprints. Then, we evaluate each fingerprinting method and the properties that are used in the fingerprint. Finally, we discuss our evaluation results.

3.3.1 Data Collection

As noted before, `Fybrid` unifies the most widely used fingerprinting methods on the Web. To conduct a study on the exploited methods, we asked a number of members of the `microWorkers` marketplace [40] to each to visit `Fybrid` and submit their browsers' fingerprint into our database. `Microworkers` is an online platform for completing online tasks. We collected browser fingerprints for 1,523 different members. Each fingerprint in our database is a vector of 74 properties (or features) of the user's system. Table 3.2 shows the number of features extracted using each fingerprinting method.

Table 3.2: The number of properties extracted using each fingerprinting method

Fingerprinting Method	**Number of Features**	**Description**
JavaScript Object Fingerprinting	20 + 12 (32)	accessible properties through `navigator` and `screen` objects
JavaScript-based Font Detection	1	number of available fonts on the user's system (of 460 fonts)
Canvas Fingerprinting	1	image data of the canvas element
Flash-based Fingerprinting	1 + 39 (40)	one feature for available fonts on the user's system and the remaining for the system-related properties
All methods	74	total features in the fingerprint

3.3.2 Experimental Evaluation

We used the Shannon's entropy formula (Equation 3.1) [41] to quantify the importance of properties (or features) in the fingerprint. This formula calculates the amount of

information that is produced by each feature in the fingerprint and has been used in the previous studies on fingerprinting [14, 7]. In this formula, the unit of the produced information is called "bit". We adopted this formula for the purpose of calculating the importance of features in the fingerprint as well as for each fingerprinting method. For a given feature F in the fingerprint with n different values (for all 1,523 browser fingerprints), the entropy or the produced information is specified as follows:

$$H(F) = -\sum_{i=1}^{n} p_i \times \log_2 p_i \tag{3.1}$$

In the Equation 3.1, p_i is the probability of occurrence of the i^{th} value. Using the Equation 3.1, a feature which has different values for all 1,523 browser fingerprints (the feature has 1,523 unique values) has the maximum entropy value of 10.57 bits and can be used as an identifier to uniquely identify each browser instance.

To evaluate the importance of features in the fingerprint, we calculate their entropy values. Features with higher entropy values are more informative (have more variant values) and make the fingerprints more unique. Figure 3.2 shows the top 10 attributes with the highest entropy values (see Appendix A for entropy values of all attributes). The `plugins` property (in Figure 3.2) of the `navigator` object has the maximum entropy value which is 9.97 bits. This means that a randomly chosen browser from 1,523 browsers in our database shares the same plugins with at most one browser in the other 1,002 browsers ($2^{9.97}$). As we noted earlier, the value of the `plugins` property for each browser in our dataset is the concatenation of string values of the name, filename, and description properties for all `Plugin` objects. Each `Plugin` object represents an installed plugin on the browser. Using only `plugins` as the browser's

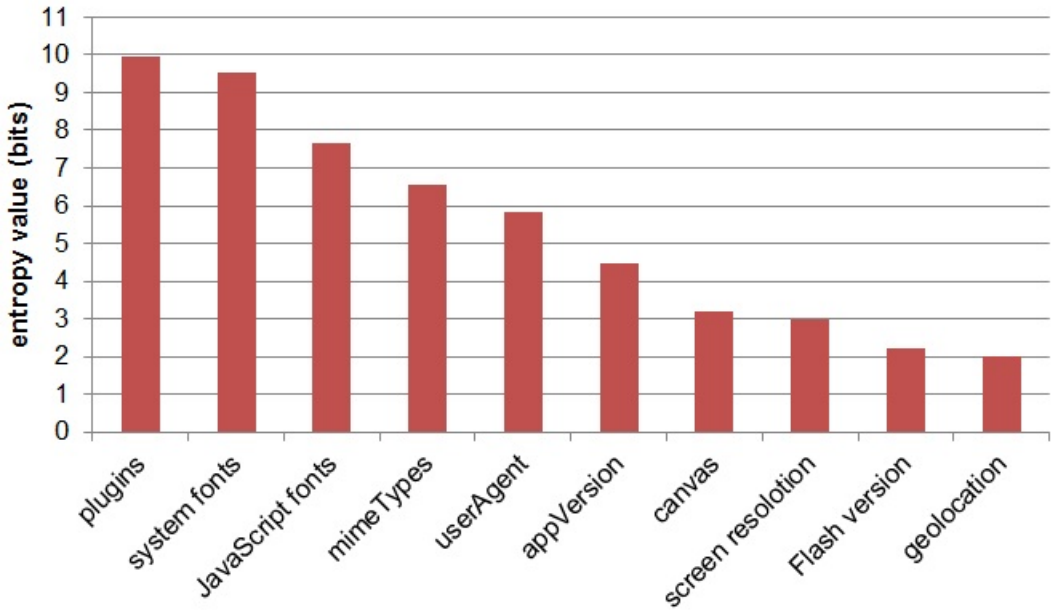

Figure 3.2: Top 10 attributes with the highest entropy values

fingerprint, 82% of fingerprints are unique in our database, whereas when using all the features, 90.41% of browsers have unique fingerprints. This clearly shows that `plugins` property plays a significant role in making fingerprints more unique.

In our experimental analysis, after `plugins`, `system fonts` (collected using Flash plugin) and `JavaScript fonts` (collected using the JavaScript-based font detection method) have the highest entropy values. It should be noted that we only used a set of 460 fonts for JavaScript-based font detection. Thus, adding more fonts would make the entropy value of JavaScript-based font detection closer to the entropy value of the font enumeration using Flash plugin.

Moreover, from the analysis of the collected fingerprinting data, we found that canvas fingerprinting (`canvas` in Figure 3.2) has the entropy value of 3.2 bits. While this technique is shown to be effective for identifying hardware/software configurations of the user's system and is prevalent among top websites of Alexa [17], alone it is not

an ideal choice for fingerprinting due to its low entropy value in our experiment.

Prior to our experiment, our assumption was that all features are necessary to uniquely fingerprint a browser and/or system. However, our experiment showed that most of the attributes (e.g., `appCodeName`) implemented by the existing fingerprinters might be redundant and hence do not contribute to the actual fingerprinting activities.

In order to find out about the information required by fingerprinters to identify a particular browser, we adopted a feature selection algorithm [42]. Our goal is to find a minimum subset of features that are sufficient to uniquely identify a given browser. In machine learning, feature selection is a process used to identify an optimal subset of features for model development. The goal of feature selection is to find the best subset by searching the entire feature space repeatedly and maximizing an objective score (e.g., the accuracy of a given classifier). Since searching all possible subsets for a feature space consisting of N features is expensive (2^N), different heuristic search methods have been introduced in the literature. Among the existing feature selection algorithms, we adopted a Greedy Forward Feature Selection (GFFS) algorithm [43]. GFFS begins with an empty set and continues adding a feature at a time to the set of already selected features until adding more features does not improve the objective function.

In our case, the GFFS algorithm begins by evaluating the entropy for all features individually. It chooses the feature with the maximum objective score which is the entropy value. Next, it adds another feature from the remaining features and chooses the best subset consisting of two features. This procedure is repeated on the subsets with three, four, and more features until adding more features to the current subset does not affect the entropy value. In order to evaluate the entropy for a subset of

features, in our case, we concatenate the contents of the features and consider the resulting value as one feature for all fingerprints. We applied the GFFS algorithm on all features (74 features) included in the fingerprint. Based on the feature selection algorithm, only eight features which are called dominant features (`plugins`, `userAgent`, `screen resolution` (i.e, `width` × `height`), `geolocation` (i.e, `latitude` & `longitude`), `onLine`, `system fonts`, `screenDPI`, and `canvas`) are sufficient for identifying a given browser. The identification accuracy of using these eight features is equal to the identification accuracy of using all features. Table 3.3 shows the selected eight features with their entropy values and the number of unique values for each feature.

In addition to calculating entropy values for all features, we perform the same experiment for all the fingerprinting methods discussed earlier. More specifically, we calculate entropy values for each fingerprinting method by concatenating the values of the features that are extracted using each method and considering it as one feature. Therefore, we transform the fingerprints in our database from 74 features to only four features. Table 3.4 shows the calculated entropy values for each fingerprinting method along with the percentage of unique browsers (from 1,523 browsers) considering each method as the browser's fingerprint.

As shown in Table 3.4, *JavaScript Objects Fingerprinting* has the highest entropy value (10.27 bits), and 89.69% of the browsers have unique JavaScript objects' properties. Moreover, *JavaScript-based Font Detection* can be a good alternative to *Flash-based Fingerprinting* when the Flash plugin is not present or is disabled on the browser. Finally, as illustrated, unifying all fingerprinting methods could increase the identification accuracy (90.41%).

Table 3.3: Entropy values for the dominant features

Feature Name	Entropy Value (bits)	Unique Values (number)
plugins	9.97	1249
userAgent String	5.83	342
screen resolution (width & height)	2.99	39
geolocation	2.02	118
onLine	0.04	2
system fonts (Flash plugin)	9.55	1091
Screen DPI (Flash plugin)	0.07	4
canvas	3.21	78

Table 3.4: Entropy values for fingerprinting methods

Fingerprinting Method	Entropy Value (bits)	Unique Browsers (%)
JavaScript Objects Fingerprinting	10.27	89.69
Flash-based Fingerprinting	10.07	83.25
JavaScript-based Font Detection	7.64	43.79
Canvas Fingerprinting	3.21	0.05
All Methods	**10.28**	**90.41**

3.4 Metrics for Fingerprinting

As noted in Section 3.1, there are a number of indicators for each fingerprinting method which show the attempts for obtaining system-related information and thus fingerprinting. After analyzing the code of popular fingerprinters found in the previous large-scale experimental studies [16, 17, 23] (e.g., `bluecava` [21], `ThreatMatrix` [44], `AddThis` [45], etc.), we identify a set of metrics for each fingerprinting method that would correspond to a future fingerprinting attempt (9 metrics in total as shown in Table 3.5).

JavaScript Objects Fingerprinting: As discussed earlier, this fingerprinting method consists of multiple accesses to the informative objects' properties. As a result, we

consider the total number of accesses made by a webpage to the `navigator` and `screen` objects' properties as a metric (*Metric 1*) for this fingerprinting method. In addition, the `Plugin` and `MimeType` objects have properties that are accessible using JavaScript. We also consider the total number of accesses to the properties of these two objects as the second metric (*Metric 2*) for JavaScript objects fingerprinting.

JavaScript-based Font Detection: As described in the previous section, to check for the presence of a particular font, the fingerprinter first sets the font on an HTML element. Then, it measures the offset properties of the HTML element. We consider the total number of loaded fonts (*Metric 3*) as well as the total number of accesses to the offset properties of the HTML elements (*Metric 4*) as metrics for recognizing this fingerprinting method.

Canvas Fingerprinting: The canvas fingerprinting process consists of two steps. The first step is to write a content, often a pangram with different colors, in a canvas element. The second step is to retrieve the produced image data. As a result, we consider the writes and reads to a canvas element as the first metric (*Metric 5*) for this fingerprinting method. In addition, canvas elements that are used for the purpose of fingerprinting are either hidden or dynamically added to the document (e.g., by using the `document.createElement` method of JavaScript). As a result, the second metric (*Metric 6*) shows the canvas visibility status and if it is created dynamically on the fly.

Flash-based Fingerprinting: Flash files that are used for fingerprinting contain methods and classes for obtaining system-related information. Moreover, the Flash files contain methods to transfer the collected information to a remote server or a JavaScript function, or to store the information as cookies on the user's system. In

addition, based on our analysis of popular fingerprinting codes, the Flash files are hidden, small, or added dynamically (e.g., using `document.createElement` JavaScript method) at runtime. As a result, we define the following four metrics for identifying the Flash files that are used with the purpose of fingerprinting. The first metric (*Metric 7*) shows whether the Flash file contains methods for enumerating system fonts or accessing system-related information. The second metric (*Metric 8*) shows the presence of methods for transferring or storing information using Flash, while the third metric shows the visibility of the Flash file (either small or hidden) or if it is added dynamically (*Metric 9*).

Table 3.5: Metrics for fingerprinting methods

Metric	**Description**
Metric 1	specifies the number of accesses to the `navigator` and `screen` objects' properties
Metric 2	specifies the number of accesses to the properties of the `Plugin` and `MimeType` objects
Metric 3	specifies the number of fonts loaded using JavaScript
Metric 4	specifies the number of accesses to the offset properties of HTML elements
Metric 5	specifies whether a canvas element is programmatically accessed (writes and reads)
Metric 6	specifies visibility status (hidden or visible) of a canvas element that is programmatically accessed
Metric 7	specifies the existence of methods for enumerating system fonts and collecting system-related information in the source code of a Flash file
Metric 8	specifies the existence of methods for transferring the collected information
Metric 9	specifies the visibility status of a Flash file (hidden, visible, or small)

3.5 Linking Fingerprints with Users

In this section, we show how we integrated `Fybrid` with a social networking service (i.e., Facebook) to collect browser fingerprints. We chose a social media platform with the goal of understanding whether we can make an association between a browser fingerprint and a user identity such that it reveals privacy leakage through fingerprinting. If this is shown to be true, we can conclude that a browser fingerprint can be a reliable source for identifying individuals as well as identifying the browser itself. We first describe a number of APIs that are available through social networking services that allow a seamless integration of `Fybrid`. Then, we present our study on fingerprinting of the users through the integration of `Fybrid` and highlight the privacy concerns.

3.5.1 Social Media and User Profile

In general, fingerprinting creates identifiers for browsers not individuals. However, it is very complicated, if not possible, to link the browser fingerprint to individuals who use the browser directly. There are a number of reasons contributing to this challenge. For example, a device can be shared among a number of users, and multiple browsers of a user can run on a single device or multiple instances of a browser can run in different modes. So far, none of the existing studies have addressed the probability of linking the browser's fingerprint to individuals. However, we observed that integrating fingerprinting with Social Networking Services (SNSs) can not only allow for linking the browser's fingerprint to individuals but also act as a means for collecting a list of browsers that a user interacts with. More specifically, being very popular among users (with more than one billion monthly active users) [46], a SNS such as Facebook has the potential to offer such exploitation if a well-designed fingerprinting tool is

integrated with it through its APIs.

Facebook provides a platform for developers, using which they can easily build and integrate applications (or apps). Through the integrated apps, users can play online games, listen to music, watch movies, or even share their interests with their friends around the world. Integrated apps play a key role in attracting more users. Facebook currently has more than 10 million integrated apps and websites [47] and more than 400,000 registered developers [48]. In addition, websites extend their functionalities by integrating with Facebook. For example, a website can easily authenticate users by using their Facebook account (i.e., through the `Facebook Login API`) rather than asking them to register in the website. However, websites and apps which a user interacts with at least have access to the publicly available information (public profile) of the user. The public profile includes information such as the user's id, first name, last name, link to the user's profile, gender, and locale. As a result, integrating the browser's fingerprint with Facebook might lead to linking the browser's fingerprint to the public profile of individuals instead of the browser itself.

In what follows, we describe the Facebook platform and explain a number of APIs that are available for accessing the user's personal information. Then, we provide a scenario that shows the possibility of correlating the browser's fingerprint to the public profile of users.

3.5.2 Integrating with the Facebook Platform

The Facebook platform is a framework for developers to build applications that interact with the core features of Facebook. `Facebook Login API` is one of the components of the Facebook platform which enables applications to authenticate users without

the need for registration and thus speeds up the login process. Applications that authenticate users using this API, have access to the public profile of users. Moreover, applications that require more personal information about the users and need to read or publish contents on Facebook on behalf of them, need to ask for specific permissions. For example, an application that requires the user's email address needs the user's permission. For security reasons, applications that ask for more information than the public profile, friends list, and email address of the users may need to be reviewed by Facebook before granting access to the user's information [49].

Another component of the Facebook platform is `Canvas`. `Canvas` is an `iframe` in which an application can be directly embedded in Facebook to run on personal computers (i.e., desktops and laptops) [50]. An `iframe` is an HTML element for displaying a webpage within another webpage. `Canvas` not only integrates the application with the core components of the Facebook platform but also provides developers with the opportunity of using any language or tool that they use for Web development (e.g., PHP, Python, and Flash). Examples of the Facebook platform components are the `Graph API` and `Facebook Login API`. `Graph API` is an interface for reading and writing data to the Facebook's social graph. Through the `Graph API`, it is possible to programmatically query the user's data, post new status, and upload photos on Facebook on behalf of the users. It should be noted that the `Canvas` applications can communicate with remote databases as Web applications do.

3.5.3 Leakage Scenario

Attentive or careful Facebook users can preserve their privacy by properly setting their privacy by going through the `Privacy Settings` option. Leaking personal information of Facebook users, intentionally or unintentionally, through different methods (e.g., request URL and referrer header) has been reported in the previous studies [51, 52]. However, to the best of our knowledge, none of the previous studies addressed the integration of fingerprinting practices with SNSs. The objective of this section is to show the possibility of such integration which would potentially lead to personal information leakage of users. These users are eventually the target of third party tracking companies or malicious users who leverage fingerprinting results for economic-gain. We integrate our hybrid fingerprinting tool, `Fybrid`, with Facebook through Facebook `Canvas` as well as `Facebook Login API`. Integrated Fybrid (`iFybrid`) only asks for the user's public profile and collects the user's id. Therefore, `iFybrid` does not need to be reviewed by `Facebook`. The Facebook user id is an individual-specific identifier for the members of this SNS. Using this id, it is possible to know about the user's public profile information including her first name, last name, locale, gender, etc. For privacy reasons, `iFybrid` does not store the actual id of the users; instead it randomly maps the user's id to a unique identifier. `iFybrid` sends the generated user id and the collected browser fingerprint to a remote database. Figure 3.3 shows the scenario of fingerprinting users through Facebook using `iFybrid` as explained below.

1. The user logins to Facebook or logins to the `iFybrid` Web application through the `Facebook Login API` using her username and password.

2. The user visits the `iFybrid` application.

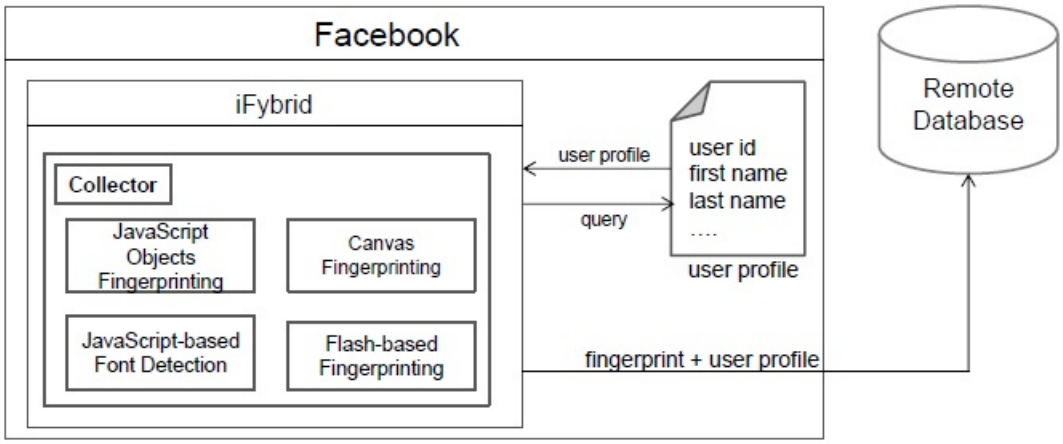

Figure 3.3: Fingerprinting through integrating with Facebook

3. The user clicks on the submit link to submit the browser fingerprint with an id which is uniquely generated for his/her.

4. `iFybrid` sends the collected data to a remote database.

It is worthwhile mentioning that an unknown tracker is able to perform the step 3 without the user's knowledge by using asynchronous requests (e.g., `XMLHttpRequest` of JavaScript). Furthermore, instead of generating a unique random identifier, the tracker could send the actual user id along with the user's browser fingerprint to a remote database. In this way, it is possible to create a rich repository of individuals and their browsers' fingerprints over time. In addition, this provides an opportunity for the trackers to become aware of the number of devices that a user has and keep track of the changes of the browsers to create more stable fingerprints with the time passage.

3.6 Summary

In this chapter, we conducted a study to identify indicators for fingerprinting attempts. We first presented the most widely used fingerprinting methods and their related workflow. We further unified the existing fingerprinting methods and developed a hybrid fingerprinting tool (i.e., `Fybrid`). The tool is developed as a Web application and is available for public use[3]. Next, we conducted an experiment and collected browser fingerprints for 1,523 different members of the `microWorkers` marketplace. We also analyzed and evaluated the importance of each feature in the fingerprint as well as the importance of each fingerprinting method individually. This allowed us to reverse engineer the workflow of the fingerprinting methods and extract metrics which are indicators for recognizing fingerprinting attempts. Finally, we integrated `Fybrid` with a social networking service (i.e., Facebook) and presented a scenario in which the browser fingerprint can be linked to the personal information of the users and therefore can be used for individual identification.

[3]http://www.fademi.com/fp

Chapter 4

Fingerprinting Detection and Prevention

In this chapter, we introduce the proposed approach for runtime detection and prevention of fingerprinting attempts. We first present an overview of the workflow of our proposed approach. In particular, we describe the details of the detection approach in Section 4.1.1 and present the adopted prevention techniques in Section 4.1.2. Finally, we present the implementation details of our approach in Section 4.2.

4.1 FPGuard: Overview

In the previous chapter (Chapter 3), we investigated the most popular fingerprinting tools to answer the following questions: (1) Which properties are used to precisely represent a browser fingerprint (fingerprinting indicators); (2) How do fingerprinters perform fingerprinting (workflow); and (3) How do fingerprinters hide themselves from anti-fingerprinting tools (undetectability). We then built a fingerprinting tool (i.e., `Fybird`) by unifying all the identified properties and deployed it as a Web service to collect browser fingerprints. More specifically, we collected fingerprints for 1,523 unique browsers and applied a feature selection algorithm to quantify the importance of features in the fingerprint. This helped us define fine-grained fingerprinting

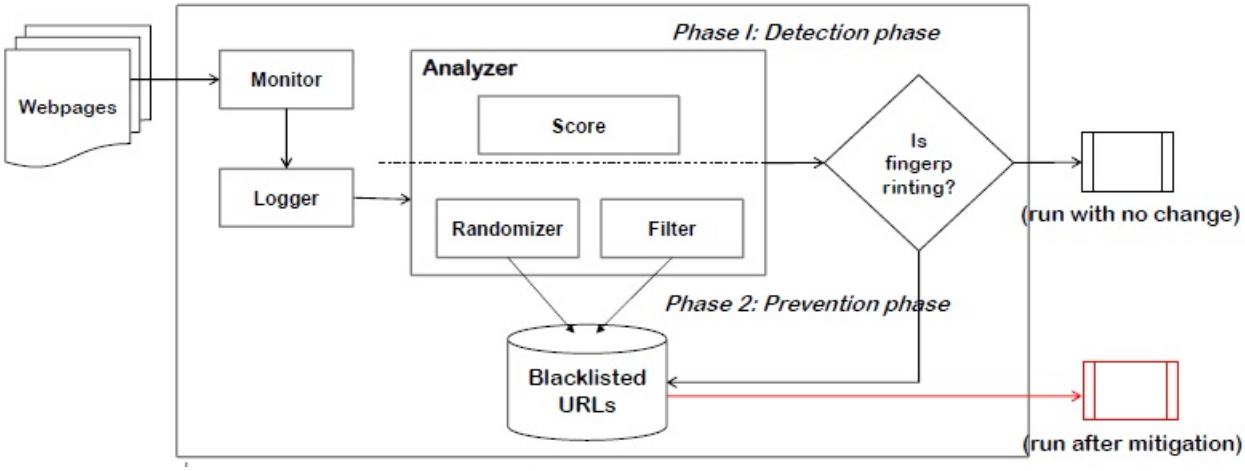

Figure 4.1: An overview of FPGuard

indicators and metrics.

In this chapter, based on the identified indicators and metrics (see Section 3.4) we propose a runtime tool, named `FPGuard`, for the detection and prevention of fingerprinting attempts at runtime. Figure 4.1 shows an overview of `FPGuard`. Given a webpage running on the user's browser, `FPGuard` monitors and records its activities on the user's browser from the time the webpage has started loading. Then, it extracts the metrics relative to each fingerprinting method and builds a signature for the webpage. Next, it uses various algorithms to distinguish normal webpages from fingerprinting webpages. In case, a webpage is flagged as fingerprinter, `FPGuard` notifies the user with an alert and stores the URL of the webpage in a blacklist. The user has the option to either trust the webpage and let it run as is or prevent it from fingerprinting the user's browser using one of the designed prevention techniques. `FPGuard` runs in two phases: *detection* and *prevention*. In what follows, we describe these phases.

4.1.1 Phase I: Detection

In detection phase, `FPguard` identifies the fingerprinting-related activities. The core components of this phase are: *Monitor*, *Logger*, and *Analyzer*. The Monitor component is responsible for collecting websites' activities on the browser. First, it injects the Logger component to the Document Object Model (DOM) [53] tree of the webpage before the loading of other resources. It should be noted that upon visiting a webpage, all major browsers first retrieve the HTML document of the webpage. Then, they parse the HTML document and create a DOM tree for the webpage. The DOM represents the contents of the HTML document as objects in a tree format. It also provides an API to interact with (e.g., modify) the contents of the HTML document. The Logger component runs at the background of the browser and records all the activities of the webpage. This component then parses the logs and extracts metrics for each corresponding fingerprinting method. These metrics will be used by the Analyzer component to look for fingerprinting patterns. Once the Analyzer flags the webpage as fingerprinter, `FPGuard` displays an alert to the user and stores the URL of the webpage with the observed metrics into the blacklist database. In what follows, we discuss the metrics (See Table 3.5) used by the Analyzer component.

Typically, *JavaScript Objects Fingerprinting* method requests several accesses to the `navigator` and `screen` objects' properties. Our first metric (*Metric 1*), can therefore be defined as the total number of accesses made by a webpage to these properties. The `Plugin` and `MimeType` objects also have some properties that are accessible through JavaScript, and the total number of accesses to the properties of these objects represents *Metric 2*. The Logger module records the number of accesses to the following properties of the `navigator`, `screen`, `Plugin`, and `MimeType` objects

whenever they are looked up by a webpage (see Section 3.1.1).

- `navigator`: all accessible properties (20 properties)
- `screen`: all accessible properties (12 properties)
- `Plugin`: name, file name, and description
- `MimeType`: type, suffixes, and description

Using the *JavaScript-based Font Detection*, to check for the presence of a particular font, such fingerprinter first sets the font on an HTML element and then measures the offset properties of the HTML element. Thus, to recognize this fingerprinting method, we defined *Metric 3* and *Metric 4* as the total number of loaded fonts and the total number of accesses to the offset properties of the HTML elements, respectively.

As we previously discussed, *Canvas-related* fingerprinters write something (often a pangram text) in a canvas element and retrieve the produced image data. As a result, we consider the writes and reads to a canvas element (*Metric 5*) as an indicator for fingerprinting. In addition, canvas elements that are used for the purpose of fingerprinting are either hidden or dynamically added to the document (e.g., by using the `document.createElement` method of JavaScript). As a result, for a canvas element where *Metric 5* is true, we consider a metric that shows the canvas visibility status and if it is created dynamically on the fly, as *Metric 6*.

Flash-based fingerprinters employ Flash files containing methods and classes to obtain system-related information and methods for transferring the collected information to a remote server or a JavaScript function and storing this information as cookies on the user's system. Flash files that are used for fingerprinting can be hidden, small, or dynamically added using JavaScript. As a result, we define the following

three additional metrics for identifying Flash files used for fingerprinting purposes. *Metric 7* indicates whether a Flash file contains methods for enumerating system fonts or collecting system properties and *Metric 8* captures the presence of methods for transferring or storing information using Flash, and finally activities that exhibit the visibility of the Flash file or if it is added on the fly are encoded in *Metric 9*.

Using the above metrics (9 metrics), the Analyzer assigns a score (Score engine) depending on the level of suspiciousness. Thus, we define three levels of suspiciousness for performing fingerprinting using those metrics. For example, for *JavaScript Objects Fingerprinting* method, the Analyzer assigns three levels of suspiciousness for accessing the properties of the informative (i.e., `navigator` and `screen` objects), `Plugin`, and `MimeType` objects. A webpage has the first and second level suspicious access to the mentioned JavaScript objects' properties, if *Metric 1* and *Metric 2* surpass the corresponding predefined thresholds. A webpage has also third-level suspicious access when it has first-level and second-level suspicious accesses at the same time (see Listing 4.1).

Listing 4.1 Algorithm for assigning suspiciousness level to JavaScript objects fingerprinting attempts

```
1. Input: Metric 1, Metric 2, Threshold 1, Threshold 2
2. Output: suspiciousness_level
1. if Metric 1 > Threshold 1 then
2.  suspiciousness_level = 1
3. if Metric 2 > Threshold 2 then
4.  suspiciousness_level = 2
5. if Metric 1 > threshold 1 and Metric 2 > Threshold 2 then
6.  suspiciousness_level = 3
7. return suspiciousness_level
```

For *JavaScript-based Font Detection*, using *Metrics 3* and *4*, the Analyzer compares the obtained metrics against the predefined thresholds and any attempt that surpasses to the given threshold value is flagged as fingerprinting (see Listing 4.2).

Listing 4.2 Algorithm for recognizing JavaScript-based font detection attempts

```
1. Input: Metric 3, Metric 4, Threshold 3, Threshold 4
2. Output: is_fingerprinter
3. is_fingerprinter = false
4. if Metric 3 > threshold 3 and Metric 4 > Threshold 4 then
5.  is_fingerprinter = true
6. return is_fingerprinter
```

The Analyzer checks *Metrics 5* and *6* and considers a canvas element as first-level suspicious, if these metrics indicate writes and reads performed on the canvas element. The Analyzer considers a canvas element as second-level suspicious, if the canvas element is first-level suspicious, and is also hidden or dynamically created (see Listing 4.3).

Listing 4.3 Algorithm for assigning suspiciousness level to canvas elements

```
1. Input: Metric 5, Metric 6
2. Output: suspiciousness_level
3. if Metric 5 = true then
4.  suspiciousness_level = 1
5. if Metric 5 = true and Metric 6 = true then
6.  suspiciousness_level = 2
7. return suspiciousness_level
```

Finally, the Analyzer labels each Flash file with a suspicious level from one to three where a level three suspicious Flash file indicates a high probability of a fingerprinting

attempt. In particular, the Flash file is considered as first-level suspicious when *Metric 7* is true. When *Metric 8* is true for a first-level suspicious Flash file, then we consider the Flash file as second-level suspicious. A second-level suspicious Flash file that is hidden, small, or created dynamically at runtime (i.e., *Metric 9* is true) is considered as a third-level suspicious Flash file (see Listing 4.4).

Listing 4.4 Algorithm for assigning suspiciousness level to Flash files

```
1. Input: Metric 7, Metric 8, Metric 9
2. Output: suspiciousness_level
3. if Metric 7 = true then
4.  suspiciousness_level = 1
5. if Metric 7 = true and Metric 8 = true then
6.  suspiciousness_level = 2
5. if Metric 7 = true and Metric 8 = true and Metric 9 = true then
6.  suspiciousness_level = 3
7. return suspiciousness_level
```

4.1.2 Phase II: Prevention

In this phase, the Analyzer module is responsible for preventing fingerprinting attempts by changing the browser's fingerprint every time the user visits a website. However, many websites need information about the browser (e.g., `userAgent` string, screen resolution, etc.) to customize their service for different browser or operating system configurations to run properly. Thus, changes made on the browser should be able to correctly represent the browser without affecting the user's browsing experience. For all URLs in the blacklist, before loading the webpage, `FPGuard` first consults the mitigation engines and then lets the webpage run on the user's browser without interruption. For this purpose, we combine randomization (*Randomizer* engine) and

filtering (*Filter* engine) techniques to achieve a nearly correct representation of the browser.

A robust randomization technique should be able to i) change the browser's fingerprint between multiple visits and ii) return values that represent the properties of the browser almost correctly. For example, the contents of a canvas element are pixels while the content of a `userAgent` property is a string representing the browser's name, version, and the underlying platform. Therefore, a particular randomization technique is needed for each property. Moreover, our filtering technique should be able to remove the fingerprinting attempts (e.g., suspicious Flash files) for mitigation instead of disabling a plugin (e.g., the Flash plugin) for the whole webpage or the whole browser. Below, we describe the randomization modules that we implemented.

The Randomizer component implements four core engines: *objectRand*, *pluginRand*, *CanvasRand*, and *fontRand* to handle the respective fingerprinting attempts. More specifically, the *objectRand* engine generates a random object at runtime to change the objects' properties between multiple visits. In this way, the objects' properties become an unreliable source of identity due to their randomized values. For this purpose, Randomizer retrieves the `navigator` and `screen` objects of the browser, applies some changes such as changing the subversion of the browser or adding noises to the current location of the user (i.e., latitude and longitude of the user's location), and replaces the generated objects with the native objects. However, for `plugins` and `mimeTypes` properties, this approach does not work, because changing the information of a browser's plugin might disable the plugin and cause loss of functionality. Therefore, the *pluginRand* engine adds a number of non-existing virtual `Plugin` and `MimeType` objects to the list of the current `plugins` and `mimeTypes` of the browser.

It also changes the order of these lists upon every visit of the user to the URls in the blacklist. As canvas fingerprinting depends on the contents of canvas elements, the *CanvasRand* engine simply adds minor noises to the contents of a canvas element that is considered suspicious by the Analyzer component. Finally, the *fontRand* engine randomly reports the loaded available fonts as unavailable after the defined threshold has passed a limit. In this way, it changes the list of available fonts on the system and thus changes the fingerprint randomly. As a result, it assures that the webpage cannot employ JavaScript-based font detection for fingerprinting.

For combating Flash-based fingerprinting, instead of disabling the Flash plugin for the browser, we adopt two approaches (*flashFilter*) : i) filtering a Flash file that is identified as suspicious by the Analyzer component, and ii) disabling the Flash plugin for each fingerprinter individually. In the former case, the Logger stores the URL of the suspicious Flash file in the blacklist. Then, Analyzer watches for the URLs of the exsiting Flash files in the blacklist and prevents the browser from loading them. In the latter case, based on the Monitor (the component that flags the webpage as fingerprinter), the Logger stores the URL of the webpage in the blacklist. In subsequent visits, FPGuard disables the Flash plugin for the URL of webpages that are present in the blacklist. For example, the Flash plugin can be disabled for a fingerprinter included as an `iframe` (which is considered as third-party) in another webpage. However, the Flash plugin can be enabled for the webpage that the user directly interacts with.

4.2 Implementation

FPGuard is developed as a combination of an instrumented version of the Chromium browser (version `38.0.2090.0`) and a browser extension integrated with the Google Chrome browser. The reason of such combination is because we could not find a particular solution to implement all the detection and prevention mechanisms in a single tool (i.e., either a browser extension or an instrumented browser). Table 4.1 shows the features that are supported by the modified Chromium browser and the FPGuard's browser extension. We chose to implement our approach in the Chromium browser, because it is an open source browser and it also shares the majority of the source code and features with the Google Chrome browser — the most widely used browser in the last two years (Figure 4.2). In the reminder of this section, we provide details information on the implementation of both the browser extension and the instrumented browser. In addition, we highlight the features that are supported by each one.

Table 4.1: Features supported by the FPGuard's modified Chromium browser and browser extension

Fingerprinting Method	**Browser Extension**		**Modified Chromium Browser**	
	Detect	**Prevent**	**Detect**	**Prevent**
JavaScript Objects Fingerprinting	yes	yes	no	no
JavaScript-based Font Detection	yes	no	yes	yes
Canvas Fingerprinting	yes	yes	no	no
Flash-based Fingerprinting	yes	yes	no	no

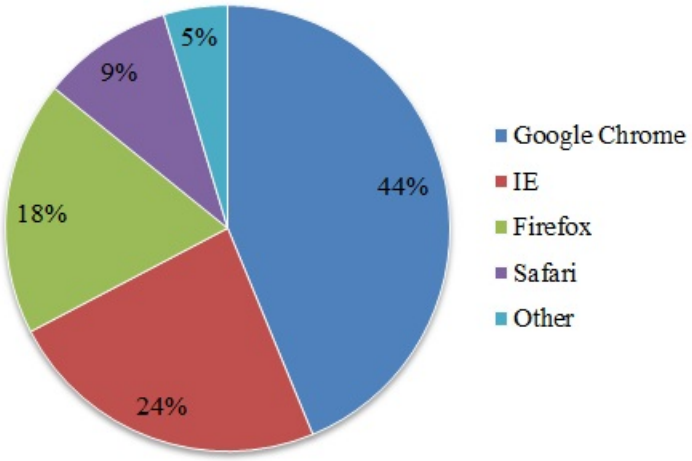

Figure 4.2: Market share of the widely used browsers in 2013 and 2014

4.2.1 Browser Extension

The `FPGuard` extension runs on the background of the browser and watches for fingerprinting-related activities on the browser. Let us assume that a webpage is opened in a tab. The `FPGuard` extension first injects a JavaScript program (i.e., Logger) to the beginning of the webpage's DOM before the loading of other resources. The Logger program records the logs for the activities of the webpage and extracts fingerprinting metrics. In order to log the access to the `navigator`, `screen`, `Plugin`, and `MimeType` objects' properties, we override the getter method of these properties using the `Object.defineProperty` method. In JavaScript, `Object.defineProperty` method [54] can be used to define a new property for an object or modify an existing property of the object. Moreover, it is possible to attach a function to a property so that the binded function is called whenever the property is looked up. We used this feature to log access to the properties (for detection) and to return the values of the objects' properties (for prevention). Similar to the `Object.defineProperty` method, the `__defineGetters__` method can be used for binding a function to an

object's property so that the binded function is called whenever the property is called. However, this method is deprecated from the Web [55]. It should be noted that using `Object.defineProperty` method, fingerprinters are able to see the definition of the function that is binded to an object's property. This means that fingerprinters are able to check whether `FPGuard` is installed or not. However, they might not be allowed to disable `FPGuard`.

In order to record suspicious canvas elements, we override the methods of the `Object.prototype` of canvas elements that are used for writing (e.g., `fillText`) and retrieving the canvas contents (e.g., `getDataURL`) and add our implementation codes (in JavaScript) for recording the writes, reads, and the visibility status of the canvas elements (of the `Object.prototype` of canvas elements). For JavaScript-based font detection, we monitor the changes of fonts for DOM nodes at runtime. `FPGuard` uses the `MutationObserver` DOM API for this purpose. The `MutationObserver API` monitors and records all changes to the contents of the DOM for the specified nodes [56]. Using this API, we check whether the font of the HTML elements in the webpage has changed or not.

Another task of the Logger is to record the suspicious Flash files that are loaded by the webpage. To this end, we first collect the URL of the Flash files that are loaded by the webpage. Then, we decompile them using open source libraries. We use the `as3-commons` libraries [57] for parsing and decompiling Flash files. We build a Flash decompiler by augmenting the `as3-commons` libraries to decompile and traverse the source code of Flash files. The decompiler itself is deployed as a Flash file (in `.swf` format) which has methods for interacting with JavaScript programs. The `FPGuard` extension injects the decompiler's Flash file (with the size of `132.4KB`) to

the beginning of the webpage's DOM after the webpage is loaded. Next, the Logger sends the URL of Flash files to the decompiler. The decompiler first decompiles the Flash files and then traverses the obtained source codes for extracting the related metrics (see Section 4.1.1). Finally, the decompiler sends back the obtained metrics for each URL to the Logger.

As noted before, the `FPGuard` extension is able to enumerate the fonts that are loaded by the webpage at runtime using JavaScript. However, we could not find a solution to get the loading process of the fonts under control (e.g., limiting the number of fonts that can be loaded) using JavaScript. As a result, the mitigation of JavaScript-based font detection attempts is the only shortcoming of the `FPGuard` extension which is implemented in the instrumented Chromium browser.

4.2.2 Browser Instrumentation

We modified the source code of a Chromium browser (version `38.0.2090.0`) with the purpose of detecting and mitigating JavaScript-based font detection attempts. As noted in Section 3.1.2, this fingerprinting method consists of loading custom fonts on an HTML element and then measuring the offset properties of the element. Therefore, we identified the spots in the source code of the Chromium browser in which the browser loads the system fonts for HTML elements. In addition, we specified the spots in the source code where the offset properties of HTML elements are returned upon being called through JavaScript. Then, we added our implementation codes (in C++) for logging and extracting the metrics that are described in Section 4.1.1. For this purpose, we modified the following classes with the purpose of extracting the metrics and mitigating JavaScript-based font detection.

- `CSSFontSelector.cpp`: This class contains methods which are called upon the loading of the fonts for HTML elements. We added our implementation for enumerating the system fonts that are loaded by an HTML document. This modification generates the first metric for recognizing JavaScript-based font detection attempts. In case of mitigating (i.e., the webpage URL is in the blacklist), we extracted the metric (i.e., the number of loaded fonts) and after a predefined threshold, the modified method randomly announces an available font as unavailable for mitigation purposes.

- `Element.cpp`: This class contains methods for obtaining information about HTML elements such as offset properties (e.g., `offsetWidth` and `offsetHeight`) using JavaScript. In particular, we modified three methods that are used for returning the offset properties of HTML elements: `offsetWidth`, `offsetHeight`, and `getBoundingClientRect`. The `offsetWidth` and `offsetHeight` methods return the width and height of an HTML element. The third method can also be used for measuring the width (`getBoundingClientRect().width`) and height (`getBoundingClientRect().height`) of an HTML element. The above methods are modified for extracting the second metric (i.e., number of accesses to the offset properties of HTML elements) for recognizing JavaScript-based font detection attempts.

4.3 Summary

In this chapter, we presented our proposed approach, `FPGuard`, for runtime detection and mitigation of fingerprinting attempts. We first described the basis of our approach for detecting fingerprinting-related activities. More specifically, we described

that we first extract the metrics related to each fingerprinting method and then employ the metrics for identifying fingerprinters. Then, we presented an overview of `FPGuard` by describing the detection and mitigation workflows separately. Our detection approach is based on the metrics that are identified for each fingerprinting method. For each fingerprinting method, we proposed a particular algorithm which calculates a suspiciousness level for the activity. The higher the level of suspiciousness, the higher the probability of performing fingerprinting. To protect users against fingerprinting, we combined randomization-based and filtering techniques. To keep the browser functionality, we adopted different randomization policies suitable for each fingerprinting method. We also proposed filtering techniques to keep the browser functionality and reduce the probability of Flash-based fingerprinting. Finally, we presented the implementation details and the challenges we faced when implementing `FPGuard`. Moreover, we explained the reason of developing `FPGuard` as a combination of an instrumented browser and a browser extension.

Chapter 5

Experimental Evaluation

In the previous chapter, we introduced our approach for detecting and preventing fingerprinting attempts at runtime. In particular, we described the metrics related to each fingerprinting method. Then, we proposed algorithms for calculating a suspiciousness level of an attempt in performing fingerprinting based on the extracted metrics. We also presented an implementation of our approach as a browser extension integrated with a modified Chromium browser (named FPGuard). In this chapter, we first present our evaluation objectives as well as the methodologies we used for evaluating FPGuard from detection, prevention, and overhead points of view (Section 5.1). Here, we also describe the tools and fingerprinters that are employed for the purpose of evaluation. The capability of FPGuard in detecting fingerprinting-related events is presented in Section 5.2. In Section 5.3, we examine FPGuard against four fingerprinting providers to evaluate its effectiveness in frustrating fingerprinting attempts. We measure the performance overhead of FPGuard in Section 5.4.

5.1 Objective and Overview

Our evaluation of FPGuard aims to answer the following three research questions:

1. *Is FPGuard able to correctly identify fingerprinting-related activities at runtime?*
2. *How does FPGuard protect users against the existing fingerprinting algorithms at runtime?*
3. *What is the performance overhead of FPGuard?*

With respect to the first question, similar to previous studies [16, 17], we evaluate FPGuard using the top 10,000 (10K) websites from Alexa. To automate the process of visiting the websites and data collection, we use iMacros [58] extension for the Google Chrome browser. iMacros is an extension for performing repeated tasks such as clicking on links and visiting a number of websites automatically. To collect the metrics and analyze them for each website, we developed an in-house script (Listing 5.1) for iMacros in which for all websites, we first visit the homepage and wait 5 seconds for the loading of the resources and once the website is fully loaded, FPGuard records the metrics and identifies the activities of the webpage as either fingerprinting or normal. In case the website is not loaded in 5 seconds, we wait 10 more seconds and then visit the next website in our list. In this way, we managed to collect metrics for 9,264 websites since some of the websites were not accessible at the time of experiment, were identified as phishing attacks or considered as malware by the browser.

Listing 5.1 Script for automating data collection

```
1. Open a new tab
2. For each website in the list
3.    Open the website in the tab
4.    Wait 5 seconds for the loading of the website
5.    Wait 10 more seconds if the website is not loaded
```

To answer the second question, we use two types of fingerprinting providers to evaluate the effectiveness of FPGuard for preventing the existing fingerprinting algorithms: (i) popular commercial fingerprinting providers named `bluecava` [21] and `coinbase` [59], and (ii) proof-of-concept fingerprinters named `browserleaks` [60] and `fingerprintjs` [61]. We choose to test FPGuard against the mentioned fingerprinters, because they generate a fingerprint for the browser upon visiting them and the generated fingerprint is obtainable in the form of an ID. We use the ID to check whether FPGuard's prevention mechanism can change the ID for a given browser in multiple visits or not. The goal is to change the ID for a browser instance in every visit to a fingerprinting provider. Table 5.1 shows the fingerprinting methods employed by each fingerprinter. Here, we describe each fingerprinting provider and the way we obtained the generated browser fingerprint: (i) `bluecava`, (ii) `coinbase`, (iii) `browserleaks`, and (iv) `fingerprintjs`.

Bluecava [21]

`Bluecava` is a popular fingerprinting company that uses different methods to fingerprint users' browsers. Similar to other fingerprinting providers, `bluecava` offers an opt-out page in which the users can choose to stop sharing their browser's fingerprint. The opt-out page also provides the generated browser fingerprint as an ID. The ID is an encrypted value of the browser's information (e.g., `426D-F151-3461-0C84-44A9-6D61-8F50-58B9`). Moreover, this page shows whether the user is already opted out or not.

Coinbase [59]

`Coinbase` is a popular company which offers an international online wallet for buying, storing, and exchanging bitcoins. Bitcoin is a virtual currency in which people can

send money to each other directly (peer-to-peer) in the form of bitcoins. `Coinbase` offers a payment button that websites can include it in their page and start accepting bitcoins as a method of payment. However, payment buttons include scripts that perform fingerprinting. Then, the generated browser fingerprint is encrypted (which is the browser's ID) and stored as a browser cookie (e.g., `62391a5d87300158cc-62cff4f06fdff7`). In later visits or future tasks, the cookie is sent back to the `coinbase` server upon every request of the user.

Browserleaks [60]

`Browserleaks` is an online web browser security check list which has collected a list of information that is observable to the webpages through the browser. It includes various fingerprinting methods. However, we only test the canvas fingerprinting method of this website which implements a proof-of-concept canvas fingerprinting inspired by a previous study on canvas fingerprinting [25] (see Section 3.1.3). This website employs canvas fingerprinting and generates a unique ID for the browser. For example, upon visiting the canvas fingerprinting link[1], `browserleaks` identifies the name of the underlying browser and the running operating system (e.g., in case of visiting the link using a Google Chrome browser on Windows 7, `browserleaks` reports that "It is very likely that you are using [**Chrome**] on [**Windows**]") and presents the generated ID (e.g., `C5794B43`). It also notifies the user about the existence of the ID in its database. We chose against testing FPGuard for other fingerprinting techniques implemented by `browserleaks` (i.e., other than canvas fingerprinting) as the feedback produced by those techniques is not in the form of ID.

Fingerprintjs [61]

[1] https://www.browserleaks.com/canvas

`Fingerprintjs` is an open source fingerprinting library which implements JavaScript objects fingerprinting and canvas fingerprinting, where canvas fingerprinting can be enabled or disabled. This library reports the browser fingerprint (or ID) as an integer number. We developed a fingerprinting webpage and included this JavaScript library in it. Upon visiting the webpage, two browser IDs are reported by `fingerprintjs`. The first ID (e.g., `229087589`) is when the canvas fingerprinting is disabled and the second ID (e.g., `739337070`) is when it is enabled.

Table 5.1: Presence of fingerprinting methods in fingerprinters

Fingerprinter	JavaScript Objects Fingerprinting	JavaScript-based Font Detection	Canvas Fingerprinting	Flash-based Fingerprinting
bluecava	yes	yes	no	yes
coinbase	yes	yes	no	no
browserleaks	no	no	yes	no
fingerprintjs	yes	no	yes	no

To answer the last research question, we measure the performance overhead imposed by the modified Chromium browser using a JavaScript performance test suite. More specifically, we use `dromaeo` [62] — a JavaScript performance test suite developed by Mozilla — to measure the browser performance when FPGuard runs. We run `dromaeo` separately on the modified browser and an original Chromium browser with the same version. Then, we compare the performance results in terms of runs/second. It is noteworthy that `dromaeo` runs various sets of tests targeting the JavaScript engine, DOM, etc. to measure the performance of the JavaScript engine of a Web browser. For example, it repeatedly modifies the DOM contents and performs string and array operations and mathematical calculations. In addition, we measure the overhead of the FPGuard browser extension by running our fingerprinting tool (i.e., Fybrid). The overhead measured as the difference between the execution times of

Fybrid when the extension is enabled and disabled. It should be mentioned that FPGuard browser extension is equipped with methods for detecting and preventing all of the fingerprinting methods that Fybrid implements. Therefore, running Fybrid measures the total overhead of the browser extension.

We conduct all the experiments on a desktop machine with the following configurations:

- Operating System: `Windows 7`
- CPU: `Intel core i7-860 CPU @ 2.8 GHz processor`
- RAM: `4 GB`

5.2 Detection

As mentioned before, we use the iMacros extension [58] to automatically collect and analyze the metrics for the visited websites. For each website (of 10K websites), first we visit the homepage and wait for 5 seconds until all the resources are fully loaded. Once the webpage is fully loaded, FPGuard starts recording the metrics and identifies the activities of the webpage as either fingerprinting or normal. Here, we report the number of fingerprinting attempts that are discovered by FPGuard for each fingerprinting method: (i) JavaScript objects fingerprinting, (ii) JavaScript-based font detection, (iii) canvas fingerprinting, and (iv) Flash-based fingerprinting.

5.2.1 JavaScript Objects Fingerprinting

By using *Metric 1* and *Metric 2* (see Section 4.1.1), respectively, we enumerate the number of calls for the `navigator` and `screen` objects (informative objects) and the

number of calls for the `Plugin` and `MimType` objects. The related thresholds for both metrics are defined accordingly; the maximum thresholds are when the websites look for all properties of the `navigator` and `screen` objects, and when they access a property for all plugins or MIME types, respectively. For our dataset, FPGuard found that the visited websites looked up for only 15 and 8 properties of the `navigator` and `screen` objects, respectively, while 20 unique properties from the `navigator` object and 12 unique properties from the `screen` object are accessible using JavaScript(see Section 3.1.1). On average, the visited websites have looked up more than 5 properties of the `navigator` object and 3 properties of the `screen` object. In fact, more than 39% of the visited websites have called the properties of the informative objects with more than the average (i.e., more than 5 properties of the `navigator` object and 3 properties of the `screen` object). As a result, returning invalid values for the properties of these objects might cause functionality loss of the browser as mentioned before.

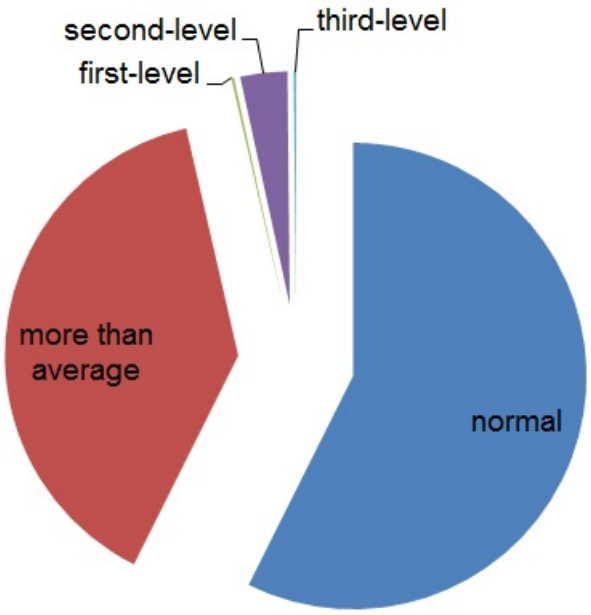

Figure 5.1: Suspiciousness access level distribution for the Alexa's top 10K websites

We also computed the suspiciousness level in performing fingerprinting based on the number of accesses to the properties of the informative objects for each individual website in our dataset. Figure 5.1 shows the result of measuring the suspiciousness level computation where most of the websites have normal (less than average) number of accesses to properties of the informative objects (57.4%). However, the number of websites that have suspicious access of first, second, and third level is considerably low in comparison to the number of websites that have the average number of accesses to the properties.

Among the visited websites, only 101 of them requested for more than 80% properties of the informative objects. That is to say, more than 17 properties of all 22 properties have been called. For example, `Letitbit.net` is one of the websites requested for most properties of the informative objects and the detailed information of

all plugins and MIME types. This is due to the fact that `Letitbit` contains a third-party script from `MaxMind` [63], as we manually investigated. It should noted that `MaxMind` offers an online fraud prevention solution using fingerprinting. Moreover, we visited `bluecava` and `coinbase` and for both, FPGuard reported second-level suspiciousness access to the informative objects' properties. Finally, 15 websites from the dataset looked up for all properties of the `navigator` and the `screen` objects plus the detailed information of all browser plugins and MIME types. Accessing all the information of `plugins` and `mimeTypes` is the characteristic of fingerprinting attempts. Based on our findings (see Section 3.3.2), using only these two properties, we can identify a browser instance in our dataset of 1,523 browsers with almost 90% accuracy.

5.2.2 JavaScript-based Font Detection

Based on our empirical studies, we revealed that most of the websites request less than 50 fonts. Figure 5.2 shows the number of fonts that are loaded using JavaScript by each website of top 10,000 websites of Alexa. Thus, we set 50 as the threshold for the number of fonts that a website can load. The threshold for the average number of accesses to the offset properties of HTML elements is set to 65. Therefore, a given website that loads more than 50 fonts and the number of its accesses to the offset properties of HTML elements surpasses 65 is considered as a fingerprinting candidate. FPGuard identified 22 websites that were loading more than 50 fonts and accessed the offset properties of HTML elements more than 69 times. The websites that loaded more than 300 different fonts and had more than 69 accesses to the offset properties of HTML elements are: `yad2.co.il` (388), `junkmail.co.za` (359), `vube.com`

(327), and `people.com.cn` (325). The mentioned websites either employ fingerprinting themselves or contain a third-party script from a fingerprinting provider (See Appendix B.1 for a list of JavaScript programs that contain methods for performing JavaScript-based font detection).

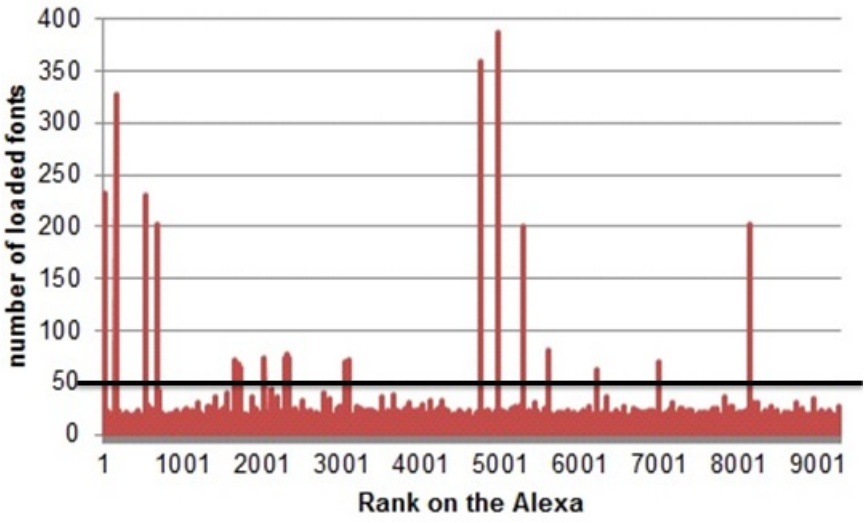

Figure 5.2: Number of fonts that are loaded by Alexa's top 10K websites

5.2.3 Canvas Fingerprinting

Figure 5.3 shows the distribution of suspicious (first and second-level) canvas elements among the visited websites. FPGuard found that 191 websites from our dataset include first and second-level suspicious canvas elements. However, only 85 out of 191 are second-level suspicious elements performing canvas fingerprinting, (e.g., by analyzing their image data (Figure 5.4)). As shown in Figure 5.4, in all cases, the fingerprinter inserts an arbitrary text, often a pangram with different colors to increase the uniqueness of the fingerprint.

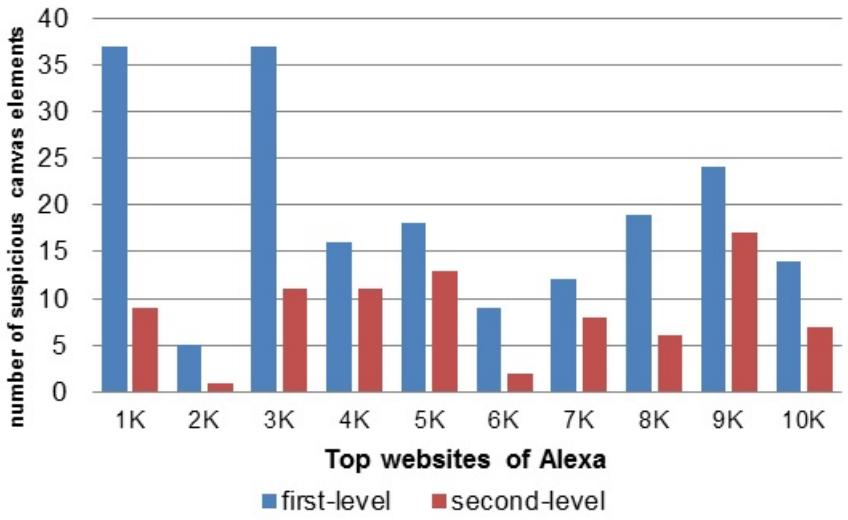

Figure 5.3: Number of suspicious canvas elements found from our dataset

Figure 5.4: Different canvas images used for fingerprinting browsers

However, a recent study identified canvas fingerprinting as the most prevalent type of fingerprinting among the top 10K websites of Alexa (4.93%) [17]. The study found that 95% of the canvas fingerprinting attempts come form a third-party script which belongs to a single fingerprinting provider (i.e., `AddThis` [22]). In fact, after manual investigation, we found that `AddThis` disabled the canvas fingerprinting as of mid July, 2014 [64]. Websites that use canvas fingerprinting belong to categories such as

dating (e.g., `pof.com`), news (e.g., `reuters.com`), file sharing (e.g., `letitbit.com`), deal-of-the-day (e.g., `groupon.com`), etc. These websites either include a third-party script whose main job is fingerprinting (e.g., `reuters.com` and `letitbit.com`) or they actually perform fingerprinting (e.g., `pof.com` and `groupon.com`). For example, `reuters.com` includes a script from `Audience Science` [65] which is an advertising company (from `revsci.net` domain) that implements canvas fingerprinting. This company uses `fingerprintjs` [61], which implements canvas fingerprinting. (See Appendix B.2 for a list of JavaScript programs performing canvas fingerprinting).

5.2.4 Flash-based Fingerprinting

FPGuard found 124 websites including Flash files that contain methods for enumerating system fonts in their homepage and considered them as first-level suspicious. We reveal that 120 websites loaded Flash files that include methods for storing information on the user's system, sending information to a remote server, or interacting with JavaScript programs. Hence, they are marked as second-level suspicious Flash files. From the 120 websites, only 59 of them contain third-level suspicious Flash files, meaning that the Flash files are hidden, small (less than 2×2 pixels), or added dynamically. We manually investigated the source code of the third-level suspicious Flash files and observed that they indeed perform fingerprinting. In addition, four second-level suspicious Flash files are observed to perform fingerprinting (See Appendix B.3 for a list of Flash files performing Flash-based fingerprinting).

5.3 Prevention

To assess the accuracy of FPGuard's prevention technique (see Table 5.2), we repeatedly visited each fingerprinter for 100 consecutive times and collected the generated IDs. Then, we checked the uniqueness of the generated IDs with the goal of achieving the highest unique value from the list of IDs. In other words, a single browser can have 100 different IDs, when one prevention technique is enabled, and thus the fingerprinter cannot uniquely identify the browser upon multiple visits. We define uniqueness as the fraction of the number of unique IDs and the total number of IDs in the dataset.

As shown in Table 5.2, the IDs that are generated by fingerprinters are unique (i.e., 100% uniqueness) for all fingerprinting methods. 100% uniqueness means that the fingerpinter generated 100 different fingerprints for the browser upon each visit. This means that the fingerprinter is not able to uniquely identify the browser between multiple visits. However, in two cases, our approach cannot produce unique identifiers (1% in Table 5.2). 1% uniqueness means that the fingerprinter generated one unique fingerprint for the browser and could uniquely identify the browser in all 100 visits. The first case is when using *flashFilter* against `bluecava` and the second is when using *fontRand* against `coinbase`. In the first case, by disabling the Flash plugin or removing the suspicious Flash file, we only removed the information obtained by the Flash plugin without applying any randomization to the fingerprint. This shows that only disabling the Flash plugin is not quite effective in stopping fingerprinters. In the second case, we investigated the source code of `coinbase` and found that for performing JavaScript-based font detection (see Section 3.1.2), `coinbase` first adds an HTML element and then it loads 263 different fonts (using JavaScript) on the HTML

element. It should be noted that after the loading each font, `coinbase` asks for the offset properties (i.e., `offsetWidth` and `offsetHeight`) of the added element to check whether the font is available on the system or not. However, the offset properties of the element is zero after loading of each font. Therefore, `coinbase` checks for the presence of 263 fonts on the user's system, but cannot detect if any of the fonts are available on the user's system or not.

Table 5.2: FPGuard vs. Fingerprinters

Fingerprinter	**objectRand**	**pluginRand**	**canvasRand**	**fontRand**	**flashFilter**
bluecava	100%	100%	-	100%	1%
coinbase	100%	100%	-	1%	-
browserleaks	-	-	100%	-	-
fingerprintjs	100%	100%	100%	-	-

To evaluate the capability of the existing fingerprinting algorithms, we slightly changed some browser information (e.g., version of the browser) to change the browser's fingerprint in order to check whether the fingerprinter can detect the changes and thus generate an identical ID for the same browser. Surprisingly, none of the fingerprinting providers can detect slight changes made to the fingerprint. For example, `bluecava` generated different IDs for a browser instance upon every visit when the order of the elements in the browser's plugin list was changed randomly. Moreover, `bluecava` could not detect the changes in the browser's version and generated different IDs when changes were performed on the browser's version. We evaluated `browserleaks` and `fingerprintjs` by adding a single-pixel random noise to the content of canvas elements and checked whether they generate an identical ID for the browser or not. They both generated new IDs, demonstrating that they highly depend on the canvas contents and cannot detect minor changes.

Our analysis of the four fingerprinters showed that the current fingerprinting algorithms are not mature enough to detect slight changes of the browser's fingerprint. In addition, gradual changes such as updating the browser version cannot be detected by even popular fingerprinters. Our randomization and filtering engines are used to make the changes undetectable to prevent fingerprinting (i.e., fingerprinters are not able to identify the changes occurred to the browser's fingerprint and thus cannot identify the browser).

5.4 Performance Overhead

Here, we describe an experiment for assessing the overhead of FPGuard. The objective of our experiment is to compare the modified Chromium browser (version `38.0.2090.0`) which includes the implementation of our approach with an unmodified (or original) Chromium browser with the same version. As mentioned before, we use `dromaeo` to measure the browser performance of FPGuard. We run `dromaeo` separately for each browser[2] and respectively obtained 2033.43 runs/second and 2041.07 runs/second. A browser instance with a higher number of runs per second has a better performance. The performance difference between the two browsers is almost negligible (approximately 0.004%). This difference is mainly due to our implementation for metric extraction (e.g., there is a major difference between the `getWidth` and `getHeight` methods of Prototype JavaScript library[3] of the modified and original browsers, which is 93.60 runs/second vs. 169.78 runs/second and 98.40 runs/second vs. 180.92 runs/second). The reason of such a difference is that we added methods

[2]The results of running test suites on modified and original browsers can be seen here: http://dromaeo.com/?id=226795, 226802

[3]http://prototypejs.org/

for recording the access to the `offsetWidth` and `offsetHeight` properties of HTML elements and we also checked their extracted numbers to see if they surpassed the thresholds.

To evaluate the overhead of the FPGuard browser extension, we run our fingerprinting tool (i.e., Fybrid) on an original Chromium browser (version `38.0.2090.0`) and measure the execution time of Fybrid with and without the FPGuard browser extension installed and report it in terms of milliseconds (ms). However, we observed that the execution time of Fybrid changes with every run when not having the FPGuard installed. Therefore, we decided to run Fybrid separately for 100 times on the browser with and without FPGuard installed and measure the average execution time of Fybrid in the form of ($average \pm standard\ deviation$). In this way, we measured the noise of the environment (i.e., standard deviation) on the execution time. In our case, the standard deviation measures the dispersion of execution times from the average execution time. The obtained average execution times with and without FPGuard installed are (180 ± 41.617) ms and (159.400 ± 24.546) ms, respectively. As a result, the total overhead that the FPGuard browser extension imposes on the execution time of the JavaScript programs of webpages is 20.6 ms on average. It should be noted that the measured overhead (20.6 ms) is less than the standard deviation of the execution times when FPGuard is not installed (24.546 ms). Therefore, the added overhead of FPGuard is negligible and does not exceed the inherent noise (i.e., standard deviation) in the reported execution times when FPGuard is not installed.

5.5 Summary

In this chapter, we presented our experiments on evaluating FPGuard from detection, prevention, and overhead point of views. In order to check whether FPGuard is able to detect fingerprinting attempts at runtime, we visited the top 10K websites of Alexa when using FPGuard. FPGuard assigns a suspiciousness level to each fingerprinting-related activity. We found that many of the visited websites have more than the average number of accesses to the JavaScript objects' properties. However, we found a few number of websites which accessed all the properties of `navigator` and `screen` objects. We identify having access to all the properties of `navigator` and `screen` objects as an indicator of performing JavaScript objects fingerprinting. In addition, we found that while many websites load less than 50 fonts (99.7% of 10K websites), there are few websites which load more than 300 fonts. Moreover, we found that canvas fingerprinting is exploited by websites from different categories ranging from dating to news websites. We also found evidences for canvas fingerprinting which demonstrate a decrease in the prevalence of this fingerprinting method in comparison to findings of a previous study. FPGuard found that suspicious Flash files that include methods for enumerating system fonts, have methods for transferring the collected fonts to a remote server or a JavaScript program and to store the fonts as Flash cookies on the user's system. Additionally, some of the Flash files that were invisible, small, or added dynamically at runtime, were mainly used for performing Flash-based fingerprinting. Finally, we found that the performance overhead that the FPGuard imposes on the browser is negligible.

Chapter 6

Related Work and Comparative Analysis

This chapter discusses the existing research work related to fingerprinting. Section 6.1 describes the techniques exploited to fingerprint users' browsers along with the capability of these techniques in identifying browser instances. Section 6.2 details the existing techniques that are used for detecting fingerprinting activities on the Web. Section 6.3 describes the existing tools that are used for protecting users against fingerprinting and highlights the strengths and weaknesses of these tools. Finally, we compare FPGuard with each of the existing tools.

6.1 Fingerprinting Practices

The properties of a browser that are accessible through the `navigator` and `screen` objects together can serve as a unique identifier (or fingerprint) for the browser [38]. In a 2009 study [38], Mayer first concatenated the contents (string values) of the `navigator`, `screen`, `plugins`, and `mimeTypes` objects of a browser. Then, an encryption method was applied on the obtained string to create an identifier (or fingerprint) for the browser. In this experiment, 96.23% of 1,328 users' browsers had unique fingerprints. In addition, Nikiforakis et al. [23] showed that the number of properties

and even their order for the `navigator` and `screen` objects are different for various browsers or different versions of a browser. Therefore, this feature can be used for identifying the browser's name and version.

Following Mayer [38], Eckersley in the `Panopticlick` work [14] conducted a large-scale study on browser fingerprinting and collected browser fingerprints for nearly half a million browsers. In this study, the browser fingerprint is considered as a combination of some properties such as the `userAgent` string, `system fonts`, `time-zone`, etc. It should be noted that the Flash plugin is used for collecting system fonts (e.g., the `enumerateFonts` method of the `Font` class). As the author reported, 94.2% of the browsers had unique fingerprints. In addition, the author discussed that even though various events such as updating plugins or installing new fonts can frequently change the browser fingerprint over time, it is possible to identify the changes of the browser fingerprint with over 99% precision using a simple heuristic algorithm.

In a 2012 study [26], the `CSS-visited` bug [66] is used for history sniffing of the users' browsers. The `CSS-visited` bug is the result of the difference between the color of visited (e.g., purple) and unvisited links (e.g., blue) in the browser. This feature is helpful for user identification. The assumption behind user identification using history sniffing is that the browsing history for each of the Web users is unique and thus users could be identified using their browsing history. In order to extract the browsing history of the user's browser, the fingerprinter simply adds a number of links to its webpage and checks the color of the links. If the color is purple for a link, then the link is visited by the user (the link is present in the browser history), otherwise the link is not visited. In this way, the authors in the study were able to uniquely identify 70% of 368,284 users by testing if a user has visited any websites of

a list consisting of 500 websites. It should be noted that this bug is currently fixed in most browsers [23].

In another study, Mowery et al. [25] used the HTML5 canvas element for browser fingerprinting. The idea behind this approach is that the image data (or pixels) of the canvas element depends on the underlying browser and operating system. As a result, the image data of a canvas element is different for various combinations of operating systems and browsers. In this study, the authors first inserted a pangram text (i.e., "The quick brown fox jumps over the lazy dog") on a canvas element using different fonts. Then, they collected the produced image data from the canvas element and considered the extracted data as the browser's fingerprint. Using this approach, the authors developed a fingerprinting algorithm and collected fingerprints for 294 browsers. They found that 116 out of 294 (39.45%) fingerprints were unique. In other words, they showed that it is possible to identify the name and version of the browser as well as the name of the running operating system.

The difference between the performance, design, and implementation of JavaScript engines of browsers is also used as a means for browser fingerprinting. In particular, Mowery et al. [27] exploited the performance difference of JavaScript engines of browsers. In this study, the authors executed different JavaScript benchmarks (39 benchmarks) on the browser and measured their execution times. Then, they considered the browser fingerprint as a vector of 39 features, where each feature is the execution time for each benchmark. Next, they applied a classification method (i.e., nearest sample using the Euclidean distance metric) on a sample of 1,015 browser fingerprints and were able to correctly classify 810 browser fingerprints (79.8% accuracy). They were able to identify the browser's name (e.g., Google Chrome, Firefox,

etc) with more that 98% accuracy. It should be noted that this approach needs more than three minutes (190.8 seconds) to perform the fingerprinting algorithm, while the average time for viewing a webpage is 33 seconds [67].

Javascript engines of browsers not only are different in terms of performance but also are different in terms of their design and implementation. For example, Mulazzani et al. [28] showed that JavaScript engines of browsers are different in terms of the number of failures in running standard JavaScript test suites. In this study, the authors collected a number of standard JavaScript test suites and executed them on different browsers. Then, they simply extracted the number of failures for each test suite. In this way, they were able to correctly identify the browser's name (Firefox, Opera, Internet Explorer, and Google Chrome) for 175 browsers in their dataset in a reasonable time (i.e., less than one second).

It should be noted that fingerprinting JavaScript engines of browsers (using the performance, design, and implementation differences) can increase the browser identification accuracy. However, none of the previous studies have addressed the presence of such fingerprinting methods on the Web.

6.2 Fingerprinting Detection

To the best of our knowledge, FPdetective [16] is the first large-scale study on detecting browser fingerprinting attempts. In this study, the authors developed a framework by combining dynamic and manual analysis and focused on finding JavaScript-based font detection and Flash-based fingerprinting-related activities. They found that Flash-based fingerprinting (97 websites) is more prevalent than JavaScript-based font detection (51 websites) among the top 10K websites of Alexa. The authors embedded

the discovered URLs of scripts and Flash objects in a browser extension, which can be used as a blacklist-based approach for finding the previously recognized fingerprinting attempts. However, the FPdetective extension cannot detect newly added fingerprinting scripts or Flash objects.

To compare FPGuard with FPdetective, we first included the blacklisted URLs that exist in the FPdetective extension in FPGuard. After visiting each website from our dataset, FPGuard identified 36 websites that contain URLs from the blacklist. This means that these 36 websites contain methods for fingerprinting (i.e., using *JavaScript-based Font Detection* and *Flash-based Fingerprinting* methods) and are detected by FPdetective. It should be mentioned that FPdetective identified 9 websites (out of 36 websites) as fingerprinter but we did not find any fingerprinting-related activities within these websites (i.e., FPGuard did not identify them as fingerprinter). We manually investigated the URLs that are identified by FPdetective as sources of fingerprinting and did not find any fingerprinting attempts which match with fingerprinting patterns of *JavaScript-based Font Detection* and *Flash-based Fingerprinting* methods. It should be noted that each of the identified websites (i.e., 9 websites) include a Flash file[1,2] which FPdetective identified as suspicious but our analysis showed that the file contains methods for storing `evercookies` [13] or Flash cookies on the user's system (i.e., there are neither any attempts for collecting system-related information nor for enumerating system fonts).

In a recent study, Acar et al. [17] conducted a large-scale study on the prevalence of the canvas fingerprinting method among Alexa's top 100K websites. The authors modified a Chromium browser to record the accesses (writes and reads) to

[1] `http://bbcdn-bbnaut.ibillboard.com/server-static-files/bbnaut.swf`
[2] `https://aa.online-metrix.net/fpc.swf?session=f556fecf8[...]3136`

canvas elements. They combined manual and dynamic analysis to identify fingerprinting attempts and found that more than 5% of the visited websites leveraged this type of fingerprinting method (i.e., canvas fingerprinting). In particular, the authors reported that most of the canvas fingerprinting attempts (95%) belong to a single fingerprinting provider (i.e., `AddThis` [22]). As noted before, `AddThis` stopped using canvas fingerprinting as of mid July, 2014 [64].

6.3 Anti-Fingerprinting Tools

The existing tools with anti-fingerprinting solutions are: `Tor` [35], `Privactor` [34], `Firegloves` [36], and `ExtensionCanvasFingerprintBlock` [68].

`Tor` is a browser used to connect to the `Tor` network. The `Tor` network is an anonymous network for hiding the user's online activities and traffic by encrypting the data and passing it through a number of nodes (or relays). Relays — individuals or organizations around the world — service `Tor` users. In addition to providing such an anonymous network, `Tor` is equipped with features for combating browser fingerprinting. The reason is that browser fingerprinting works well on the anonymous network and could effectively be used for identification. As explained by Perry et al. [69], `Tor` makes the browsers' fingerprints identical, meaning that the browser's properties are fixed and identical for all users. In addition, it disables all browser plugins except the Flash plugin due to its high market penetration. For the Flash files, it offers a "click-to-play" option in which the user has to authorize a Flash file to be executed. Similarly, canvas elements in `Tor` should be authorized by users. Otherwise, the canvas elements are disabled and represent empty white images. Moreover, `Tor` limits the number of fonts that a website can load. In case a document exceeds

the limit, `Tor` does not allow the website to load more fonts. It should be noted that disabling plugins or returning empty contents for canvas elements reduce the functionality of the browser.

`Privactor` is an enhancement to the private-mode of browsers proposed by Nikiforakis et al. [34]. In `Privactor`, the authors employed randomization policies on two properties of the browser that are used for fingerprinting. The idea behind this tool is to randomly change the browser's fingerprint upon each visit to a website. Thus, every visit of a user with the same browser is considered as a new user visiting the website. However, they only focused on the plugin list of the browser and the offset properties of HTML elements to prevent *JavaScript Objects Fingerprinting* and the *JavaScript-based Font Detection* attempts, receptively. To this end, `Privactor` randomly hides a number of plugins from the browser's plugin list. Moreover, it adds noises to the offset properties of HTML elements, when the properties are looked up more than 50 times.

Our analysis of Alexa's top 1K websites shows that they call offset properties of HTML elements for 64,988 times (65 calls on average). These calls are mainly performed for design purposes. Therefore, adding noises to these properties might affect the design of a website. The authors evaluated the effect of their approach by comparing the screen shot images of webpages before and after employing their approach. They reported the effect of their approach on a screen shot image of a website as 0.7% on average. Additionally, the authors only focused on the `offsetWidth` and `offsetHeight` properties of the HTML elements, while the `getBoundingClientRect` method can also be used for getting the offset properties of the HTML elements. As noted before, `Privactor` hides a number of plugins and thus causes functionality loss

of the browser (e.g., `Facebook` does not load Flash resources if the Flash plugin is hidden).

`Firegloves` [36] is a proof-of-concept Mozilla Firefox extension which was developed for research purposes. `Firegloves` returns random values for the browser properties (e.g., screen resolution). It also disables all plugins and MIME types on the browser. To prevent JavaScript-based font detection attempts, `Firegloves` limits the number of available fonts on each tab and also returns random values for the `offsetWidth` and `offsetHeight` properties of the HTML elements. However, as we explained earlier, it is also possible to get the width and height of the HTML elements using the `width` and `height` properties of the `getBoundingClientRect` method.

`ExtensionCanvasFingerprintBlock` [68] is a recently developed browser extension for protecting against canvas fingerprinting by returning an empty image for canvas elements that are accessed programmatically. As mentioned in the previous chapter (see Section 5.2.3), we found that 191 canvas elements are accessed programmatically (first-level suspicious) but only 85 of them employed fingerprinting (second-level suspicious). Therefore, returning an empty image for all suspicious canvas elements might cause functionality loss.

It is noteworthy that `Tor` and `Firegloves` block both normal and fingerprinting websites as they are not equipped with a detection mechanism. However, FPGuard first detects fingerprinting attempts and then applies a suitable prevention technique. Moreover, as mentioned earlier, FPGuard returns neither fixed nor empty values for the browser properties. Conversely, `Tor` and `Firegloves` return empty values for the plugin list. In our experiments, a property of the browser's plugin list is looked up by nearly 75.2% of Alexa's top 10K websites. Thus, returning an empty value

for the plugins list can cause functionality loss. While `Tor` disables all plugins in the browser, `Privactor` employs a randomization-based solution by randomly hiding some plugins from the browser's plugin list upon every visit of a user to a webpage. Hiding a number of plugins still causes functionality loss of the browser. In contrast, FPGuard adds non-existing plugins with real names and properties to the browser's plugins list and permutes the order of the plugins upon every visit.

`Tor`, `Firegloves`, and `ExtensionCanvasFingerprintBlock` return empty contents for all canvas elements. However, FPGuard keeps the browser's functionality by first identifying suspicious canvas elements and then adding minor noises to their contents. For preventing *JavaScript-based Font Detection* attempts, both `Tor` and `Firegloves` limit the number of system fonts for all websites. `Privactor` adds random noises to the offset properties of HTML elements after they are looked up more than 50 times. As mentioned earlier, this approach might interfere with the design of certain websites. FPGuard, on the other hand, first detects *JavaScript-based Font Detection* attempts and then after a predefined threshold (50 fonts), it randomly announces the available fonts as unavailable. In this way, it assures the randomness of browser's fingerprint on multiple visits. Moreover, it does not interfere with the webpage's design since it has no effect on the offset properties of the HTML elements and it only randomly changes the available fonts after a predefined threshold.

6.4 Summary

In this chapter, we presented the existing research work for performing, detecting, and preventing fingerprinting methods. We first discussed various approaches that

are proposed for browser identification using fingerprinting. In addition to providing detailed information of their workflow, we reported their accuracy in identifying a browser instance. Then, we described two large-scale studies on detecting fingerprinting methods on the Web. Moreover, we compared FPGuard with an existing blacklist-based browser extension in terms of the number of identified fingerprinting attempts. Finally, we presented the existing anti-fingerprinting tools and compared FPGuard with them. Our analysis shows that FPGuard could effectively identify fingerprinting attempts. Furthermore, it could prevent fingerprinting without causing functionality loss of the browser.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The Web has become a popular medium for delivering a variety of static, dynamic, and interactive contents such as images, audios, and videos. Websites or Web applications delivering such contents no longer depend only on their own contents. Instead, they largely depend on third-party services (e.g., social networks). This provides added values for both parties. For the website provider, it reduces management and maintenance costs and for the third party, it provides monetization through embedding services such as advertisement. As a result, most third-party service providers deploy techniques (e.g., browser fingerpriting) to monitor interactions and collect consumption patterns or behaviors of the Web users (without the users' consent) to personalize services accordingly. This raises privacy concerns for the Web users who prefer to remain anonymous to unknown third-party advertising companies by not sharing their browsers' fingerprint. It should be noted that due to the stateless nature of browser fingerprinting, Web users are unaware of being fingerprinted and

followed across multiple websites. In addition, a careful or attentive user might observe the fingerprinting attempts, but the user might not be able to fully control or stop these attempts. Despite the necessity of effectively preventing fingerprinting attempts, there are a few tools which mitigate fingerprinting attempts but often cause functionality loss of the browser.

To address the abovementioned problem, in this thesis, we first reverse engineered the workflow of the widely used fingerprinting methods (i.e., *JavaScript Objects Fingerprinting*, *JavaScript-based Font Detection*, *Canvas Fingerprinting*, and *Flash-based Fingerprinting*) on the Web to extract metrics related to each method. For this purpose, we developed `Fybrid` — a hybrid browser fingerprinting tool — which implements the four most prevalent fingerprinting methods on the Web. Using `Fybrid`, we collected 1,523 browser fingerprints and evaluated the importance of browser properties and fingerprinting methods in browser identification using fingerprinting. Our analysis showed that the plugin list of browsers plays a key role in fingerprinting and *JavaScript Objects Fingerprinting* is the most effective fingerprinting method compared to other methods. We also integrated `Fybrid` with a social networking service (i.e., Facebook) to collect fingerprints and tried to link the browser fingerprint with a user such that the personal information of Web users can be revealed through fingerprinting. In fact, we showed that it is possible to associate the personal information of the users with their browser fingerprints. In this way, it is possible to perform individual identification rather than browser identification, an issue which raises serious privacy concerns.

Second, we implemented `FPGuard` — an approach to detect and prevent fingerprinters at runtime. For runtime detection of browser fingerprinting, we proposed

a number of algorithms to analyze the extracted metrics. As for fingerprinting prevention, we combined randomization and filtering techniques to change the browser fingerprint in every visit the user makes to a webpage. In particular, we applied suitable randomization policies (4 policies) and filtering techniques to keep the browser functionality and combat fingerprinting attempts.

We evaluated `FPGuard`'s detection approach on Alexa's top 10,000 websites. We identified *Canvas Fingerprinting* as the most prevalent type of fingerprinting method on the Web (85 websites out of 10,000 websites). In addition, we compared `FPGuard` with a blacklist-based browser extension (i.e., `FPdetective`) in terms of the number of identified fingerprinting attempts. `FPGuard` was able to correctly identify the websites in the `FPdetective`'s blacklist as fingerprinters except for few cases (9 websites). More specifically, `FPdetective` identified 9 websites as fingerprinters exploiting the *Flash-based Fingerprinting* method, but we could not find any fingerprinting-related attempts in the source code of the Flash files that were identified as evidences of fingerprinting. In fact, we observed that Flash files contain methods for storing cookies (e.g., `evercookie` and Flash cookies) on the user's system instead of methods for enumerating system fonts or collecting system-related information.

We also evaluated `FPGuard` against four fingerprinting providers. `FPGuard` successfully changed the browser's fingerprint upon each visit to the fingerprinters. This is to say, the fingerprinters were not able to uniquely identify the browser between multiple visits. We also compared `FPGuard` with the existing anti-fingerprinting tools and pointed out the strengths and weaknesses of each tool. We observed that the existing solutions cause functionality loss of the browser for both normal and fingerprinter webpages. However, `FPGuard` frustrates fingerprinting attempts and at the

same time keeps the browser functionality by first detecting fingerprinting attempts and then using particular randomization policies and filtering techniques with respect to each attempt. We also analyzed the existing fingerprinting algorithms used in popular fingerprinting providers. Our analysis showed that the existing fingerprinting algorithms are highly dependent on the browser properties and slight changes to the properties (e.g., updating the browser version) can frustrate their attempts for browser identification.

Finally, we evaluated `FPGuard` from the overhead point of view. We separately measured the overhead for the `FPGuard` instrumented Chromium browser and the browser extension and observed that their imposed overheads are negligible.

7.2 Limitations and Future Work

As for limitations, we currently assign a suspiciousness level to each fingerprinting method to show the possibility of performing fingerprinting attempts by a website. As a result, we cannot precisely determine whether a website is leveraging any of the fingerprinting methods or not. To determine if a website is definitely exploiting fingerprinting methods, we need to analyze the information flow of the fingerprinting methods to check whether the collected information is transfered to a remote server or not. `FPGuard` currently does not employ information flow analysis, instead it assigns a suspiciousness level with each fingerprinting method. A higher level of suspiciousness shows a higher probability of performing fingerprinting attempts. We plan to employ information flow analysis to precisely detect fingerprinting attempts in future. Moreover, due to our implementation approach in the FPGuard's browser extension, fingerprinters are able to check if the browser extension is installed on the

browser or not. However, they may not be able to disable FPGuard. In future, as we keep collecting fingerprinting data, we intend to develop robust algorithms and build analytic models that can adjust the thresholds (see Section 5.2) adaptively. We also plan to extend `FPGuard` and integrate it with a browser instance (e.g., Chromium) and make it available as an enhancement to the private mode of the Chromium browser.

Bibliography

[1] C. Dwyer, "Behavioral Targeting: A Case Study of Consumer Tracking on levis.com," 2009.

[2] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and Defending Against Third-party Tracking on the Web," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2012.

[3] C. J. Hoofnagle, J. M. Urban, and S. Li, "Privacy and modern advertising: Most us internet users want'do not track'to stop collection of data about their online activities," in *Amsterdam Privacy Conference*, 2012.

[4] A. M. McDonald and L. F. Cranor, "Beliefs and Behaviors: Internet Users? Understanding of Behavioral Advertising," in *Proceedings of the 2010 Research Conference on Communication, Information and Internet Policy*, 2010.

[5] J. Turow, J. King, C. J. Hoofnagle, A. Bleakley, and M. Hennessy, "Americans reject tailored advertising and three activities that enable it," *Available at SSRN 1478214*, 2009.

[6] T. (2009), "2009 Study: Consumer Attitudes About Behavioral Targeting." `http://www.scribd.com/doc/18050719/TRUSTe-TNS-Study-Consumer-Attitudes-about-Behavioral-Targeting`, 2014.

[7] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre, "User tracking on the web via cross-browser fingerprinting," in *Information Security Technology for Applications*, pp. 31–46, Springer, 2012.

[8] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle, "Flash Cookies and Privacy," in *AAAI Spring Symposium: Intelligent Information Privacy Management*, 2010.

[9] F. Roesner, "Detecung and defending against third–party tracking on the web," 2012.

[10] Ghostery, "Ghostery." `https://www.ghostery.com/en/`, 2014.

[11] A. Plus, "Adblock Plus." `https://adblockplus.org/en/chrome`, 2014.

[12] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh, "An Analysis of Private Browsing Modes in Modern Browsers," in *USENIX security symposium*, pp. 79–94, 2010.

[13] S. Kamkar, "evercookie–never forget," *New York Times*, 2010.

[14] P. Eckersley, "How Unique is Your Web Browser?," in *Privacy Enhancing Technologies*, pp. 1–18, Springer, 2010.

[15] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, "A practical attack to de-anonymize social network users," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 223–238, IEEE, 2010.

[16] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "FPDetective: Dusting the Web for Fingerprinters," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 1129–1140, ACM, 2013.

[17] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The Web Never Forgets: Persistent Tracking Mechanisms in the Wild,"

[18] ThreatMatrix, "Device Fingerprinting and Fraud Protection Whitepaper." `http://www.cse.iitd.ernet.in/~siy107537/sil765/readings/Device-Fingerprinting-and-Online-Fraud-Protection-Whitepaper.pdf`, 2014.

[19] Iovation, "Solving Online Credit Fraud Using Device Identification and Reputation." `https://meritalk.com/uploads_legacy/whitepapers/Solving-Online-Credit-Fraud.pdf`, 2014.

[20] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking Internet Malware," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 443–457, IEEE, 2012.

[21] bluecava, "BlueCava." `http://bluecava.com/opt-out`, 2014.

[22] AddThis, "Addthis - get likes, get shares, get followers." `http://www.addthis.com/privacy/opt-out`, 2014.

[23] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 541–555, IEEE, 2013.

[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," in *ACM Sigplan Notices*, vol. 45, pp. 1–12, ACM, 2010.

[25] K. Mowery and H. Shacham, "Pixel Perfect: Fingerprinting Canvas in HTML5," *Proceedings of W2SP*, 2012.

[26] L. Olejnik, C. Castelluccia, A. Janc, *et al.*, "Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns," in *5th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2012)*, 2012.

[27] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting Information in JavaScript Implementations," *Proceedings of W2SP*, 2011.

[28] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. C. Wien, "Fast and Reliable Browser Identification with Javascript Engine Fingerprinting," in *Web 2.0 Workshop on Security and Privacy (W2SP)*, vol. 5, 2013.

[29] Adobe, "Adobe Flash Runtimes Statistics." `http://www.adobe.com/products/flashruntimes/statistics.html`, 2014.

[30] G. Cluley, "How to Turn off Java in Your Browser - And Why You Should Do It Now." http://nakedsecurity.sophos.com/2012/08/30/how-turn-off-java-browser/, 2012.

[31] NoScript, "NoScript." http://noscript.net/, 2014.

[32] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi, "Host Fingerprinting and Tracking on the Web: Privacy and Security Implications," in *NDSS*, 2012.

[33] G. Acar, "Obfuscation for and Against Device Fingerprinting Position Paper for Symposium on Obfuscation," February 2014.

[34] N. Nikiforakis, W. Joosen, and B. Livshits, "PriVaricator: Deceiving Fingerprinters with Little White Lies," 2014.

[35] M. Perry, E. Clark, and S. Murdoch, "The Design and Implementation of the Tor Browser[draft]." https://www.torproject.org/projects/torbrowser/design/, July 2014.

[36] K. Boda., "Firegloves." http://fingerprint.pet-portal.eu/?menu=6, 2014.

[37] G. Wilson, "User-agent switcher for chrome." https://chrome.google.com/webstore/detail/user-agent-switcher-for-c/djflhoibgkdhkhhcedjiklpkjnoahfmg, 2014.

[38] J. R. Mayer, "Any Person... a Aamphleteer?: Internet Anonymity in the Age of Web 2.0," *Undergraduate Senior Thesis, Princeton University*, 2009.

[39] AdobeFlash, "Capabilities - as3." http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/system/Capabilities.html\#propertySummary, 2014.

[40] MicroWorkers, "MicroWorkers." https://microworkers.com/index.php, 2014.

[41] C. E. Shannon, "Prediction and Entropy of Printed English," *Bell system technical journal*, vol. 30, no. 1, pp. 50–64, 1951.

[42] M. Dash, K. Choi, P. Scheuermann, and H. Liu, "Feature Selection for Clustering - A Filter Solution," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, (Washington, DC, USA), pp. 115–, IEEE Computer Society, 2002.

[43] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *The Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[44] ThreatMatrix, "ThreatMatrix - Building Trust on the Internet." http://www.threatmetrix.com/, 2014.

[45] AddThis, "AddThis - Terms of Service, month = july, year = 2014, howpublished = "http://www.addthis.com/tos."

[46] E. Bakshy, I. Rosenn, C. Marlow, and L. Adamic, "The Role of Social Networks in Information Diffusion," in *Proceedings of the 21st international conference on World Wide Web*, pp. 519–528, ACM, 2012.

[47] FACEBOOK, "FACEBOOK INC. 2012 Annual Report." http://investor.fb.com/common/download/download.cfm?companyid=AMDA-NJ5DZ&fileid=

658233&filekey=46826077-D2FD-4E84-9BBE-C3F844B547A0&filename=FB_2012_10K.pdf, 2014.

[48] C. Patsakis, A. Asthenidis, and A. Chatzidimitriou, "Social networks as an attack platform: Facebook case study," in *Networks, 2009. ICN'09. Eighth International Conference on*, pp. 245–247, IEEE, 2009.

[49] Facebook, "Facebook Login Overview." https://developers.facebook.com/docs/facebook-login/overview/v2.1, 2014.

[50] Facebook, "Facebook Canvas." https://developers.facebook.com/docs/games/canvas, 2014.

[51] G. A. F. Emily Steel, "Facebook in Privacy Breach." http://online.wsj.com/news/articles/SB10001424052702304772804575558484075236968, 2014.

[52] B. Krishnamurthy and C. E. Wills, "On the Leakage of Personally Identifiable Information via Online Social Networks," in *Proceedings of the 2nd ACM workshop on Online social networks*, pp. 7–12, ACM, 2009.

[53] L. Wood, V. Apparao, L. Cable, M. Champion, M. Davis, J. Kesselman, T. Pixley, J. Robie, P. Sharpe, and C. Wilson, "Document Object Model (DOM) Level 1 Specification," *w3C recommendation*, vol. 1, 1998.

[54] M. D. Network, "Object DefineProperty." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty, 2014.

[55] M. D. Network, "Object Prototype DefineGetter." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/__defineGetter__, 2014.

[56] M. D. Network, "Mutation Observer." https://developer.mozilla.org/en/docs/Web/API/MutationObserver, 2014.

[57] as3commons, "As3 Commons Bytecode." http://www.as3commons.org/as3-commons-bytecode/index.html, 2014.

[58] imacros, "iMacros for Chrome." https://chrome.google.com/webstore/detail/imacros-for-chrome/cplklnmnlbnpmjogncfgfijoopmnlemp?hl=en, 2014.

[59] CoinBase, "CoinBase - An International Digital Wallet." https://coinbase.com/, 2014.

[60] BrowserLeaks.com, "BrowserLeaks." https://www.browserleaks.com/, 2014.

[61] Valve, "Fingerprintjs." https://github.com/Valve/fingerprintjs, 2014.

[62] Mozilla, "Dromaeo - javascript performance test suit." http://dromaeo.com/, 2014.

[63] MaxMind, "Maxmind - ip geolocation and online fraud prevention." https://www.maxmind.com/en/home, 2014.

[64] AddThis, "The facts about our use of a canvas element in our recent r&d test." http://www.addthis.com/blog/2014/07/23/

the-facts-about-our-use-of-a-canvas-element-in-our-recent-rd-test/\#.VBdxSfldU8k, 2014.

[65] AudienceScience, "Audiencescience - enterprise advertising management system." http://www.audiencescience.com/, 2014.

[66] Bugzilla@Mozilla, "Mozilla - Bug 57351." https://bugzilla.mozilla.org/show_bug.cgi?id=57351, 2014.

[67] cbsnews, "Measuring Time Spent On A Web Page." http://www.cbsnews.com/news/measuring-time-spent-on-a-web-page/, 2014.

[68] Appodrome, "Canvasfingerprintblock." https://chrome.google.com/webstore/detail/canvasfingerprintblock/ipmjngkmngdcdpmgmiebdmfbkcecdndc, 2014.

[69] S. M. Mike Perry, Erinn Clark, "Tor: Cross-Origin Fingerprinting Unlinkability." https://www.torproject.org/projects/torbrowser/design/#fingerprinting-linkability, 2014.

Appendix A

Entropy Values - All Properties

A.1 JavaScript Objects

Table A.1 shows the calculated entropy values for the system properties that are accessible through the `navigator` and `screen` JavaScript objects properties for 1,523 different browsers.

Table A.1: Entropy values of JavaScript objects' properties

navigator		**screen**	
Property Name	**Entropy Value (bits)**	**Property Name**	**Entropy Value (bits)**
plugins	9.97	resolution (width & height)	2.99
mimeTypes	6.55	colorDepth	0.60
userAgent	5.83	pixelDepth	0.64
appVersion	4.49	bufferDepth	0.19
geolocation	2.02	deviceXDPI	0.24
buildID	1.87	deviceYDPI	0.24
language	1.62	logicalXDPI	0.19
oscpu	1.52	logicalYDPI	0.19
doNotTrack	1.22	systemXDPI	0.20
productSub	1.14	systemYDPI	0.20
vendor	1.14	updateInterval	0.10
javaEnabled	0.66		
browserLanguage	0.34		
systemLanguage	0.29		
platform	0.24		
cpuClass	0.21		
product	0.16		
onLine	0.04		
appCodeName	0.00		
cookieEnabled	0.00		

A.2 Flash Plugin

Table A.2 shows the calculated entropy values for the system properties that are accessible through the Flash plugin for 1,523 different browsers.

Table A.2: Entropy values of the system properties accessible through the Flash plugin

Property Name	**Entropy Value (bits)**	**Property Name**	**Entropy Value (bits)**
Fonts	9.55	Mouse.cursor	0.06
screenResolution	2.48	flash.ui.Keyboard.hasVirtualKeyboard	0.06
version	2.22	Mouse.supportsCursor	0.06
os	1.89	Keyboard.physicalKeyboardType	0.06
manufacturer	1.19	hasTLS	0.03
supports64BitProcesses	0.59	supports32BitProcesses	0.03
language	0.48	avHardwareDisable	0.02
languages	0.48	hasAudio	0.02
System.vmVersion	0.31	hasEmbeddedVideo	0.02
hasIME	0.30	hasMP3	0.02
playerType	0.22	hasScreenBroadcast	0.02
hasAccessibility	0.14	hasScreenPlayback	0.02
touchscreenType	0.09	hasStreamingAudio	0.02
isDebugger	0.08	hasStreamingVideo	0.02
screenDPI	0.07	isEmbeddedInAcrobat	0.02
pixelAspectRatio	0.07	localFileReadDisable	0.02
hasAudioEncoder	0.06	maxLevelIDC	0.02
hasPrinting	0.06	screenColor	0.02
hasVideoEncoder	0.06	System.ime	0.02
cpuArchitecture	0.06	System.useCodePage	0.02

Appendix B

Fingerprinter URLs

B.1 JavaScript-based Font Detection

Table B.1 shows the URLs for some scripts that perform JavaScript-based Font Detection and are detected by FPGuard.

Table B.1: URLs of scripts performing JavaScript-based font detection

Script URL
http://www.junkmail.co.za/sbbi/?sbbpg=sbbShell&gprid=DT&sbbgs=[...]&ddl=10935
http://m1.thestaticvube.com/v/969.1008/web3/js/web-en.min.js
https://www.etsy.com/ (an inline script in the homepage)
http://ds.bluecava.com/v50/AL/BCALIF5.js
https://coinbase.com/assets/jquery-82d48e54e30bce171e48ec3f05683068.js
http://tags.mdotlabs.com/tracking.php?siteID=V3ei&customUserValue=

B.2 Canvas Fingerprinting

Table B.2 shows the URLs for some scripts that perform canvas fingerprinting and are recognized by FPGuard.

Table B.2: URLs of scripts performing canvas fingerprinting

Script URL
http://device.maxmind.com/js/device.js
http://az590556.vo.msecnd.net/app/ec[...]a9/client.js
http://www.groupon.com/layout/assets/[...]15b1e66a03.js
http://d1g3gvqfdsvkse.cloudfront.net/assets/featurekicker.js
http://www.pof.com/ (an inline script in the homepage)
http://js.revsci.net/gateway/gw.js?csid=C12310&auto=t
http://static.audienceinsights.net/t.js
http://ox-d.rantsports.com/w/1.0/jstag
http://dup.baidustatic.com/tpl/wh.js
http://tracker.supercart.dolka.ru/thankyou.admitad.pixel.js?t=1410551579764
http://static.yieldmo.com/ym.min.js?_=1410551726789
https://www.gstatic.com/swiffy/v6.0.2/runtime.js
http://src.kitcode.net/fp2.js
http://pd0.imp.revsci.net/baycon/bayconfp.js?version=1.07

B.3 Flash-based Fingerprinting

Table B.3 shows the URLs for some Flash files that are flagged as third-level suspicious by FPGuard.

Table B.3: URLs of third-level suspicious Flash files

Flash URL
http://securepaths.com/sp.swf
http://www.bluehost.com/media/shared/general/trackr.swf
http://sjs.sinajs.cn/blog7swf/fonts.swf
http://device.maxmind.com/flash/Device.swf
http://lookup.bluecava.com/flash/guids3.swf
http://mp.pianomedia.eu/bucket/novosense.swf
http://www.justhost.com/media/shared/general/trackr.swf
http://www.hostmonster.com/media/shared/general/trackr.swf
http://sync.graph.bluecava.com/flash/bc.swf
https://mpsnare.iesnare.com/stmgwb2.swf
https://www.kickstarter.com/assets/FontList.swf