A Large-Scale Analysis of Browser Fingerprinting via Chrome Instrumentation

Mohammadreza Ashouri
Institute of Computer Science - University of Potsdam
Potsdam, Germany
email: ashouri@uni-potsdam.de

Abstract—**In this work, we introduce FPTracker as a standalone, portable and practical browser that utilizes static and dynamic analysis to obtain concise results on a large set of websites. In contrast to the previous works, which rely on native code instrumentation that have low performance and high cost for monitoring each fingerprint Application programming interface (API), FPTracker is developed as an independent tool that does not need to interact with users' web browsers. In order to prove the usefulness of FPTracker, we have evaluated the top 10,000 European websites (according to Alexa.com) that comprise 1,393,426 links. We have chosen popular European websites to discern how these websites employ user tracking third parties concerning the EU General Data Protection Regulation (GDPR). Accordingly, we found that 117,012 links out of 1,393,426 use invisible user fingerprinting systems. For instance, one of the biggest European banks and a leading advertising website still fingerprint their visitors.**

Keywords - browser fingerprinting*; *privacy*; *security*; *web API*; *encryption.

I. INTRODUCTION

Browser fingerprinting is a user tracking technique which extracts a wide range of unique information about online users through web browser APIs. Numerous methodologies have been proposed in order to dispute with browser fingerprinting, often are based on static analysis techniques and predefined blacklists. However, despite recent advances in privacy enhancement in web browsers, fingerprinting systems are not only improving their accuracy but also trying to avoid being caught by anti-tracking systems.

The information obtained in this technique includes web browser software and version (e.g., Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge, Opera), operating system (Windows, Mac OS, Linux), screen resolution (mobile, tablet or desktop), established fonts, browser plugins and extensions, time zone, ad-blockers, language, and the hardware properties of users device.

These items may not be individually identifiable; for instance, millions of users may use the same version of Chrome browser. However, the combination of these pieces of information is enough to identify users uniquely. In other words, there is enough distinction among the aforementioned features so that only one in several thousand can have the same combination of blueprints, which is perfectly accurate for the fingerprinting objectives (e.g. advertising). Hence, a successful web fingerprinter relies on this slight variance between users devices in order to create and label online users by unique hash strings (unique fingerprint IDs).

When for the first time, Electronic Frontier Foundation (EFF) published research called "How Unique is Your Browser?" in May 2010, browser fingerprinting got widespread attention. Via analyzing over a million visits to their research website [36], they found that 83.6% of the browsers seen had a unique fingerprint; 94.2% between those with Java or Flash enabled. It also proved that the combination of its fingerprint algorithm had at least 18.1 bits of entropy, meaning that users had a 1 in 286,777 chance of receiving the same fingerprint hash string as another one. The interesting point is that they got this fingerprints only through 8 web browser features (e.g., fonts, plugins, timezone, supercookies, cookies enabled, user agent, Hypertext Transfer Protocol (HTTP) accept, screen resolution). However, due to the advancement in web browser software and privacy systems, some of these features are out of function (e.g. Flash). Therefore, it sounds that by elaborating of functional features of new browsers, it is possible to extract accurate browser fingerprint of the users which is a desirable target for advertising companies.

Moreover, browser fingerprinting is on a debated subject with privacy laws. Compared to more well-known tracking cookies, browser fingerprinting is complicated for users and browser extensions to contend: websites can do it without disclosure, and it is difficult to modify browsers. As cookies are more noticeable and more comfortable to tackle, corporations have been more moved to turn to trickier fingerprinting techniques. Companies also have to obey the law as well as the residents of the European Union. However, during this study, we found that still many European websites including online banks still use this technique regardless of the law.

Web browser fingerprinting methods have been changing based on the new features providing by the full-fledged web browsers as well as the recent development in anti-tracing and anti-fingerprinting systems (e.g. ad-blockers). Hence, in this work we introduce FPTracker, which analyzes the presence of advanced and encrypted fingerprinting scripts and third-parties in websites. FPTracker can also enhance the privacy level of web users via identifying and reporting the latest fingerprinting tricks on the web.

Accordingly, we built FPTracker based on similar technologies used in previous works, such as FP-STALKER [26], FPGaurd [18], and FPDetective [20]. However, it has several key differences that provide more accurate, thorough, and reliable analysis. Hence, the key contributions in our work are as follows:

1) **Minimal cost of monitoring new fingerprinting APIs**. While most of the previous works rely on native instrumentation code [20]–[22] (produce a high maintenance cost and a high cost-per-API monitored), in FPTracker the monitoring cost of new fingerprinting APIs is minimal. It means that fingerprinting metrics can be enabled or disabled by users without web browsers recompilation or rendering engines manipulation. This feature also permits users to conduct their customized analysis.
2) **Performing in-depth analysis**. FPTracker uses a a new combined method based on static and runtime analysis, which enables us to perform more in-depth analysis and get accurate results.
3) **Analyzing encrypted scripts**. Since the industrial fingerprinting libraries tries to utilize encryption methods to evade anti-fingerprint systems (e.g. ad-blockers), FPTracker analyzes encrypted scripts to recognize their actual intention via a light and innovative approach (this is explained in the next sections).
4) **Running independently**. Our approach works based on the instrumentation of Chrome driver [33]. Therefore, our proposed tool does not need to communicate through the installed user web browsers (e.g., Chrome, Firefox, Opera) and makes FPTracker standalone and agile in comparison with similar tools.
5) **Simulating user interactions**. In order to bypass anti-crawler mechanisms in industrial browser fingerprint third parties, FPTracker performs arbitrary user interactions on websites to deceive the anti-crawler systems and traces the hidden APIs that work by user interactions.
6) **Identifying new fingerprinting techniques**. In FPTracker we have specified a long set of distinct metrics almost based on the recent fingerprinting approaches (e.g., WebRTC, WebGL, etc.) . We also consider other common techniques, such as canvas fingerprinting and font enumeration. Hence, we have achieved concise and more reliable results in comparison with previous studies which only focus on one or a few metrics.
7) **Analyzing secondary web pages**. Many popular web sites are interested in tracking their visitors based on their search results or favorite items (which are typically located on secondary pages, such as searching or booking pages). Thus, unlike previous studies that only analyze homepages, FPTracker detects the presence of web tracking scripts in all website pages including homepages and secondary pages.

In the following section, we introduce Browser fingerprinting techniques. Next, in the methodology section, we present our approach, and in the implementation section, we describe the details of FPTracker implementation. In the experimental results section, we express our evaluation results. Finally, in the related works section, we will compare our proposed system with similar works, and we represent the advantages of FPTracker in comparison with the previous works.

II. BACKGROUND

Web-Based fingerprinting requires certain types of information which are collected from user's systems. A client's browser requests such information. Executed web scripts like JavaScript which are placed on different web pages determine the type of collected information. The operations of client-side fingerprinting scripts are done based on HTTP header fields. It is one of the primary data transfer mechanism in many popular browsers. Unparalleled retrievement of data from web servers is one of the critical features of client-side fingerprinting which gives the web trackers, such as advertisement third parties, the ability to authorise online users based on their environment properties directly.

The computed fingerprint is a unique identifier which stores collected data on specific databases. The contents of these databases are defined after the completion of necessary processes in the browser. Eventually, the acquired results are sent to the web server.

In more advanced web-based fingerprinting schemes, the website provider is replaced by a third-party agent. Fingerprinting script is anonymously inserted in the website's main code only under permission of owners or business partners. In other words, such web-scripts are invisibly retrieved by visitors. The new scheme provides a more comprehensive set of information about users, and they are referring to websites. This technique might be useful only in cases where the website owner agrees on publishing advertising content on different parts of web pages.

A. Browser behavior

Every browser (type and version) has its unique characteristics which lead to unprecedented behavior. Such differences can be used for the creation of unique fingerprints.

JavaScript can gather information about screen properties of user devices. "Windows. Screen" object is the unique tool allocated to this process. Characters like depth, width, height and colour of devices screen can be gathered by such object which leads to a determination of the type of device (smartphone, tablet, two screened PCs). Lack of generalisation is a severe problem for this technique since different browsers use different run-time codes which interrupts the comparability of fingerprints. Some features of users systems like the number of CPUs or available free RAM can be useful for browser fingerprinting. The extraction of such information is not possible for all browsers, but the rest of the collected data can have critical applications for advertisers and web-owners.

B. Font detection/ enumeration

Any information in the OS is shown using fonts. These visual symbols are considered as unique features of devices. Also, installed software like MS-word or Acrobat creative suite use their special fonts, and on their front, the installation of customized fonts by users increase the diversity of available fonts in any system. The fingerprinting procedure might use these differences in order to attribute a unique signature to any device. The list of enumerated fonts is a meaningful basis for collecting a significant amount of data about the system's situation. The more comprehensive the font database, the higher the accuracy of fingerprinting based on font features.

C. Canvas fingerprinting

Canvas is an HTML5 API which is used to draw graphics and animations on a web page via scripting in JavaScript. However, apart from this, Canvas can be used as further entropy in web-browser's fingerprinting and applied for online fingerprinting targets. The approach is based on the fact that the corresponding canvas image may be rendered uniquely in distinct machines. This occurs for numerous causes. At the image format level, web browsers use diverse image processing engines and compression level; therefore, the final images can get distinctive checksum [25]. At the system level, operating systems have many fonts which use different algorithms for pixel rendering. Listing 1 represents an example of Canvas fingerprinting in JavaScript.

Listing 1. Sample code showing how Canvas fingerprinting works

```
function doFingerprinting(evt) {
    let canv = document.createElement("CANVAS");
    canv.height = canv.width = windSize;
    canv.style.height = canv.style.width = windSize + "em";
    let ctx = canv.getContext("2d");
    ctx.fillStyle = "black";

    let d = document.createElement("DIV");
    d.style.position = "fixed";
    d.style.left = d.style.top = "0px";
    d.style.zIndex = -1000;
    d.style.visibility = "hidden";
    document.body.appendChild(d);
...
```

D. WebGL fingerprinting

WebGL is a JavaScript API for rendering interactive graphics within any web browser without using extra plugins. Since GPU makes the rendering process of interactive graphics, the effect can be different in machines with different GPUs [24].

E. WebRTC tracking

WebRTC is an API with a complex set of protocols for establishing communications applications. It is created for applications such as voice and video chat (e.g., Facebook Messenger, OpenTokRTC, Google Hangouts) [32]. WebRTC offers TCP-like reliable and UDP-like unreliable data channels. Since WebRTC has recently become generally accessible and well established in web browsers, this API has become attractive for user fingerprinting purposes.

III. METHODOLOGY

Regarding our studies, almost all previous studies detect browser fingerprinting scripts in web pages based on web crawling and performing static analysis. This traditional approach is losing its functionality due to the recent advancements in fingerprinting techniques (such as using anti-crawling, anti-robot and code encryption methods to track real users instead of web bots). Moreover, we have observed that the popular browser fingerprinting libraries (e.g., Bluecava, fingerprint.min.js, client.js, google analytics, etc.) only fingerprint the actual users. In other words, these scripts do not run their fingerprinting functions until online users interact on web pages.

In this work, we attempt to support all of the practical features in FPTracker. Hence, we have created a static analyzer that co-operates with a runtime analyzer. We have also specified our fingerprint metrics based on the new techniques and recent updates in the popular user tracking third parties.

In this section, we explain the outline of our method, and in the next section, we present the details of the implementation.

Figure 1 represents an overview of FPTracker, a standalone web browser inside FPTracker is responsible for loading websites, enabling the analyzers to work on the interactive web pages and performing user interaction on the loaded web pages. After loading a website, our investigation will be conducted based on the following steps:

1) The static analyzer starts as a background thread in order to find any match between the enabled fingerprint APIs (Fingerprint Metrics) and the web page loaded objects (e.g., DOM, JavaScript, and CSS).
2) The runtime analyzer performs user interactions on the loaded web pages to trigger and trace the potential **hidden fingerprinting functions**.

A. Static Analyzer

When a webpage is entirely loaded on the internal browser, the Static Analyzer begins to parses the webpage as a set of DOM objects and separates the JavaScript, CSS, and HTML scripts. Then, the analyzer looks for any match between the tool enabled fingerprinting APIs and the extracted web APIs of the web page (Figure 2 shows the static analyzer in FPTracker).

We present some of the web metrics that we have specified for FPTracker in Table I. The purpose of the specification of these metrics is their usability in browser fingerprinting as well as their output values, which combines with each other to precisely identify online users. We obtained these metrics by investigating popular fingerprinting libraries and scripts. The Static Analyzer iterates this process to analyze all of the web pages and third party libraries of an examined website. After ending the process, FPTracker submits the identified fingerprints to the scoring module, which is responsible for providing the details of identified fingerprinting APIs in a website for the report database.

B. Runtime Analyzer

The Runtime Analyzer module is responsible for detecting and tracking suspicious JavaScript calls in websites, which

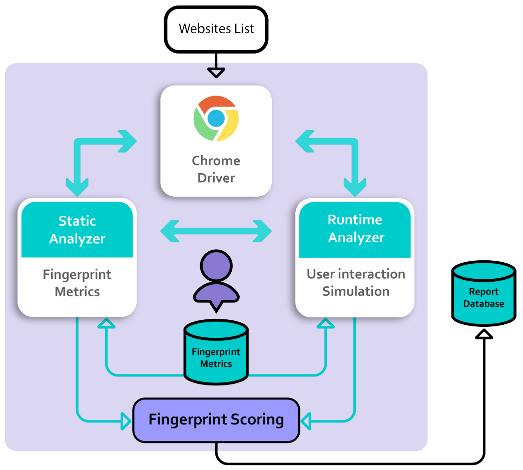

Figure. 1. Framework overview

TABLE I. SOME OF THE SPECIFIED JAVASCRIPT METRICS IN FPTracker WHICH CAN BE USED IN BROWSER FINGERPRINTING

Object Name	Usage
navigator.plugins	List of Plugins
navigator.userAgent	User agent header sent by the browser
navigator.language	Preferred language of the user
navigator.languages	Preferred languages of the user
navigator.cpuClass	CPU class of the user's OS
navigator.geolocation	Access to the location of device
navigator.platform	Platform of the browser
navigator.hardwareConcurrency	Number of logical processor cores available
navigator.deviceMemory	Device memory in gigabytes
navigator.doNotTrack	User do-not-track preference
navigator.appName	Official name of the browser
navigator.connection	Information about network connection
screen.colorDepth	Bit depth of color palette for displaying images
screen.height	Total height of users screen in pixel
screen.width	Total width of users screen in pixel
window.localStorage	Access a session storage object for the document origin
window.sessionStorage	Access a session storage object for the current origin
window.indexedDB	Store the data inside user browser
window.RTCPeerConnection	WebRTC connection
window.AudioContext	Audio-processing graph

either are encrypted or stimulated by user interactions. This module, which is shown in Figure 3, has two main threads:

The first thread is responsible for simulating user interactions on web pages (e.g., moving the mouse, clicking on buttons, etc.). In the meanwhile, the runtime analyzer traces hidden fingerprinting calls that serve for user interactions. For instance, in the case of **fingerprint.js**, which is a popular browser fingerprinting library, user uniques hash ID will be generated when an online user start to interacts on web pages (Listing 2 presents a sample hash function.).

Listing 2. Sample browser fingerprinting hash function

```
var my_hasher = new function(value, seed){ return value.length %
    seed; };
var fingerprint = new Fingerprint({hasher: my_hasher}).get();
```

The second thread of the **Runtime Analyzer** (RA) is responsible for intercepting and monitoring encrypted JavaScript codes inside of a web page. During our research, we have discerned that real-life user tracking third parties take advantage of web encryption algorithms to evade being caught and blocked by anti-tracking/ anti-fingerprinting systems (e.g. ad-blocking web extensions). Hence, it is inevitable that FPTracker must be able to disclose fingerprinting scripts even in encrypted formats. Therefore, the RA module analyzes the actual purposes of the encrypted scripts through monitoring web pages function calls, which is done via lightweight instrumentation of JavaScript engine. We describe the analysis of this technique in the next section.

IV. IMPLEMENTATION

In this section, we describe the implementation of FPTracker. As we mentioned earlier, our tool consists of three main modules: an internal instrumented browser, a static analyzer and a runtime analyzer. The entire platform is built on Python, **Selenium** [23], and **Chrome web driver** [33]. Selenium is a compact framework for inquiring web appli-

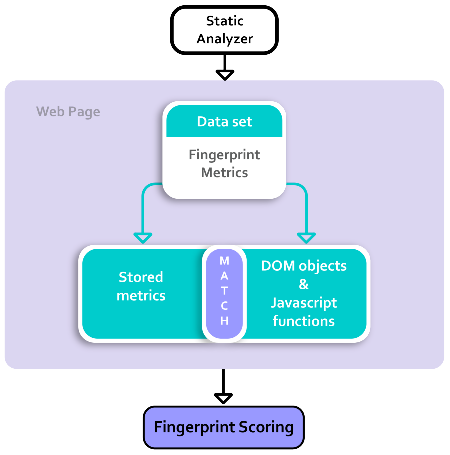

Figure. 2. Static Analyzer in FPTracker

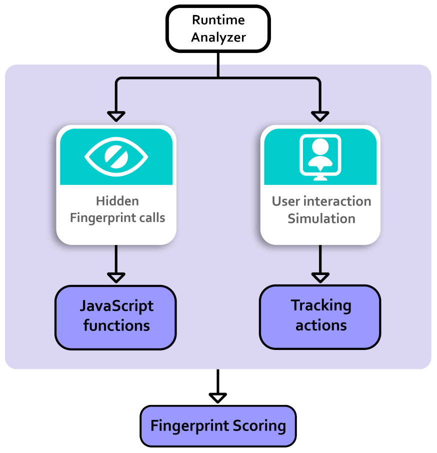

Figure. 3. Runtime Analyzer in FPTracker

cations. Selenium gives a playback tool for creating sound, browser-based regression automation tests. We also use **Beautifulsoup** [35] for parsing the contents of web pages. Beautiful Soup is a library that makes it simple to parse HTML and XML documents. Moreover, It produces a parse tree for parsed pages that can be used to obtain data from HTML, which is helpful for web scraping.

A. Internal Browser

The internal browser in FPTracker is responsible for presenting reality and support for Web APIs. We considered a variety of choices to drive measurements, i.e., to instruct the browser to visit a set of pages and to perform a set of actions on each. The two main categories to choose from are lightweight browsers like PhantomJS and full-fledged browsers like Chrome and Firefox. We decided to use Selenium which is a cross-platform web driver for Firefox, Chrome, and PhantomJS. By using our internal browser, all technologies that web users would have access to (e.g., HTML5 storage options, Canvas, WebGL, etc.) are also supported by FPTracker. The alternative, PhantomJS, does not support WebGL, HTML5 Audio and Video, CSS 3-D, and browser plugins making it challenging to run measurements on the use of these technologies. In retrospect, this has proved to be a sound choice. Without full support for new web technologies, we would not have been able to discover and measure the use of new fingerprinting techniques (such as webRTC and WebGL) or interacting on web pages.

In order to use Selenium, a real browser should be instrumented. Therefore, we have chosen Chrome Driver to be instrumented by Selenium because Chrome is the most popular web browser which takes 64% of the global browser market based on Browser Market Share Worldwide in December 2018 [34]. Hence, using the Chrome Driver make our internal web browser similar to an ordinary web user.

B. Static Analyzer Module

The Static Analyzer (SA) runs as a background thread, and it extracts web APIs of a given web page and looks for tracking scripts in the load web pages. All the specifications including fingerprint metrics are stored in a plain text file which is called **Fingerprint Metrics**. Users can update this file without recompilation of FPTracker or struggling with low-level data structures. Listing 3 shows how users can update and specify Fingerprint Metrics in a plain text file.

Listing 3. How users can update the fingerprint metrics in FPTracker

```
navigator.plugins : TRUE ;
navigator.hardwareConcurrency : FALSE ;
navigator.connection : TRUE ;
window.localStorage : FALSE ;
window.AudioContext : TRUE ;
```

The SA module examines the whole DOM objects of a given web page, and then it looks for any match between the stored metrics with parsed DOM objects and JavaScript functions. For instance, Figure 1 shows a Canvas API that is commonly used in fingerprinting libraries. In this sample, Listing 4 presents how the SA module detects canvas fingerprint functions in a given web page.

Listing 4. Sample code showing how FPTracker identifies Canvas fingerprinting

```
{
 var methods = []
 var canvasMethods = ['getImageData', 'getLineDash',
       'measureText','getContext','isPointInPath']
 canvasMethods.forEach(function (method) {
   var item = {
     type: 'Canvas',
     objName: 'CanvasRenderingContext2D',
     propName: method
   }

   methods.push(item)
 })
```

C. Runtime Analyzer Module

As we mentioned earlier, the Runtime Analyzer (RA) has two goals, simulating user interactions on websites and analyzing encrypted scripts inside of web pages.

Listing 5. Sample code showing how CrytoJS encrypt scripts in web pages

```
var encrypted = CryptoJS.AES.encrypt("Message", "Secret
      Passphrase");
var decrypted = CryptoJS.AES.decrypt(encrypted, "Secret
      Passphrase");
document.getElementById("example1").innerHTML = encrypted;
document.getElementById("example2").innerHTML = decrypted;
```

The main purpose of the JavaScript encryption is to make the understanding of the code logic difficult even though the functionality must be unchanged. Listing 5 shows an example of using CryptoJS library in JavaScript for code encryption. Moreover, Encrypting fingerprinting scripts can help malicious web scripts to twist static analyzers and avoid being detected by anti-fingerprinting tools. For instance, Listing 6 represents a simple ”Hello World” JavaScript code before the encryption.

Listing 6. Simple Hello World JavaScript code before the encryption

```
function hi() {
  console.log("Hello World!");
}
hi();
```

Listing 7 shows the simple ”Hello World” snippet code after the encryption.

Listing 7. Encrypted Hello World JavaScript code

```
var _0x5ec0=['log'];(function(_0x83958c,_0xc60544){var
      _0x45802b=function(_0x510b72){while(--_0x510b72)
{_0x83958c['push'](_0x83958c['shift']());}};
_0x45802b(++_0xc60544);}(_0x5ec0,0x123));var
      _0x551f=function(_0x361366,_0x1e7a76)
{_0x361366=_0x361366-0x0;
var _0x31149b=_0x5ec0[_0x361366];
return _0x31149b;};function
      hi(){console[_0x551f('0x0')]('Hello\x20World!');}hi();
```

Through the runtime analyzer component, we can recognize the malicious behavior of a encrypted code; however only if some conditions are true. If those conditions are never met, the malicious behavior of the code with performing runtime analyzing cannot be spotted. A condition could be a check if the actual environment of execution is not virtualized and if this condition is false, the code will not execute. This happened with the some of the fingerprinting libraries we have analyzed (e.g., FingerprintJS, ClientJS, etc.).

Listing 8. Example code presenting how the RA module matches inline onload events via regular expressions

```
def onLoad(evestr):
regex = r"""
(onload)\s*=(\s*\"\S*\")
"""
oncl = []

matches = re.finditer(regex, evestr, re.IGNORECASE | re.MULTILINE
      | re.VERBOSE)
for matchNum, match in enumerate(matches):
matchNum = matchNum + 1

for groupNum in range(0, len(match.groups())):
groupNum = groupNum + 1
if groupNum % 2 == 0:
oncl.append(match.group(groupNum))
for val in oncl:
stri = str(val)
stri = stri.replace("\"", "")
Eventscr.append(stri)
return oncl
```

1) Analyzing encrypted codes

There are several techniques for encrypting and decoding JavaScript (JS) codes, which is explained in [38]. However, we used our innovative approach for decoding JS scripts in web pages. Our decoding techniques is optimal and does push overhead to the analysis process.

Precisely, our method is based on the monitoring of function calls inside of web pages so that the Runtime Analyzer controls whether JavaScript function calls and related access properties are related to fingerprinting.

In order to identify function calls and related to fingerprinting purposes, we have already analyzed all of the standard browser fingerprinting scripts and libraries and extracted their common used objects and methods. Whenever an encrypted JavaScript code or library, attempt to make any function call associated with any of our blueprinted functions, The Runtime Analyzer labels that encrypted code as browser fingerprinting. Moreover, because almost all standard browser fingerprinting scripts trace online users based on a unique hash code, if these functions are made, the RA identifies them as fingerprinting suspicious scripts.

Listing 9. Setting Proxy for Navigator.plugins object

```
Object.defineProperty(window, "navigator", {
value: new Proxy(window.navigator, {
get: function(target, name) {
if (name == "plugins") {
console.log("Navigator.plugins is accessed");
}
```

We perform the function call tracing via lightweight instrumenting of JavaScript Engine in our internal web browser. The method of instrumentation in FPTracker is relatively easy. It works by handling a proxy listener for all JavaScript native objects in a given website. Listing 9 shows a sample of the instrumentation for navigator.plugins object, and Listing 10 shows a sample of encrypted code which performs Plugin enumeration navigator.

Listing 10. Sample encryption function which performs Plugin enumeration

```
var _0xe2d9=["\x6C\x65\x6E\x67\x74\x68",
"\x70\x6C\x75\x67\x69\x6E\x73",
"\x50\x6C\x75\x67\x69\x6E\x73\x3A\x20",
"\x6E\x61\x6D\x65","\x3C\x62\x72\x2F\x3E",
"\x6C\x6F\x67"];
var x=navigator[_0xe2d9[1]][_0xe2d9[0]];
txt= _0xe2d9[2];
for(var i=0;i< x;i++)
{
txt+= navigator[_0xe2d9[1]][i][_0xe2d9[3]]+ _0xe2d9[4];
}
```

Lastly, overhead due to the use of cryptographic algorithms in encrypted JavaScript codes in web pages can also help us for identifying browser fingerprinting libraries. For instance, by measuring the estimated overhead based on the version of our internal web browser, we can recognize the encryption function calls and the type of browser fingerprinting libraries. Although this method is not accurate, in some cases, such as large scale analysis, can speed up the investigation. Table II represents our experimental result performed by FPTracker for having the comparison of the performance between popular JavaScript libraries used for encryption in Chrome and Firefox. In this comparison, we have evaluated the performance of **SHA256** and **AES-CBC** algorithms on Chrome and Firefox web browsers. We also chose **asmcrypto**, **CryptoJS** and **SJCL** as the most widely used libraries for performing the encryption tests in JavaScript.

2) User interactions simulation

The RA module simulates user interaction on web pages via DOM objects information that has taken from the SA module (e.g., ID, Name, Object related events, etc.). The user interactions starts by scamping DOM objects in an arbitrary way. For instance, moving mouse on the web images, clicking on the form elements, entering data to the web form inputs. As an illustration in Listing 11, the RA module creates click events on three types of DOM elements to trigger JavaScript event handlers and potential hidden fingerprinting functions.

Listing 11. Three types of DOM elements

```
<button>tag
<input type = button>
<input type = submit>
```

Based on our study and experimental observation, the page load event is one of the most common places where trigger fingerprinting scripts. The scripts usually start to fingerprint visitors after some purposeful delays to avoid most of the anti-fingerprint browser extensions. These delays are made by **window.requestIdleCallback()** or **WindowOrWorkerGlobalScope.setTimeout()** JavaScript methods. The RA module also triggers document **onload** event and trace the potential fingerprinting scripts behind of this event. The RA module triggers events via **dispatchEvent** method. Listing 12 presents how the RA module triggers page onload event.

Listing 12. Sample code showing how the RA triggers the page onload event

```
var load_event = document.createEvent('Event');
load_event.initEvent('load', false, false);
window.dispatchEvent(load_event);
```

V. EXPERIMENTAL RESULTS

In this section, we summarize the results of our experiments with FPTracker. We begin by outlining our benchmark and experimental setup, describe some representative fingerprints found by our analysis and interpret the results.

In this paper, we have analyzed the top 10K European websites (according to Alexa.com). The total number of links that our tool actually reached is **1,393,426**. Our evaluation was conducted on a single Linux machine with an Intel Core i7-8500Y and 16GB memory from 7 February 2019 until end of April 2019.

A. Fingerprinting links

The total number of distinct fingerprinting URLs where FPTracker found is **117,012**, and 8% of the total reached links are browser fingerprinting while 92% are non-fingerprinting.

Moreover, **110,583** links have plain-text browser fingerprinting scripts and **6429** have encrypted scripts. In other words, **806** domains use browser fingerprinting scripts that are in plain-text, and **44** domains use the encrypted scripts (Figure 4 presents the percentages of fingerprinting links without/ with encryption).

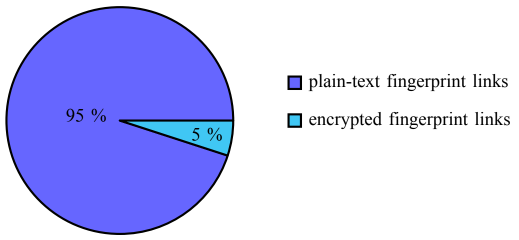

Figure. 4. Percentages of fingerprinting links without, with encryption

B. Third parties analysis

Since most of the fingerprinting scripts that FPTracker revealed are third-party libraries, it would be interesting to know about these libraries as well. Therefore, we first concentrated on identifying the most popular third parties' libraries in our benchmark. Accordingly, we determined how many distinct third-party libraries were called. In Figure 5, the 10 most used fingerprinting libraries in our evaluation are shown.

C. The adoption of the GDPR

It seems after the adoption of the General Data Protection Regulation (GDPR), websites warn users about using cookies. However, they are still interested in using browser fingerprinting scripts to collect more information about online visitors (especially in news websites which monetizing options are limited). For example, we found that **Fingerprint2** (a popular browser fingerprinting library) has been used at least in some popular crowd-sourcing websites such as **www.ebay-kleinanzeigen.de** (a popular online advertising service in German-speaking counties.).

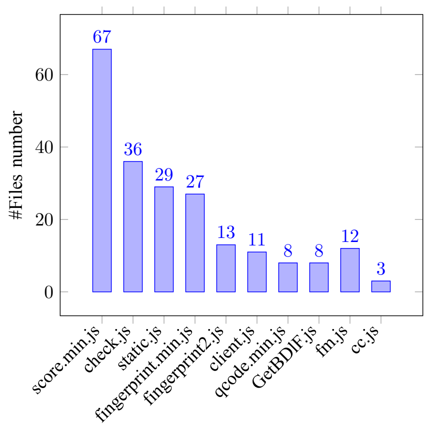

Figure. 5. The most used browser fingerprinting libraries

TABLE II. THE ANALYSIS OF THE OVERHEADS IN WEB ENCRYPTION

Cryptography Libraries	Chrome	Firefox
asmcrypto.js[1]	SHA256= 51 MiB/s AES-CBC: 47 MiB/s	SHA256: 144 MiB/s AES-CBC: 81 MiB/s
CryptoJS.js[2]	SHA256= 5.6 MiB/s AES-CBC: 3.6 MiB/s	SHA256: 28.8 MiB/s AES-CBC: 27 MiB/s
SJCL.js[3]	SHA256= 5.6 MiB/s AES-CBC: 2.35 MiB/s	SHA256: 7.2 MiB/s AES-CBC: 10.12 MiB/s

D. Fingerprinting in secondary pages

Since almost all of the previous works have studied the presence of browser fingerprinting only in the homepages of websites, the using of browser fingerprinting in the secondary pages of websites has been remained unexplored. During our experiment, we found 850 domains out of 1000 use browser fingerprinting. However, only 112 domains still use fingerprinting in the homepages, and the rest of the **688** websites fingerprint their visitors in the secondary pages. It seems that fingerprinting is becoming popular in the secondary pages of websites instead of the homepages. We presume there are two main grounds for this.

The first reason is the valuable information about users' interests (e.g. searched results) are more likely to be shown in the sub-pages (e.g., search pages, booked items, registration forms, etc.). For instance, websites such as online shopping or travel agencies can fingerprint their visitors based on their search results. The second reason can be because most of the previous large scale analysis focused only on the home pages for fingerprinting detection. Therefore, popular websites listed in Alexa.com have attempted to conceal their fingerprint scripts by placing them in the secondary pages.

TABLE III. THE MOST USED FINGERPRINT TECHNIQUES IN OUR BENCHMARK

Canvas	WebGL	WebRTC	Font Enumeration
6.7%	2.2%	0.87%	0.69%

E. Most used fingerprinting techniques

The most commonly used fingerprint techniques we found via FPTracker are Canvas, WebGL, WebRTC and Font enumeration fingerprinting (Table III). Also, the most called fingerprinting JavaScript objects in our analysis as shown in Figure 6. We have noticed that many of the domains in our benchmark, use a combination of fingerprinting techniques instead of one single method. For instance, in most domains where Canvas fingerprinting has been used, WebGL fingerprinting has existed as well.

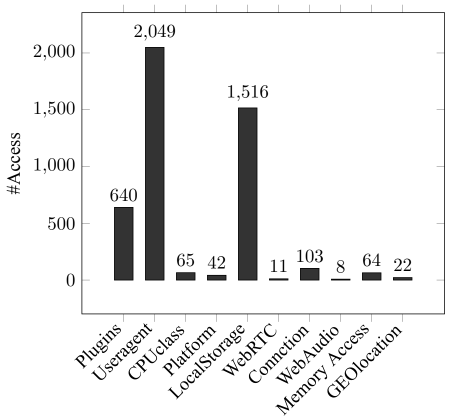

Figure. 6. The most fingerprinting access objects in the benchmark

F. Canvas fingerprinting

Our experiment recognized that **6.7%** domains out of 10,000 use Canvas fingerprinting. However, because Canvas methods can have typical applications (e.g. online gaming), we mark only those URLs as browser fingerprinting that either have any of identified fingerprinting third-party libraries or any hash computation functions related to Canvas fingerprinting. We also perceived that using Canvas fingerprinting cause about **30** milliseconds process overhead to the visitors' web browsers. Listing 13 represents an example method in FPTracker that recognizes Canvas fingerprinting. Especially, FPTracker classifies **fillText(), strokeText(), toDataURL(), getImageData()** as the most used function calls in the Canvas fingerprinting method.

Listing 13. Sample code showing how Canvas detection method in FPTracker works

```
var origToDataURL = HTMLCanvasElement.prototype.toDataURL;
HTMLCanvasElement.prototype.toDataURL = function() {
  var r = origToDataURL.apply(this, arguments);
  window.tourl++;
  return r;
  };
...
var origgetimagedata =
      CanvasRenderingContext2D.prototype.getImageData;
CanvasRenderingContext2D.prototype.getImageData = function() {
  var r = origgetimagedata.apply(this, arguments);
  window.getimagedata++;
  return r;
  };
```

G. Font detection results

Our experiment results demonstrate that **0.69%** of the domains use font enumeration fingerprinting. Enumerating installed fonts on the user web browser can have normal application for web design purposes, for instance, enumerating fewer than 20 fonts does not indicate any web browser fingerprinting activities. However, the most of the non-fingerprinting websites enumerate fewer than 16 fonts, but some of the fingerprinting websites request between 100 to 500 fonts. The maximum number of inquired fonts in our experiment was 500 fonts (the result of Font detection is shown in Figure 7).

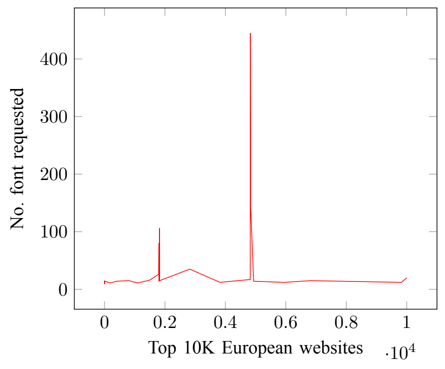

Figure. 7. Number of fonts requested for top 10K European websites. The most of the non-fingerprinting websites enumerate fewer than 16 fonts, but some of the fingerprinting websites request between 100 to 500 fonts.

We set 45 as the threshold for the number of fonts that a website can load — the threshold for the average number. Therefore, when a given website requests to load more than 45 fonts, we label its domain as sceptical about adopting font enumeration fingerprinting. FPTracker identified 16 websites that were loading more than 100 fonts. Also, during our analysis, we found that the time of font enumeration fingerprinting has more overhead in comparison with the other studied techniques introduced in this research (the average overhead for font enumeration method is approximately **90ms** on desktop Chrome).

H. WebGL fingerprinting

Based on our analysis **2.2%** of the domains use WebGL fingerprinting, using this method for fingerprinting is becoming popular since it is reliable for websites to track their users' device even if users surf the websites with several different browsers. In other words, this technique works based on user' hardware regardless of the attitudes of the web browsers. Generally, there are two methods for WebGL fingerprinting:

1) The entire of WebGL Browser Report Table will be retrieved and examined. In some cases, it is converted into a hash for faster analysis.
2) A hidden 3D image is rendered and hashed in the browser. Because the ending result depends on users' hardware which performs the computation, this method yields distinct values for different devices and drivers.

Among WebGL methods that FPTracker found in our benchmark, many of them draw a gradient with **drawArrays()** and converts it to Base64 string with toDataURL(). They also enumerate advanced WebGL extensions by **getSupportedExtensions()**. Another common extension, which interests trackers, is WEBGL_debug_renderer_info and provides information about users' graphics driver. FPTracker identifies this fingerprinting method via monitoring JavaScript function calls relate to these functions.

I. WebRTC fingerprinting

Tracing users through webRTC method is done through STUN servers. A STUN server allows users to find out their public IP address, the type of Network address translation (NAT) with a particular local port. Chrome, Firefox and Safari have implemented WebRTC which enables requests to STUN servers be made that will return the local and public IP addresses for users. In our analysis, 0.87% of the domains use WebRTC fingerprinting method and FPTracker detects the method (in Listing 14) which is commonly used for checking the compatibility of webRTC in the users' browser [10]. Moreover, we have observed that some websites use *iframe* tag to bypass webRTC blocking in client web browsers. This method is shown in Listing 15.

Listing 14. Checking the possibility to use WebRTC in Chrome, Firefox and Safari

```
var RTCPeerConnection = window.RTCPeerConnection
      || window.mozRTCPeerConnection
      || window.webkitRTCPeerConnection;
   var useWebKit = !!window.webkitRTCPeerConnection;
```

Listing 15. Using the iframe tag to bypass webRTC blocking in fingerprinting codes

```
   if(!RTCPeerConnection){
     //<iframe id="iframe" sandbox="allow-same-origin"
          style="display: none"></iframe>
      //<script>...getIPs called in here...
      //
      var win = iframe.contentWindow;
      RTCPeerConnection = win.RTCPeerConnection
          || win.mozRTCPeerConnection
          || win.webkitRTCPeerConnection;
      useWebKit = !!win.webkitRTCPeerConnection;
   }
```

J. Avoiding static analyzers by intentional delay

New fingerprinting libraries to avoid being detected by static analyzers do not fingerprint visitors right after page load event. Instead, they wait for a few seconds based on the visitor's web browser software (in average 30 seconds). Also, our results show that fingerprint libraries are removing font enumeration as a classic fingerprinting method due to its process overhead (between 80ms to 2000ms) that increases the chance of being detected by ad-blocker plugins (e.g., uBlock Origin, Adblock Plus, and AdBlock). The overhead of font enumeration is unusually high, especially in mobile phone web browsers (e.g. Firefox mobile edition).

TABLE IV. THE AVERAGE OVERHEAD RELATED TO COMMON FINGERPRINTING METHODS

Canvas	**WebGL**	**WebRTC**	**Font Enumeration**
10 ms	35 ms	30 ms	40 ms

VI. RELATED WORKS

Peter Eckersley is the founder of Panopticlick website. In his studies [2], three sources were used for the collection of features which are HTTP protocol, JavaScript and Flash API. Eckersley extracted 470161 fingerprints in total. There is a potential bias in collected data since these data represent those users that consider web privacy as an essential matter. Attributes such as list of installed fonts or list of installed plugins were the most dominant ones in Eckersley's data. In recent years, the capability of the JavaScript engine as an appropriate basis for identification of the client's browser type and version (even operating system) had been confirmed. Relevant benchmarks, JavaScript conformance test [6] and website rendering analysis [9], were all used for the creation of unique fingerprints. Some other techniques which are placed in the same category of fingerprinting but depend on their unique mechanism are Profiling [3], use of Evercookies [4] and History stealing [5]. Firegloves [30] is a Mozilla Firefox extension which replaces random values for the browser properties such as screen resolution. Firegloves restricts the number of available fonts on each browser tab and also returns random values for the offsetWidth and offsetHeight attributes of the HTML elements. However, it is possible to get the width and height of the HTML elements using the width and height attributes of the getBoundingClientRect method.

Another similar work is ExtensionCanvasFingerprint-Block [31] which is a browser extension for protecting users against canvas fingerprinting by returning empty images for canvas elements. Similarly, FP-Block [27] is an interesting prototype web browser extension which maintains the use and management of web identities. The web identities produced by FP-Block are distinct and logical (e.g. they are not generally randomized) that is achieved by implementing a Markov model for the generating of attribute values.

The above works diminish the fingerprinting surface by disabling browser functionalities because they do not have detection capability to perform before blocking, they block both regular and fingerprinting websites. This eventually will build annoying experience users due to the reduced functionality of the browser.

Metwalley, S. Traverso, and M. Mellia [28] proposed an algorithm based on web services that often exchange users identifiers as parameters in HTTP GET format. They look for HTTP GET requests and possible signs for the presence of user identifiers. It worth remarking that the assumption that web services transfer user identifiers through GET request is not practical because in many cases user identifiers are sent by POST queries. Another interesting work for detecting and analyzing browser fingerprinting is FPDetective [20]. They investigated the presence of fingerprinting on the home pages, using JavaScript-based font probing. FPDetective considers font-based fingerprinting as the main evidence for the existence of fingerprinting, and it proposes that fingerprinting is more popular than previously believed.

Another exciting research is [20], which analyzes the prevalence of canvas fingerprinting on the web. They discovered that Canvas fingerprinting is the most generally used tracking system which is existed in more than 5.5% of the top 100K Alexa websites. They found a total of 20 Canvas fingerprinting provider domains that are active in 5542 of the top 100K websites. FPGuard [18] is another browser fingerprinting detection tool which evaluated Alexa's top 10K focusing on Canvas and Flash-based Fingerprinting. They identified Canvas fingerprinting as the most common type of fingerprinting method on the Web. Another most recent similar works is FP-STALKER [26] which compares fingerprints to determine either they arise from the same browser instance, or they come from unknown cases. They constructed two options of FP-STALKER, a rule-based option that is fast, and a hybrid option that uses machine learning to improve precision, but it is prolonged. They identified languages and user-agent attributes as essential fingerprinting features.

Even though FPTracker similar to the previous works analyzes the presence of fingerprinting methods in websites, there are some remarkable distinctions between our work and these studies. For instance, these works have not analyzed encrypted scripts and libraries that potentially can be used for suspicious purposes (such as browser fingerprinting). Moreover, none of these previous works measures WebGL fingerprinting (which is a powerful and popular fingerprinting method these days). Moreover, during our studies we noticed that most of the state-of-the-art browser fingerprinting/ user tracking libraries use anti-crawler mechanisms to avoid Internet bots and spoofed fingerprints. The principle of these mechanisms is based on hiding their fingerprinting related functions unless online users interact with web pages. Thus, FPTracker is the first tool that traces and analyzes the professional browser fingerprinting scripts through reproducing user interactions on web pages (which help us to achieve more concrete results.). Furthermore, it seems that placing fingerprinting scripts in secondary web pages (instead of home pages) is becoming popular. This is because popular websites may attempt to evade their browser fingerprinting actions to be caught by anti browser fingerprinting systems and analysis. Websites may also desire to gather more information about their visitors based on their search results and favorite items which usually are placed on sub pages (like shopping pages). All of the works mentioned above only analyze homepages; however, FPTracker investigates all web pages including the homepage and sub pages of each website.

TABLE V. THE COMPARISON BETWEEN FPTracker AND SIMILAR WORKS

Tools	FPTracker	FP-STALKER	FPDetective	FPGaurd
JavaScript Objects	✓	✓	✓	✓
Canvas	✓	✓	-	✓
Font Enumeration	✓	✓	✓	✓
WebGL	✓	✓	-	-
WebRTC	✓	-	-	-
Encrypted Methods	✓	-	-	-
User Interaction	✓	-	-	-

Finally, in order to demonstrate the usefulness of our proposed approach, we compare FPTracker with other standard works. Accordingly, we have selected **FPDetective**, **FPGuard** and **FP-STALKER** as three well-known tools with similar functionality with our proposed tool. The result of the comparison (shown in Table 4) presents the advantages of our work over these tools.

VII. LIMITATIONS AND FUTURE WORKS

A. Server-side hash calculation

Even though in the past, browser fingerprinting were done mostly in the user web browser through Flash, JavaScript, HTML and CSS. However, our analysis indicates that the new browser fingerprinting libraries try to calculate the user profile hash IDs in the server side. Therefore, they use JavaScript agents that collect the value of the browser APIs and send them to server-side applications intimately.

This technique not only preserves the fingerprint libraries from being detected by anti-tracing systems, but also it protects these methods from reverse engineering and spoofing of fingerprints by Internet bots.

We have not supported this approach in this current version of FPTracker yet. However, we are working on potential solutions for tracking the potential fingerprinting data flows related to the third party servers as a future extensions of our work.

B. New applications of fingerprinting

Based on our experiments, browser fingerprinting methods are becoming popular in anti-phishing and fraud detection systems, especially for online financial and communication services.

Another advantageous application of this technique can be in detecting unusual activities on websites. For example, activities such has account harvesting, DOS attacks or identifying malicious users and Internet bots (automatically fill up web forms, or crawl the websites' information). For instance, during our studies we have noted that one of the European banks uses browser fingerprinting in one of its online services, we think that they might use it for the fraud detection purposes.

C. Client-server architecture for FPTracker

Our proposed tool is implemented as a standalone browser. Since the analyzing process can be time and memory consuming, we want to design a future extension with a client-server architecture. By doing this, a central server is responsible for patterning new fingerprint techniques, and clients use the detection functions to indicate whether a particular website comprises fingerprinting codes. Moreover, the results from clients can be used to retrain as a function for increasing accuracy. The important benefit of this plan can be detecting other type of threats on websites such as crypto-jacking, which is widespread presently.

D. Network sniffing

Another interesting extension for FPTracker would be shifting the analysis from web browser to network sniffer, to have data also on real navigation cases. Since the calls to the tracker are made from the browser, it would be essential to understand the output of the requests sent to the tracker domains, to recognize it in HTTP packets. It is worthwhile to know whether encrypted scripts can also be found in the network traffic and not only in the users' web browser.

VIII. CONCLUSION

In this research, we presented FPTracker as a standalone and practical tool for fingerprinting detection by the combination of static and runtime analysis on a large set of websites. To implement our tool, we used Selenium and Chrome driver as an instrumented standalone web browser. Moreover, our tool is able to identify encrypted fingerprints with a lightweight instrumentation approach. FPTracker also conducts user interactions on web pages to trace hidden fingerprinting scripts that used anti-crawler mechanisms. Finally, we evaluated FPTracker on the top 10K European websites including 1,393,426 links, and we have proved that standard techniques, such as font and plugin enumerations, are substituting with new technologies such as WebGL and WebRTC.

ACKNOWLEDGEMENT

The author would like to thank Prof. Dr. Christoph Kreitz for his helpful guidance and excellent comments during the review process, and the anonymous reviewers for sharing their valuable reviews.

References

[1] A. Hintz, *"Fingerprinting websites using traffic analysis."*, In Proceedings of the 2nd international conference on Privacy enhancing technologies, pp. 171-178. Springer, Berlin, Heidelberg, 2002.

[2] P. Eckersley, *"How Unique is Your Web Browser?"*, In International Symposium on Privacy Enhancing Technologies Symposium, pp. 1-18. Springer, Berlin, Heidelberg, 2010.

[3] T. Paulik, A.M. Foldes, G.G. Gulyas, *"Blogcrypt: Private Content Publishing on the Web."*, In 2010 Fourth International Conference on Emerging Security Information, Systems and Technologies, pp. 123-128. IEEE, 2010.

[4] Evercookie – virtually irrevocable persistent cookies, samy.pl/evercookie/, [retrieved: August, 2011]

[5] J. Grossman, I know where you have been, jeremiahgrossman.blogspot.com, [retrieved: August, 2011]

[6] P. Reschl, M. Mulazzani, M. Huber, and E. Weippl, *"Poster abstract: Efficient browser identification with javascript engine fingerprinting"*, In Proc. of Annual Computer Security Applications Conference (ACSAC). 2011.

[7] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, *"Fingerprinting information in javascript implementations."*, In Proceedings of W2SP, vol. 2, no. 11. 2011.

[8] K. Boda, Á.M. Földes, G.G. Gulyás, and S. Imre, *"User Tracking on the Web via Cross-Browser Fingerprinting."*, In Nordic conference on secure it systems, pp. 31-46. Springer, Berlin, Heidelberg, 2011.

[9] K. Mowery, and H. Shacham, *"Pixel perfect: Fingerprinting canvas in HTML5."*, Proceedings of W2SP (2012): 1-12.

[10] A. Reiter, and A. Marsalek, *"WebRTC: your privacy is at risk."*, In Proceedings of the Symposium on Applied Computing, pp. 664-669. ACM, 2017.
[11] T. Unger et al., *"Shpf: Enhancing http (s) session security with browser fingerprinting."*, In 2013 International Conference on Availability, Reliability and Security, pp. 255-261. IEEE, 2013.
[12] V. Bernardo and D. Domingos, *"Web-based Fingerprinting Techniques."*, In Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016), V. 4, SECRYPT, pp. 271-282, 2016.
[13] P. Laperdrix, W. Rudametkin, and B. Baudry, *"Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints."*, In 2016 IEEE Symposium on Security and Privacy (SP), pp. 878-894. IEEE, 2016.
[14] S. Luangmaneerote, E. Zaluska, and L. Carr, *"Inhibiting browser fingerprinting and tracking."*, In 2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security (ids), pp. 63-68. IEEE, 2017.
[15] T. Saito et al., *"Tor Fingerprinting: Tor Browser Can Mitigate Browser Fingerprinting?"*, In International Conference on Network-Based Information Systems, pp. 504-517. Springer, Cham, 2017.
[16] P. Laperdrix, *"Browser Fingerprinting: Exploring Device Diversity to Augment Authentification and Build Client-Side Countermeasures."*, PhD diss., Rennes, INSA, 2017.
[17] N.M. Al-Fannah and W. Li, *"Not all browsers are created equal: comparing web browser fingerprintability."*, In International Workshop on Security, pp. 105-120. Springer, Cham, 2017.
[18] A. FaizKhademi, M. Zulkernine, and L. Weldemariam, *"FPGuard: Detection and prevention of browser fingerprinting."*, In IFIP Annual Conference on Data and Applications Security and Privacy, pp. 293-308. Springer, Cham, 2015.
[19] N. Takei, T. Saito, K. Takasu, and T. Yamada, *"Web browser fingerprinting using only cascading style sheets."*, In 2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA), pp. 57-63. IEEE, 2015.
[20] G. Acar et al., *"FPDetective: dusting the web for fingerprinters."*, In Proceedings of the 2013 ACM SIGSAC conference on Computer communications security, pp. 1129-1140. ACM, 2013.
[21] C. Neasbitt et al., *"Webcapsule: Towards a lightweight forensic engine for web browsers."*, In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 133-145. ACM, 2015.
[22] K. Singh, A. Moshchuk, H.J. Wang, and W. Lee, *"On the incoherencies in web browser access control policies."*, In 2010 IEEE Symposium on Security and Privacy, pp. 463-478. IEEE, 2010.
[23] S. Avasarala, *"Selenium WebDriver practical guide."*, Packt Publishing Ltd, 2014.
[24] G. Nakibly, G. Shelef, and S. Yudilevich, *"Hardware fingerprinting using HTML5."*, arXiv preprint arXiv:1503.01408 (2015).
[25] G. Acar et al., *"The web never forgets: Persistent tracking mechanisms in the wild."*, In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 674-689. ACM, 2014.
[26] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, *"FP-STALKER: Tracking browser fingerprint evolutions."*, In 2018 IEEE Symposium on Security and Privacy (SP), pp. 728-741. IEEE, 2018.
[27] C.F. Torres, H. Jonker, and S. Mauw, *"FP-Block: usable web privacy by controlling browser fingerprinting."*, In European Symposium on Research in Computer Security, pp. 3-19. Springer, Cham, 2015.
[28] H. Metwalley, S. Traverso, M. Mellia, S. Miskovic, and M. Baldi, *"The online tracking horde: a view from passive measurements."*, In International Workshop on Traffic Monitoring and Analysis, pp. 111-125. Springer, Cham, 2015.
[29] S. Englehardt, and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis.", In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 1388-1401. ACM, 2016.
[30] K. Boda, fingerprint.pet-portal.eu/?menu=6, [retrieved: March, 2019]
[31] Appodrome, Canvasfingerprintblock, http://goo.gl/1ltNs4, [retrieved: December, 2018]
[32] D. Fifield, and M.G. Epner, *"Fingerprintability of WebRTC."* arXiv preprint arXiv:1605.08805 (2016).
[33] chromedriver, sites.google.com/a/chromium.org/chromedriver, [retrieved: December, 2018]
[34] statcounter, gs.statcounter.com/browser-market-share, [retrieved: December, 2018]
[35] beautifulsoup4, pypi.org/project/beautifulsoup4, [retrieved: February, 2019]
[36] panopticlick, panopticlick.eff.org, [retrieved: January, 2019]
[37] M. Juarez et al., "A critical evaluation of website fingerprinting attacks.", In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 263-274. ACM, 2014.
[38] M. AbdelKhalek, and A. Shosha, "Jsdes: An automated de-obfuscation system for malicious javascript.", In proceedings of the 12th International Conference on Availability, Reliability and Security, p. 80. ACM, 2017.