POLITECNICO DI TORINO
Repository ISTITUZIONALE

Unveiling Web Fingerprinting in the Wild Via Code Mining and Machine Learning

*Original*
Unveiling Web Fingerprinting in the Wild Via Code Mining and Machine Learning / Rizzo, Valentino; Traverso, Stefano; Mellia, Marco. - In: PROCEEDINGS ON PRIVACY ENHANCING TECHNOLOGIES. - ISSN 2299-0984. - ELETTRONICO. - 1:1(2021), pp. 44-63. [10.2478/popets-2021-0004]

*Availability:*
This version is available at: 11583/2852687 since: 2020-11-13T15:15:50Z

*Publisher:*
Sciendo

*Published*
DOI:10.2478/popets-2021-0004

*Terms of use:*

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

*Publisher copyright*

(Article begins on next page)

01 September 2025

Valentino Rizzo*, Stefano Traverso, and Marco Mellia

# Unveiling Web Fingerprinting in the Wild Via Code Mining and Machine Learning

**Abstract:** Fueled by advertising companies' need of accurately tracking users and their online habits, web fingerprinting practice has grown in recent years, with severe implications for users' privacy. In this paper, we design, engineer and evaluate a methodology which combines the analysis of JavaScript code and machine learning for the automatic detection of web fingerprinters.

We apply our methodology on a dataset of more than 400, 000 JavaScript files accessed by about 1, 000 volunteers during a one-month long experiment to observe adoption of fingerprinting in a real scenario. We compare approaches based on both static and dynamic code analysis to automatically detect fingerprinters and show they provide different angles complementing each other. This demonstrates that studies based on either static or dynamic code analysis provide partial view on actual fingerprinting usage in the web. To the best of our knowledge we are the first to perform this comparison with respect to fingerprinting.

Our approach achieves 94% accuracy in small decision time. With this we spot more than 840 fingerprinting services, of which 695 are unknown to popular tracker blockers. These include new actual trackers as well as services which use fingerprinting for purposes other than tracking, such as anti-fraud and bot recognition.

**Keywords:** Tracking, Fingerprinting, Machine Learning, Static Code Analysis, Dynamic Code Analysis

DOI 10.2478/popets-2021-0004
Received 2020-05-31; revised 2020-09-15; accepted 2020-09-16.

## 1 Introduction

Web fingerprinting is an advanced technique for gathering information about users when they browse the Internet. Its deployment is aimed to uniquely identify users without relying on cookies or other kinds of client-side state. Based on information obtained from the browser and device, fingerprinting practice builds precise signatures to uniquely re-identify them across different web services. Fingerprints can be obtained on-the-fly by injecting specialized JavaScript code, which the browser executes in a transparent way to the user.

Nowadays, fingerprinting is a common practice, and it has been fueled by the online advertising industry, which relies on the collection of personal information to design ad campaigns tailored to match users' interests and maximize conversion rates. Fingerprinting has been widely studied by the research community because of its deep consequences [11, 24, 28]. Indeed, it allows trackers to be more precise at recording users' online behavior, putting users' privacy at risk [10, 12, 24].

Nevertheless, fingerprinting is used for purposes other than tracking users [4]. In fact, it allows websites to execute security-related tasks, such as fraud detection, or to discriminate visits generated by bots. For instance, application providers (e.g., web mails, social networks, bank websites, etc.) leverage fingerprinting to track devices linked to accounts, and notify users when their credentials are used from unknown devices.

In this study we aim at understanding web fingerprinting in the wild through two principal contributions. First, we design a scalable and extensible methodology for the identification of fingerprinting script providers (*fingerprinters* in the rest of the paper) based on static or dynamic analysis of JavaScript code and machine learning. Second, we apply such methodology on a peculiar dataset of pages visited by a set of 1,000 actual users during their navigation. This allows us to characterize fingerprinting usage and fingerprinters' penetration in the wild, albeit from a limited point of view. Our approach complements the catalog of tools that have been adopted so far to study fingerprinting which mostly build on dynamic analysis of JavaScript code. Indeed, by comparing our results obtained from static versus dynamic analysis, we demonstrate they provide different and complementary perspectives.

More in detail, the contributions presented in this paper are the following:

- We design, develop and engineer a methodology which combines code-mining techniques and supervised machine learning approaches to automatically detect web

***Corresponding Author: Valentino Rizzo:** Ermes Cyber Security S.R.L., Turin, Italy, E-mail: v.rizzo@ermes.company
**Stefano Traverso:** Ermes Cyber Security S.R.L., Turin, Italy, E-mail: s.traverso@ermes.company
**Marco Mellia:** Politecnico di Torino & Ermes Cyber Security S.R.L., Turin, Italy, E-mail: marco.mellia@polito

cc BY-NC-ND

fingerprinters. Our methodology achieves 94.2% accuracy, with a modest 5.6% false positive rate, and requires about 190ms per-script decision time on off-the-shelf hardware. We compare this against approaches based on dynamic code analysis [7, 11]. Interestingly, we observe that static code analysis spots several *latent* fingerprinting patterns, i.e., pieces of JavaScript code which activate under given circumstances only (e.g., on specific browser versions and settings). We conclude that static and dynamic code analysis are complementary, and should be combined to maximize detection.

- We use our methodology to detect scripts using fingerprinting on a unique dataset of JavaScript files downloaded by real users. 1.1% of scripts are classified as fingerprinters and delivered by 842 unique domains, some of which were previously unknown.
- We deeply characterize fingerprinters, and the techniques they adopt. Surprisingly, most recent and accurate fingerprinting techniques such as Canvas and Audio are the least used (21% and 4% of fingerprinting scripts, respectively), whereas more traditional techniques (e.g., enumeration of plug-ins and MIME types) are the most popular (85% of fingerprinting scripts).
- We match the list of fingerprinters we obtain against a list of tracking domains provided by popular tracker blockers. We identify 695 unknown fingerprinters that combine modern fingerprinting techniques with traditional ones more often than already known systems. This stresses the need for automatic detection systems such as the one proposed in this paper.
- Deepening on the class of unknown fingerprinters, we observe it includes trackers which deliver their scripts directly from the domain of websites hosting them, thus possibly circumventing tracker-blockers. Other services use fingerprinting for non-tracking purposes, such as fraud detection and bot recognition. These observations testify there is a great variety in fingerprinters' ecosystem, and further ingenuity is required to identify actual privacy-offending fingerprinting.

We strongly believe the methodology and the results presented in this paper are interesting for researchers working on online privacy, as well as for developers building privacy-preserving technologies such as anti-tracking blocklists, privacy-aware browsers and tracker blockers. For this reason we share our ground-truth dataset with researchers to stimulate further studies.

The rest of the paper is organized as follows: Section 2 presents a brief overview of known fingerprinting techniques, as well as works related to this study. In Section 3 we present the ground-truths we use for training and testing our classifiers as well as the dataset we obtain from real users. In Section 4 we detail our methodology, its parameter tuning and performance evaluation, and compare static and dynamic code analysis approaches. Then, in Section 7 we present a characterization of fingerprinters based on data obtained from real users. In Section 8 we discuss the limitations of our approach. Finally, Section 9 concludes the paper.

# 2 Background and related work

In this section we describe fingerprinting practice, techniques and countermeasures (Section 2.1 and Section 2.2), and the body of work related to this study (Section 2.3), which we divide in three categories. The first studies its usage and pervasiveness. The second focuses on techniques to identify web fingerprinting. The third proposes countermeasures to mitigate it.

## 2.1 Fingerprinting techniques

Fingerprinting is the process that leverages the browser to collect information about the device running it. Such collection might have multiple purposes. The most privacy-offending is the identification and tracking of browser instances or devices in a *stateless* manner, e.g., without using HTTP cookies. In general, a user encounters fingerprinting scripts during navigation, and, once executed by the browser, these scripts collect and report sets of attributes and properties whose combination is likely unique for the user's device configuration. Considering the specific case of tracking services, fingerprinting makes their activity more difficult to detect and block, as they do not install identifiers on the users' device.

In the last decade fingerprinting techniques have evolved dramatically. This evolution has been driven prominently by trackers' need of identifying users even when, e.g., cookies are turned off. We briefly present the state of the art of techniques in the following.

**Browser Fingerprinting:** It builds on the collection of attributes simply obtainable from users' browser, such as, for instance, installed fonts, plug-ins, MIME types, screen properties, user agent, etc. These can be easily retrieved through `navigator` and `screen` objects, HTTP headers, Java and Flash browser's plug-ins. When combined, this information may build unique identifiers to track users [10, 19, 24], or discriminate visits generated by automatic crawlers [7]. However, this technique has some disadvantages. First, it is prone to instability as fingerprints can change because of browser upgrades or modifications of its configuration (e.g., new font installation). Second, it cannot distinguish browser instances identically configured on different devices [10].

**Canvas and Audio fingerprinting:** These two techniques are more recent than those described above and we group them together as they both build on the same idea of leveraging APIs which, when fed with specific input, return values which vary depending on users' system configuration and hardware. For instance, Canvas fingerprinting builds on rendering text and graphical components on screen areas using the HTML5 Canvas element and on the acquisition of pixel data returned by the corresponding API. More in detail, the fingerprinting script obtains a 2D-graphics context and draws text and figures in the canvas. Then, it executes the `toDataURL()` JavaScript API to obtain a Base64-encoded version of the PNG image rendered from the canvas, which is ultimately used to generate a hash. A similar process can be executed with 3D objects. This simple fingerprinting technique has been demonstrated for the first time by Mowery and Shacham in [21]. Many factors influence the generation of the hash. These are operating system, browser version, graphics card and anti-aliasing management. As a consequence, the hash acts as a unique identifier. Canvas fingerprinting is appealing for trackers as it provides high entropy, it is consistent over time, transparent to the user and fully implementable in JavaScript, thus easy to execute without any special requirement. Nevertheless, hash can change across browsers, and cannot be used to discriminate users sharing the same graphical stack. Audio fingerprinting builds on a similar mechanism which exploits the creation and recording of an audio waveform [11].

**Other fingerprinting approaches:** There exist other, rarer to encounter or less efficient, fingerprinting techniques. *JavaScript Engine fingerprinting* builds on JavaScript conformance tests such as ECMA's Test262 test suite  [3]. As Mulazzani *et al.* show in [22], this method keeps track of failed conformance tests in user's browser to pinpoint its version. *Cross-browser fingerprinting* builds on APIs of hardware made accessible via JavaScript. For instance, specifically for mobile devices, Bojinov *et al.* have shown in [9] how one can measure accelerometer calibration imprecisions to build a signature and uniquely identify the device. This fingerprinting approach detects the device being used, and works independently from the browser in use. *WebRTC* offers features for real-time communication which, in order to find the best route between two communicating endpoints, allows to collect information on IP addresses of interest, including private ones used by local network interfaces. Such information can again be used to further enrich the robustness of a fingerprint [27]. *Battery fingerprinting* consists in taking track of the capacity and status of devices' battery using HTML5 Battery Status APIs. This is another piece of information which one could use to increase entropy and, thus, uniqueness of fingerprints [25]. It is worth mentioning that Firefox, as well as many other browsers, dropped the support for such set of APIs in 2015, thus preventing web fingerprinters to collect data on users' battery status [1].

## 2.2 Countermeasures to fingerprinting

The privacy concerns raised by fingerprinting techniques made users' demand for privacy-preserving solutions grow considerably in the last decade. Countermeasures for private users come often implemented as browser extensions, which can be divided in two main families: *active blockers*, browser extensions which either prevent the browser to execute fingerprinting code (e.g., No-Script[1] and Privacy Badger[2]), or block or alter specific attributes (e.g., Canvas Blocker[3] and Ultimate User Agent[4]). The second family is *URL blockers*. These are extensions, apps or proxies (considering the corporate scenario), which use pre-built blocklists of URLs to prevent the browser to contact fingerprinters, and thus, download and execute their code (e.g., Adblock Plus[5] and Disconnect[3]). Blocklists are usually built manually, and contain domains or URLs which have been found linked to some sort of tracking activity.

## 2.3 Related work

**Fingerprinting pervasiveness:** Many studies have quantified the diffusion of web tracking in the last years. Krishnamurthy and Willis were the first to longitudinally describe tracking services and how they massively increased their presence in the web between 2005 and 2008  [17]. Following studies have shown a worryingly consistent growth of web trackers' pervasiveness [19, 20].

[1] No-Script, https://github.com/hackademix/noscript.

[2] Privacy Badger, https://www.eff.org/privacybadger/faq.

[3] Canvas Blocker, https://github.com/kkapsner/CanvasBlocker.

[4] Ultimate User Agent, http://iblogbox.com/chrome/useragent/alert.php.

[5] Adblock Plus https://adblockplus.org/.

Considering specifically web fingerprinting, Eckersley was the first to provide a comprehensive study of available techniques with EFF's Panopticlick project. In [10] he showed how fingerprinting improves traceability, severely harming users' privacy, especially when Java and Flash plug-ins are installed in the browser.

Nikiforakis *et al.* examined the JavaScript code of popular fingerprinters [24]. They show some techniques are so pervasive they can track the IP address of devices even in presence of HTTP proxies, or install browser plug-ins without the user's authorization. Englehardt and Narayanan presented similar observations in [11].

Trackers keep developing novel techniqes exploiting browsers' data leakage. For instance, Olejinik *et al.* were the first to denounce in [25] how browsers' Battery Status APIs can be used for fingerprinting purposes.

**Identification based on dynamic code analysis:** A number of studies have analyzed fingerprinting and developed identification techniques based on dynamic analysis of scripts contained in webpages. Acar *et al.* built a set of heuristics to understand how browser properties such as `navigator`, `window.screen` and `HTMLElement` are systematically used by trackers to perform browser and device fingerprinting [7]. In 2014, G. Acar *et al.* presented in [6] the results of analyzing the presence of ever-cookies, cookie-syncing practice and Canvas fingerprinting on the 100,000 most popular websites according to Alexa ranking. Interestingly, Canvas fingerprinting was present on 5.5% of considered websites. In 2016, Englehardt and Narayanan conducted a similar experiment on a wider scale, i.e., considering Alexa's top 1M websites [11]. These new results demonstrated that only 1.6% of websites contained some Canvas fingerprinting code, leading to conclude that previous studies induced web trackers to stop using this pervasive technique. Englehardt and Narayanan surveyed methodologies based on Canvas font, AudioContext, BatteryAPI and WebRTC, which all resulted to be less diffused than Canvas fingerprinting.

Section 5 shows that studies based on dynamic code analysis alone tend to underestimate the actual usage of fingerprinting in the web, and in Section 6 we compare our approach with those presented in [7] and [11].

**Identification based on static code analysis:** The number of studies addressing the problem of identifying trackers and fingerprinters using static analysis of JavaScript code is rather limited. Recently, Ikram *et al.* presented in [16] a methodology which combines code mining and machine learning to identify pieces of JavaScript code performing tracking activity – not fingerprinting, though. Ikram *et al.* encountered our same difficulties during the design of their methodology, such as, e.g., the need of manually labeling tracking scripts to build a ground-truth dataset for the classifiers, and the inability to correctly analyze obfuscated scripts.

Van Zalingen and Haanen proposed in [15] an approach that has inspired us in the design of our methodology. Similarly to them, we leverage Abstract Syntax Tree to perform code analysis, and SVM to classify fingerprinting scripts. However, despite these common points, our methodology differs substantially from multiple perspectives. Amongst the most important ones, our machine-learning approach addresses considerably more fingerprining techniques, and has been trained and tested using a much bigger ground-truth dataset containing thousands of JavaScript scripts. In fact, Van Zalingen and Haanen's dataset contains only few tens of scripts. Finally, we achieve much better results.

**Mitigation of fingerprinting:** Other studies propose systems to protect devices from fingerprinting. In particular, by limiting or modifying its execution at run time, when the browser is asked to call specific fingerprinting APIs [8, 13, 14, 23]. In general, these systems alter values provided by fingerprinting APIs by returning either partial or empty sets of properties, randomizing values for attributes such as the screen size or DOM elements' offsets, and introducing noise to the images produced by `HTMLCanvasElement`. However, these approaches could cause the browser to be even more unique and, as such, easier to recognize in a multitude [24].

Browsers themselves have introduced some mechanisms to limit the collection of easily identifiable user data. Laperdrix *et al.* in [18] repeated the experiments Eckersley presented in [10]. By comparing results, Laperdrix *et al.* observed a noticeable reduction of identifying capability of lists of fonts and plug-ins.

This paper advances the state of the art on web fingerprinting by presenting a methodology that complements existing techniques for fingerprinter detection, so it can be used to build curated privacy-preserving blocklists, or extended to identify on the fly specific JavaScript pieces of code to block.

## 3 Datasets

In this section we describe the datasets we collected and use throughout this paper.

## 3.1 Dataset for fingerprinter analysis

Our study builds on a unique dataset containing HTTP and HTTPS logs provided by volunteering users who participated in a web measurement project run by Politecnico di Torino between April and August 2017. Users joining the project provided their explicit consent to install an HTTP/S traffic monitoring tool, Ermes Proxy, on their PCs (running Windows, macOS or Linux) for a period of at least one month. Ermes Proxy acts as a software proxy installed on the device and performs HTTP and SSL inspection. It processes all HTTP/S transactions generated by browsers on the device, and dumps to file the information extracted from headers (i.e., removing all payloads): For each HTTP/S transaction, Ermes Proxy registers the timestamp, the HTTP method, the requested URL, page referer, user agent, and server response status code in a local log, which is periodically uploaded to a remote server.

In total, 982 volunteers installed Ermes Proxy on their PCs for more than 1 month. We collected about 250GB of traffic logs obtained from more than 52M HTTP/S transactions. In the remainder of the paper we will refer to this dataset as *HTTPDataset*.

**Compliance with data protection law:** We performed our data collection in Europe, where since May 2018 any kind of data collection process must be compliant with the European General Data Protection Regulation (GDPR in short) [5]. Despite EU's GDPR was not yet active at the moment of the collection, we used its draft as a policy reference to conduct our measurement collection: Users who joined the data collection have been properly informed about the purposes of the research project; They received a description of the data collection methodology, including information about the risks they could incur into, as well as a description of the mechanisms we adopted to prevent data leaks and guarantee privacy and security. The whole collection was based on the *opt-in* principle, and users voluntarily accepted to join the experiment. Finally, our data collection has been funded by Politecnico di Torino and approved by its Privacy and Security Board.

**Privacy preservation and security:** We designed the measurement collection process to be fully secure and respect users' privacy. HTTPS connections are the most critical from privacy and security perspectives as they often carry information users would like to keep private. For this, Ermes Proxy pseudo-anonymizes logs to mitigate the risk of leaking personal information. In particular, Ermes Proxy does not register any user identifier, nor it stores Personal Identifiable Information (PIIs) such as IP addresses. Plus, to avoid collecting PIIs contained in URLs, it strips query parameters contained in URLs (by removing text after "?"). To prevent attackers to gain access to our data collection, we implement strict security policies. We limit the access to the server hosting data to few authorized people who can access it through selected machines.

We encouraged users to join the project using rewards. They could obtain a gift card to spend on a popular online retailer at the end of the data collection period. To avoid the dataset to be biased or polarized towards specific communities, we advertised our initiative at different events (e.g., fairs, classes, etc.), platforms (Facebook groups), and specific online communities. In the end, the set of participants consists mainly in males (85%) between 18 and 31 years old (91%). Participants are from different parts of Italy, with 19% from the metropolitan area of Politecnico di Torino. The visited websites fairly represent the typical web activity of the average user. In fact, 68.1%, 42.8% and 17.3% of Alexa's top 1,000, 10,000 and 100,000 websites are present in the set of visited websites, respectively.[6] Conversely, 68.8% of visited websites is out of Alexa's top 1M, being these mostly local websites.

We use *HTTPDataset* to collect JavaScript files that we use to design and test our methodology. First, we extract all URLs containing JavaScript scripts by performing a substring search for ".js" files (about 716,000 records). Next, we download each JavaScript file using `wget`, parse it and generate a hash code from its content. We use the hash to identify duplicate files. The final dataset consists of 419,824 JavaScript files, of which 236,217 are unique, from a total of 29,851 different services, that we identify by the domain name in the URL. We refer to this dataset as *JSWild*.

Our dataset provides a different angle compared to data collections like crawling the top websites in rankings (e.g., Alexa), or passively sniffing HTTP traffic. First, it considers regular PCs, each with different browsers, browsing histories, operating systems, and hardware configurations. Second, it factors actual users' habits while they browse websites in which they are interested. Hence, we can analyze scripts from internal website pages, possibly protected by login, yet publicly accessible, as well as scripts downloaded only after explicit user actions (e.g, click on cookie banner).

[6] Alexa Top Websites, https://www.alexa.com/topsites.

## 3.2 Ground truth for static code analysis

To design and test a supervised machine-learning methodology, we need *labeled* data. In our case, labels refer to the script being fingerprinters or not. To this end, we proceed in two phases.

**First phase: building a seeding dataset.** First, we gather JavaScript files connected to websites that have low probability of embedding ads and trackers, and, hence, fingerprinters. Such websites are, for instance, academic and government sites. Then, we integrate this collection with previously known fingerprinting scripts contained in Princeton's WebCensus database [2] which Englehardt and Narayanan built for their experiments [11]. Next, we manually analyze each script to verify whether it contains fingerprinting code or not. For this we proceed as follows:

1) We check if the script is in clear text and easy to read, or at least, minification does not compromise the readability of the code.

2) We check if the script domain corresponds to some tracking service. For this, we use a list of tracking domains, named *TrackerList*, built by merging different lists (Disconnect[7], EasyPrivacy[8] and EasyList[9]).

3) Similarly to the approach used by Englehardt and Narayanan, we check the presence of specific fingerprinting APIs (see Table 3 for the full list).

4) We analyze the code to understand its procedure, the results it would obtain if executed and if these would be sufficient to perform fingerprinting based on the knowledge we have on available techniques, and we extrapolate the context to understand the aim of the code.[10] For instance, for the case of font enumeration, we check the presence of any kind of code implementing the following steps: a) apply browser's default text on a pre-defined text and calculate its size, b) iterate over a list of predefined fonts, and at each cycle, apply a font from the list on the text, c) compare the size of rendered text with the size of text rendered using the browser's default font, d) register the considered font id in a list in case of mismatch (that means, the font is installed as the browser has not used default font as fallback), e) the list is then saved in a structure which is sent to a server or saved in a cookie. At the end of this process we obtain 309 manually verified scripts where 170 are labeled as fingerprinters.

| | Non-fingerprinting | Fingerprinting | Total |
|---|---|---|---|
| #scripts | 1,169 | 733 | 1,902 |
| #domains | 278 | 493 | 752[12] |

**Table 1.** Characterization of ground-truth dataset used to train and test our methodology based on static code analysis *JSStaticGroundTruth*.

**Second phase: extending the manually labeled dataset.** We build on these 309 scripts to build a preliminary version of our classifier. We then use it to quickly classify a subset of $100,000$ randomly picked scripts from the *HTTPDataset*. At the end of this process we again manually check them as described above. In case of ambiguous decision, we skip the script. This results in 563 scripts manually verified as fingerprinters and $1,030$ as non-fingerprinters.

In the end, we obtain in total more than $1,900$ manually validated scripts, 39% of which are fingerprinters. We refer to this dataset as *JSStaticGroundTruth*, as detailed in Table 1. For the sake of transparency, we share with the community this ground-truth to let other researchers verify its content.[11]

## 3.3 Ground truth for dynamic code analysis

To compare our approach based on static analysis against dynamic code analysis we need to build a separate dataset. In fact, dynamic code analysis builds on the availability of data collected during the execution of a piece of code. For our specific purpose, we need to log fingerprinting APIs when executed by the browser. For this, we modify OpenWPM[13], the web crawler introduced in [11] to override JavaScript functions and APIs which are typically used for fingerprinting purposes. Any time a JavaScript script requires the browser to execute one of the instrumented APIs, we log i) the API name, ii) the URL from which the script has been

[7] Disconnect, https://disconnect.me.

[8] EasyPrivacy, https://easylist.to/easylist/easyprivacy.txt.

[9] EasyList, https://easylist.to/easylist/easylist.txt.

[10] We remark that we label as fingerprinters all scripts which do contain actual fingerprinting code, and not for some mere feature probing.

[11] Ground-truths available at https://www.pimcity-h2020.eu/publication/unveiling-web-fingerprinting-in-the-wild-via-code-mining-and-machine-learning/ .

[12] Some domains host both non-fingerprinting and fingerprinting scripts. As such, the total number of domains is different from the sum of the number of domains in the two classes.

[13] OpenWPM, https://github.com/mozilla/OpenWPM.

| | Non-fingerprinting | Fingerprinting | Total |
|---|---|---|---|
| #scripts | 206 | 222 | 428 |
| #domains | 101 | 161 | 262 |

**Table 2.** Characterization of the ground-truth dataset used to train and test our classifier based on dynamic analysis *JSDynamicGroundTruth*.

downloaded, and iii) the execution timestamp, and we save the JavaScript file.

We instruct our OpenWPM crawler to visit the top 1M websites of the Alexa rank. For each website, we visit the landing page and five internal links at random. The measurement campaign took 94 days to navigate all websites using 8 browser instances running in parallel on 2 AWS machines equipped with a 2.5GHz CPU core and 8GB of RAM. In total, we collect 9.8M scripts and 111GB logs of calls to JavaScript APIs. Out of this dataset, we randomly extract a sample consisting of about 156,000 scripts connected to 17,000 websites. In the rest of the paper, we refer to this dataset as *JSAlexa*.

At last, we create a second labeled dataset that we use to train and test a classifier based on dynamic code analysis data, capable of detecting the same set of fingerprinting techniques identified by our static analysis. This second ground truth, namely *JSDynamicGroundTruth*, consists of 428 scripts downloaded by our crawler, that we manually verified using a procedure similar to that used to build *JSStaticGroundTruth*. We summarize *JSDynamicGroundTruth* in Table 2.

# 4 Classification using static code analysis

In this section we present the methodology we design to automatically identify JavaScript files performing fingerprinting based on static analysis of their code. It builds on the assumption that we can determine whether a script performs fingerprinting by checking the presence of code patterns typically employed to perform fingerprinting. Our methodology consists of four steps that we describe in the following.

## 4.1 Code de-obfuscation and beautification

In order to minimize their footprint, JavaScript files are usually delivered to the browser in a minified form, which replaces variable and function names with short random strings and wipes out unnecessary spacing. Because of this process, minified scripts are hardly human-readable, and, thus, difficult to analyze. Sometimes code is obfuscated too. Code obfuscation enables the protection of intellectual and industrial property, but it makes reverse engineering impracticable, with severe consequences for privacy and security [29]. In this phase we first de-minify JavaScript files using the same approach used in [15], which combines JSBeautifier[14] with a set of tools developed specifically for this purpose. Then, we attempt to perform de-obfuscation, which allows us to maximize the amount of scripts to process. More in detail, de-obfuscation is effective against a number of obfuscating tools and techniques: JavaScript Obfuscator[15], Dean Edward's *packer*[16] and url-encoding. However, our de-obfuscation approach cannot succeed when strong obfuscation techniques are employed. We discuss the impact of this limitation on our analysis in Section 8.

## 4.2 Syntactic structure analysis and string-matching search

We analyze the code's syntactic structure by generating an Abstract Syntax Tree (AST). Given a script, its AST provides a tree representation of the syntactic structure of the code. Each node of the tree describes a code construct in the script, and integrates data about the type of construct, position and construct-specific properties. In other words, we obtain information describing what would happen when the browser executes the analyzed code. To accomplish this task we build on *Esprima*[17] to generate the AST, and *Estraverse*[18] to analyze it.

## 4.3 Fingerprinting patterns

The features we use for classification build on APIs used by scripts performing fingerprinting. In Table 3 we re-

[14] JSBeautify, https://github.com/beautify-web/js-beautify.

[15] JavaScript Obfuscator, https://javascriptobfuscator.com.

[16] Dean Edward's packer, http://dean.edwards.name/packer.

[17] Esprima, http://esprima.org/.

[18] Estraverse, https://github.com/estools/estraverse.

port the list of APIs we consider in this study, grouped by category. Since the presence in the code of one of such APIs is not sufficient to determine the presence of fingerprinting (APIs can be used for different purposes), we must define *fingerprinting patterns* around them. A pattern is a portion of JavaScript instructions containing one or more calls to APIs in Table 3, and satisfying specific conditions. For instance, non-fingerprinting scripts often use Canvas APIs, hence we define specific criteria to determine when Canvas APIs, e.g., `toDataURL()`, are likely to be used for fingerprinting. For instance, Canvas fingerprinting is based on the presence of micro-differences between the image provided as input and the rendered one. Thus, we create a pattern for which we check that the image format used by `toDataURL()` to output the figure is lossless, since a lossy one would be deprived of the needed details to perform fingerprinting. We define patterns for APIs detected with both AST and string-matching approaches. For the AST, we create in total 79 patterns ranging from simple enumerations of `navigator` properties to calls to specific APIs (`toDataURL()`, `createOscillator()`, etc.).

Figure 1 reports an example of fingerprinting code, together with a graphical representation of part of the corresponding AST. For the sake of brevity, we omit to report the full AST and hide irrelevant sub-trees (drawn with dashed contour). The figure shows how we employ the fingerprinting pattern we developed to detect plug-in enumeration leveraging the AST. Our pattern consists of the blocks highlighted in red in the figure. In particular, we record the variable declaration in line 1 where `navigator.plugin` is called, the `for` statement and the subsequent calls to `navigator.plugin`'s properties `version` and `name`. The final fingerprinting pattern also verifies that the properties are not compared to static strings or used under a conditional expression. If all these conditions are met, we finally record the pattern as a possible enumeration attempt, and create the relative feature to be used by the classifier.

Unfortunately, AST generation (often) fails on obfuscated or malformed code. In fact, it fails on 7.33% of the processed scripts contained in *JSStatic-GroundTruth*. In this case it either produces an incomplete result (3.67%), or it does not produce a result at all (3.66%). To mitigate this issue, we also rely on simple string-matching approach to register the presence of fingerprinting patterns in obfuscated scripts. Indeed, despite obfuscation, it is common that some calls to fingerprinting APIs may be still present in clear text, and, thus, easy to detect. For string matching, we build 63 fingerprinting patterns, i.e., all those designed for AST, except enumerations (fonts, plug-ins and MIME types) as string matching does not allow us to verify conditions for enumeration-based fingerprinting techniques.

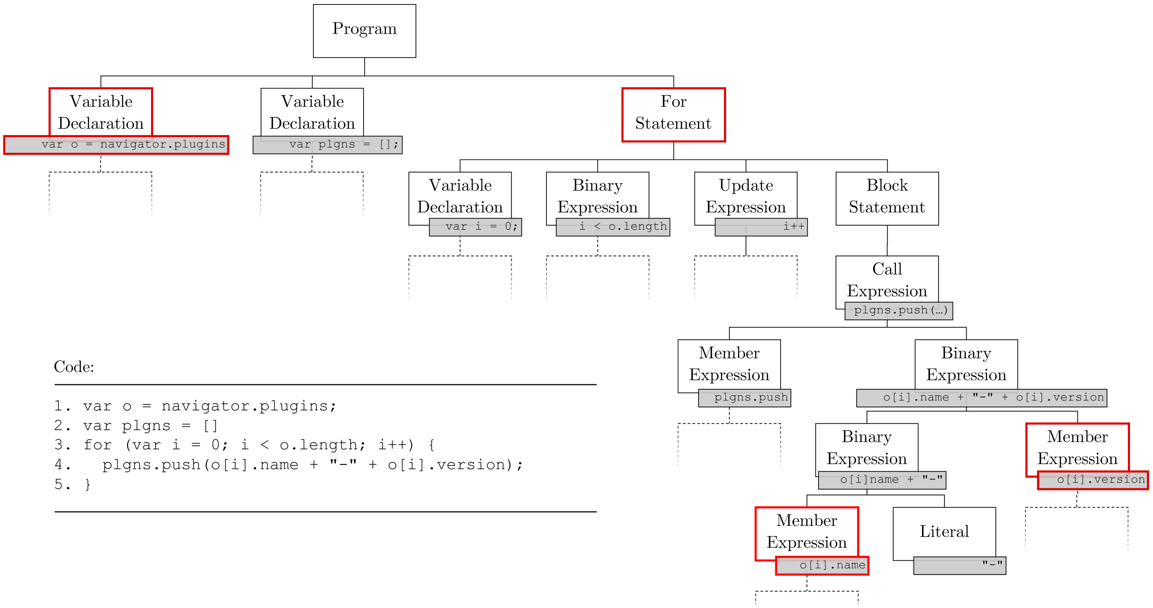
## 4.4 Classification with machine learning

Given the multidimensional space and variety in AST data, machine learning is both crucial and a natural choice to accelerate the creation of reliable classifiers. Moreover, machine learning algorithms offer explanations on why a decision has been taken, e.g., showing the most important features that drive a decision. Hence, we rely on supervised machine learning to train classifiers able to distinguish fingerprinting from non-fingerprinting JavaScript code. To do so, we first define the features to be used as input, then we select a proper supervised classifier, train and optimize parameters to ultimately maximize classification performance. In the following, we describe each step.

**Feature engineering:** For each fingerprinting pattern identified in the AST or by using string-matching, we create a feature based on its number of occurrences. In details, given a JavaScript file, we create a map of key-value pairs where fingerprinting patterns are the keys, and the corresponding number of occurrences in the file are the values. We then use such features to train and test two supervised machine-learning classifiers.

**Machine learning model selection:** Given the limited size of the labeled dataset, we select Support Vector Machines (SVM) and Random Forest (RF) as models to train. We choose these because they are considered among the best performing models with relatively small datasets, and they build on substantially different approaches to classification. Both consider non-linear models. SVM has been chosen because it captures complex relationships between samples, without requiring us to perform cumbersome data transformations. RF has been selected because of its flexibility and its capability to automatically perform feature selection. RF models are also easy to understand and interpret, and this considerably helps us to debug and improve the methodology. We excluded other, more complex, models such as those based on Neural Networks or Associative Rules. In fact, these models perform very well when fed with large amounts of data, which is not our case. For the implementation, we use Scikit-Learn [26], a popular Python library for machine learning.

**Classification performance indices:** To evaluate the performance of the trained models, we rely on standard classification indices such as precision and recall. Given



```
var o = navigator.plugins;
var plgns = []
for (var i = 0; i < o.length; i++) {
  plgns.push(o[i].name + "-" + o[i].version);
}
```

**Fig. 1.** Example of JavaScript code performing fingerprinting using plug-in enumeration, and the graphical representation of part of corresponding Abstract Syntax Tree. Red blocks represent the fingerprinting pattern we developed to detect plug-in enumeration.

a class label, they are defined as follows:

$$Precision = \frac{|\,true\ positives\,|}{|\,true\ positives\,| + |\,false\ positives\,|} \quad (1)$$

$$Recall = \frac{|\,true\ positives\,|}{|\,true\ positives\,| + |\,false\ negatives\,|} \quad (2)$$

Precision evaluates the classifier's exactness, whereas, recall represents its completeness. We also consider the accuracy, a standard measure of the generic goodness of the classifier, irrespective of the considered class.

$$Accuracy = \frac{|\,true\ positives\,| + |\,true\ negatives\,|}{|\,all\ items\,|} \quad (3)$$

We conduct a thoughtful tuning of parameters for each classifier to identify the configurations providing the best results. We report details in Appendix A.1.

Finally, we analyze feature importance assigned by the RF classifier. This gives a hint about which APIs RF considers to be determinant for classification. Interestingly, all most important APIs come from Browser Parameters, Plug-in and MIME Types and Screen categories, while Canvas' API `toDataURL()` looks considerably less determinant. As described in detail in Section 7, this reflects the actual usage of fingerprinting techniques in a real scenario.

## 4.5 Final performance evaluation

We now report classification results in details. Table 4 details the per-class precision, recall, and the overall accuracy for the best hyperparameters configurations. Again, to avoid overfitting, all results are averaged over a 10-fold cross validation using the best performing hyperparameters. First, we observe that all indices for both SVM and RF overcome 90%. Interestingly, both classifiers reach a large recall at recognizing non-fingerprinting scripts, but this decreases for fingerprinting scripts. This means that it is more likely for both classifiers to mis-classify a true fingerprinting script than mis-classifying a true non-fingerprinting. In other words, actual non-fingerprinting scripts are easier to classify. This confirms the intuition that a non-fingerprinting script rarely uses patterns used by fingerprinters. As such, it is easier to classify them. Nevertheless, some patterns used by fingerprinting scripts are also seldom used by non-fingerprinters. This is the case for instance of scripts that enumerate plug-ins to check whether to execute code blocks. In general, this means that separating the two classes is not trivial, and both classifiers create robust models.

We manually check the sets of misclassified fingerprinting scripts produced by SVM and RF, and find that they rarely overlap. That is, each classifier tends to fail on different sets of fingerprinting scripts, while the sets of misclassified non-fingerprinting scripts share many elements. Since we aim at maximizing the probability of identifying actual fingerprinters (i.e., maximizing true positives), we can combine the output of both classifiers – SVM ∪ RF in the table – so that we consider a

| Categories | Browser Parameters | Plug-ins & MIME types | Screen | WebGL Parameters | Canvas |
|---|---|---|---|---|---|
| Set of APIs | navigator.appCodeName<br>navigator.product<br>navigator.productSub<br>navigator.vendor<br>navigator.vendorSub<br>navigator.onLine<br>navigator.appVersion<br>navigator.language<br>navigator.cookieEnabled<br>navigator.javaEnabled<br>navigator.doNotTrack | navigator.plugins<br>navigator.plugins.length<br>navigator.plugins[i].name<br>navigator.plugins[i].filename<br>navigator.plugins[i].description<br>navigator.mimeTypes<br>navigator.mimeTypes.length<br>navigator.mimeTypes[i].enabledPlugin<br>navigator.mimeTypes[i].description<br>navigator.mimeTypes[i].suffixes<br>navigator.mimeTypes[i].type | window.screen.height<br>window.screen.width<br>window.screen.colorDepth<br>window.screen.pixelDepth<br>window.screen.availLeft<br>window.screen.availTop<br>window.screen.availHeight<br>window.screen.availWidth<br>window.screen.deviceYDPI<br>window.screen.deviceXDPI<br>window.screen.systemXDPI<br>window.screen.systemYDPI<br>window.screen.logicalXDPI<br>window.screen.logicalYDPI<br>window.screen.updateInterval | getExtension("WEBGL_debug_renderer_info")<br>UNMASKED_VENDOR_WEBGL<br>UNMASKED_RENDERER_WEBGL<br>RENDERER<br>ALIASED_POINT_SIZE_RANGE<br>ALIASED_LINE_WIDTH_RANGE<br>MAX_RENDERBUFFER_SIZE<br>MAX_CUBE_MAP_TEXTURE_SIZE<br>MAX_COMBINED_TEXTURE_IMAGE_UNITS<br>MAX_TEXTURE_IMAGE_UNITS<br>MAX_TEXTURE_SIZE<br>MAX_VARYING_VECTORS<br>MAX_VERTEX_ATTRIBS<br>MAX_VERTEX_TEXTURE_IMAGE_UNITS<br>MAX_VERTEX_UNIFORM_VECTORS<br>MAX_VIEWPORT_DIMS | readPixels<br>getImageData<br>toDataURL<br>toBlob<br>mozGetAsFile<br>mozFetchAsStream<br>extractData<br>fillText<br>strokeText |

| Categories | Audio | Hardware Information | Timezone | Font |
|---|---|---|---|---|
| Set of APIs | createOscillator<br>createanalyzer<br><br>createDynamicsCompressor<br>getChannelData<br>getFloatFrequencyData | navigator.platform<br>navigator.hardwareConcurrency<br>navigator.cpuClass<br>navigator.maxTouchPoints<br>navigator.msMaxTouchPoints<br>navigator.oscpu<br>window.devicePixelRatio | getTimezoneOffset | measureText<br>offsetWidth<br>offsetHeight<br>getBoundingClientRect<br>getFontData |

**Table 3.** Grouping of JavaScript APIs used for fingerprinting purposes.

| Classifier | Class | Precision | Recall | Accuracy |
|---|---|---|---|---|
| **SVM** | Non-fingerprinter | 0.939 | 0.954 | 0.939 |
| | Fingerprinter | 0.937 | 0.916 | |
| **RF** | Non-fingerprinter | 0.931 | 0.968 | 0.940 |
| | Fingerprinter | 0.955 | 0.903 | |
| **SVM ∪ RF** | Non-fingerprinter | 0.958 | 0.941 | 0.942 |
| | Fingerprinter | 0.922 | 0.944 | |
| **SVM ∩ RF** | Non-fingerprinter | 0.914 | 0.982 | 0.936 |
| | Fingerprinter | 0.973 | 0.875 | |

**Table 4.** Classification results using SVM and Random Forest obtained using our ground-truth dataset *JSStaticGroundTruth*.

script as fingerprinter if at least one of the two classifiers returns a positive match. This choice increases the recall, but lowers the precision (since we accept some more false positives). The performance results of this combination in Table 4 shows that precision on the fingerprinter class decreases a little, but recall gets closer to 0.95. In the remainder of the paper we use this combination of SVM and RF. We also show what happens if one favors precision over recall, i.e., SVM ∩ RF. In this case, the overall accuracy is still high, but as classification gets more precise at identifying actual fingerprinting scripts, it mis-classifies many of them as non-fingerprinting. Contrarily, almost all non-fingerprinting scripts are correctly labeled as such (recall equals 0.982), but with less precision (0.914).

At last, we describe in Appendix A.2 the experiments we conducted to ensure our classifiers are not affected by overfitting.

## 5 Classification using dynamic code analysis

Static analysis of JavaScript code fails in case of code obfuscation or in case of malformed JavaScript. We quantified the extent of this limitation in Section 4.3. A possible solution to overcome this limit is using dynamic code analysis, which is performed by executing the code under exam and observing its operation, by, e.g., keeping track of instructions being executed. This approach allows us to understand the actual workflow of a piece of code, even in case its source code is not available (e.g., the program is compiled or its code is obfuscated).

Dynamic code analysis has some significant drawbacks too. First, code obfuscation prevents manual validation of classification based on dynamic analysis. Indeed, we cannot verify the classification verdict obtained with dynamic code analysis when a script is irreversibly obfuscated, i.e., the code is unreadable and impossible to understand. Second, it fails with scripts programmed to prevent the execution of (portions of) code. For instance, some fingerprinting patterns are executed under specific conditions (e.g., browser in use, visited website, installed plug-ins, etc). Finally, dynamic code analysis is resource consuming because it requires ad hoc analysis frameworks to keep track of executed APIs and analyze the execution workflow.

Hence, static and dynamic code analysis show some pros and cons, and we use them to complement each other as they lead to different visibility on code structure and execution. Here we aim to understand how the two approaches actually compare and complement: we augment our methodology to process data gathered using dynamic analysis of JavaScript code, and compare its performance against the static analysis approach.

### 5.1 Classifier engineering

We adapt the methodology described in Section 4.4 to process logs in *JSAlexa*. For each script, we count at runtime the occurrences of APIs in Table 3. Once completed, features reporting the name of the API and number of its occurrences are used to feed a classifier which is responsible for distinguishing fingerprinting scripts.

For training and testing our classifier we use the same steps described in Section 4.4 using ground-truth *JSDynamicGroundTruth*. For the sake of simplicity, we use a Random Forest as classification engine. The resulting Random Forest achieves a 0.952 precision on the fingerprinting class, which is in line with results obtained on *JSStaticGroundTruth*.[19]

### 5.2 Comparison

We apply both methodologies on the about 156, 000 scripts contained in *JSAlexa*. We observe the following results: First, we observe that classification based on static code analysis tags 1764 scripts as fingerprinting: of these, 692 present only slight differences from the others, mostly consisting in custom identifiers and settings used by trackers, resulting in 1072 actually different fingerprinters. As shown in Table 5, only 794 are labeled as such by dynamic code analysis. Such difference is caused by the intrinsic differences of these two approaches. In fact, dynamic analysis can identify fingerprinting patterns only if these have actually been executed by the browser. However we observe a considerable amount of fingerprinting scripts which contain *latent* fingerprinting patterns, i.e., portions of code executed under given conditions (e.g., "browser is Google Chrome", "cookies are not enabled", etc.).[20] Only static analysis can capture these. For these reasons, of the 1072 scripts labeled as fingerprinters using static analysis, the dynamic one agrees in 678 of the cases whilst 394 are "missed", i.e., roughly 30% of scripts. Second, of the 794 fingerprinters detected using dynamic analysis, 116 are "missed" by static analysis.[21] By manually inspecting them, we observe most of these are deeply obfuscated scripts which do actually generate logs of fingerprinting patterns at execution time. These are impossible to examine with static analysis, which misses then about 15% of scripts.

In summary, the results of this experiment show that static and dynamic code analysis complement each other and both approaches must be considered to achieve a reasonable trade-off between accuracy, development costs and resource footprint.

## 6 Evaluating state of the art

Now we run experiments to gauge how our approach improves the state of the art. For this, we compare the performance achieved by our classifiers against the two most prominent solutions for fingerprinter detection, FPDetective [7] and Princeton's heuristics [11].

Considering FPDetective, unfortunately, the open-source repository hosting its code is no longer maintained and the code building the system is obsolete. Hence, to reproduce the behavior of FPDetective, we implement the heuristics presented in Section 4.2 of its paper [7]. Notice that, since it was built back in 2013, FPDetective does not consider many fingerprinting techniques which have been introduced in the recent years: Canvas, WebGL, Audio fingerprinting. Nevertheless, it contemplates now dead Flash technology.

Considering Princeton's proposal, also in this case we cannot use the code contained in the repository provided in [11]. This hosts the code of the web scraper, OpenWPM, but not the implementation of the heuristics described in the paper. Hence, we can only try to

[19] We follow the same grid-search-based approach described in Appendix A.1 to optimize hyperparameters.

[20] There exist advanced techniques for dynamic analysis whose aim is to comprehensively examine all execution workflows contained in code. However, such techniques (e.g., *concolic testing*) are extremely resource consuming, and hard to automatize and implement in the browser environment.

[21] These numbers are not reflected in Table 5 as the numbers reported there are computed considering Princeton's and FPDetective's approaches too.

| ∩ | Static | Dynamic | Princeton | FPDetective |
|---|---|---|---|---|
| **Static** | **1072** | - | - | - |
| **Dynamic** | 678 | **794** | - | - |
| **Princeton** | 260 | 307 | **330** | - |
| **FPDetective** | 115 | 131 | 101 | **199** |
| **Exclusive** | **388** | **66** | **23** | **62** |
| **Coverage** | **88%** | **65%** | **27%** | **16%** |

**Table 5.** Fingerprinters found in *JSAlexa* using static and dynamic code analysis, as well as Princeton's approach [11] and FPDetective [7]. Each cell reports the number of fingerprinters detected by both approaches on corresponding row and column. Each cell in "Exclusive" row reports the number of fingerprinters detected exclusively by the approach on relative column. Each cell in "Coverage" row reports the percentage of fingerprinters detected by the approach on the corrisponding column, computed over the total number of unique detected fingerprinters (1223).

reproduce the heuristics described in Sections 6.1-6.5 of [11]. Unfortunately, three fingerprinting techniques described in the paper are not reproducible. In details,

- **Audio:** The authors simply list the Audio APIs which they monitored, but they do not provide any actual algorithm to conclude whether a script is performing Audio fingerprinting or not. This lack of information prevents us from implementing this heuristic.
- **WebRTC:** This heuristic is not meant to be automatic. The paper describes which APIs authors monitored to check WebRTC usage, but the test to understand whether this was used for fingerprinting purposes (i.e., tracking IP addresses) is performed manually.
- **Battery:** also in this case the authors do not detail any algorithm to check when fingerprinting based on Battery properties is performed. They simply list the properties to monitor, and claim the fingerprinting purposes are confirmed by the presence of other techniques.

As a result, the only heuristics we can reproduce entirely are those for detecting Canvas and Canvas Font fingerprinting. However, we remark that the usage of Audio, WebRTC and Battery APIs for fingerprinting is quite infrequent: as we show in Section 7, Audio is used in about 4% of fingerprinting scripts. According to [11], WebRTC has been observed in 0.7% of websites in the top 1M, and Battery is used by just 2 scripts. Hence, we are confident that the lack of these three techniques in our implementation of Princeton's heuristics does not influence conclusions presented in the following section.

## 6.1 Comparison

As done in Section 5, we use again *JSAlexa* to compare the performance of our classifiers against FPDetective and Princeton's heuristics, and report the results in Table 5. As shown, FPDetective and Princeton's heuristics identify 199 and 330 fingerprinters, respectively. All in all, focusing on the percentage of fingerprinters detected by each approach ("Coverage" row), we observe Princeton's heuristics and FPDetective can spot only 27% and 16% of overall 1223 fingerprinters, respectively.

Checking the fingerprinters identified exclusively by these approaches, by manual verification we have that of the 23 scripts labeled as fingerprinters by Princeton's heuristics, 2 are true positives, 18 are false positives, and 3 are obfuscated scripts impossible to check. Similarly, FPDetective labels as fingerprinters 62 scripts which are not captured by other approaches, but 34 of these are clear false positives. Of the 25 true positives, many present very similar content, and they can be collapsed to just 2 scripts, reducing the FPDetective's additional true positives down to 5 scripts. In the end, Princeton's heuristics and FPDetective together detect 7 new true positives corresponding to actual false negatives of our classifiers. For instance, FPDetective concludes `moatad.js`[22] is fingerprinting as it performs plug-in enumeration and checks the status of the battery (possibly for bot recognition). Similarly, Princeton's heuristics label the script `tfav_adl_347.js`[23] as fingerprinting because it performs font enumeration. In both cases, our RF model does not achieve the same conclusion.

The results of these old methodologies are rather poor and expected: First, both do not cover all fingerprinting techniques considered by our classifiers. Second, they both build on dynamic code analysis only, thus, they cannot spot latent fingerprinting APIs.

# 7 Fingerprinting in the wild

Now we leverage our *JSWild* dataset to analyze the adoption of fingerprinting scripts in the web. We first quantify the actual spread of fingerprinting in JavaScript files, and analyze which techniques are the most used in Section 7.1. Then, we study how they are used in combination in Section 7.2. Finally, in Sec-

[22] https://z.moatads.com/martinwilliamssyngenta953159580698/moatad.js

[23] https://j.adlooxtracking.com/ads/js/tfav_adl_347.js

| Class | Total scripts | Unique scripts | Domains |
|---|---|---|---|
| Non-fingerprinting | 415280 (98.9%) | 231543 (98.9%) | 29678 |
| Fingerprinting | 4544 (1.1%) | 2674 (1.1%) | 842 |

**Table 6.** Fingerprinting scripts identified by SVM and Random Forest classifiers combined as shown in Section 4 on *JSWild*.

tion 7.3, we study the pervasiveness of domains associated to fingerprinting activity, and whether known trackers adopt specific fingerprinting techniques.

## 7.1 Analyzing use of fingerprinting APIs

We run our classifier on all JavaScript files contained in *JSWild* dataset. Table 6 shows the results. Only 1.1% of scripts is classified as actual fingerprinting.

Since the same APIs used for fingerprinting can be used for purposes other than tracking (e.g., Canvas APIs can be used to render actual images, or Audio APIs can be used to play a piece of music), we aim to analyze whether fingerprinters use some specific APIs. For this, we leverage the coarse grouping reported in Table 3, and represent in Figure 2 the percentage of non-fingerprinting (cyan) and fingerprinting (red) JavaScript files that use APIs belonging to different categories. First, we notice that non-fingerprinting scripts do employ APIs used for fingerprinting, but with a much lower probability. In fact, a not-negligible percentage of non-fingerprinting scripts employs these APIs: more than 15% and 8% of non-fingerprinting scripts use Browser Parameters and Screen APIs, respectively. Given the large cardinality of non-fingerprinting scripts, these fractions correspond to considerably large numbers. Second, API usage fractions are quite heterogeneous. Indeed, basically all fingerprinters use Browsers Parameters, while Canvas and Audio APIs are used less frequently, i.e., in 20% and 3% of cases, respectively.

## 7.2 Analyzing usage of fingerprinting API combinations

Fingerprinters usually combine different techniques to obtain high-entropy identifiers and improve tracking accuracy, and we now aim to understand how fingerprinting APIs are used together. For this, we rely on conditional probability for an API category $I$ to be used given the use of category $H$, that we compute as follows.

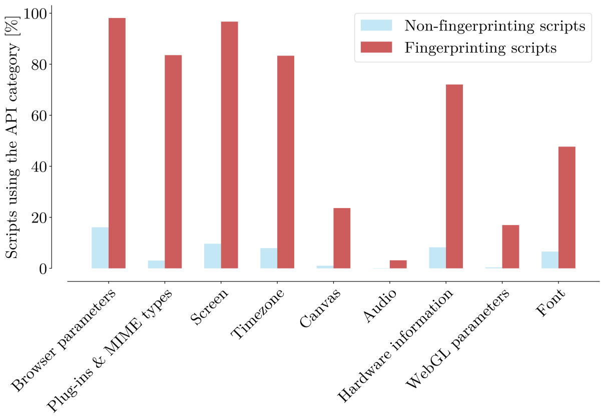


**Fig. 2.** Per-API-category percentages of fingerprinting and non-fingerprinting scripts. Results obtained using scripts in *JSWild*.

$$P(I \mid H) = \frac{\mid scripts\ with\ I\ and\ H \mid}{\mid scripts\ with\ H \mid} \tag{4}$$

Figure 3 reports the conditional probability for a given API category on the x axis to be employed given the use of categories on the y axis. First, results show that fingerprinters tend to combine APIs in Browser Parameters, Screen, Timezone, Plug-in and MIME types, Hardware Information and Fonts. For instance, the probability that Hardware Information APIs are combined with Browser Parameters, Screen and Timezone APIs are 98.7%, 97.2% and 80.2%, respectively. Second, we observe that scripts using APIs from Browser Parameters up to Font are very unlikely to use also WebGL Parameters, Canvas and Audio APIs. However, the opposite is not true: APIs in Audio, Canvas, and WegGL Parameters are often used in combination with most used APIs. For instance, all scripts using Audio APIs also use those in Browser Parameters. Interestingly, the probability that scripts employing Audio APIs also use APIs from WebGL Parameters is 91.4%, but the probability that WebGL Parameters is combined with Audio is just 16.6%. This is explained by the fact that Audio APIs are not so frequent to encounter in fingerprinting scripts (see Figure 2). These observations suggest there are clusters of APIs often used together (e.g., Browser Parameters, Screen, Timezone, Hardware Information), and others used in mutual exclusion (e.g., scripts using Font APIs do not use APIs from Audio).

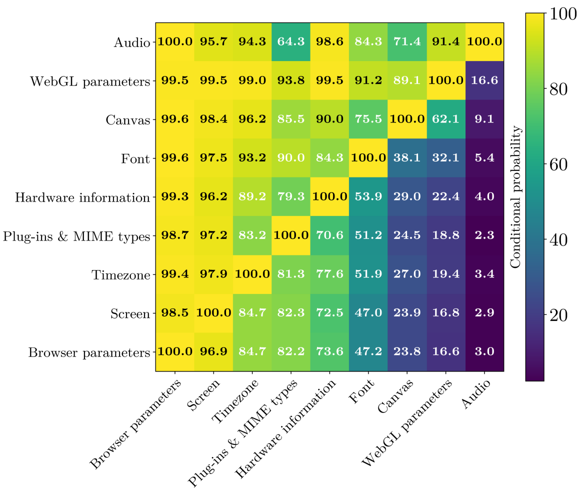


**Fig. 3.** Usage of fingerprinting API combinations measured using conditional probability obtained empirically from scripts in *JSWild*. Each cell reports the probability for an API category on the x axis to be employed given the use of category on the y axis.

## 7.3 Fingerprinting usage for known trackers and unknown fingerprinters

We now investigate whether there exist fingerprinting techniques which are peculiar for tracking services, or, conversely, for services which rely on fingerprinting to perform other tasks. We take a conservative approach, and restrict our analysis to well-know trackers only contained in *TrackerList* (see Section 3.3). Next, we extract the set of fingerprinting scripts identified in *JSWild* and match the domain contained in the corresponding URL against the tracking domains contained in *TrackerList*. Hence, we obtain a list of scripts classified as fingerprinters by our classifier and that are served by known tracking services. The 4,544 fingerprinting scripts in our dataset are served by 842 services. 1,705 of those scripts are served by 147 "known trackers". The remaining 2,839 fingerprinting scripts come from 695 services, "unknown fingerprinters", including actual tracking services not present in *TrackerList*, services using fingerprinting for anti-fraud, security and recognition of bots or web scrapers. For each domain in these two classes, we obtain the set of fingerprinting scripts it delivers and gather the list of fingerprinting APIs contained in such scripts. This way for each domain we obtain the list of fingerprinting techniques implemented by the corresponding scripts. We then count the number of different fingerprinting APIs used by each domain and build its empirical distribution. Results in Figure 4 show that known trackers and unknown fingerprinters exhibit similar trends. In fact, domains in both classes tend to use 3 to 6 different fingerprinting APIs, with around 20% of domains combining 8 different APIs at the same time.

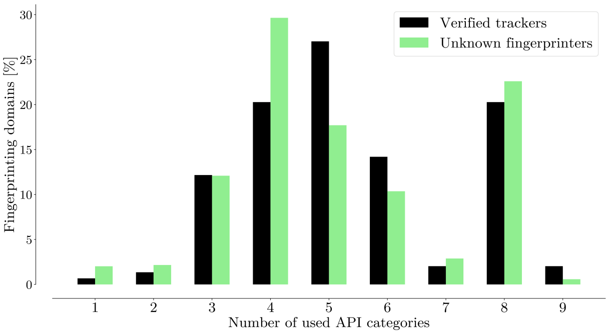
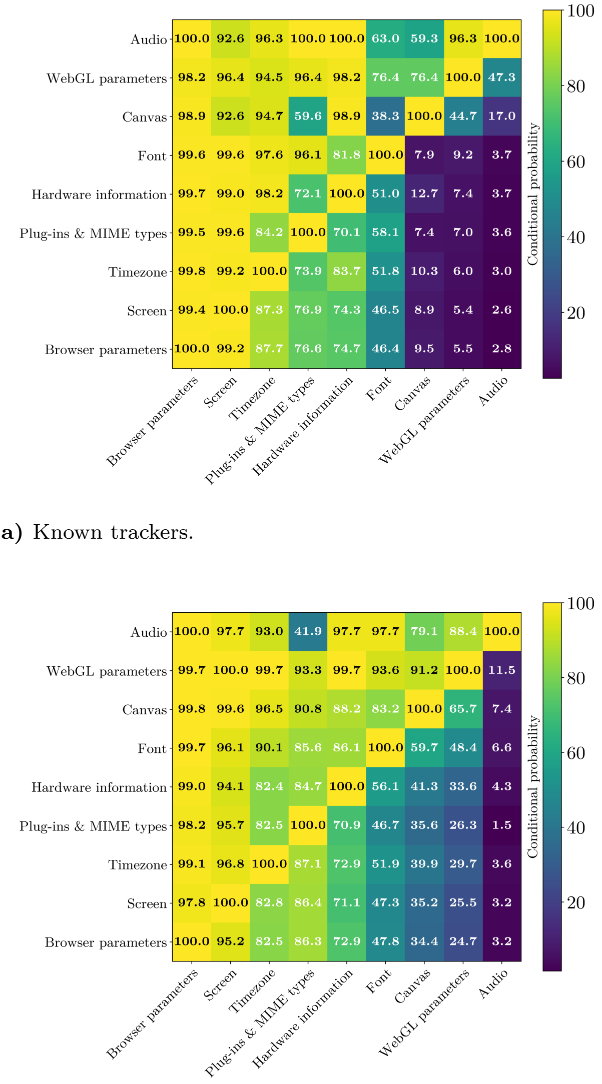


**Fig. 4.** Empirical distribution of domains based on the number of fingerprinting APIs they use. Black (green) bars refer to known trackers (unknown fingerprinters). Results obtained from *JSWild*.

Next, we dig further. In particular, we are interested in analyzing how frequently users incur in fingerprinters and which are the most encountered during navigation. For this, Table 7 reports 30 of the 147 known trackers using fingerprinting which are the most contacted by users in *HTTPDataset*. The table details the number of users in our population who downloaded the script, as well as the list of API categories used within. Domains are ranked based on users' penetration in decreasing order. Focusing on services, we observe that most of them are very well-known tracking domains often run by big players of the web. Results confirm that these scripts extract information using a combination of typically more than 3 APIs. As before, modern techniques like Canvas and Audio are less used. For instance, Audio APIs are used by just 4 services of Table 7. Considering the whole set of 147 trackers, 12% and 28% of them use Audio, and Canvas APIs, respectively. Conversely, 88% trackers implement techniques based on enumeration of plug-ins and MIME types. There are a few trackers (highlighted in bold) which use APIs from almost all categories. Among these, *addthis.com* and *moatads.com* are particularly pervasive as they have been downloaded by 57% and 46% of users in our dataset, respectively. By manually checking the code in the scripts, we notice that some trackers collect fingerprinting information possibly for purposes other than tracking. Indeed, *moatads.com* and *adtechus.com* use fingerprinting to discriminate accesses generated by automatic bots or crawlers.

Now, similarly for what we did in Table 7 we report in Table 8 the 30 most contacted unknown fingerprinters. Again, domains are ranked based on users' penetration in decreasing order. These are either new track-



|  | Browser parameters | Screen | Timezone | Plug-ins & MIME types | Hardware information | Font | Canvas | WebGL parameters | Audio |
|---|---|---|---|---|---|---|---|---|---|
| Audio | 100.0 | 92.6 | 96.3 | 100.0 | 100.0 | 63.0 | 59.3 | 96.3 | 100.0 |
| WebGL parameters | 98.2 | 96.4 | 94.5 | 96.4 | 98.2 | 76.4 | 76.4 | 100.0 | 47.3 |
| Canvas | 98.9 | 92.6 | 94.7 | 59.6 | 98.9 | 38.3 | 100.0 | 44.7 | 17.0 |
| Font | 99.6 | 99.6 | 97.6 | 96.1 | 81.8 | 100.0 | 7.9 | 9.2 | 3.7 |
| Hardware information | 99.7 | 99.0 | 98.2 | 72.1 | 100.0 | 51.0 | 12.7 | 7.4 | 3.7 |
| Plug-ins & MIME types | 99.5 | 99.6 | 84.2 | 100.0 | 70.1 | 58.1 | 7.4 | 7.0 | 3.6 |
| Timezone | 99.8 | 99.2 | 100.0 | 73.9 | 83.7 | 51.8 | 10.3 | 6.0 | 3.0 |
| Screen | 99.4 | 100.0 | 87.3 | 76.9 | 74.3 | 46.5 | 8.9 | 5.4 | 2.6 |
| Browser parameters | 100.0 | 99.2 | 87.7 | 76.6 | 74.7 | 46.4 | 9.5 | 5.5 | 2.8 |

**(a)** Known trackers.

|  | Browser parameters | Screen | Timezone | Plug-ins & MIME types | Hardware information | Font | Canvas | WebGL parameters | Audio |
|---|---|---|---|---|---|---|---|---|---|
| Audio | 100.0 | 97.7 | 93.0 | 41.9 | 97.7 | 97.7 | 79.1 | 88.4 | 100.0 |
| WebGL parameters | 99.7 | 100.0 | 99.7 | 93.3 | 99.7 | 93.6 | 91.2 | 100.0 | 11.5 |
| Canvas | 99.8 | 99.6 | 96.5 | 90.8 | 88.2 | 83.2 | 100.0 | 65.7 | 7.4 |
| Font | 99.7 | 96.1 | 90.1 | 85.6 | 86.1 | 100.0 | 59.7 | 48.4 | 6.6 |
| Hardware information | 99.0 | 94.1 | 82.4 | 84.7 | 100.0 | 56.1 | 41.3 | 33.6 | 4.3 |
| Plug-ins & MIME types | 98.2 | 95.7 | 82.5 | 100.0 | 70.9 | 46.7 | 35.6 | 26.3 | 1.5 |
| Timezone | 99.1 | 96.8 | 100.0 | 87.1 | 72.9 | 51.9 | 39.9 | 29.7 | 3.6 |
| Screen | 97.8 | 100.0 | 82.8 | 86.4 | 71.1 | 47.3 | 35.2 | 25.5 | 3.2 |
| Browser parameters | 100.0 | 95.2 | 82.5 | 86.3 | 72.9 | 47.8 | 34.4 | 24.7 | 3.2 |

**(b)** Unknown fingerprinters.

**Fig. 5.** Analysis of usage of fingerprinting API combinations expressed using conditional probability obtained empirically from scripts in *JSWild* dataset, separately for known trackers (Figure 5a), and unknown fingerprinters (Figure 5b).

ing services not yet included in the lists building *TrackerList*, or services which use fingerprinting for purposes other than tracking. First, we observe that, similarly to known trackers, basically all of them rely on legacy fingerprinting techniques. In fact, APIs in Plug-ins and MIME types are used by 92% of unknown fingerprinters. This similarity holds also for recent techniques. For instance, Canvas APIs are used in 31% of the cases. However, as also suggested by results in Table 8, Audio APIs are less used in this class, with only 4% of unknown fingerprinters employing them. Second, focusing on services, we notice many popular web portals such as *repstatic.it* or *wired.it* embed actual fingerprinting scripts. Further investigation shows that they deliver a script provided by *webtrekk.com*, a well-known third-party tracker which delivers its tracking scripts directly through the domain of the first party. In total, out of the 30 in the table, we count 14 domains using this approach. As such, those domains are actual trackers not contemplated in *TrackerList*, i.e., by most popular tracker blockers. Next, we observe domains which use many fingerprinting APIs despite not being trackers. For instance, *poste.it*, the web portal of the Italian postal service, uses APIs from many categories, possibly for security purposes. *cloudflare.com* uses such APIs to distinguish visits generated by bots and web scrapers. Finally, we observe that domains in Table 8 exhibit a smaller penetration than those in Table 7. This is because trackers in Table 7 connect as third-party services to multiple websites. Thus, they have higher chances to be contacted by users. Instead, domains in Table 8 are mostly first-party websites, and their penetration corresponds to their (relative) popularity among users.

Results above suggest there exist almost no differences in the usage of fingerprinting APIs between known trackers and other services. As a further check, we repeat the experiment of Figure 3, separately for these two classes, and report the results in Figure 5. In this case, we perform the analysis at a finer granularity, i.e., considering the APIs used in unique scripts in *JSWild*, and not aggregated by domain. Interestingly, the two plots show there exist different API usage patterns between the two classes. In detail, scripts from trackers (Figure 5a) rarely use APIs from WebGL parameters, Canvas and Audio in combination with APIs in Browser Parameters up to Font. Differently, scripts from unknown fingerprinters (Figure 5b) do employ more recent fingerprinting techniques, especially Canvas and WebGL Parameters: for known trackers the probabilities of having APIs from WebGL Parameters and Canvas used together with Font APIs is 9.2% and 7.9%, respectively, whereas, these increase up to 48.4% and 59.7% for unknown fingerprinters. We inspect the scripts responsible for this increase and check results are not polarized by the influence of few. Instead, they come from a multitude of domains (217 for the case of Canvas) implementing fingerprinting for very different purposes, including tracking, anti-fraud and bot recognition. Understanding the root cause behind this difference is complex. We conjecture it might be due to new trackers relying on more recent fingerprinting techniques and which anti-tracking lists building *TrackerList* do not include yet.

Finally, we conclude there is no fingerprinting technique which is peculiar to a given family of services. Thus, it is hard to grasp the purpose of fingerprinters (i.e., tracking or security) based on the list of fingerprinting techniques they implement.

## 8 Limits

Our tools, methodology and datasets present some limits that we summarize hereafter.

- **Limits of Ermes Proxy:** As Ermes Proxy cannot provide us a copy of scripts downloaded by users, we resort to `wget` to download scripts later on. This lead us to miss about 40% of scripts actually encountered by users (mainly because scripts are no more available, or their URLs have been truncated for anonymization).
- **Spatial, temporal and geographical limits of *HTTPDataset*:** We could only study fingerprinting JavaScript files downloaded by a limited number of users and for a limited amount of time. Hence, while we are confident that *HTTPDataset* fairly represents typical users' browsing patterns, it only partially represents the most visited websites. Second, *HTTPDataset* comes from a single country, i.e., Italy. Therefore, our findings might not generalize to other regions, and fingerprinters active in such country might be inactive or behave differently because of different local regulations.
- **Considered APIs**: Our methodology builds on a wide number of fingerprinting APIs. For the sake of simplicity, it does not contemplate some known, yet rare to encounter, fingerprinting techniques. Those are JavaScript Engine recognition, devices' sensors fingerprinting, WebRTC and Battery.
- **Limits of approach to analyze fingerprinting usage**: Our analysis of fingerprinting usage only contemplates scripts labeled as fingerprinters by our classifier based on static code analysis. Hence, it misses strongly obfuscated and malformed scripts.

Because of all of the limitations described above, results in Section 7 represent an underestimation of actual usage of fingerprinting in the wild.

## 9 Discussion and conclusions

We explored the possibility of identifying fingerprinting services by combining static code analysis and machine learning. For this, we designed, engineered and evaluated a methodology building on these two techniques and, leveraging a ground-truth dataset including more than 1,900 labeled JavaScript files, we demonstrated the feasibility of this approach, showing that we can achieve up to 94% accuracy at identifying fingerprinters automatically. Plus, our approach is scalable and easy to extend to consider other fingerprinting techniques.

We adapted the methodology to build on JavaScript execution logs obtained by performing dynamic code analysis. By comparing the two approaches we showed they complement each other. In fact, dynamic analysis spots obfuscated fingerprinting code impossible to detect with static analysis, whereas, static analysis finds fingerprinting patterns the browser executes only under specific conditions, and, thus, not always detectable by dynamic analysis. Hence, studies based solely on static or dynamic code analysis provide an incomplete view of actual usage of fingerprinting in the web.

By applying our methodology on a dataset of JavaScript files downloaded by 982 users during their navigation, we obtained several results. Out of the about 420,000 scripts contained in the dataset, we identified more than 4,500 performing fingerprinting distributed by 842 different domains. Interestingly, we observed that only 17% of these domains are known trackers. The unknown fingerprinters are new trackers not yet included in popular anti-tracking lists and services which rely on fingerprinting for security and anti-fraud purposes. By characterizing fingerprinting techniques those scripts use, we found that modern and possibly more accurate approaches based on Canvas and Audio fingerprinting are the least used, while traditional techniques, such as those based on enumerations of browser's plug-ins and MIME types are 5 times more popular. Finally, the 695 unknown fingerprinters employ the same techniques used by verified trackers, making it hard to understand services' nature based on adopted techniques.

In conclusion, the contributions presented in this paper demonstrate that our approach to the automatic identification of fingerprinters is novel, feasible and accurate. Nevertheless, our results show that further ingenuity is needed to discriminate services conducting user-tracking activity leveraging fingerprinting code analysis. We plan to investigate this aspect in our future work.

## 10 Acknowledgments

This research has been funded from the European Union's Horizon 2020 Research and Innovation program under Grant Agreement No. 871370 (PIMCity).

# References

[1] Battery Status API has been removed from Firefox. https://www.fxsitecompat.dev/en-CA/docs/2016/battery-status-api-has-been-removed/.

[2] Princeton web census. https://webtransparency.cs.princeton.edu/webcensus/.

[3] test262 Test Suite. https://github.com/tc39/test262.

[4] Device tracking by web sites can be a good thing. https://www.zdnet.com/article/device-tracking-by-web-sites-can-be-a-good-thing/, 2013.

[5] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union L119* (May 2016), 1–88.

[6] Acar, G., Eubank, C., Englehardt, S., Juárez, M., Narayanan, A., and Díaz, C. The web never forgets: Persistent tracking mechanisms in the wild. In *ACM Conference on Computer and Communications Security* (2014).

[7] Acar, G., Juarez, M., Nikiforakis, N., Diaz, C., Gürses, S., Piessens, F., and Preneel, B. Fpdetective: Dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 1129–1140.

[8] Boda, K., Földes, A. M., Gulyás, G. G., and Imre, S. User tracking on the web via cross-browser fingerprinting. In *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications* (Berlin, Heidelberg, 2012), NordSec'11, Springer-Verlag, pp. 31–46.

[9] Bojinov, H., Michalevsky, Y., Nakibly, G., and Boneh, D. Mobile device identification via sensor fingerprinting. *CoRR abs/1408.1416* (2014).

[10] Eckersley, P. How unique is your web browser? In *Privacy Enhancing Technologies* (Berlin, Heidelberg, 2010), M. J. Atallah and N. J. Hopper, Eds., Springer Berlin Heidelberg, pp. 1–18.

[11] Englehardt, S., and Narayanan, A. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1388–1401.

[12] Englehardt, S., Reisman, D., Eubank, C., Zimmerman, P., Mayer, J., Narayanan, A., and Felten, E. W. Cookies that give you away: The surveillance implications of web tracking. In *Proceedings of the 24th International Conference on World Wide Web* (Republic and Canton of Geneva, Switzerland, 2015), WWW '15, International World Wide Web Conferences Steering Committee, pp. 289–299.

[13] FaizKhademi, A., Zulkernine, M., and Weldemariam, K. FPGuard: Detection and Prevention of Browser Fingerprinting. In *29th IFIP Annual Conference on Data and Applications Security and Privacy (DBSEC)* (Fairfax, VA, United States, July 2015), P. Samarati, Ed., vol. LNCS-9149 of *Data and Applications Security and Privacy XXIX*, Springer International Publishing, pp. 293–308. Part 7: Network and Internet Security.

[14] Ferreira Torres, C., Jonker, H., and Mauw, S. Fp-block: Usable web privacy by controlling browser fingerprinting. 3–19.

[15] Haanen, S., and van Zalingen, T. Detection of browser fingerprinting by static javascript code classification.

[16] Ikram, M., Asghar, H. J., Kaafar, M. A., Mahanti, A., and Krishnamurthy, B. Towards seamless tracking-free web: Improved detection of trackers via one-class learning. *Proceedings on Privacy Enhancing Technologies 2017*, 1 (2017), 79 – 99.

[17] Krishnamurthy, B., and Wills, C. Privacy diffusion on the web: A longitudinal perspective. In *Proceedings of the 18th International Conference on World Wide Web* (New York, NY, USA, 2009), WWW '09, ACM, pp. 541–550.

[18] Laperdrix, P., Rudametkin, W., and Baudry, B. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 878–894.

[19] Lerner, A., Simpson, A. K., Kohno, T., and Roesner, F. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association.

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

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

[22] Mulazzani, M., Reschl, P., Huber, M., Leithner, M., Schrittwieser, S., Weippl, E., and Wien, F. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)* (2013), vol. 5, Citeseer.

[23] Nikiforakis, N., Joosen, W., and Livshits, B. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web* (Republic and Canton of Geneva, Switzerland, 2015), WWW '15, International World Wide Web Conferences Steering Committee, pp. 820–830.

[24] Nikiforakis, N., Kapravelos, A., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy* (May 2013), pp. 541–555.

[25] Olejnik, Ł., Acar, G., Castelluccia, C., and Diaz, C. The leaking battery. In *Data Privacy Management, and Security Assurance* (Cham, 2016), J. Garcia-Alfaro, G. Navarro-Arribas, A. Aldini, F. Martinelli, and N. Suri, Eds., Springer International Publishing, pp. 254–263.

[26] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[27] Reiter, A., and Marsalek, A. Webrtc: Your privacy is at risk. In *Proceedings of the Symposium on Applied*

*Computing* (New York, NY, USA, 2017), SAC '17, ACM, pp. 664–669.

[28] Upathilake, R., Li, Y., and Matrawy, A. A classification of web browser fingerprinting techniques. In *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)* (July 2015), pp. 1–5.

[29] Xu, W., Zhang, F., and Zhu, S. The power of obfuscation techniques in malicious javascript code: A measurement study. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on* (2012), IEEE, pp. 9–16.

# A Appendix A

## A.1 Hyperparameter tuning

Each classifier is characterized by internal parameters - called hyperparameters - which shall be accurately tuned to optimize performance. For this, we run an initial set of experiments to perform hyperparameter configuration. We follow the best practice, using 10-fold cross-validated grid-search optimization also to limit overfitting. For SVM, we consider combinations of the `kernel`, $\gamma$, $C$ and `class_weight` hyperparameters. For RF, we consider `n_estimators`, `max_features`, and `min_samples_leaf`.

We report the results of the grid-search optimization in  Figure 6 and Figure 7, for SVM and RF, respectively. Figures report the overall accuracy achieved by classifiers with different hyperparameter configurations. Focusing on SVM, we consider both linear and Radial Basis Function (RBF) kernels. For brevity, we report the results for RBF kernel only, which consistently outperforms the simple linear kernel, and detail the case where the classifier ignores class weights (top plot in Figure 6), i.e., classes are treated equally even if unbalanced, and when considering balanced class weights (bottom plot in Figure 6), i.e., the classifier keeps track of the class frequencies in the training set and counterbalances them by assigning weights to classes. On the x and y axes we vary $\gamma$ and $C$, respectively. $\gamma$ is inversely proportional to the radius of the area of influence of support vectors. If too large, the radius is smaller, and data points cannot influence each other. If too small, the influence region is too large and the model is not able to represent the complexity of data. $C$ determines the trade off between the complexity of the decision function and accuracy. Intuitively, larger values of $C$ increase the probability of overfitting data. Results in Figure 6 show that $\gamma$ has a considerable impact on accuracy, while $C$ and `class_weights` only marginally affect it. We achieve the best accuracy with SVM for the lowest value of $\gamma$, $C$ equal to 100 and balanced class weights mode disabled. Interestingly, while classes are little unbalanced (39% of fingerprinting scripts vs 61% of non-fingerprinting) we observe minor differences when balanced mode is active.

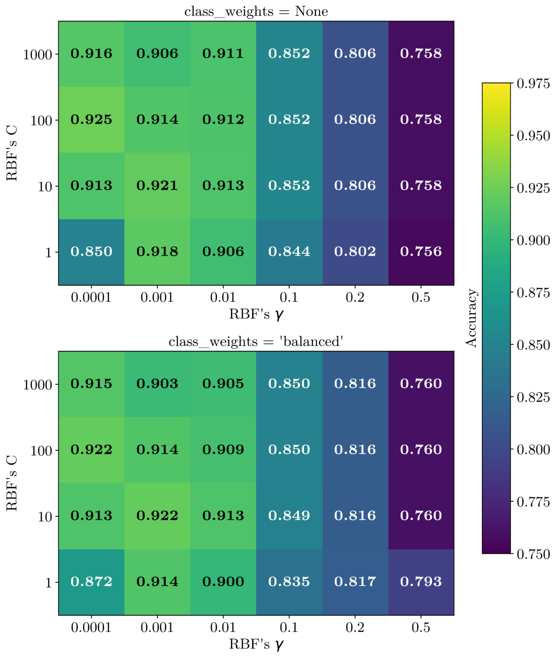


**Fig. 6.** Grid-search optimization of hyperparameters for SVM classifier with RBF kernel. *JSStaticGroundTruth* dataset.

Switching to RF, the hyperparameter space we consider is defined by `n_estimators`, `max_features`, and `min_samples_leaf`. These represent, respectively, the number of trees to use in the forest, the maximum number of features to split a node, and the minimum number of data points a leaf node can contain. Each grid in Figure 7 refers to a different value of `n_estimators` (from 15 to 64), and reports the accuracy when varying `max_features` (y axis) and `min_samples_leaf` (x axis). As shown, accuracy varies marginally across configurations in $[0.921, 0.950]$. We achieve the best performance with 45 estimators, and `min_samples_leaf` and `max_features` set to 2 and 50, respectively.

Considering time to complete a test, SVM takes almost 20 seconds to perform 10-fold cross validation on the whole *JSStaticGroundTruth* on a server equipped with a 4-core 8-threads 1.8-4.0 GHz CPU, averaging 1.93s for training and $3.84 \times 10^{-2}$s for testing on each fold. RF takes less than 1 second on the same hardware, spending $7.58 \times 10^{-2}$s for training and $4.34 \times 10^{-3}$s for testing on average. However, we remark that the most time-consuming operation is script processing, which takes on average 193ms to complete for each script. More specifically, on average, 100ms for code de-obfuscation, 85ms for AST generation and 8ms for counting features.

## A.2 Excluding overfitting

We take advantage of the experiment described in Section 5 to verify that our methodology based on static analysis is not affected by overfitting. In particular, we use the above classification results to calculate the ac-

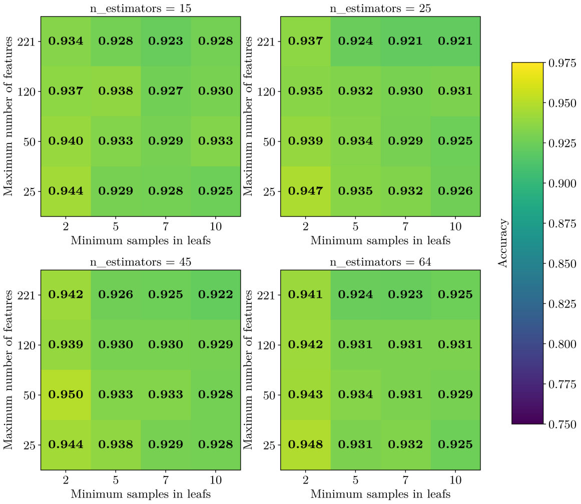


**Fig. 7.** Grid-search optimization of hyperparameters for RF classifier. *JSStaticGroundTruth* dataset.

curacy achieved by static analysis classifier on an independent dataset such as *JSAlexa*. In particular, we manually check the 1,764 scripts labeled as fingerprinting, and we obtain a precision of 0.951 which is similar to result reported in Table 4 and obtained on a completely independent dataset, *JSStaticGroundTruth*. Given the large amount of scripts labeled as non-fingerprinting in *JSAlexa*, it is impossible for us to manually verify all of them. Hence, to get a rough estimation of the fingerprinting class recall, we pick all scripts tagged as non fingerprinting and containing at least seven fingerprinting APIs, and manually observe that none of them are fingerprinters. Even if non comprehensive, this analysis suggests that the rate of false negatives in non-fingerprinting class is low. Hence, we can exclude our classifier is affected by overfitting.

# B Appendix B

Table 7 reports the 30 most contacted trackers (i.e., present in *TrackerList*) in *JSWild* using fingerprinting APIs. The table details the number of users in our population who downloaded the corresponding script, and the list of APIs used within. Domains are ranked based on users' penetration in decreasing order. Table 8 reports the same information for the 30 most contacted fingerprinters in *JSWild* dataset whose domain is not contained in *TrackerList*.

| Domain | # Users | Font | WebGL | Hardware Information | Audio | Canvas | Plug-ins & MIME types | Browser Parameters | Screen | Timezone |
|---|---|---|---|---|---|---|---|---|---|---|
| **addthis.com** | 562 (57%) | ● | | ● | | ● | ● | ● | ● | ● |
| bing.com | 541 (55%) | | | | | | ● | ● | ● | |
| hotjar.com | 491 (50%) | | | ● | | | ● | ● | ● | ● |
| **moatads.com** | 449 (46%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| doubleverify.com | 327 (33%) | | | ● | | ● | | ● | ● | ● |
| siftscience.com | 265 (27%) | | | ● | | ● | ● | ● | ● | ● |
| mathtag.com | 252 (26%) | ● | | ● | | | ● | ● | ● | ● |
| theadex.com | 197 (20%) | | | | | | ● | ● | ● | ● |
| hs-analytics.net | 182 (19%) | | | ● | | | | ● | ● | ● |
| scorecardresearch.com | 144 (15%) | | | ● | | | | ● | ● | ● |
| perfdrive.com | 134 (14%) | | | ● | | ● | ● | ● | ● | ● |
| adtechus.com | 132 (13%) | ● | | ● | | | ● | ● | ● | ● |
| **adf.ly** | 130 (13%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| adsrvr.org | 118 (12%) | | | | | | ● | ● | ● | ● |
| digitru.st | 111 (11%) | | | | | | ● | ● | ● | |
| ioam.de | 108 (11%) | | | | | | ● | ● | ● | |
| **mediaplex.com** | 104 (11%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| kissmetrics.com | 94 (10%) | | | | | | ● | ● | ● | |
| **rubiconproject.com** | 94 (10%) | ● | ● | ● | ● | | ● | ● | ● | ● |
| **penx.com** | 90 (9%) | ● | ● | ● | ● | | ● | ● | ● | ● |
| coremetrics.com | 89 (9%) | | | | | | ● | ● | ● | ● |
| y-track.com | 81 (8%) | | | ● | | ● | ● | ● | ● | ● |
| **cdn-net.com** | 72 (7%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| **gumgum.com** | 67 (7%) | ● | ● | ● | ● | | ● | ● | ● | ● |
| globalwebindex.net | 64 (7%) | | | ● | | | ● | ● | ● | ● |
| firstimpression.io | 61 (6%) | | | ● | | | ● | ● | ● | ● |
| doug1izaerwt3.cloudfront.net | 60 (6%) | | | | | | ● | ● | ● | |
| **advertising.com** | 53 (5%) | ● | ● | ● | ● | | ● | ● | ● | ● |
| **webtrends.com** | 48 (5%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| **yandex.ru** | 42 (4%) | ● | ● | ● | | ● | ● | ● | ● | ● |

**Table 7.** The 30 most contacted trackers in *JSWild* dataset performing fingerprinting.

| Domain | # Users | Font | WebGL | Hardware Information | Audio | Canvas | Plug-ins & MIME types | Browser Parameters | Screen | Timezone |
|---|---|---|---|---|---|---|---|---|---|---|
| cedsdigital.it$^{\textbf{webtrekk}}$ | 346 (35%) | | | | | | ● | ● | ● | ● |
| paypal.com | 284 (29%) | | | ● | | ● | ● | ● | ● | ● |
| mediaset.net$^{\textbf{webtrekk}}$ | 232 (24%) | | | | | | ● | ● | ● | ● |
| alicdn.com | 221 (23%) | ● | | ● | | ● | | ● | ● | ● |
| repstatic.it$^{\textbf{webtrekk}}$ | 173 (18%) | | | | | | ● | ● | ● | ● |
| stbm.it$^{\textbf{webtrekk}}$ | 157 (16%) | | | | | | ● | ● | ● | ● |
| stgy.ovh | 155 (16%) | ● | | ● | | | ● | ● | ● | ● |
| grouponcdn.com | 151 (15%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| ilfattoquotidiano.it$^{\textbf{webtrekk}}$ | 148 (15%) | | | | | | ● | ● | ● | ● |
| github.com | 144 (15%) | ● | | | | ● | ● | ● | ● | ● |
| ansa.it$^{\textbf{webtrekk}}$ | 132 (13%) | | | ● | | | ● | ● | ● | ● |
| yotpo.com | 130 (13%) | ● | | ● | | | ● | ● | ● | ● |
| youmath.it | 126 (13%) | | | | | | ● | ● | ● | ● |
| poste.it | 114 (12%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| areyouahuman.com | 113 (12%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| cloudflare.com | 103 (10%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| ilgiornale.it | 100 (10%) | | | | | | ● | ● | ● | ● |
| plug.it$^{\textbf{webtrekk}}$ | 99 (10%) | | | ● | | | ● | ● | ● | ● |
| tiscali.it$^{\textbf{webtrekk}}$ | 98 (10%) | | | | | | ● | ● | ● | ● |
| jsdelivr.net | 92 (9%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| wired.it$^{\textbf{webtrekk}}$ | 90 (9%) | | | | | | ● | ● | ● | ● |
| tiqcdn.com | 90 (9%) | | ● | ● | | ● | ● | ● | ● | ● |
| libero.it | 89 (9%) | | | ● | | ● | ● | ● | ● | ● |
| editmysite.com | 89 (9%) | | | ● | | | ● | ● | ● | ● |
| 24o.it | 87 (9%) | | | | | | ● | | ● | ● |
| yimg.com | 83 (8%) | ● | ● | ● | | ● | ● | ● | ● | ● |
| mymovies.it$^{\textbf{webtrekk}}$ | 70 (7%) | | | | | | ● | ● | ● | ● |
| skuola.net$^{\textbf{webtrekk}}$ | 68 (7%) | | | | | | ● | ● | ● | ● |
| lastampa.it$^{\textbf{webtrekk}}$ | 65 (7%) | | | | | | ● | ● | ● | ● |
| flixbus.de | 65 (7%) | | | ● | | | ● | | ● | |

**Table 8.** The 30 most contacted domains in *JSWild* dataset performing fingerprinting, but not present in *TrackerList*.[24]