POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master's degree thesis

Machine Learning Approaches for Automatic Detection of Web Fingerprinting

Advisor

prof. Marco Mellia

Candidate

Valentino Rizzo

Company tutor

Stefano Traverso, PhD

Ermes Cyber Security

Academic Year 2017 – 2018

To my parents, Erica and Fulvio, and to my brother Alessandro

Summary

Web tracking techniques are constantly evolving, becoming ever more pervasive and harmful for user privacy and company security. The last years witnessed the birth and diffusion of stateless tracking mechanisms, which go under the name of "fingerprinting", that allow device identification without the need for client-side storage. These methodologies rely exclusively on JavaScript scripts injected into the source code of webpages and the use of particular APIs, initially created for other purposes, but which ultimately allow the retrieval of device-identifying information. As JavaScript scripts are widely diffused on the web and the APIs used for fingerprinting are in most cases essential for the correct functioning of webpages, users have no means to avoid being tracked by services employing such methodologies, unless by using tools which severely harm user experience and webpage functionality by selectively allowing the execution of Javascript scripts.

Motivated by the privacy and security risks, the purpose of this thesis has been the development of automatic detection methodologies able to recognize fingerprinting scripts by statically analysing their code, through the use of code-mining and machine learning algorithms, in order to be ultimately able to identify scripts and block fingerprinting attempts. The static approach presents advantages like the possibility to perform a totally offline analysis and the lack of need for dependencies other than the set of analysed scripts, but also limitations such as the impossibility to analyse strongly obfuscated scripts and a much lower amount of extracted information. In particular, the system developed in this thesis relies on the identification of API components and code patterns which are commonly used in web fingerprinting methodologies, the creation of features from the extracted information and the use of machine learning classifiers in order to label the analysed scripts as fingerprinters or not.

The developed system obtained positive results and the static analysis through machine

learning algorithms has been proven to be a valid approach for the automatic detection of fingerprinting scripts in the web, although it presented some limitations which should be addressed in the future by improving the current methodology or integrating it with a dynamic system.

Acknowledgements

I would like to thank my corporate tutor Stefano Traverso, who mentored me during all the phases of the thesis, my advisor prof. Marco Mellia for his support, and Sjors Haanen and Tim van Zalingen for their previous work and the publication of the related code, which has been highly helpful for the development of this thesis.

My greatest thanks goes to my family and in particular to my parents, Erica and Fulvio, and to my brother Alessandro, who together supported me and shared joys and difficulties throughout the academic years, and without whom I would not be the person I am today. Thank you.

Contents

Chapter 1

Introduction

The modern web represents the perfect environment for advertisements, as it offers important advantages over other platforms which can be crucial for advertising purposes. Tracking techniques allow the discovery of users' interests and preferences, and the knowledge of this information is a significant advantage for advertisers as it becomes possible to target advertisements to the users who are more probably interested in the related products, considerably increasing the effectiveness of advertising activities. For these reasons, more than 20,000 services, called trackers, build their business on the collection of users' data, in order to produce analytics and statistics about them and being able to target them with advertisements concerning products which they are probably interested into. While this mechanism could be considered beneficial to the end users, as the displayed advertisements concern products which they consider interesting, it also represent a privacy threat, as the amount of collectable information by using tracking techniques is alarmingly large. In addition, web tracking represent a security threat for companies too, as tracking the web activity of employees can lead to the leak not only of personal information about the employees, but also of sensitive company data such as strategic plans and research and development information.

Web advertisements base their working principles on auctioning mechanisms called real-time bidding [16, 23]. Webpage areas dedicated to advertisements are frequently managed by services called "ad exchanges" which, when the related webpages get loaded, expose the collected data regarding the current visiting user, including his preferences, habits, age, sex, geographical region and other information, to other services called "demand-side

platforms", used by the advertised companies in order to access ad exchanges. Demand-side platforms offer their bid for the advertisement, based on the received user information and the advertised product, and at this stage the ad exchange service selects the platform offering the best bid and displays the related ad.

Most of the free services offered on the web base their business model on advertisements, and as these services multiply users' personal information is increasingly exposed to the trackers' collection activities and, consequently, privacy risks for users are constantly increasing.

At the birth of web tracking, its functioning was exclusively based on the use of cookies. Cookies allow tracking services to assign to each user a unique identifier and store it into the user's web browser. Thanks to this system, tracking services can recognize the user on multiple websites by reading the mentioned identifier, and therefore track the websites the user visits. This technique is highly identifying, as each user is characterized by a different identifier, but it presents a major problem: the lack of cookies persistence. Indeed, not only web browsers store a limited number of cookies, but users can also easily manage them and delete unwanted ones. This possibility limited the effectiveness of the tracking activity, as a previously met user who deleted cookies related to a tracking service appeared to the latter as a new, unknown user. This problem led to the development of new tracking methodologies, based on alternative storage locations as Flash cookies [2], Silverlight cookies [3], ETags [8], web cache, `window.name` DOM property, HTML5 session, global and local storage, HTML5 IndexedDB, and force-cached PNGs using HTML5 Canvas tags to read pixels, used as cookies, back out [3]. By using alternative storages for saving the users' unique identifiers, tracking services managed to increase the complexity, for a web user, to avoid being tracked, as the system became more resilient. The use of the listed storage possibilities led to the creation of respawning cookies, also called "evercookies" [18]: since the same identifier is saved in multiple locations, in the case that only a part of the copies gets deleted by the user it is possible to respawn the removed values from any storage that persists, without affecting the tracking activity.

Figure 1.1 exemplifies this mechanism in the case the user identifier is stored both in Flash and HTTP cookies and the latter gets removed by the user.

Another commonly used workaround for the deletion or total blockage of cookies is embedding unique identifiers in URL queries of HTTP requests [12]. By using this technique, it is extremely difficult for a user to avoid being tracked as the query containing the

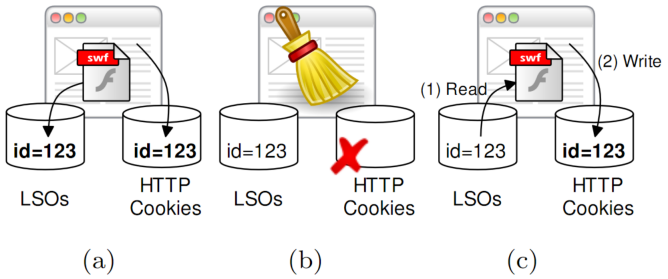

Figure 1.1: From [2]. Respawning HTTP cookies from Flash cookies: (a) the webpage stores an HTTP and a Flash Cookies (LSO), (b) the user removes the HTTP cookie, (c) the webpage respawns the HTTP cookie by copying the value From the Flash cookie.

corresponding identifier can be mixed together with other functional queries, and as such it becomes challenging to detect, other than impractical.

An additional tracking methodology, developed in order to enhance tracking accuracy, has been cookie synchronization, also known as cookie matching [2, 6]. Through this practice, tracking services exchange the assigned identifier to a given user, so as to extend the amount of retrieved information about that particular user.

Despite the already high pervasiveness of the exposed tracking methodologies, new stateless tracking mechanisms have been developed in the last years allowing to recognize devices which have been previously encountered by a tracking service without the need to depend upon stateful systems, like cookies and HTML5 storage. These techniques rely on JavaScript scripts, widely used in the web and difficult to control by an end user without breaking the relative web page functionality, in order to collect a vast amount of information regarding both hardware and software characteristics of the machine on which they are being executed, allowing for its unique identification. These procedures, which goes under the name of "fingerprinting" since a fingerprint of the executing machine is produced in order to uniquely identify it, represent a major privacy threat as it becomes extremely difficult to avoid being tracked, since only an expert user can distinguish fingerprinting scripts from those which are functional for the webpage. Additionally, even in the case of an expert user, the process of selecting which of the scripts contained in a webpage to execute and which to block is remarkably complex and tedious, and for these reasons it cannot represent a valid countermeasure against these tracking methodologies.

The purpose of this thesis has been the creation of a system capable of automatically detecting fingerprinting scripts by statically analysing them, and therefore without the

need to execute the contained code, through the use of code-mining and machine learning algorithms.

The following work is structured as follows: chapter 2 describes the existing fingerprinting techniques, how they operate and the analysis and results obtained by previous works which investigated web fingerprinting mechanisms; chapter 3 and chapter 4 illustrate the developed system, its working principles and the undertaken decisions during its development; chapter 5 reports the system results, both during the development and at its end; finally, in chapter 6 are expressed some observations about the system, the chosen approach, the obtained results and possible improvements to be considered in future work.

Chapter 2

Background and related work

In the following sections, the main fingerprinting techniques are introduced by explaining their working principles, analysing their effectiveness and presenting the results obtained by noticeable previous works which examined them and, in some cases, proposed possible defence mechanisms.

2.1 Canvas fingerprinting

According to [2], canvas fingerprinting is the most widespread fingerprinting technique on the web. It bases its operation on the use of HTML5 canvas element and it has been first discovered by Keaton Mowery and Hovav Shacham in [13]. The final image rendered through the mentioned HTML5 element presents, in fact, slight diversities among different machines based on differences in the operating system, set of installed fonts, graphics card, graphic driver, physical display and even internet browser used by the machines rendering it. Moreover, the technique is completely transparent to the users, as drawing elements into a canvas context and managing it can be done without displaying the obtained image on the webpage. Mowery and Shacham estimated an entropy of at least 10 bits for the fingerprint obtained by using the HTML5 canvas element, meaning that one in a thousand users share the same fingerprint. This estimation is considered very conservative as Benoit Jacob, in [7], already approximated an entropy of 9 bits caused exclusively by the GPU model. In Mowery and Shacham's experiment, 94.2% of the analysed devices were characterized by a unique fingerprint.

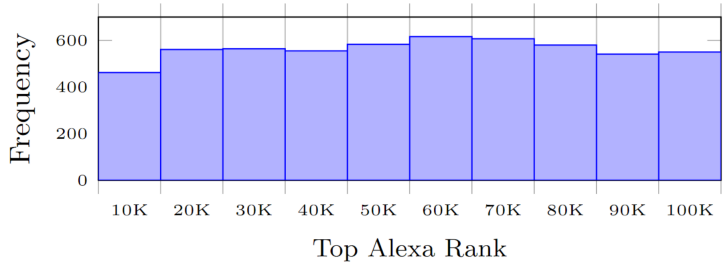

Figure 2.1: From [2]. Frequency of canvas fingerprinting scripts on the home pages of the Top Alexa 100K sites.

Canvas fingerprinting works by creating a particular image, aimed at exposing possible peculiarities of the machine on which the corresponding code is being executed, and then retrieving the image by converting it into a string (and, in some cases, hashing the resulting string). In order to uncover these peculiarities, the image contains one or more strings, characterized by a given font and size, aimed at highlighting font rendering differences between the analysed devices. Other than diversities caused by graphical rendering properties, such as font rasterization, anti-aliasing and smoothing, the string also aims at verifying the support of different writing scripts and at checking how particular Unicode characters, like those corresponding to emojis, are rendered, exposing the different implementations adopted by operative systems and their versions. The two methods commonly used for the described purposes are `fillText()` and `strokeText()`. Additional common checks in canvas fingerprinting concern `globalCompositeOperation` support, which manages how overlapping images are drawn, and the test for the presence of a specific point in a path by drawing two or more geometrical shapes and using the `isPointInPath()` method. Figure 2.2 illustrates the rendered figure when performing canvas fingerprinting through the use of the Fingeprintjs2 library[1]. "Cwm fjordbank glyphs vext quiz" is a famous pangram[2] commonly used in canvas fingerprinting, with the aim of drawing every letter of the alphabet in the canvas context in order to maximize the probability of rendering differences between the fingerprinted devices. Printing a string containing all the Latin letters in alphabetical order is also a common practice.

Finally, multiple methods can be used in order to return the created image: `readPixels()`,

[1]FingerprintJs2 - Modern & flexible browser fingerprinting library: http://valve.github.io/fingerprintjs2/

[2]Pangram: a sentence containing every letter of the alphabet.

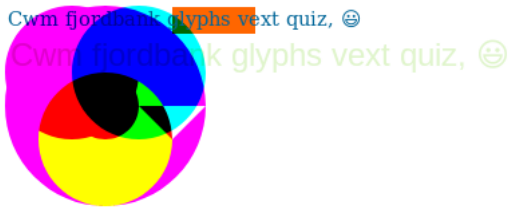

Figure 2.2: The rendered image in Fingeprintjs2's canvas fingerprinting.

`getImageData()`, `toDataURL()`, `toBlob()`, `mozGetAsFile()`, `mozFetchAsStream()` and `extractData()` [2, 11, 13] all represent valid alternatives for retrieving the information contained inside the rendered image and use it as a device identifier. In particular, the most used methods in canvas fingerprinting are `toDataURL()`, which returns a base64-encoded URI containing a representation of the drawn image, and `getImageData()`, that returns an `ImageData` object which includes, in its `data` property, the RGBA values corresponding the image. The other methods, as stated in [2], are rarely used for fingerprinting purposes as they would require extra steps without gaining any advantage in the final result.

As reported in [5], the usage of canvas fingerprinting has noticeably decreased among tracker services in the last years as it brought negative public perception to those using it. Nevertheless, its diffusion on the web has increased considerably, as knowledge of the technique has spread and more obscure trackers have not been concerned about public perception.

2.2 AudioContext fingerprinting

AudioContext fingerprinting is another fingerprinting methodology which bases its functioning on hardware and software differences between the machines on which it is applied. In particular, the internet browser and the audio stack, including the specific audio card mounted on the machine, the audio driver and the operating system, are the causes for the differences between machines which allow their fingerprinting.

This technique is usually deployed using two alternative processes, illustrated in Figure 2.3. The first method, illustrated in the upper part of the figure, is obtained through

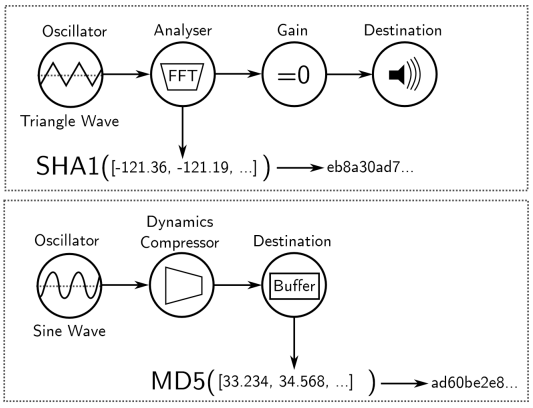

Figure 2.3: From [5]. AudioContext node configurations used to generate a fingerprint. Top: Used by `www.cdn-net.com/cc.js` in an `AudioContext`. Bottom: Used by `js.ad-score.com/score.min.js` in an `OfflineAudioContext`.

the use of an `AudioContext`, which is an audio-processing graph built from audio modules linked together. It makes use of an `OscillatorNode` in order to create an audio wave, whose type is not relevant for the fingerprinting purposes. This node is followed by a second node, of type `AnalyserNode`, which is able to provide real-time frequency and time-domain analysis information. This is the most important node for the fingerprinting purposes, as it allows to analyse the audio wave produced by the system and to collect its frequencies components. For this purpose, a Fast Fourier Transform (FFT) is used by the node in order to sample the wave signal over a period of time and divide it into its frequency components. A `GainNode` is follows the Analyser, setting the audio gain to `0` in order to avoid reproducing audible sounds, making the fingerprinting technique transparent to te tracked user. The frequencies values used for creating the unique identifier are commonly retrieved using the `getFloatFrequencyData()` method of the `AnalyserNode` object. `getByteFrequencyData()` could be a valid alternative, but its use has not been noticed for fingerprinting purposes nor in previous works nor in this thesis' study. Finally, the gathered values are hashed and the resulting string is used as the fingerprinted device identifier.

The latter method, illustrated in the lower part of the figure, employs an `OfflineAudioContext`, which differs from the standard `AudioContext` because it doesn't render the audio to the device hardware but it outputs the result to an `AudioBuffer`. In this case a `DynamicCompressorNode` is used instead of the `AnalyserNode` exploited in the previous method. This node is used to tweak the audio wave that is successively passed to

an `AudioDestinationNode`, which represents the final destination of the audio wave. From this node, the buffer representing the processed audio wave buffer is retrieved and, similarly to the previous methodology, it gets hashed in order to create the device-identifying string.

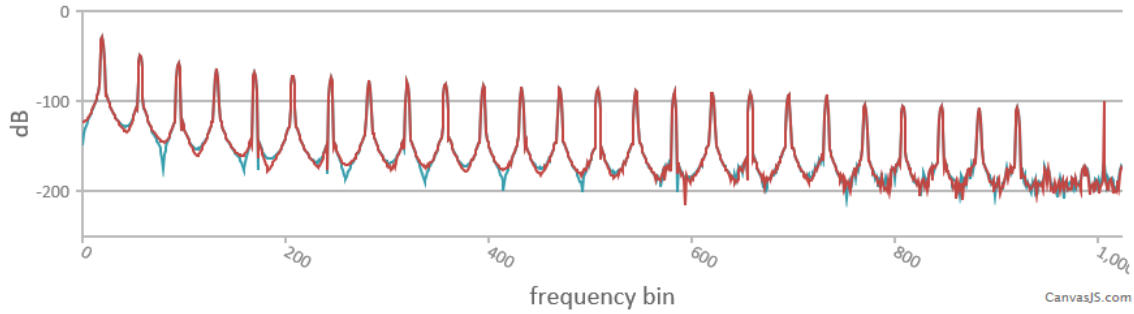

Figure 2.4: Audio frequency samples obtained by two different machines: the red samples are produced using Google Chrome on Android, the blue ones by running Microsoft Edge on Windows. The graphical representation has been generated by using Princeton CITP's AudioContext Fingerprint Test Page[3].

In [5], Steven Englehardt and Arvind Narayanan estimated an entropy for the described fingerprinting methodology of 5.4 bits, basing their evaluation on a set of 18,500 devices which produced 713 different fingerprints. This value is not sufficient for the described technique to uniquely identify the targeted devices, but it can be used in conjunction with other methodologies in order to create a fingerprinting system based on multiple techniques.

2.3 WebGL fingerprinting

WebGL is a graphics API used for rendering interactive 2D and 3D objects in the browser and manipulate them through JavaScript without the need for external plugins, allowing GPU-accelerated usage of physics, image processing and effects as part of the web page canvas. However, differently from standard canvas elements, this API does not use `CanvasRenderingContext2D` but a dedicated context, which is `WebGLRenderingContext`.

Possible fingerprinting methodologies offered by this API have been initially analysed

[3]Princeton CITP's AudioContext Fingerprint Test Page: https://audiofingerprint.openwpm.com/.

in [13] and were based on the same working principles which characterize canvas fingerprinting: the rendered image, in fact, presents slight differences based on the graphics cards of the machines on which it is rendered, even if drawing a simple scene consisting of 200 polygons, a black and white texture applied on their surfaces, a simple ambient and directional lights. The motivation for these differences is to be found on the different graphical stacks characterizing the tested machines, similarly to what causes analogues differences in canvas fingerprinting. Given the similar dependencies to the already introduced fingerprinting methodology, this technique has not faced the same diffusion on the web as canvas fingerprinting, since the latter's complexity is considerably lower while providing comparable information.

Nevertheless, the API presents another privacy vulnerability which can be exploited for obtaining information about the device on which the rendering context is created: initially designed for supporting developers but available for any purpose, some of the methods offered by the API expose informations about the physical GPU, the operative system and the related environment which characterize the machine on which they are called. In particular, the `WEBGL_debug_renderer_info` interface provides two properties, the WebGL vendor and the WebGL renderer, which return the name of the GPU vendor and model, respectively. In addition, `WebGLRenderingContext` offers, through the `getParameter()` method, an access to over 90 parameters which highly depend on the graphics stack of the device executing the code, and all together represent a noticeable amount of information which can support other fingerprinting techniques in order to better identify a device.

Finally, in Appendix B it is reported an example of fingerprinting code using the WebGL API and the described mechanism.

2.4 Battery fingerprinting

The Battery Status API provides information about the battery charge level, its charging or discharging status, the amount of time remaining for the battery to get completely charged or discharged and can also be used in order to get notifications, through an event-based system, about changes in the mentioned information. It has been first presented in April 2011 [9], became a W3C Candidate Recommendation in May 2012 [10], and web browsers started supporting it, both on mobile and desktop platforms, in the same year [14, 22], but as Łukasz Olejnik et al. found several privacy vulnerabilities derived by a

possible API misuse in 2015 [17], browsers started limiting or removing support to the API. The mentioned paper showed how the Battery API can lead to user tracking both by using small amounts of battery readouts, in which case the tracking would be valid in the short-term, and by using high amount of readouts, which allow the reconstruction of the battery capacity and a long-term tracking procedure. In particular, the battery charge `level` property allows retrieving the necessary readouts, which make possible short-term tracking mechanisms. This value, if combined with the public IP address of the tracked device, can lead to the identification of a device in the short-term, as a device characterized by the same IP address of a previously recorded device and a battery level coherent with the amount of time that has passed from the last encounter has high probabilities of being the same device. Additionally, at the time of the discovery, Mozilla Firefox on Linux presented an additional weakness, as it returned a double-precision floating-point value corresponding to the exact battery charge level detected by the operating system, and even on other browser-OS combinations, on which this value was truncated before being exposed by the API, it was possible to calculate an higher measurement precision than the one obtained by reading the charge `level` by combining its value with the one of `chargingTime` or `dischargingTime`. By gathering and collecting such information, it is possible to estimate the tracked device's battery capacity and use it alongside other device properties in a broader device fingerprint [15]. As of today, Mozilla Firefox and WebKit removed the API support, Microsoft Edge never supported it but it is listed in the Microsoft Edge web platform features status and roadmap, currently labelled as "Under consideration"[4], and Google Chrome still fully supports the API, returning truncated values for the battery readouts. Being Chrome the currently most widely used web browser, occupying over 62% of the browser market share in June 2018 according to NetMarketShare[5], this tracking technique has been included among the methodologies detected by the developed system.

[4]Microsoft Edge web platform features status and roadmap - Battery API: https://developer.microsoft.com/en-us/microsoft-edge/platform/status/batterystatusapi/

[5]NetMarketShare Market Share Statistics for Internet Technologies - Browser Market Share: https://netmarketshare.com/browser-market-share.aspx

2.5 Font enumeration

Font enumeration represent another common technique used in order to create a device fingerprint, which bases its working principles on the probe of a high number of fonts, checking for their support on the tracked machine. This process leads to the extraction of device-identifying information since the set of supported fonts depends not only on the operating system and its version, but also on the software which is installed on the machine. Moreover, since fonts are OS-dependent, they allow linking different browsers running on the same device [1].

This procedure can be obtained in two alternative ways, the first based solely on JavaScript and `HTMLElement` properties, commonly referred to as JavaScript-based font enumeration, and the second based on HTML5 `<canvas>` element and the corresponding properties provided by `CanvasRenderingContext2D`, and for this reason referred to as canvas font fingerprinting.

JavaScript-based font enumeration is the most diffused methodology of the two, as it allows retrieving similar amount of information in a simpler way. The technique relies on the browsers' behaviour in case a non-supported font is set for writing a string in the HTML document: in fact, in this case browsers automatically switch to a default font and write the text by using that font instead of completely failing the writing attempt. Exploiting this practice, in order to perform the necessary font probing the technique starts by creating a specific HTML element and writing inside it a string characterized by a non-existent font. This string gets automatically written using the default font, and at this stage the dimensions of the mentioned HTML element are measured through the use of `offsetWidth` and `offsetHeight` properties or by calling the `getBoundingClientRect()` method, and saved into a dedicated variable. At this stage it is possible to start probing for the support of other fonts, by repeating the described process for each of them and comparing the measured element's dimensions to those obtained by the default font: since each font is characterized by slightly different measures on the same text, an equality to those of the default font can be intended as the lack of support for the tested font on the device, and the consequent automatic switch to the default one. Using this mechanism, a considerable amount of fonts are tested (circa 500 in the most diffused JavaScript snippet), retrieving identifying information about the analysed device. Figure 2.5 illustrates the diffusion of the described methodology on the homepages of the top 1 million Alexa websites in 2013.

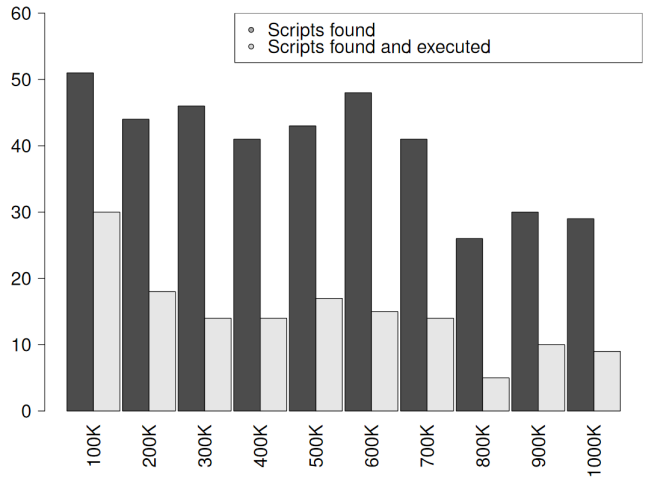

Figure 2.5: From [1]. JavaScript-based font probing scripts on homepages of top 1 million Alexa sites.

Canvas font fingerprinting is obtained in a very similar way to JS-font fingerprinting, performing the same operations in a `CanvasRenderingContext2D` and measuring the text size through the `measureText()` method. In [5] Steven Englehardt and Arvind Narayanan encountered this technique on 3,250 first-party sites over the top 1 million Alexa websites, corresponding to less than 1% of them but, as they highlight, the technique is more heavily used on the top sites, reaching 2.5% on the top 1,000, and in the majority of cases (90%) the corresponding JavaScript script was served by a single third-party, mathtag.com.

2.6 Plugin and mimetype enumerations

Plugin and mimetype enumerations are part of the most widespread fingerprinting techniques, as they are very simple to realize and, according to [4], the most identifying fingerprinting methodology, as their results are characterized by an entropy which goes from 16.5 to 17.7 bits. The code needed for producing such fingerprints is minimal, as it is only needed a loop construct to analyse every plugin or mimetype respectively exposed by the DOM properties `navigator.plugins` and `navigator.mimetypes`. In particular, the retrieved information is provided by the `.name`, `.filename`, `.description` and `.version` properties of the `Plugin` objects contained in the `PluginArray` provided by `navigator.plugins`, and the `.enabledPlugin`, `.description`, `.suffixes` and `.type` properties of the `MimeType` instances in the `MimeTypeArray` provided by `navigator.mimetypes`. As it is intuitable, the

amount of information collected through the presented properties in considerable, as it fetched not only the support or not of a plugin or mimetypes, but also other details as its version, the name of the file corresponding to the plugin, the list of supported suffixes by a mimetype and other details which allow the achievement of the 17.7 bits of entropy calculated in the aforementioned paper. In order to produce a fingerprint, as it is often the case with other techniques, the retrieved values are concatenated into a string and, by applying an hashing algorithm on it, the device identifier used for the pursued intents is created.

2.7 DOM properties collection

The collection of generic DOM properties represents one of the oldest and most simple fingerprinting techniques, as it relies exclusively on the collection of various property values offered by the `navigator` and `screen` APIs, which highly depend on the characteristics of the fingerprinted device. In Appendix A it is specified the list of over 30 properties which are taken into consideration in the developed system, as they are particularly common in fingerprinting scripts collecting information about the device by using this technique. The obtained information varies from the browser vendor to its language, from the screen's colour depth to its DPIs, from the number of cores of the device's CPU to the maximum number of touches detected by the screen at the same time: as it is noticeable analysing the mentioned list, the gathered data highly depends on the web browser, the operating system and the physical hardware of the fingerprinted device. As with other methodologies, also in this case the collected data is concatenated and successively hashed in order to obtain a device identifier.

2.8 Fingerprinting applications

Fingerprinting techniques are not enough, if taken singularly, to uniquely identify a device corresponding to a user. As the global population was estimated to 7.6 billions people in 2017 [19], the necessary amount of entropy to uniquely identify every human being is:

$$Entropy = -\log_2 \frac{1}{7600000000} = 32.823 \text{ bits}$$

No fingerprinting technique reaches this entropy level. Nevertheless, by combining the presented fingerprinting methodologies, it is possible to move closer to it, and in fact this

is the path taken by fingerprinters on the web. Figure 2.6 shows the results obtained by EFF's Panopticlick 3.0 fingerprinting test page[6]. As it is noticeable from the figure, the test uses most of the presented methodologies in order to produce a device fingerprint. In the test, the obtained fingerprint resulted to be unique among the 1,813,420 tested in the previous 45 days, and it produced at least 20.79 bits of identifying information.

Browser Characteristic	bits of identifying information	one in x browsers have this value	value
Limited supercookie test	0.38	1.3	DOM localStorage: Yes, DOM sessionStorage: Yes, IE userData: No
Hash of canvas fingerprint	13.9	15238.75	3bc532113c693692365c4430391d18f7
Screen Size and Color Depth	2.47	5.54	1920x1080x24
Browser Plugin Details	13.91	15367.89	Plugin 0: Shockwave Flash; Shockwave Flash 30.0 r0; NPSWF64_30_0_0_134.dll; (Adobe Flash movie; application/x-shockwave-flash; swf) (FutureSplash movie; application/futuresplash; spl).
Time Zone	2.53	5.76	-120
DNT Header Enabled?	1.26	2.39	False
HTTP_ACCEPT Headers	8.03	261.6	text/html, */*; q=0.01 gzip, deflate, br it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Hash of WebGL fingerprint	6.26	76.53	7a06e495f624fb535c8b5acac4c78409
Language	6.93	121.86	it-IT
System Fonts	9.77	871.41	Arial, Arial Rounded MT Bold, Arial Unicode MS, Book Antiqua, Bookman Old Style, Calibri, Cambria, Cambria Math, Century, Century Gothic, Century Schoolbook, Comic Sans MS, Consolas, Courier, Courier New, Garamond, Georgia, Helvetica, Impact, Lucida Bright, Lucida Calligraphy, Lucida Console, Lucida Fax, Lucida Handwriting, Lucida Sans, Lucida Sans Typewriter, Lucida Sans Unicode, Microsoft Sans Serif, Monotype Corsiva, MS Gothic, MS Outlook, MS PGothic, MS Reference Sans Serif, MS Sans Serif, MS Serif, MYRIAD PRO, Palatino Linotype, Segoe Print, Segoe Script, Segoe UI, Segoe UI Light, Segoe UI Semibold, Segoe UI Symbol, Tahoma, Times, Times New Roman, Trebuchet MS, Verdana, Wingdings, Wingdings 2, Wingdings 3 (via javascript)
Platform	3.0	7.98	Win64
User Agent	8.45	350.15	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:61.0) Gecko/20100101 Firefox/61.0
Touch Support	0.6	1.52	Max touchpoints: 0; TouchEvent supported: false; onTouchStart supported: false
Are Cookies Enabled?	0.22	1.17	Yes

Figure 2.6: Device fingerprinting results obtained by Panopticlick 3.0 using Mozilla Firefox 61.0.1 on Windows 10.

Finally, it is important to remark that device fingerprinting is not used exclusively

[6]Panopticlick 3.0 - Is Your Browser Safe Against Tracking?: https://panopticlick.eff.org/

for tracking purposes. As the presented techniques evolved, their use for fraud detection purposes became more and more common [5], as users logging in from unknown devices can be the result of possible account thefts, and security-critical services (e.g: online banking) can trigger additional checks if needed. As the final purpose of device fingerprinting cannot be discerned on the client-side, since it depends on the fingerprinting services internal uses of the retrieved identifiers, the automatic detection system developed in this thesis makes no distinction between the two usage categories, detecting fingerprinting procedures regardless of their final purposes.

Chapter 3

Code Analysis

The chosen approach for automatically detecting fingerprinting scripts is based on the use of binary classifiers in order to distinguish JavaScript scripts which attempt to create a fingerprint of the visiting device, gathering information trough known JavaScript APIs, from "safe" (i.e. non-fingerprinting) scripts. The analysis is fully static, which means that the code contained in the analysed scripts is never executed during the process but its text is parsed and processed before the successive execution of a machine learning classifier, producing the features that the classification algorithm will later use. The static approach is rarely to be found in previous works as it brings different limitations in the amount of information that can be extracted when analysing the code, and in particular it impedes an effective analysis of obfuscated scripts and prevents from knowing object's values at execution time. Nevertheless, static analysis allows analysing a script even without being in possess of its dependencies as it would be necessary when executing the code in a dynamic approach, and more importantly it allows a totally offline analysis. These advantages, in conjunction with the methodology being almost unexplored until today for the detection of fingerprinting scripts, led to a static approach to the problem in this work, accepting its limitations in particular with obfuscated scripts.

During an early research about previous works addressing the JavaScript fingerprinting detection problem, only "Detection of Browser Fingerprinting by Static JavaScript Code Classification" by Tim van Zalingen and Sjors Haanen, 2018 [21] has been found to have a static approach. Its authors used a deobfuscator/beautifier in order to try to deobfuscate scripts which have been obfuscated through common weak obfuscation techniques, then

created an Abstract Syntax Tree of the deobfuscated code and used it in order to "expand" every member expression contained in the code, meaning that if a value has been assigned to a variable through a variable declarator or an assignment expression, the correspondence was recorded and in the following occurrences of that variable it has been substituted by the value it has been assigned (the process will be examined in depth in section 3.2). Finally, the number of occurrences of JavaScript APIs commonly used in fingerprinting techniques was counted and the count values were used as features in a Support Vector Machine binary classifier in order to determine which scripts attempted to create a unique fingerprint for the device executing the code.

In the aforementioned project an extremely limited number of scripts consisting of 25 samples has been used in order to train and test the classifier, the Abstract Syntax Tree has not been fully exploited in order to extract the maximum amount of information from the analysed code and, more importantly, the only type of fingerprinting technique addressed by above-mentioned work is the one based on `plugins` or `mimetypes` enumeration and general DOM properties collection. Moreover, differently from the project developed in this thesis, the classification as fingerprinter or safe did not refer to single JavaScript scripts but to entire domains, analysing all of the scripts retrieved from a domain and then marking the entire domain as a fingerprinter or not.

Given that the mentioned work's code is publicly available at [20] and that the structure of the used methodology reflects what was the initial concept for the project developed in this thesis and appeared as a valid approach from the results obtained by the cited study, it has been used as a starting point and modified in order to adapt it to this work's goals. The following sections describe in detail the phases that compose the scripts' code static analysis characterizing the system developed in this thesis, while in chapter 4 the classification phase, obtained through machine learning algorithms, is examined.

3.1 Code deobfuscation and beautification

In this phase a JavaScript script executed in Node.js is used in order to deobfuscate scripts which have been obfuscated using common weak obfuscation techniques: in particular,

the addressed obfuscation methods and tools are Javascript Obfuscator [1], Dean Edward's `packer` [2], urlencoded JavaScript code and an old tool called `MyObfuscate`, now discontinued, which mostly based its obfuscation procedure on urlencoding JavaScript code.

This deobfuscation procedure remained mostly untouched if compared with the one offered by Tim van Zalingen and Sjors Haanen in their project, since deobfuscating JavaScript code is not one of the main goals of this work, given that static deobfuscation is possible only with a limited number of obfuscation methods and that, by choosing a static approach to the fingerprinting scripts detection problem, the possibility of being unable to properly process obfuscated scripts was accepted from the beginning, and any success in this sense has been considered a bonus more than an actual objective.

The present phase consists of two sub-phases: in the first it is attempted, during the analysis of a script, to detect obfuscation and eventually deobfuscate the code, while in the second phase the code is beautified and shaped in a more human-readable form.

In order to detect the code which has been obfuscated through `JavaScript Obfuscator` a regular expression in used, searching for matches on obligatory patterns in the resulting code. Once that obfuscated code has been detected, the hexadecimal numbers characterizing this technique, since they are used in order to replace literal characters, are converted back into ASCII characters and the resulting script is saved for further, subsequent processing.

Scripts compressed through `packer` are detected by searching, again by using a regular expression, for a static pattern common to all of the scripts compressed in this way. The `eval` function, used in these scripts in order to execute the obfuscated code, is temporarily overwritten by a substitute function that, after unpacking the code (i.e. made it readable, and so statically analysable), collects its deobfuscated text into a variable instead of executing it, as it would happen with a regular `eval` function. This process can defeat also some other `eval`-based obfuscators, as they pack the code in a very similar way to what `packer` does. The produced code is stored for the following processing steps.

Finally, the detection and deobfuscation of scripts obfuscated through the `MyObfuscate`

[1] `Javascript Obfuscator` (https://javascriptobfuscator.com/) bases its obfuscation procedure on converting ASCII characters which constitute the code of the script into hexadecimal numbers, which are then converted back at execution time and the resulting code is then executed.

[2] Dean Edward's `packer` (http://dean.edwards.name/packer/) is an eval-based JavaScript compressor that aims at reducing the code size but also causes the code to be illegible (obfuscated) until it is not converted back to a convetional form through the execution of the `eval` function.

tool or the `urlencode` function is obtained in a similar way, since these techniques are comparable, and it consists in searching for known patterns characterizing scripts obfuscated through these methods, detecting strings corresponding to obfuscated code and using the `unescape()` JavaScript function, which decodes url-encoded strings, in order to obtain a clear, readable version of the code.

Once that the deobfuscation attempts have taken place, the `js-beautify` npm[3] package is exploited in order to reformat and reindent the code, enhancing its readability. This last step is not strictly necessary for the fingerprinting detection process to work, but it is useful in order to successively check manually the goodness of the classification results.

3.2 Abstract Syntax Tree, member expression expansion and patterns check

For the following phase an Abstract Syntax Tree (AST) is created, based on the deobfuscated code. Similarly to the previous one, this phase is obtained through a JavaScript script executed in Node.js. In order to create the AST and explore it, so as to expand the member expressions contained in the script and identify patterns and APIs occurrences which are commonly use in web fingerprinting techniques, two npm packages are used, respectively `esprima`[4] and `estraverse`[5].

3.2.1 Member expression expansion

Working principles

An Abstract Syntax Tree allows creating a tree representation of the syntactic structure of the code in which each node stands for a code construct in the analysed script, providing, for every node, information about its construct type, its position and additional, construct-specific properties. This allows recording, for each scope, variable declarations and assignations, and also permits the identification of particular code patterns which are

[3] npm is the default package manager for JavaScript in the runtime environment Node.js. Other than a local, command-line client, able to download, install and manage packages, it offers an online database called "npm registry", where the available packages can be browsed.

[4] `esprima` npm package: https://www.npmjs.com/package/esprima

[5] `estraverse` npm package: https://www.npmjs.com/package/estraverse

useful to analyse in order to produce features for the following classification phase, extracting, within the limits of a static analysis, a noticeable amount of information about what will probably happen once that the code will be executed.

The successive classification phase is based on features consisting in the number of occurrences of code patterns and known JavaScript API particularly common in fingerprinting scripts, therefore counting the right amount of occurrences is fundamental for the classification phase to work properly. In this sense a mere string-matches count over the deobfuscated file would produce poor results, since after every variable assignation the subsequent occurrences of the left operand in the assignation would not be counted as occurrences of the right operand. The following example shows a basic case in which string-matching over the deobfuscated script would not work in order to identify the access to `window.screen` properties.

```
var ws = window.screen;
var wsw = ws.width;
var wsh = ws.height;
var wsc = ws.colorDepth;
var wsp = ws.pixelDepth;
var wsvdpi = ws.verticalDPI;
var wshdpi = ws.horizontalDPI;
```

Listing 3.1: Code snippet on which a string-matching search would not be adequate in order to count the number of occurrences of calls to `window.screen` properties.

If searching for string-matches over `window.screen.height` or any other `window.screen` property, the occurrences count in the illustrated code would be equal to zero, given that an exact, complete string representing the access to the mentioned properties in not present in the code. This is a first, fundamental case in which the use of an Abstract Syntax Tree is crucial: by recording, for each scope, all of the variable assignations it is possible to substitute the following occurrences of the left assignation operand with the right operand. By successively reporting on an output file every member expression contained in the code and expanding them (i.e. substituting a member expression with its assigned operand in a previous assignment expression in the same scope) it becomes possible to count the right amount of occurrences through string-matching. Listing 3.2 depicts what the described process would produce as output file when executing it on the upper code snippet.

```
window.screen
window.screen.width
window.screen.height
window.screen.colorDepth
window.screen.pixelDepth
window.screen.verticalDPI
window.screen.horizontalDPI
```

Listing 3.2: Member expression expansion phase output.

Performing a string-matching search over Listing 3.2 is now a reasonable approach in order to count `window.screen` properties occurrences.

Expanding member expressions

In order to perform the member expression expansion, each node of AST created from the analysed code must be processed. The AST can be easily obtained through the `esprima` package, passing to its `parse` method the code that is going to be analysed. The method returns an object corresponding to the AST, which is successively passed as an argument to the `traverse` method of the `estraverse` package together with two functions, `enter` and `leave`, which are executed respectively when entering and leaving a node during the tree traversal and that, jointly, contain all of the processing logic that will be applied on each node.

When analysing a node, the first step consists in a check on its `type` property: it specifies the kind of syntactic construct to which the node corresponds, and as such it is the main trait that characterizes the node. The first test regards the eventual creation of a scope: nodes that have a type corresponding to `FunctionDeclaration`, `FunctionExpression` or `Program` create a new scope, which is a fundamental information in order to expand member expressions since the correspondences established by variable assignations are valid only within their scope, and as such a collection of scopes and relative assignations is kept during the AST traversal in the form of different associative arrays, one for each scope. Node type is again the inspected property in the following check, aimed at detecting an assignment operation: the searched node types are `VariableDeclarator`, as during its declaration a variable may be initialized, and `AssignmentExpression`. When one of these node types in encountered, the left and right operands of the assignation are memorized into an associative array containing all the assignations that are part of the corresponding scope,

saving the left operand as a key of the associative array and the right one as the relative value. The name of the right operand is inspected before its memorization inside the array, in order to expand it in case it already corresponds to another expression. This check is performed recursively until the initial expression is found. Finally, when a node with type `MemberExpression` is found, it gets analysed in order to expand it and store the expanded value on the script's output file. A member expression can contain one or more other member expressions inside as it is, for example, with `w.screen.height`, since it contains the member expression `w.screen` and the property identifier `height`. In order to properly manage these cases, only the first expression is analysed when processing the corresponding node, as the inner ones correspond to other nodes that will be analysed later in the AST traversal. In the cited example, `w.screen` is, in turn, composed by the object `w` and the property `screen`: in the case `var w = window` was one of the previous expressions in the same scope, the analysed member expression will be expanded to `window.screen.height`.

3.2.2 Plugin and mimetype enumerations detection

The Abstract Syntax Tree is also useful in order to detect the most identifying fingerprinting techniques: plugin and mimetypes enumeration. Since these techniques rely on loops in order to detect installed plugins and supported mimetypes by the browser, it is possible to isolate calls to `navigator.plugins`, `navigator.mimetypes` or their properties that are located inside a loop by analysing the AST and checking if their relative nodes have, among their parent nodes, a node corresponding to a loop, meaning that they are placed inside a loop construct.

There are four loop types in JavaScript: while, do-while, for and for-in loops, which correspond to the node types `WhileStatement`, `DoWhileStatement`, `ForStatement` and `ForInStatement` inside the AST. When traversing the tree, it is checked if the currently analysed node is characterized by one of these types. If it does, the node is recorded into an array containing all the nodes which start a loop. When the `leave` function will be called on the same node which started a loop, meaning that all of its children nodes have been parsed and the analysis is leaving the code corresponding to the loop, the node is removed from the looping nodes array. In this way it is possible to know if a node is part of a loop or not by checking the looping nodes array's size: if it is different from zero, then the currently analysed node is contained inside a loop, otherwise it is not. Once that it is known that

the currently analysed AST section is part of a loop, for each `MemberExpression` node it checked if its expanded version contains a string identifying a call to `navigator.plugins`, `navigator.mimetypes` or one of their properties: if it does, the `-detectedLoop` suffix is added to the expanded value which will be reported on the current phase's output file, in order to distinguish occurrences of the mentioned APIs appearing inside a loop from those which does not.

The plugin and mimetype enumerations detection system has been improved during the whole thesis' development phase, as it showed some problems which have been gradually addressed. The issues were due to the fact that, as it is intuitable, not every occurrence of the concerned APIs which is contained inside a loop aims at enumerating plugins or supported mimetypes. Consequently, additional conditions and checks necessary to mark an occurrence as a possible enumeration attempt have been added during the work.

The accessed `navigator.plugins` or `navigator.mimetypes` index should not be a constant number or string but a variable: this because it is very common for the developer to check for the presence or not of a plugin by trying to access its index and then test on the returned value in order to understand if the plugin is installed or not. However, if the accessed index is constant, it is very probable that the developer is checking for the presence of a limited number of plugins, and so the API occurrence should not be marked as a potential enumeration attempt. This test is performed by checking the node type of the AST node corresponding to the index: if the type is `Literal`, then the index is not a variable but a string, and thus the `-detectedLoop` suffix, identifying possible enumeration attempts, is not added to the expanded value of the corresponding member expression.

Another condition regards the position of the API call occurrence: in the AST, children nodes of a loop-identifying node are not only the ones corresponding to syntactic constructs contained inside the loop block, but also those occurring inside the loop conditional expression. In these cases an enumeration attempt is extremely unlikely, and as such these occurrences should not be marked by the aforementioned suffix. Every AST node corresponding to a loop is characterized by the `test` property, containing the node of the syntactic construct coinciding with the loop's test condition. By excluding the member expressions contained in this node is then possible to avoid marking as potential enumeration attempts the API calls positioned into the loop condition. An additional check is performed when analysing a for loop, given that other than a test condition and a body it is characterized by the presence of an initialization field and an update field, also present

as properties in the relative AST node. While the update field is normally processed, since the enumeration process can take place inside it, it is unlikely for it to occur into the initialization field, which therefore is excluded from the enumeration check.

```
/* In the following example the call appearing in the test condition of the for
    loop is not marked as a possible enumeration attempt, as occurrences
    located there are exluded from this inspection. However, the call inside
    the if conditional expression in this case would be marked as an
    enumeration attempt, even if it is not: this issue will be solved in the
    following steps */
var flash = false;
for (var i = 0; i < navigator.plugins.length; i++) {
   if(navigator.plugins[i].name.indexOf("Adobe Flash") != -1) {
      flash = true;
      break;
   }
}

/* The following example is characterized by two navigator.plugins occurrences
    inside a for loop. While the first one is not marked as an enumeration
    attempt since it is located in the test condition of the loop, the second
    one is noted as such, being placed into the update field. */
var pluginsEnumeration = "";
var np = navigator.plugins;
for (var i = 0; i < np.length; pluginsEnumeration += np[i].name + "-" +
    np[i].description + "-" + np[i].version + "|", i++)
/*
```

Listing 3.3: Examples illustrating a case in which the abovementioned APIs occurrences are ignored and one in which they are detected as a possible enumeration attempt.

As commented in Listing 3.3, a problem persists: it is a common practice for developers to loop on the whole plugins collection in order to check if some plugins or mimetypes, necessary for their code to work properly, are installed on the browser. In these cases, exemplified by the aforementioned code sample, calls to the indicated APIs would occur inside a loop block and, as such, they would be marked as enumeration attempts. In order to avoid so, a check on nodes of type `IfStatement` having among their parents a node with a type indicating a loop (and as such being the corresponding code located inside a loop) is performed. In case the if statement's test condition implements a check on the accessed plugin by comparing its name with a constant string through an equality operator or by using the `indexOf` method and checking its return value, the accesses to

the concerned APIs are excluded form the enumeration attempts detection. This condition avoids errors when such a test is performed through the use of an `if` statement, but in JavaScript another common syntax is used in order to execute a test, achieved by the usage of logical expressions based on the operator `&&`: in the case of the operator in question, if the left operand returns `false` then the right one is not evaluated, since it would be superfluous as the final result would be `false` in any case. Exploiting this behaviour, the mentioned syntax is used to achieve a conditional execution, by using the condition as the left operand of logical expression and the code that needs to be executed if the condition is satisfied as the right operand. Given the above, the same tests performed on `if` statements are applied to the described construct, detected through the AST nodes having type `LogicalExpression`.

```
var pdfViewer = false;
var np = navigator.plugins;
for (var i = 0; i < np.length; i++) {
   np[i].name == "Edge PDF Viewer" && pdfViewer = true;
}
```

Listing 3.4: Code sample illustrating a case in which the mentioned test is fundamental in order not to mark as possible enumeration attempt the described construct, if a test on the plugin's name is contained.

3.2.3 Other detected constructs

The AST is useful not only for expanding all the member expressions and detecting plugins or mimetypes enumerations, but also in order to detect other types of enumerations and fulfil additional conditions on other recognized API calls before reporting them as possible fingerprinting attempts. In particular, in a similar way to what is done with plugin and mimetype enumerations, font enumerations can be detected by identifying occurrences of APIs located inside loops bodies and commonly used in order to perform them. As described in the previous chapters, there are two main types of font enumerations: canvas font enumerations, obtained by measuring the size of a text written in a canvas rendering context through the `measureText()` method, and JS-based font enumerations, through the `offsetHeight` and `offsetWidth` properties offered by the `HTMLElement` interface or the `getBoundingClientRect()` method provided by `Element`. Also in these cases the

`-detectedLoop` suffix is added to the output string when the mentioned methods are detected inside a loop, in order to distinguish them from calls appearing outside any loop construct.

Finally, the AST is exploited also in order to check the arguments of calls to the `toDataURL()` method of the `HTMLCanvasElement` interface, commonly used in order to produce canvas fingerprints. This method accepts two optional parameters: a `type` parameter, indicating the image format and whose default value is `image/png`, and an `encoderOptions` parameter, indicating the compression level if the chosen image format is `image/webp` or `image/jpeg`. For a canvas fingerprint to be effective the canvas representation contained in the returned data URI should not be compressed, as compression would cause minor modifications to the image which would be deleterious for the fingerprinting purposes. For this reason the used parameters are checked and, in case the requested image format is `image/webp` or `image/jpeg`, the API call is not counted among the occurrences marked as possible canvas fingerprinting attempts. As previously described, since the default value for the `type` optional parameter is `image/png`, when such parameter is not set in the API call the occurrence is not excluded from the count of possibly fingerprinting API occurrences. However, as it will be described in section 4.2, additional conditions must be met in order to finally count the analysed `toDataURL()` call among the aforementioned occurrences.

Chapter 4

Classification

The final step of the developed methodology consists in the analysed scripts classification into "fingerprinting" and "safe" (non-fingerprinting) ones. For this purpose, two machine learning classifiers are used: Support Vector Machine and Random Forest, both provided by the `scikit-learn` Python package. The set of features used in the classification process is determined by the execution of a Python script, whose objective is to count the number of occurrences of APIs commonly used in fingerprinting scripts. In the following, the classification model and its evolution during the thesis development are described in depth, after a brief introduction to supervised machine learning algorithms.

4.1 Introduction to supervised machine learning

Machine learning algorithms can be divided into two groups, based on their learning methodology: supervised and unsupervised learning algorithms. Supervised machine learning is characterized by the presence of a pre-labelled dataset, called "training set", used by developers in order to train the classification model, acting as a guide to teach the algorithm what conclusions it should produce based on input data characteristics. Unsupervised machine learning algorithms are, instead, provided with unlabelled data, and they automatically determine the groups in which data should be divided by analysing similarities and differences among the contained items, without any guidance by external data used as a model. For the automatic fingerprinting detection system developed in this thesis the used machine learning algorithms belong to the first group, as the classes

in which the processed scripts are divided and the features to be analysed are fixed and pre-determined.

Features represent the distinctive properties derived from input data which a machine learning algorithm employs in order to learn from the training dataset and successively classify the analysed items. Feature extraction is the process of transforming the input data into relevant values which can be interpreted by the classifier. Different algorithms exist for the extraction of features, of which an example is "term frequency–inverse document frequency" (tf-idf), commonly used in text classification. Feature extraction can be followed by two additional processes, called feature transformation and feature selection, which respectively aim to transform previously extracted features in order to improve the accuracy of the algorithm and to reduce the number of features, removing those which are superfluous and reducing redundancy. In the developed system, the chosen information to be transformed into features consist in the number of occurrences of particular API methods and properties which provide device-identifying information and are commonly used in fingerprinting techniques. Indeed, each fingerprinting methodology described in chapter 2 can be realized through the use of limited sets of API components, therefore their occurrences are detected in order to verify, by analysing their frequency and the satisfaction of predetermined conditions, if they are exploited for fingerprinting purposes or not.

After that the set of features to be analysed has been determined and the classification algorithm has been selected, the classification model can be built. A classification model is a mathematical model, characterized by parameters which are set in order to tune it and allow the transformation of the model's input into outputs consisting in the classes assigned to the analysed data. These parameters are set by adjusting them accordingly to the training data, in a process called "model fitting". This process represents the phase through which the algorithm learns to classify the analysed data and, as it is intuitable, the training set quality is crucial for achieving good classification performance.

The model is successively validated by evaluating its performance on an independent dataset. Cross validation is the most diffused model validation technique, and it has been used in the this project in order to assess how the model results generalize to an independent dataset. This technique is based on the division of the initial dataset into multiple subsets and their use for training and validation. In particular, k-fold cross validation is the methodology which has been adopted in this thesis, which is characterized by the division of the dataset into k equally sized groups. One of the created groups is used as a validation

set, on which the model performance is evaluated, while others are used as training data. This process is repeated k times, until all of the subgroups have been used for the model validation, and the final results are based on those obtained by each iteration.

4.2 Feature extraction

Differently from the initial work developed by Tim van Zalingen and Sjors Haanen in [21], the purpose of this thesis has not been to detect fingerprinting attempts by identifying only those carried on through the collection of `navigator` and `window.screen` properties, but also to identify canvas fingerprinting, WebGL fingerprinting, AudioContext fingerprinting, canvas font enumerations and JS font enumerations. In order to do so, different features have been added to the initial set provided by the mentioned project, while others have been removed or modified in order to improve the classification accuracy.

As introduced in chapter 3, the features used by the classifiers consist in the number of occurrences of a set of APIs obtained through string matching. In their work, van Zalingen and Haanen used alternatively occurrences counted from the deobfuscated script or those counted from the member expression expansion phase output as features for their classifier. In this thesis, after a first analysis of the results, it has been chosen to use both sets at the same time, since in this way it is possible to identify occurrences which are not recognized in one of the two detection steps: this allows minimizing errors and it is useful also in order to try to correctly classify some obfuscated scripts that the deobfuscation phase has not been able to deobfuscate. Indeed the Abstract Syntax Tree is not useful in these cases, since being created from obfuscated code causes it to have no significant information contained inside its `MemberExpression` nodes, and therefore the member expression expansion output is not useful. Nevertheless, some obfuscation techniques are based on run-time code composition starting from a set of strings containing the collection of used APIs concatenated and separated by pre-determined delimiters: in these cases, a string-matching search over the obfuscated script can detect the APIs and, as described in the following chapter, in some cases it can bring to a correct classification of part of the obfuscated fingerprinting scripts. However, it must be stressed that, given the static nature of this work's approach, classifying obfuscated scripts is not one of its objectives and any success in this respect is taken as a bonus rather than a pursued goal.

In addition to the conditions introduced in the AST traversal phase, further requirements are imposed in the current phase over a set of API calls for them to be counted among the possibly fingerprint-identifying ones. In the case of canvas fingerprinting, the set of detected methods is composed by `readPixels()`, `getImageData()`, `toDataUrl()`, `toBlob()`, `mozGetAsFile()`, `mozFetchAsStream()` and `extractData()`. However, their occurrence is not sufficient for the purposes of canvas fingerprinting, as text must be written into the canvas rendering context in order to complete the fingerprinting procedure [2, 5, 11]. For this reason the `fillText()` or `strokeText()` method calls must be present inside the script in order to count the mentioned APIs occurrences among those considered as possible fingerprinting attempts (and passed as features to the classifiers), and this check is conducted in the current phase.

In Appendix A it is presented the list of strings representing noticeable API calls in fingerprinting techniques, for which the number of occurrences is counted on the deobfuscation and member expression expansion phases' outputs. Some API occurrences are counted only if the `-detectedLoop` suffix is also present in the relative string produced in the member expression expansion output, as those API calls are particularly important only in case of an enumeration attempt and thus an occurrence positioned outside loop constructs would not be relevant.

Finally, it is proposed an example of a fingerprinting script - in which the device fingerprint is obtained through canvas fingerprinting, plugin enumeration, mimetypes enumeration and the collection of other `navigator` and `screen` properties:

```
function deviceprint_software() {
   var a = "";
   var b = "";
   var s = navigator.plugins;
   var o = navigator.mimeTypes;
   if (s.length > 0) {
      var m = s.length;
      for (var i = 0; i < m; i++) {
         var plugin = s[i];
         a += plugin.name + "-" + plugin.description + plugin.filename + "|"
      }
   }
   if(o.length > 0) {
      var m = o.length;
      for (var i = 0; i < m; i++) {
         var mimetype = s[i];
```

```
        b += mimetype.name + "-" + mimetype.description + mimetype.filename +
            "|"
      }
   }
   return a + b;
};

function deviceprint_display() {
   var a = "";
   if (self.screen) {
      a += screen.colorDepth + SEP + screen.width + SEP + screen.height + SEP +
          screen.availHeight
   }
   return a;
};

function deviceprint_java() {
   var a = (navigator.javaEnabled()) ? 1 : 0;
   return a;
};

function getCanvasFp() {
var a = [],
   c = document.createElement("canvas");
   c.width = 2E3;
   c.height = 200;
   c.style.display = "inline";
   var b = c.getContext("2d");
   return b.rect(0, 0, 10, 10), b.rect(2, 2, 6, 6), a.push("canvas winding:" +
       (!1 === b.isPointInPath(5, 5, "evenodd") ? "yes" : "no")),
       b.textBaseline = "alphabetic", b.fillStyle = "#f60", b.fillRect(125, 1,
       62, 20), b.fillStyle = "#069", this.options.dontUseFakeFontInCanvas ?
       b.font = "11pt Arial" : b.font = "11pt no-real-font-123",
       b.fillText("Cwm fjordbank glyphs vext quiz, \ud83d\ude03",
   2, 15), b.fillStyle = "rgba(102, 204, 0, 0.2)", b.font = "18pt Arial",
       b.fillText("Cwm fjordbank glyphs vext quiz, \ud83d\ude03", 4, 45),
       b.globalCompositeOperation = "multiply", b.fillStyle = "rgb(255,0,255)",
       b.beginPath(), b.arc(50, 50, 50, 0, 2 * Math.PI, !0), b.closePath(),
       b.fill(), b.fillStyle = "rgb(0,255,255)", b.beginPath(), b.arc(100, 50,
       50, 0, 2 * Math.PI, !0), b.closePath(), b.fill(), b.fillStyle =
       "rgb(255,255,0)", b.beginPath(), b.arc(75, 100, 50, 0, 2 * Math.PI, !0),
       b.closePath(), b.fill(), b.fillStyle = "rgb(255,0,255)", b.arc(75, 75,
       75,
   0, 2 * Math.PI, !0), b.arc(75, 75, 25, 0, 2 * Math.PI, !0),
       b.fill("evenodd"), c.toDataURL && a.push("canvas fp:" + c.toDataURL()),
       a.join("~")
```

```
};

var fp = deviceprint_software() + "|" + navigator.appCodeName + "|" +
    navigator.cpuClass + "|" + naviagtor.platform + "|" + deviceprint_display()
    + "|" + deviceprint_java() + "|" + getCanvasFp();
```

Listing 4.1: Example of a fingerprinting script.

```
(exp)navigator.plugins-detectedLoop,2
(exp).name-detectedLoop,2
(exp).filename-detectedLoop,2
(exp).description-detectedLoop,2
(exp)navigator.plugins.length,2
(exp).toDataURL,2
(exp)navigator.appCodeName,1
(exp)navigator.javaEnabled,1
(exp)screen.height,1
(exp)screen.width,1
(exp)screen.colorDepth,1
(exp)screen.availHeight,1
(exp)navigator.cpuClass,1

(dob).toDataURL,2
(dob).appCodeName,1
(dob).javaEnabled,1
(dob)screen.height,1
(dob)screen.width,1
(dob)screen.colorDepth,1
(dob)screen.availHeight,1
(dob).platform,1
(dob).cpuClass,1
```

Listing 4.2: The set of features extracted from the script.

Every feature is characterized by a `(dob)` or `(exp)` prefix, whose purpose is differentiating those obtained from the deobfuscation phase output from those acquired from the member expression expansion phase output.

4.3 Classification models

While T. van Zalingen and S. Haanen used a Support Vector Machine for the final classification phase, in the current work both Support Vector Machine and Random Forest have

been used for the purpose. Both classifiers use as features the number of occurrences related to output files deriving from the deobfuscation phase and the member expression expansion phase, as previously described. The classification scripts are written in Python and the Scikit-learn library (`sklearn` package) is used for retrieving machine-learning algorithms.

The best hyperparameters for both the classifiers have been found by executing a cross-validated grid search optimization over a set of 4 hyperparameters with 96 possible combinations and 1879 scripts in the dataset, divided in 1148 safe-labelled and 731 fingerprinting-labelled scripts. The first set has been constructed by gathering JS scripts from known, non fingerprinting websites consisting mainly of government and academic websites, whereas the second set has been collected using the database[1] offered by Princeton University's WebTAP project, which is the result of Steven Englehardt and Arvind Narayanan's work in Online Tracking: A 1-million-site Measurement and Analysis, 2016 [5], and by manually collecting scripts for fingerprinting methodologies which does not appear in the mentioned database. Moreover, scripts listed in the considered database have evolved between the time in which they have been analysed and the time of their download for the current work, and as such some of them, which initially did fingerprint the machine on which they were executed, did not once that they have been downloaded for this thesis work, as the fingerprinting code have been removed. For this reason many of them have been manually analysed after their download, in order to move non-fingerprinting script into their relative set. The described collection process led to the acquisition of 1879 scripts. Additional details about this dataset are described in section 5.1. The used hyperparameters for the two classifiers are:

```
C=100, coef0=0.0,
class_weight=None,
decision_function_shape='ovr',
degree=3, gamma=0.001,
kernel='rbf',tol=0.001
```

Listing 4.3: SVM hyperparameters

```
bootstrap=True, class_weight=None,
    criterion='gini',
max_depth=None, max_features='auto',
    max_leaf_nodes=None,
min_impurity_decrease=0.0,
    min_impurity_split=None,
min_samples_leaf=10,
    min_samples_split=2,
min_weight_fraction_leaf=0.0,
    n_estimators=64, n_jobs=1
```

Listing 4.4: RF hyperparameters

[1] Database available at https://webtransparency.cs.princeton.edu/webcensus/

The Random Forest classifier's inner workings can be easily interpreted as it relies on a set of decision trees characterized by randomly selected features. The following, partially cut tree represents one of the decision trees produced by the classifier, and serves to provide an idea of the classifier's decision mechanisms:

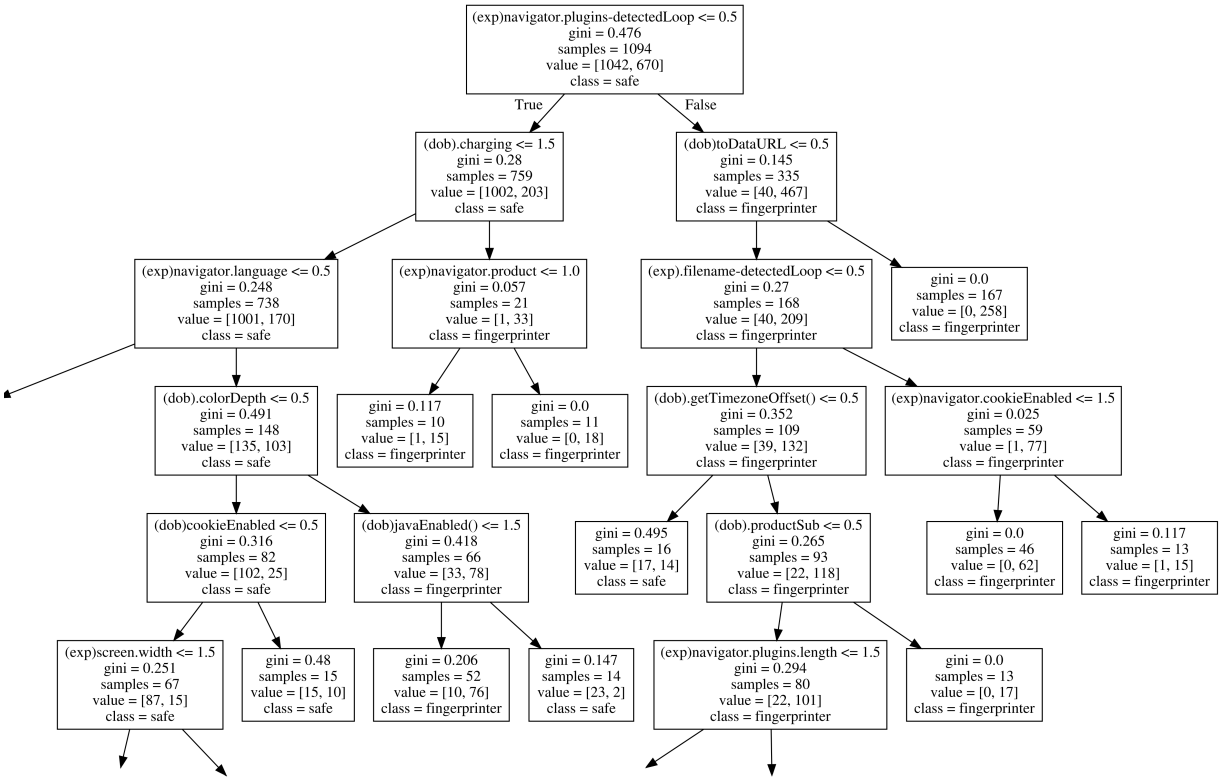

Figure 4.1: Representation depicting part of one of the 64 trees produced by the Random Forest algorithm.

Chapter 5

Results

In this chapter the results obtained by the developed automatic detection system are presented, explaining how they have been influenced by the improvements introduced during the development phase and comparing them to those of previous related work.

5.1 The used datasets

During the thesis work, two distinct datasets have been exploited in order to develop the entire system: the one that has been introduced and discussed previously in section 4.3, consisting of 1879 scripts obtained by government and academic websites, the mentioned Princeton Web Census database and manually inserted script and another, larger dataset collected during the initial phase of the thesis. In the following, the first set will be mentioned as the "development" dataset, whilst the latter will be called the "wild" dataset, as it provides a good representation of the script variety found in the wild. It is important to specify that the development dataset has been improved throughout the duration of the thesis, in order to increase its entropy and make it more representative of the real scripts' variety found on the web. This enhancement operation has been fundamental for the improvement of the final results, since the considered dataset is used for the creation of the classifiers' training set and thus to make the classifier learn to distinguish safe scripts from fingerprinters. Initially, the development dataset consisted of 789 safe scripts and 262 fingerprinting scripts. The safe scripts which were present at the time, whose only source were academic and government websites, have been later reduced to 300 since the

limited source diversity caused many scripts to be very similar among them. Both safe and fingerprinting script sets have been later enlarged up to 1,879 total items, divided in 1,148 safe and 731 fingerprinting scripts, by manually labelling scripts contained in the wild dataset and copying them into the development dataset, in order to improve it.

	Safe	Fingerprinters	Total
# of scripts	1,148	731	1,879
# of domains	561	324	885

Table 5.1: Final dimensions of the development dataset.

The wild dataset has been gathered starting from a database containing traces of volunteering users' activities while surfing the web, in which each record represents an HTTP request or response transmitted by the browser. The database counts 52,063,801 records, collected from April to August 2017 and generated by 982 different users. Starting from a set containing all the records, all those having an associated HTTP method different from `GET` or `POST`, or being responses to previous requests have been removed. Then, analysing the remained records' URLs, only those ending with `.js`, and as such presumably consisting in JavaScript scripts, have been kept. The remained scripts have been grouped by domain, determining it through the `PublicSuffixList` Python package, and successively downloaded in order analyse them. The described processing led to a set containing 716,328 records, coming from 40,148 different domains. These records have been further analysed in order to group scripts which always appeared as third-parties, script which always appeared as first-party and those which had been used in both ways. In order to do so, the domain derived from the script URL has been compared to the one indicated by the `referer` HTTP header field. Whist this can appear as a valid approach, it presents some flaws: it's not uncommon for scripts not to be requested directly from the webpage which the user is visiting, but from other, third-party elements which are present on the page. In those cases, if the element making the request is characterized by the same domain of the requested script, the described methodology would label the script occurrence as a first-party, even if both the requesting element and the script are finally loaded in a website having a different domain. The exposed problem have not been solved, as the database containing the records did not report any information about the website on which the scripts were going to be loaded or the possible chain of requests which brought to the one regarding the script. Given the difficulties in recreating a possible requests chain in a

set of records containing no information about the relative relations and that the knowledge about a script being used as third-party or first-party did not represent a critical information for the thesis' purposes, the possible errors derived from the discussed complication have been accepted. The described mechanism led to the collection of the following sets: 474,415 scripts from 31,385 domains which always appeared as first-parties, 237,191 scripts from 8,278 domains which always appeared as third-parties and 4,722 scripts from 480 domains which appeared both as first-parties and third-parties considering the entirety of the records. These sets have been finally downloaded. As expected, a noticeable part of the dataset could not be downloaded as the scripts related to the reported URLs were not available any more. Additionally, many scripts, even if downloaded from different URLs, contained the same code, as reusing publicly available JavaScript snippets is a common practice when creating websites. For this reason, the dataset has been further reduced by calculating an MD5 hash for each of the downloaded scripts, comparing the hashes and deleting those scripts which were found to be equal. The final dataset dimensions have been the following: 89,029 scripts always identified as first-party, 70,479 scripts labelled as third-party and 1,443 scripts used in both ways, for a total of 160,951 items in the database, grouped by methodology of use (first or third-party) and domain.

	First-party	Third-party	Used in both ways	Total
# of scripts	89,029	70,479	1,443	160,951
# of domains	8,843	4,224	340	12,554

Table 5.2: Final dimensions of the wild dataset.

The development and wild dataset present a major difference: while in the first set every contained script is labelled, dividing scripts which use fingerprinting techniques from those which does not, and as such it can be used as training and test set in order to fit, evaluate and, based on the achieved results, improve the classifier, it is much more arduous to do so by using the wild dataset, which lacks any information about the presence of fingerprinting code in the contained scripts. Considering this, the development dataset has been the one initially used to develop the automatic detection system, while the wild dataset, larger and containing more diverse scripts, has been used only later so as to evaluate the system performance with data that better represents the real JS scripts variability in the web.

5.2 Tests overview

The following table summarizes the most important tests which have been conducted during the project development and which are successively described in depth in section 5.4 and section 5.5, in order to provide the reader with an overview for a clearer understanding. The table is later reproduced at the and of section 5.5 to recapitulate the respective results.

Test name	**#Safe & #Fp in test set**	**Data origin**	**Comment**
Initial test; SVM	#Safe: 7 #Fp: 3 (4-fold CV)	Small initial dataset	Using the classification model provided in [20], new features have been added for detecting all targeted fingerprinting techniques. Features are extracted by the deobfuscation or member expression expansion phases' output alternatively. Table 5.5, Table 5.6.
Development test 1; SVM	#Safe: 279 #Fp: 195 (4-fold CV)	Development dataset	Introduction of the development dataset for model training and testing, removed irrelevant features, union of the deobfuscation and member expression expansion phases' features. Table 5.7.
Development test 2; SVM	#Safe: 111 #Fp: 78 (10-fold CV)	Development dataset	Introduction of advanced loop detection, enhanced canvas fp detection, new hyperparameters through CV grid search optimization. Table 5.8.
RF introduction	#Safe: 111 #Fp: 78 (10-fold CV)	Development dataset	Testing the Random Forest algorithm performance. Table 5.9.
Wild precision (SVM)	#Total: 43,580	Wild dataset	First approximation of the system's precision score on the wild dataset.Table 5.10.
Wild recall (SVM)	#Total: 27,834	Wild dataset	First approximation of the system's recall score on the wild dataset. Table 5.11.
Final wild scores (SVM & RF)	#Total: 160,951	Wild dataset	Enhanced enumeration detection capabilities, modified hyperparameters. Final results on the wild dataset. Table 5.12, Table 5.13.

Test name	#Safe & #Fp in test set	Data origin	Comment
Final development scores (SVM & RF)	#Safe: 111 #Fp: 78 (10-fold CV)	Development dataset	Final results on the development dataset. Table 5.14, Table 5.15.

Table 5.3: Overview of the system developments and relative tests presented in this chapter.

5.3 Initial results

As previously introduced, T. van Zalingen and S. Haanen's "Detection of Browser Fingerprinting by Static JavaScript Code Classification" project's code [20] has been used as a starting point for the development of the currently presented system. This project bases its classification phase on the use of an SVM classifier, and it is characterized by a definitely modest dataset on which to perform the classifier's training and testing. The scores which have been calculated in the mentioned work and that will continue be evaluated so as to measure the performance of the system developed in this thesis are the recall, precision and F_1 scores, respectively obtained through the following formulas:

$$Recall = \frac{|TruePositives|}{|TruePositives|+|FalseNegatives|} \qquad Precision = \frac{|TruePositives|}{|TruePositives|+|FalsePositives|}$$

$$F_1 score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Reminding that the mentioned project's approach presents differences from that of the system presented in this thesis, since it aims to detect only the fingerprinting techniques which rely on `navigator` and `screen` properties collection and presents a different classification granularity, as it classifies as "fingerprinter" or "safe" whole domains and not individual scripts, the results obtained by the aforementioned project are reported in Table 5.4, in order to compare them to those of the system developed in this thesis.

The development of the automatic detection system presented in this thesis started from the modification of the aforementioned project's code so as to classify single scripts and not whole domains, and in order to detect a larger set of fingerprinting techniques. For the latter purpose, strings corresponding to API calls which are popular in canvas

SVM, K = 4	Precision	Recall	F_1 score	Test set size
Safe	0.83	0.95	0.88	5
Fingerprinters	0.84	0.70	0.73	3
Avg/Total	0.84	0.80	0.82	8

Table 5.4: Results obtained by T. van Zalingen and S. Haanen in "Detection of Browser Fingerprinting by Static JavaScript Code Classification" [21].

fonts enumeration and canvas, AudioContext, WebGL and battery API fingerprinting, have been added to the set of detected strings during the classifiers' features calculation phase. Despite WebRTC-based fingerprinting, which allows the discovery of a device IP address even if it uses a VPN, has been initially taken into consideration, it has been later removed from the set of detected methodologies since it is characterized by a low diffusion (it is found only on the 0.07% of the Alexa[1] top 1 million websites [5]) and also considering that the functions used to obtain such fingerprint are very common among all the scripts using WebRTC, a circumstance which caused many false positives in the developed system.

The classifier's features at this stage were still obtained alternatively from the member expression expansion phase's outputs or from those of the deobfuscation phase, and so without merging the two sets in order to provide the classifier with the maximum amount of information. Moreover, SVM hyperparameters `C`, γ and `class_weight` were respectively set to `1000`, `0.0001` and `balanced`: as described in section 4.3, the mentioned values will be later substituted by those obtained through a cross-validated grid search hyperparameter optimization.

After the described modifications and using 4-folds cross validation, the results on an initial dataset consisting of 43 manually gathered scripts, using features collected from the deobfuscation phase's output were the following:

SVM, K = 4	Precision	Recall	F_1 score	Test set size
Safe	0.84	0.93	0.88	7
Fingerprinters	0.75	0.64	0.69	3
Avg/Total	0.79	0.78	0.78	10

Table 5.5: Initial results on a small dataset. Features collected from the deobfuscation phase's output.

[1] Alexa Website Traffic, Statistics and Analytics: `www.alexa.com`

While using features collected by the member expression expansion phase's output led to the following scores:

SVM, K = 4	Precision	Recall	F_1 score	Test set size
Safe	0.73	1.00	0.84	7
Fingerprinters	0.50	0.28	0.35	3
Avg/Total	0.62	0.64	0.60	10

Table 5.6: Initial results on a small dataset. Features collected from the member expression expansion phase's output.

Features obtained from the member expression expansion's output caused worse results than those obtained by features collected from the deobfuscation phase's output, as the first phase could not correctly expand some expression and, other than their expanded form, it did not produce any additional information about the analysed expressions yet. Comparing these results to those reported for van Zalingen and Haanen's project, it can be noticed that the introduction of new APIs to be detected led to a deterioration of the scores, as the problem became more difficult given the broader range of fingerprinting techniques. The need for a bigger dataset to be used for the classifier's training and testing was indisputable, in order to train better the classifier and to produce more representative results.

5.4 Developments

Given the emerged necessity for a larger amount of scripts, the previously introduced development dataset has been collected. As described earlier, this dataset has been improved during the entire duration of the thesis, gradually increasing its entropy so as to provide a better representation of the scripts' variance found on the web. For this reason, the precision, recall and F_1 scores did often get worse during the thesis development, as the analysed dataset became progressively more difficult to be correctly labelled given the increments in the contained scripts' diversity. Consequently, it has been chosen to present the gradual methodology improvements by calculating the scores progressively obtained by the system on the final version of the development set, in order to avoid misunderstandings caused by possible worsening scores during the system's development.

Other than a bigger dataset, additional improvements have been brought. Indeed, while

the initial classifier used features collected from the deobfuscation or member expression expansion phases alternatively, at this stage the two sets have been merged, so as to have every information obtained by the previous phases at the same time for the classifier to analyse them. Furthermore, some initially detected strings' occurrences like `.width` and `.height`, supposed to match `window.screen.width` and `window.screen.height` calls, or `.name` and `.description`, supposed to match `navigator.plugins.name` and `navigator.plugins.description`, or even `.product`, supposed to match `navigator.product`, were removed from the set of detected strings' occurrences (used as features by the classifier) as they have been found to be too commonly matched on expressions which did not correspond to the mentioned APIs calls. In their place, only the strings representing the complete API calls were left, not particularly common in the deobfuscation phase's outputs but frequently found among the expanded member expressions.

The described enhancements led to noticeably improved scores, made even better by the bigger dataset. Table 5.7 illustrates the results obtained at this stage using 4-fold cross validation.

SVM, K = 4	Precision	Recall	F_1 score	Test set size
Safe	0.91	0.93	0.92	279
Fingerprinters	0.90	0.87	0.89	195
Avg/Total	0.91	0.90	0.90	475

Table 5.7: Results obtained by the developed system with the described improvements on the final development dataset, using the SVM classifier.

At this stage, additional enhancements were introduced in order to further improve the obtained results. The most important modification has been the introduction of loops detection through the AST traversal. This improvement, whose inner workings are described in depth in section 3.2, allowed a more rigorous detection of `navigator.plugins`, `navigator.mimetypes`, JS fonts and canvas fonts enumerations, which represent the most identifying and one of the most common fingerprinting techniques. However, in this phase, other than checking if the corresponding API calls were positioned inside a loop, no other additional conditions were checked, and as a result the detection accuracy could be improved even more as it has been done subsequently. Another improvement introduced at this stage regarded canvas fingerprinting: in order to count an occurrence of `toDataURL()`,

`readPixels()`, `getImageData()`, `toBlob()`, `mozGetAsFile()`, `mozFetchAsStream()` or `extractData()`, it has been established the need for `fillText()` or `strokeText()` to be present too, as explained in section 4.2. SVM's hyperparameters have been optimized too, thanks to the use of cross-validated grid search optimization: `C` was modified from `1000` to `100` and γ from `0.0001` to `0.001`. `class_weight` is the only parameter which has still been left on a different value compared to the final ones presented in the previous chapter (`balanced`), since its modification, in contrast with the cross-validated grid search results, will be explained later.

The new results obtained by the system with the SVM classifier after the described improvements, calculated through a 10-folds cross-validation, has been the following:

SVM, K = 10	Precision	Recall	F_1 score	Test set size
Safe	0.94	0.96	0.95	111
Fingerprinters	0.93	0.92	0.92	78
Avg/Total	0.94	0.94	0.94	189

Table 5.8: Results obtained on the final version of the development dataset by the refined system, using the SVM classifier.

The Random Forest classifier has been introduced at this point in the development phase, in order to test its performances and compare them to those of Support Vector Machines. Its hyperparameters have been optimized through a cross-validated grid search optimization, and even if the development dataset, used when performing the aforementioned optimization, evolved during the thesis' stages, the optimal hyperparameters always remained the stable. However, when comparing the hyperparameters used at this stage with those listed in section 4.3 there are some differences, as those described in the previous chapter does not represent the best hyperparameters obtained through the cross-validated grid search: in particular, at this stage `class_weight` was set to `balanced` and `min_samples_leaf` to `1`, since they represented the best values for the current dataset; however, once that the wild dataset has been introduced successively, the mentioned hyperparameters have been changed in order to better adapt them to the latter scripts set. The RF classifier results obtained at this stage, using a 10-folds cross validation, are illustrated in Table 5.9.

As it can be noticed, its results are similar to those obtained by the SVM classifier in this phase, and only marginally better.

RF, K = 10	Precision	Recall	F_1 score	Test set size
Safe	0.95	0.96	0.96	111
Fingerprinters	0.94	0.93	0.94	78
Avg/Total	0.95	0.95	0.95	189

Table 5.9: Results obtained on the final version of the development dataset by the refined system, using the RF classifier.

The following step has been to test the system, using both classifiers, on the wild dataset, so as to get an estimate of its performance on a set of scripts representing well those which are found in a normal web surfing activity. Unlike the development dataset, the wild dataset did non evolve during the course of the thesis and, more importantly, the contained scripts were not labelled as "fingerprinter" or "safe". For this reason, it has not been possible to produce accurate results when evaluating the system performance over the mentioned dataset, as only an estimation of the obtained scores has been achievable. Indeed, given the lack of labels, a manual inspection of the scripts has been necessary so as to determine if a script utilized fingerprinting techniques or not. Given the size of the analysed dataset and the need for a manual analysis, labelling of all the dataset scripts has not been practicable, and as such the calculated scores based their determination exclusively on the partial amount of scripts which has been analysed.

As formerly described, the classifier's training set originated, for every test, from the development dataset, as it provides a ground truth being composed of correctly labelled scripts. This represented a problem when classifying scripts contained in the wild dataset, since some of them were previously inserted in the development dataset in order to improve its variety and size: in fact, in order to correctly evaluate the performance of a classifier, the elements on which it is tested should not be also present in its training set. In order to avoid this problem, a Python script has been created, able to compare all the elements contained in the designated training and test sets and then remove from the training set those which are also part of the test set. The wild dataset has always been evaluated by dividing it into subsets and analysing them one at a time. Before every test, the aforementioned script has been executed so as to temporarily remove from the training set the scripts which needed to be excluded from the training phase, and later restoring them before the successive evaluation.

The initial results on the wild dataset returned an approximate distribution of 99% safe

scripts to 1% fingerprinting scripts. As a result, the precision score for the fingerprinting scripts resulted the least complicated to be determined, as it bases its value only on those scripts labelled as fingerprinters by the system (both false and true positives), which were definitely less than those classified as safe. In a first analysis over a subset of the wild dataset, consisting of 44,193 scripts, 613 of them (1.39%) have been classified as fingerprinters. The SVM classifier has been used in this phase for the system's classification phase. Of the scripts classified as fingerprinters, 144 have been manually analysed and labelled, obtaining the scores reported in Table 5.10.

Classifier	True positives	False positives	Precision
SVM	104	40	0.72

Table 5.10: Initial precision score for the "fingerprinters" class on the wild dataset.

The calculation of the recall score caused more difficulties: as its value is given from $Recall = \frac{|TruePositives|}{|TruePositives|+|FalseNegatives|}$, an estimation of the number of false negatives was necessary, to be found among the 43,580 scripts which had been classified as "safe" by the system. Also in this case a Python script has been written, in order to help determining the number of false negatives in the system's results. The script returns a list of scripts, contained in the test set, which have been classified as safe even if occurrences of relevant APIs in fingerprinting techniques have been found in their body. In the output list each script name is followed by the set of APIs whose occurrences have been found in the script body, together with the number of occurrences. The list is also sorted by the APIs set size. Initially, only some of the APIs listed in Appendix A were included in this phase since others, like occurrences of `window.screen` or `navigator` properties (excluding those referred to `navigator.plugins` and `navigator.mimetypes`), were not considered sufficient in order to efficaciously fingerprint a device if not complemented with other, more identifying ones. However, as it has been found successively (and will be discussed in section 6.2), in some cases the system was not able to detect a relevant API occurrence, and this led to missed false negatives when trying to estimate them. For this reason, the final version of the script detected the occurrences of every API known to be used in web fingerprinting techniques, in order to minimize the possibility of underestimating the number of false positives.

Using the described methodology, the recall score has been estimated by having the

automatic detection system analyse a set of 27,834 scripts. The results at this stage, considering only the scripts whose classification correctness had been verified manually, were the following:

Classifier	True positives	False negatives	Recall
SVM	69	8	0.90

Table 5.11: Initial recall score for the "fingerprinters" class on the wild dataset.

As it can be noticed scores, and in particular the precision score, faced an appreciable deterioration if compared to those obtained on the development dataset. This is caused by two factors: the distribution of the two classes in the development dataset, characterized by a large unbalance between safe and fingerprinting scripts (circa 99% safe and 1% fingerprinting, as previously mentioned), which causes prominent deterioration in the fingerprinters' precision score even when a low percentage of safe scripts are misclassified by the system, and the considerably higher diversity of the scripts contained in the wild dataset if compared to those in the development dataset, which induces an higher difficulty in the problem.

After the first estimation of the system's recall and precision scores on the wild dataset, the main error causes have been investigated and analysed, in a tentative to bring these scores to the values previously obtained on the development dataset. A first source of inaccuracy has been identified in the enumeration detection technique: as introduced earlier, no additional conditions were present at this stage other than the presence of APIs commonly used in enumeration techniques inside the block of a loop construct. As mentioned in subsection 3.2.2, developers often use these APIs inside loops for different purposes other than producing enumerations: for this reason, at this stage the additional checks described in the same section have been introduced so as to have a more accurate detection of the mentioned fingerprinting methodology. As previously described, plugin and mimetype enumerations are the most identifying fingerprinting technique and, as such, also the one of the most commonly used in fingerprinting scripts, given also the simplicity through which a snippet of code performing these operations can be written. For this reason, given the high diffusion of such techniques in the wild dataset, the newly introduced enhancement in their detection led to a considerable improvement in the system results. Additionally,

some classifiers' hyperparameters have been modified at this stage, given the different distribution of the two scripts classes in the development dataset, used as training set, and the wild dataset, used as test set. Indeed, if in the first set 39% of the contained scripts are labelled as fingerprinters (see Table 5.1), in the latter set only about 1% of them include fingerprinting code. As previously described, the `class_weight` parameter at this stage was set, for both classifiers, to `balanced`: this gave a weight for the two classes which was proportional to their distribution in the training set, which, as previously reported, is definitely different from the subdivision found in the wild dataset, assumed to be more representative of the characteristics of the entirety of scripts disseminated in the web. For this reason, the mentioned parameter has been set to `none`, as the previously assigned weights resulted in better results on the development dataset but worse results on the wild dataset. Finally, the Random Forest's `min_samples_leaf` hyperparameter, which defines the minimum number of samples that should be contained in the produced trees' leafs, has also been modified and set to `10`, which corresponds to its final value reported in section 4.3, as the value returned by the cross-validated grid search hyperparameters optimization caused overfitting over the development dataset.

5.5 Final results

After the introduction of the improvements described above, the final results on all the subsets in which the wild dataset has been previously divided have been estimated. This process required over two weeks of work, as the scripts needed to be manually analysed in order to verify the correctness of the class assigned by the system to each of them by checking for the presence of fingerprinting techniques in their code. This operation has been conducted, despite the considerable amount of work required for this process, aiming at two goals: the first has been the estimation of the system's performances basing their calculation on a significant amount of samples, so as to have a good representation of the performance on the entire dataset; the second has been the improvement of the system performance itself, as this has been the phase in which the development dataset has been enlarged the most by introducing the scripts, initially contained only in the wild dataset (and which, as such, were unlabelled) which were being manually classified. The achievement of a final version of the development dataset, characterized by the dimensions reported in Table 5.1, led not only to more representative results but also to a more accurate

classification by the system since the training set, derived from the development dataset, got enriched with new syntactic constructs which obtained a device fingerprint in different ways than those of the already contained set of scripts.

In total, over 550 scripts have been manually analysed and classified in this phase. The following results were calculated considering only the scripts which have been manually analysed. It must be highlighted that while the number of true positives and false negatives is the same between the system version using the SVM classifier and that using RF, these sets presented some noticeable differences among the two system versions, which will be later analysed in section 6.3.

Classifier	True positives	False positives	False negatives
SVM	413	79	35
RF	413	46	35

Table 5.12: Final TP, FP and FN values for the "fingerprinters" class on the wild dataset.

Classifier	Precision	Recall	F_1 score
SVM	0.84	0.92	0.88
RF	0.90	0.92	0.91

Table 5.13: Final precision, recall and F_1 scores for the "fingerprinters" class on the wild dataset.

In the end, the final results on the development dataset are reported. The related scores faced a deterioration if compared to those obtained previously, before the introduction of the last system modifications which aimed to the attainment of better results on the development dataset. This is caused by the variation of the classifier's hyperparameters, which were previously optimized on the development dataset and have been later modified in order to better fit the wild dataset, and by the low percentage of scripts contained in the first dataset which benefited from the newly introduced checks, in particular those related to the enumerations detection.

SVM, K = 10	Precision	Recall	F_1 score	Test set size
Safe	0.94	0.95	0.94	111
Fingerprinters	0.93	0.91	0.92	78
Avg/Total	0.93	0.93	0.93	189

Table 5.14: Final results on the development dataset, using the SVM classifier.

RF, K = 10	Precision	Recall	F_1 score	Test set size
Safe	0.93	0.97	0.94	111
Fingerprinters	0.95	0.89	0.92	78
Avg/Total	0.94	0.93	0.93	189

Table 5.15: Final results on the development dataset, using the RF classifier.

The discussed deterioration on this dataset has been accepted as the wild dataset, larger and characterized by an higher scripts diversity, has been considered as the main indicator of the system's performance in the wild.

Since the development dataset is entirely labelled, it has been also possible to plot the receiver operating characteristic (ROC) curve, illustrating the system performance as the threshold on the necessary probability to label a script as fingerprinter changes, for both the system's versions using SVM and RF classifiers. Both ROC curves have been calculated by performing 5 and 10-cross validation, so as to avoid overfitting and have an insight on the model's efficacy on an independent dataset.

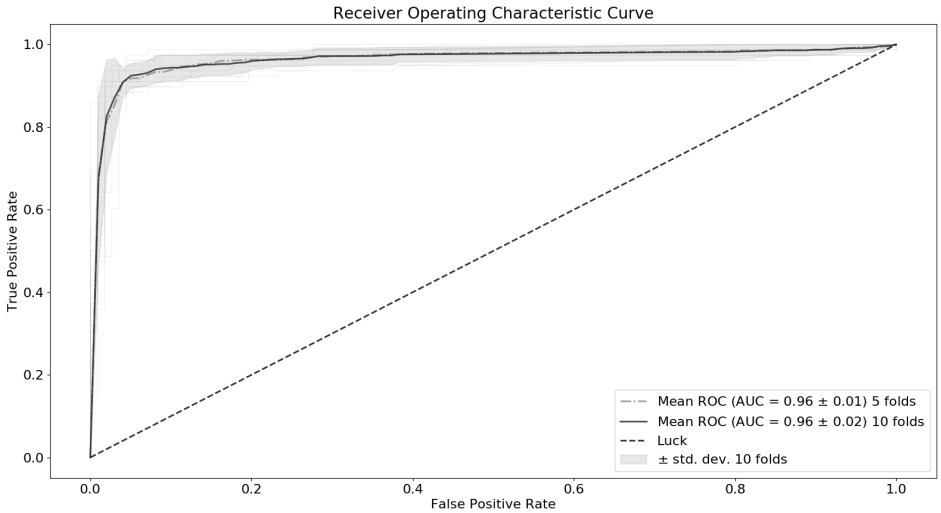

Figure 5.1: ROC curve obtained by the SVM classifier.

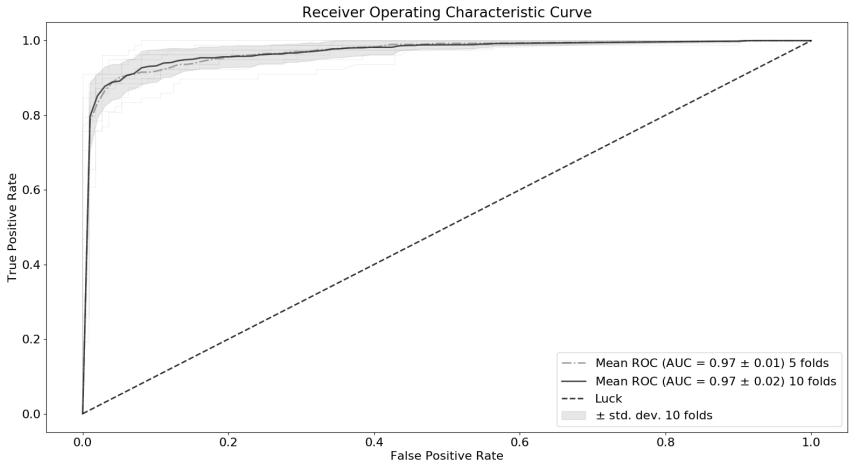

Figure 5.2: ROC curve obtained by the RF classifier.

Finally, the table initially reported in section 5.2 is reproposed, updated with the obtained results in order to provide an overview of the system developments during the thesis work.

Test name	**#Safe & #Fp in test set**	**Results**	**Comment**
Initial test; SVM	#Safe: 7 #Fp: 3 (4-fold CV)	(dob) Precision: 0.75 Recall: 0.64 F_1 score: 0.69 (exp) Precision: 0.50 Recall: 0.28 F_1 score: 0.35	Given the limited dataset size and the lack of any advanced syntax constructs detection, this initial test resulted in modest scores. Table 5.5, Table 5.6.
Development test 1; SVM	#Safe: 279 #Fp: 195 (4-fold CV)	Precision: 0.90 Recall: 0.87 F_1 score: 0.89	The introduction of a much bigger dataset to be used in the training phase and the modifications on the features, removing the unnecessary ones and joining those extracted from different preprocessing phases, resulted in an important improvement. Table 5.7.

Test name	**#Safe & #Fp in test set**	**Results**	**Comment**
Development test 2; SVM	#Safe: 111 #Fp: 78 (10-fold CV)	Precision: 0.93 Recall: 0.92 F_1 score: 0.92	Improvements like loop detection, additional conditions on canvas fingerprinting and new hyperparameters led to better scores on the development dataset. Table 5.8.
RF introduction	#Safe: 111 #Fp: 78 (10-fold CV)	Precision: 0.94 Recall: 0.93 F_1 score: 0.94	Random Forest performed slightly better than SVM on the development dataset, if using the hyperparameters obtained through grid search optimization. Table 5.9.
Wild precision (SVM)	#Total: 43,580	Precision: 0.72	Given the higher script diversity of the wild dataset, the initial precision score on this set has been significantly lower than that obtained on the development dataset. Table 5.10.
Wild recall (SVM)	#Total: 27,834	Recall: 0.90	The recall score on the wild dataset proved to be fairly good from the beginning. Table 5.11.
Final wild scores (SVM & RF)	#Total: 160,951	(SVM) Precision: 0.84 Recall: 0.92 F_1 score: 0.88 (RF) Precision: 0.90 Recall: 0.92 F_1 score: 0.91	After an analysis of the causes for the scores differences between the development and the wild dataset, the system has been improved with new checks and different classifiers' hyperparameters to adapt it to the latter set, considered more important as it better represents the set of scripts which are found in the wild. Table 5.12, Table 5.13.
Final development scores (SVM & RF)	#Safe: 111 #Fp: 78 (10-fold CV)	(SVM) Precision: 0.93 Recall: 0.91 F_1 score: 0.92 (RF) Precision: 0.95 Recall: 0.89 F_1 score: 0.92	Given the modifications introduced in the previous test, the system suffered the new hyperparameters and produced worse, but still valid results on the development dataset. Table 5.14, Table 5.15.

Table 5.16: Overview of the developments and relative tests presented in this chapter. The scores presented in the Results column refer to the "fingerprinting" class.

Chapter 6

Conclusions

In the following, some considerations about the obtained results, the static approach and the performance of the classifiers are made, highlighting advantages and disadvantages and proposing some possible improvements to the developed system and conceivable different approaches for the automatic detection of fingerprinting scripts.

6.1 The datasets and their influence over the system performance

As it can be noticed in section 5.3, section 5.4 and section 5.5, the system scores faced considerable variations based on the datasets which were used for the training and test phases of the classifiers. As noted when the first results on the development dataset has been presented, the final version of the mentioned set has been used for all the reported scores, as its enhancements caused the classification difficulty to increase and as such the results to become worse, and having the system developments producing degrading scores could have generated misunderstandings and lack of clarity about their advantageous impacts on the system performance. However, presenting the results in this way led to a big step in the scores presented in Table 5.7. This was caused by the fact that a major part of the work accomplished during the thesis consisted in the improvement of the cited dataset, as this process led to considerable improvements in the system's ability to correctly classify the multitude of scripts contained in the mentioned dataset or gathered from independent sets.

It has been also discussed the cause of the degraded scores when the wild set has been analysed by the system: the larger number and higher diversity of scripts contained in that set caused the classification to be more complex than on the development set, bringing the same methodology which previously obtained excellent results to poorer ones, before the introduction of the latter improvements.

The development dataset continued to produce different results from those obtained by the wild dataset, and only in the final outcomes they became more similar. The diversity in the mentioned datasets is caused mainly by two reasons: the smaller size of the development dataset, which as described before causes a limited scripts diversity if compared to that of the wild dataset, and the different distribution of the safe and fingerprinting scripts. As reported in section 5.4, the latter motivation caused the precision score on the wild dataset to be easily deteriorated, as small percentages of wrongly classified safe scripts had a noticeable influence over it. However, the distribution of 99 to 1 for safe and fingerprinting scripts, which characterized the wild dataset and as such it has been assumed to be representative of the real balance between the two classes in the web, could not be reproduced in the development dataset, as it would have been necessary a number of safe-labelled scripts which was not feasible to gather in the current work.

Finally, it is remarked the importance of data in the system improvements. While in section 5.4 the described enhancements mainly concern the methodology, the amount of time spent on constantly improving the development dataset, manually labelling a total of 1,540 scripts during the months of work on the thesis project, has been fundamental for the achievement of the final detection capabilities of the system, as this dataset is used for the classifiers' training phase and, as such, it represents the main learning resource of the system.

6.2 Limits of the static analysis

The static approach adopted in this work presents a significant advantage consisting in the possibility to perform the analysis completely offline and without the need for additional dependencies other than the code snippet than is going to be analysed, but, as introduced in chapter 3, it also presents different limitations. In this section the main problems derived from the static nature of the analysis will be presented and discussed.

One of the main issues derived from the static approach is the impossibility to get the

same amount of information about the code and the context in which a particular snippet is going to be executed that could be achievable through a dynamic analysis. The most noticeable example for this problem concerns the fingerprinting techniques based on enumerations: when statically analysing the code, it is possible to detect a loop construct in many ways, and for example this is an operation which in the developed project has been performed through the use of an AST as described in subsection 3.2.2, but it is not possible to observe the number of iterations over the loop, which is an important information in order to distinguish a fingerprinting behaviour from a safe one. This is mostly caused by the fact that other than the variables names, no other information can be gathered about them and, in particular, their value in the section of code in which a possibly fingerprinting construct is identified cannot be known. This information could be crucial, other than in the case of enumerations, in many other unclear circumstances. For example, the detection of canvas fingerprinting is one of the context in which knowing the mentioned information could be decisive in order to increase the accuracy of the system. In this case, in fact, it could be important to know the canvas element's `height` and `width`, the length of the text written into the canvas, or the size of the area specified by a `getImageData()` call [5] so as to detect if the conditions which are necessary for a canvas fingerprinting operation are met or not. Another case where the additional information that can be gathered when performing a dynamic analysis would be important is the detection of JS fonts enumerations, as the number of iterations and the value of the object on which `offsetWidth`, `offsetHeight` or `getBoundingClientRect()` are called could represent critical information in order to distinguish harmless behaviours from fingerprinting ones. In general, the knowledge of variables' values is an important information in order to distinguish accurately fingerprinting scripts from safe ones, and the lack of this information has been the main source of complications during the development of the system presented in this thesis.

Another difficulty derived from the static approach has been discovered during the final phase of the work, when manually analysing the wild dataset's scripts in order to verify the correctness of the class assigned by the classifiers. Indeed, in some cases it has been noticed that a loop construct is not always necessary so as to produce an enumeration, as this could be obtained also by the use of methods which apply a function passed as an argument over each element contained in a set, allowing the production of an enumeration. In these cases the static approach impedes to retrieve more details about the type of object on which such a function is called and on the values of the variables passed as arguments, and even

if the problems derived by these cases could be mitigated, only a dynamic approach would guarantee an high accuracy. Nevertheless, enumerations obtained in the described way have been very uncommon in the dataset, and for this reason the lack of detection in these cases had marginal consequences over the final results.

A further consideration can be expressed about obfuscated scripts. As described in chapter 3, the classification of such scripts has been excluded from the goals of the developed system from the beginning, as it was known that a static approach could not be able to be effective on this type of scripts. Nevertheless, even if this statement remained valid for the majority of the obfuscated scripts, in some cases the system has been able to correctly classify some of them as fingerprinters. In fact, given that this type of scripts were present both in the development and in the wild dataset, the system included them both in the training and test sets of the classifiers, and as such learned from them and assigned a label even for those scripts. While the majority of them has been classified as safe, as the system could not be able to detect the presence of fingerprinting techniques within their code, among the set of scripts classified as fingerprinter some obfuscated scripts have been found which did use fingerprinting techniques. In particular, the SVM classifier has been the one which has been able to detect these cases, while RF never did. The cases in which the system has been effective over obfuscated scripts were those in which the code obfuscation has been obtained through the creation of a string, containing a concatenation of APIs separated by a delimiter, used to dynamically create the real script code and execute it. This caused the member expression expansion phase's output to be meaningless, as the AST could not extract significant information from obfuscated scripts, but it has been possible to identify some noticeable APIs through string matching over the deobfuscation phase's output. These matches, whose occurrences have been counted and used as features during the classification phase, have been enough for the SVM classifier to correctly detect fingerprinting techniques in some of the obfuscated scripts. Despite that, obfuscated scripts have been excluded from the calculation of the system performances, as this has been the chosen policy from the beginning of the project.

6.3 Additional observations

A noteworthy case has been the one regarding scripts belonging to the Matomo analytics service[1] (formerly Piwik): while on the official website it is stated that the service does use fingerprinting techniques in order to produce the advertised analytics, the system never classified any related script as a fingerprinter. By manually analysing the involved scripts, it has been noticed that the fingerprinting attempts consists in the check for the support of 9 pre-determined mimetypes and the collection of the values of the following properties: `navigator.javaEnabled`, `navigator.cookieEnabled`, `navigator.doNotTrack`, `screen.height`, `screen.width`, `navigator.platform` and `window.GearsFactory`. The collected properties are not sufficient in order to uniquely fingerprint a device, as presumably a vast set of users return the same values, and the enumeration of 9 mimetypes is not the most effective way for trying to uniquely identify a device, since the most identifying information would come out of uncommon supported mimetypes, that would be detected by analysing all the the entire set contained in `navigator.mimetypes` and not by imposing a predetermined group of mimetypes on which to perform the check. The developed system did not label the mentioned scripts as fingerprinters because no script contained in the training set and labelled as fingerprinter obtained a fingerprint in the described way. As even after a manual analysis the effectiveness of this technique has not been ascertained, these scripts have been excluded from the calculation of the system performance.

Finally, an observation about the two classifiers' performance is presented. As noticeable in the results reported in section 5.5, in the development dataset the SVM classifier has been able to obtain and higher recall but a lower precision than RF over the "fingerprinters" class, while the latter missed more fingerprinting scripts but also misclassified less safe scripts. This trend has been found also over the wild dataset for the majority of the scripts but, because of a set of similar fingerprinting scripts which were misclassified as safe by SVM while correctly labelled as fingerprinters by RF, the final results over this dataset reports the same amount of true positives and false negatives for the two classifiers. Nevertheless, it has been noticed that while the two scripts did often misclassify the same safe scripts, labelling them as fingerprinters and therefore producing false positives, the false negatives, meaning fingerprinting scripts which were labelled as safe, were often different between the

[1] Matomo, free web and mobile analytics software: https://matomo.org/

two classifiers. This led to the observation that a union of the sets of scripts labelled as fingerprinters produced by the two classifiers could have led to a significant enhancement for the recall score with a limited deterioration of the precision score. Hence the two sets have been joined and the resulting set produced the following scores:

True positives	False positives	False negatives	Precision	Recall	F_1 score
436	101	12	0.81	0.97	0.88

Table 6.1: Results obtained by the union of the "fingerprinters" classes produced by SVM and RF.

Similarly, an intersection of the two classifiers' results would bring to an higher precision and a lower recall score, which may be useful for some applications, but, because of the higher differences among the true positive sets and the similiarity of the false negatives, it would also cause an higher results degradation than the presented union.

6.4 Final thoughts and future work

User tracking on the web is becoming always more pervasive, as analytics services and advertisers gradually refine their tracking mechanisms in order to identify returning users in spite of the possible implementations of countermeasures by the latter. Stateless tracking methodologies have been created and developed during the last years, and faced an important adoption by the aforementioned services. Despite stateful techniques still represent the most widely used and precise tracking mechanism, stateless approaches, which mainly consist in device fingerprinting, offer a crucial advantage consisting in the much higher difficulty to avoid them and, consequently, the lack of control by the tracked users.

Motivated by the implications on users' privacy caused by this trend, in this thesis it has been developed a system which aims at automatically detecting fingerprinting JS scripts found on the web, through a static analysis of the scripts' code. The static nature of the system represents a novel approach to the problem, and despite the previously described limitations, it has proven to be effective at identifying fingerprinting scripts, as shown by the final results in section 5.5. At the current stage, in order to produce an accurate list of fingerprinting scripts, it could still be necessary a revision over the system results, as the related precision, whose score varies from 0.84 to 0.90 on the wild dataset, could not be sufficient in some applications. Nevertheless, the static analysis aimed at the

automatic detection of fingerprinting scripts has proven to be a valid methodology, and while a dynamic approach could solve most of the complications described in section 6.2, it would also introduce the related disadvantages discussed in chapter 3. For these reasons, the two approaches could be complementary for the purpose.

As described in the aforementioned section, in spite of the intrinsic limitations of the static approach, the current system can still be further improved by enlarging the classifiers' training set, refining the set of detected APIs or enhancing the AST traversal phase in order to detect some of the unidentified fingerprinting constructs discussed in section 6.2. Finally, the development of a dynamic analysis methodology to complement the system developed in this thesis would undoubtedly lead to the resolution of the static approach limitations and, as such, to an higher detection accuracy and the possibility to efficaciously analyse even obfuscated scripts.

Appendix A

List of detected APIs

Below it is presented the list of strings, representing APIs which are commonly used in fingerprinting techniques, detected by the feature production phase.

Each string is followed by 0 or 1 (or both), indicating respectively if the string is searched in the member expression expansion output or in the deobfuscation output. Concerning strings which start with a dot, when they are searched in a deobfuscation output file it is also performed a scan for a version without the initial dot, and its occurrences are counted and reported separately from those that include the dot.

```
// Navigator and window.screen
    properties
navigator.appCodeName,0
.appCodeName,0,1
navigator.product,0
navigator.productSub,0
.productSub,1
navigator.vendor,0,1
navigator.vendorSub,0
navigator.onLine,0,1
navigator.appVersion,0
.appVersion,1
navigator.language,0,1
navigator.cookieEnabled,0
.cookieEnabled,1
navigator.javaEnabled,0
.javaEnabled,1
navigator.doNotTrack,0
.doNotTrack,1
window.screen.horizontalDPI,0
.screen.horizontalDPI,1
.horizontalDPI,1
window.screen.verticalDPI,0
.screen.verticalDPI,1
.verticalDPI,1
window.screen.height,0
screen.height,0,1
window.screen.width,0
screen.width,0,1
window.screen.colorDepth,0
screen.colorDepth,0,1
.colorDepth,1
```

```
window.screen.pixelDepth,0
screen.pixelDepth,0,1
.pixelDepth,1
window.screen.availLeft,0
screen.availLeft,0,1
.availLeft,1
window.screen.availTop,0
screen.availTop,0,1
.availTop,1
window.screen.availHeight,0
screen.availHeight,0,1
.availHeight,1
window.screen.availWidth,0
screen.availWidth,0,1
.availWidth,1
navigator.platform,0
.platform,1
navigator.hardwareConcurrency,0
.hardwareConcurrency,1
navigator.cpuClass,0
.cpuClass,1
navigator.maxTouchPoints,0
.maxTouchPoints,1
navigator.msMaxTouchPoints,0
.msMaxTouchPoints,1
navigator.oscpu,0
.oscpu,1
new Date().getTimezoneOffset(),0

// Timezone
.getTimezoneOffset(),0,1
new Date().getTimezoneOffset,0
.getTimezoneOffset,0,1

// AudioContext fingerprint
.createOscillator,0,1
.createAnalyser,0,1
.createDynamicsCompressor,0,1
.getChannelData,0,1
.getFloatFrequencyData,0,1

// Battery API fingerprint
.dischargingTime,0,1
.chargingTime,0,1
.charging,0,1
battery.level,0
navigator.getBattery,0,1
navigator.getBattery(),1
battery.addEventListener,0,1

// Plugin and mimetype enumerations
navigator.plugins-detectedLoop,0
.filename-detectedLoop,0
.description-detectedLoop,0
.name-detectedLoop,0
.version-detectedLoop,0
navigator.plugins.length,0
.plugins.length,1
navigator.mimeTypes-detectedLoop,0
.mimeTypes-detectedLoop,0
.mimeTypes.enabledPlugin-detectedLoop,0
.mimeTypes.description-detectedLoop,0
.mimeTypes.suffixes-detectedLoop,0
.mimeTypes.type-detectedLoop,0

// Canvas fingerprint
.readPixels,0,1
.getImageData,0,1
.toDataURL,0,1
.toBlob,0,1
.mozGetAsFile,0,1
.mozFetchAsStream,0,1
.extractData,0,1

// Canvas font enumeration
.measureText-detectedLoop,0

// WebGl infos
.getExtension("WEBGL_debug_renderer_info"),1
.UNMASKED_VENDOR_WEBGL,0,1
.UNMASKED_RENDERER_WEBGL,0,1
.RENDERER,0,1

// JS font enumeration
.offsetWidth-detectedLoop,0
.offsetHeight-detectedLoop,0
.getBoundingClientRect-detectedLoop,0
.getFontData,0,1
```

Appendix B

WebGL fingerprinting example

The following code sample is part of a Bethesda[1] fingerprinting script which uses multiple fingerprinting techniques, retrieved at https://bethesda.net/shared/core/2/browser.min.js. The reported section concerns WebGL fingerprinting, performed through the collection of a considerable amount of graphic parameters.

```
getWebglCanvas: function () {
   var e = document.createElement("canvas"),
   t = null;
   try {
      t = e.getContext("webgl") || e.getContext("experimental-webgl")
   } catch (e) {}
   return t || (t = null), t
}

var e, t = function(t) {
return e.clearColor(0, 0, 0, 1), e.enable(e.DEPTH_TEST), e.depthFunc(e.LEQUAL),
    e.clear(e.COLOR_BUFFER_BIT | e.DEPTH_BUFFER_BIT), "[" + t[0] + ", " + t[1]
    + "]"
},
n = function(e) {
var t, n = e.getExtension("EXT_texture_filter_anisotropic") ||
    e.getExtension("WEBKIT_EXT_texture_filter_anisotropic") ||
    e.getExtension("MOZ_EXT_texture_filter_anisotropic");
return n ? (t = e.getParameter(n.MAX_TEXTURE_MAX_ANISOTROPY_EXT), 0 === t && (t
    = 2), t) : null
```

[1] Bethesda Softworks: https://bethesda.net/.

```
};
if (e = this.getWebglCanvas(), !e) return null;
var r = [],
o = "attribute vec2 attrVertex;varying vec2 varyinTexCoordinate;uniform vec2
    uniformOffset;void
    main(){varyinTexCoordinate=attrVertex+uniformOffset;gl_Position=vec4(attrVertex,0,1);}",
i = "precision mediump float;varying vec2 varyinTexCoordinate;void main()
    {gl_FragColor=vec4(varyinTexCoordinate,0,1);}",
a = e.createBuffer();
e.bindBuffer(e.ARRAY_BUFFER, a);
var s = new Float32Array([-.2, -.9, 0, .4, -.26, 0, 0, .732134444, 0]);
e.bufferData(e.ARRAY_BUFFER, s, e.STATIC_DRAW), a.itemSize = 3, a.numItems = 3;
var l = e.createProgram(),
c = e.createShader(e.VERTEX_SHADER);
e.shaderSource(c, o), e.compileShader(c);
var u = e.createShader(e.FRAGMENT_SHADER);

r.push("webgl aliased line width range:" +
    t(e.getParameter(e.ALIASED_LINE_WIDTH_RANGE)));
r.push("webgl aliased point size range:" +
    t(e.getParameter(e.ALIASED_POINT_SIZE_RANGE)));
r.push("webgl alpha bits:" + e.getParameter(e.ALPHA_BITS));
r.push("webgl antialiasing:" + (e.getContextAttributes().antialias ? "yes" :
    "no"));
r.push("webgl blue bits:" + e.getParameter(e.BLUE_BITS));
r.push("webgl depth bits:" + e.getParameter(e.DEPTH_BITS));
r.push("webgl green bits:" + e.getParameter(e.GREEN_BITS));
r.push("webgl max anisotropy:" + n(e));
r.push("webgl max combined texture image units:" +
    e.getParameter(e.MAX_COMBINED_TEXTURE_IMAGE_UNITS));
r.push("webgl max cube map texture size:" +
    e.getParameter(e.MAX_CUBE_MAP_TEXTURE_SIZE));
r.push("webgl max fragment uniform vectors:" +
    e.getParameter(e.MAX_FRAGMENT_UNIFORM_VECTORS));
r.push("webgl max render buffer size:" +
    e.getParameter(e.MAX_RENDERBUFFER_SIZE));
r.push("webgl max texture image units:" +
    e.getParameter(e.MAX_TEXTURE_IMAGE_UNITS));
r.push("webgl max texture size:" + e.getParameter(e.MAX_TEXTURE_SIZE));
r.push("webgl max varying vectors:" + e.getParameter(e.MAX_VARYING_VECTORS));
r.push("webgl max vertex attribs:" + e.getParameter(e.MAX_VERTEX_ATTRIBS));
r.push("webgl max vertex texture image units:" +
    e.getParameter(e.MAX_VERTEX_TEXTURE_IMAGE_UNITS));
r.push("webgl max vertex uniform vectors:" +
    e.getParameter(e.MAX_VERTEX_UNIFORM_VECTORS));
r.push("webgl max viewport dims:" + t(e.getParameter(e.MAX_VIEWPORT_DIMS)));
```

```
r.push("webgl red bits:" + e.getParameter(e.RED_BITS));
r.push("webgl renderer:" + e.getParameter(e.RENDERER));
r.push("webgl shading language version:" +
    e.getParameter(e.SHADING_LANGUAGE_VERSION));
r.push("webgl stencil bits:" + e.getParameter(e.STENCIL_BITS));
r.push("webgl vendor:" + e.getParameter(e.VENDOR));
r.push("webgl version:" + e.getParameter(e.VERSION));

r.join("~");
```

Bibliography

[1] Gunes Acar et al. “FPDetective: dusting the web for fingerprinters”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (pp. 1129-1140)* (2013).

[2] Gunes 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* (2014).

[3] Tomasz Bujlow et al. “A Survey on Web Tracking: Mechanisms, Implications, and Defenses”. In: *Proceedings of the IEEE* (2017).

[4] Peter Eckersley. “How Unique is Your Web Browser?” In: *Proceedings of the 10th International Symposium on Privacy Enhancing Technologies (PETS), pages 1-18, Berlin, Germany* (July 2010).

[5] Steven Englehardt and Arvind Narayanan. “Online tracking: A 1-million-site Measurement and Analysis”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016).

[6] Google Inc. *Google Cookie Matching Protocol*. URL: https://developers.google.com/ad-exchange/rtb/cookie-guide.

[7] Benoit Jacob. *Re: [Public WebGL] about the VENDOR, RENDERER, and VERSION strings*. Nov. 2010. URL: https://www.khronos.org/webgl/public-mailing-list/public_webgl/1011/msg00229.php.

[8] Samy Kamkar. *evercookie*. Oct. 2010. URL: https://samy.pl/evercookie/.

[9] Anssi Kostiainen. “Battery Status Event Specification”. In: *Battery Status Event Specification - W3C Working Draft 26* (2011).

[10] Anssi Kostiainen and Mounir Lamouri. *Battery status API*. May 2012. URL: https://www.w3.org/TR/2012/CR-battery-status-20120508/.

[11] Hoan Le, Federico Fallace, and Pere barlet-Ros. "Towards accurate detection of obfuscated web tracking". In: *IEEE International Workshop on Measurement and Networking* (2017).

[12] Hassan Metwalley, Stefano Traverso, and Marco Mellia. "Unsupervised Detection of Web Trackers". In: *Global Communications Conference (GLOBECOM)* (2015).

[13] Keaton Mowery and Hovav Shacham. "Pixel perfect: Fingerprinting canvas in HTML5". In: *Proceedings of Web 2.0 Security & Privacy (W2SP), pages 1-12* (May 2012).

[14] Mozilla. *Battery status API*. 2012. URL: https://developer.mozilla.org/en-US/Firefox/Releases/10.

[15] Łukasz Olejnik, Steven Englehardt, and Arvind Narayanan. "Battery Status Not Included: Assessing Privacy in Web Standards". In: *3rd International Workshop on Privacy Engineering (IWPE'17). San Jose, United States* (2017).

[16] Łukasz Olejnik, Tran Minh-Dung, and Claude Castelluccia. "Selling Off Privacy at Auction". In: *Annual Network and Distributed System Security Symposium (NDSS)* (2014).

[17] Łukasz Olejnik et al. "The Leaking Battery". In: *Data Privacy Management, and Security Assurance (pp. 254-263). Springer, Cham* (2015).

[18] Ashkan Soltani et al. "Flash Cookies and Privacy." In: *AAAI spring symposium: intelligent information privacy management* (2010).

[19] Department of Economic United Nations and Social Affairs. "Population Division (2017) - Data Booklet". In: *World Population Prospects: The 2017 Revision* (2017).

[20] Tim van Zalingen and Sjors Haanen. *Code for the OS3 master System and Network Engineering Research Project 1 report: "Detection of Browser Fingerprinting by Static JavaScript Code Classification"*. 2018. URL: https://github.com/Timvanz/static-javascript-fingerprint-classification.

[21] Tim van Zalingen and Sjors Haanen. *Detection of Browser Fingerprinting by Static JavaScript Code Classification*. Feb. 2018. URL: http://delaat.net/rp/2017-2018/p82/report.pdf.

[22] Webkit. *Changeset 110991. Support for Battery Status API.* 2012. URL: https://trac.webkit.org/changeset/110991.

[23] Shuai Yuan, Jun Wang, and Xiaoxue Zhao. "Real-time Bidding for Online Advertising: Measurement and Analysis". In: *ADKDD '13 Proceedings of the Seventh International Workshop on Data Mining for Online Advertising* (2013).