Detection of Browser Fingerprinting by Static JavaScript Code Classification

February 11, 2018

Tim van Zalingen
University of Amsterdam
tim.vanzalingen@os3.nl

Sjors Haanen
University of Amsterdam
sjors.haanen@os3.nl

Supervisors:
Aidan Barrington (KPMG)
Ruben de Vries (KPMG)

***Abstract*—The fast moving technologies on the Web provide users, developers and researchers with increasing challenges when it comes to user privacy. The domain of browser fingerprinting has grown in response to the limitations in other tracking techniques such as cookies. For the sake of countering browser fingerprinting, detection is an important step. This paper sets out to answer whether the detection of scripts by static code analysis and subsequent classification with machine learning is possible. A Support Vector Machine classifier is used in order to attempt classification and detection of fingerprinting websites. The classifier shows to be able to make a correct distinction most of the time. 0.05 of the non-fingerprinting websites are wrongly classified, while 0.70 of the fingerprinting websites are correctly detected. Other metrics prove the proposed method to indeed be promising for detection. Solving the limitation of size of the current dataset might improve the results.**

I. Introduction

Nowadays Internet users are extensively tracked [1]. ISPs might want to study the mobility and usage patterns of clients. Companies often track users and examine their behaviour. This tracking is performed to better serve the client or to personalise advertisements on their website. Research at Uber showed that users tend to order more hastily when their mobile phone battery is almost depleted [2]. This near empty battery can be detected by mechanisms that are used to track the identity and state of a user. Thus user tracking allows prices to be adapted to the needs of a user. This price tailoring is quite similar to the net neutrality discussion. User identification by tracking can also be used for fraud detection. This will be examined in related work, section III-A. Most often browser fingerprints are utilised by websites hosting spam or malicious activity [3]. It is suggested that users should do more than simply deleting cookies to disable tracking and thereby safeguard their privacy [4]. As a result of clearing cookies, and subsequently demonstrating uncommon behaviour, a user might even be identified more easily.

This introduces the domain of browser fingerprinting. Browser fingerprinting detects settings in the browser, operating system and hardware, via the browser, to uniquely identify a user cross-network. The Electronic Frontier Foundation (EFF) runs Panopticlick, a website on which people can test how traceable their browser is [5]. The data gathered by each user test is stored by the EFF. The traceability test includes the uniqueness of the browser fingerprint. This uniqueness is measured by comparing the fingerprint to a set of previously stored fingerprints. The EFF published a paper analysing the browser fingerprints of a sample of the first 470,161 Panopticlick participants [6]. About 86.3% of browsers could be uniquely identified at the first visit. One challenge in tracking users is that browser characteristics can change over time. A simple and crude algorithm, outlined in the aforementioned paper, was able to correctly identify the same user at least one hour after the last visit in 65% of cases. Note that, according to the EFF, this algorithm can easily be improved.

The similar AmIUnique project was able to collect 118,934 fingerprints of which 89.4% were unique [7]. Both the EFF and the AmIUnique project state that browser plugins appear to make the fingerprint more unique and thus have a negative effect on the user's privacy. Even the mere circumstance of having a privacy enhancing browser plugin makes users more unique and therefore more traceable.

Other parties might outperform Panopticlick or AmIUnique, since some parties use more or other browser characteristics to base their fingerprint on [3]. It is likely that commercial entities, which have been developing fingerprinting techniques for years, can track users even more accurately.

There exists research that surveys the Web to map fingerprinting practices, but there is limited research that focuses on static code analysis. In this paper, static code analysis is used with machine learning in order to classify fingerprinting scripts. Static analysis involves no execution of code during the examination. Dynamic code analysis, on the other hand, is the examination of code behaviour while executed.

A. Motivation

As it can facilitates user tracking, the privacy issues that result from browser fingerprinting should be taken seriously. Whereas common browser plugins that safeguard privacy help against cookies, they seem to fail at protecting a user against browser fingerprinting. The intricacies of these plugins are explained in related work, section III-B. In order to preserve user privacy, fingerprinting needs to be prevented. As mentioned in related work, this can be accomplished by changing the browser characteristics to be more generic or by randomising it every now and then. Yet, this randomisation of the website can have a negative effect on user experience. Therefore it is important to prevent fingerprinting without affecting the rest of the website. Before prevention, detection of the scripts that perform the fingerprinting is critical. If performed correctly, only the scripts responsible for fingerprinting can be targeted. The use of machine learning might improve on earlier methods that arbitrarily assign a score when features are detected and classify on an arbitrary threshold [8]. When performing this detection before execution, it is known whether a script plans on fingerprinting a user in advance instead of in hindsight.

B. Problem Definition

The aim of this study is to detect the practice of browser fingerprinting before execution with classification by machine learning. The study sets out to answer the following question:

> *Can the action of browser fingerprinting be detected before execution by analysing JavaScript code with machine learning?*

The studies discussed in related work, section III-C, show that JavaScript (JS) is used to extract most of the information used for fingerprinting and that such code stands out. Therefore, browser fingerprinting is expected to show in JS code. Approaches using static code analysis and simply looking for occurrences of calls, were also able to achieve some success. To prove this, a method of detecting browser fingerprinting in JS code is designed and tested. As the question implies, a distinction between code that fingerprints and code that does not is expected. What makes JS code for browser fingerprinting distinguishable from code that serves other functionality? When such distinctions are used to detect fingerprinting code, how accurate can this detection be? Depending on how accurate and time consuming the detection method is, it might be applicable or not. Would it be possible to use this method as part of a prevention tool?

II. Background

A. Fingerprinting

Fingerprints are obtained through the browser. During a connection, the browser already provides the server with some basic device specific information in the HTTP header. This, along with other information that the browser already discloses in the communication with the server, can be used in passive fingerprinting. A client can never know of passive fingerprinting, as all execution is server side. Active fingerprinting, on the other hand, is obtaining the device specific information through script execution on the client side. JS can serve as an interface to device specific information. In this paper, websites which attempt to fingerprint browsers are called *fingerprinters*.

Whereas cookies are stateful, browser fingerprints are stateless. A user can simply read and delete cookies, but this cannot be applied to browser fingerprints. Most browsers allow for the deletion of cookies and have an option to block cookies from third parties only, as such preserving the workings of the website [9]–[11]. Cookies can be extensively examined on the client side, allowing for the scope of third party cookies to be mapped [12]. Browser fingerprints work, on the other hand, in the shadows. A user can not simply delete fingerprints or disable third party fingerprinting. Some plugins, as discussed in related work, section III, attempt to block certain scripts, but are either unreliable or disturb the workings of the website. Because the client machine is unaware of fingerprinting being executed, the extent of the practice can not easily be mapped.

B. Support Vector Machines

In this paper, Support Vector Machines (SVMs) are frequently referred to. This section will provide some theoretical background into SVMs and techniques used to validate such a machine learning classifier.

SVMs are a collection of supervised learning methods [13], [14]. For this research, SVMs are utilised for classification purposes. SVMs attempt to separate data with a maximised margin. Such a boundary does not necessarily have to be linear. This can be achieved with a ‘kernel trick’, which allows the data to be classified in higher dimensions [15].

SVMs provide several relevant advantages. Later sections will explain how these advantages are utilised. Firstly, it works well with high dimensional spaces. As mentioned later on, tens of JS calls are used to form the input vectors, as such creating a high dimensional space. Secondly, the SVM classifier can still perform well with more dimensions than samples. Lastly, while other classification algorithms might become biased towards the largest set, SVM classification does not degrade with datasets of different sizes [16].

There exist several parameters that can be tweaked. A few of these are worth mentioning, as they are set in the experiments performed in this study. Firstly, the parameter C allows for the measure of classification errors to be weighted. C should depend on the noise of the data. A high C attempts to classify everything correctly, whereas a low C has a smoother decision surface. In conclusion, if there is noise in the data and a high value of C attempts to weigh this noise heavily, the classification might be too specific. Since not all features might be linearly separable, a 'kernel trick' is used. This second parameter, the use of a non-linear kernel, such as the Radial Basis Function (RBF) kernel, allows for a feature to be drawn to a higher dimension. Lastly, if such a non-linear kernel is used, the influence of a feature is determined by the value of γ. The larger this γ, the higher an narrower the peak in the new dimension. Therefore, a high γ requires new samples to be closer in order to be affected. Choosing a too large γ may result in overfitting, because each sample is fitted perfectly locally, but other close by samples will not be classified as similar. Section method, IV-D2, explains how these parameters are chosen.

1) Validating a classifier: The classifier is validated by using part of the dataset as test set. The exact details of this validation will be explained in section IV-D2 method. The performance of the classifier is determined by examining a Receiver Operating Characteristic (ROC) curve and calculating an F_1 score.

The ROC curve is obtained by plotting the true positive rate (TPR) against the false positive rate (FPR). The true positive rate, or recall, shows how sensitive the classifier is.

$$\text{TPR} = \text{Recall} = \frac{\text{Number of true positives}}{\text{Number of positives}} \tag{1}$$

The false positive rate shows how likely the classifier is to falsely identify a sample as positive.

$$\text{FPR} = \frac{\text{Number of false positives}}{\text{Number of negatives}} \tag{2}$$

A better result has a higher true positive rate than a false positive rate. Therefore the area under the ROC curve says something about the performance of the classifier. This Area Under the Curve (AUC) is calculated and used as a metric.

The earlier mentioned F_1 score is used to measure the accuracy of the classification. The precision and recall, in formula 1, are considered. The precision shows what fraction of positively identified values are actually true.

$$\text{Precision} = \frac{\text{Number of true positives}}{\text{Number of selected positives}} \tag{3}$$

The F_1 score is a weighted average between the precision and recall.

$$F_1 = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

Thus a higher F_1 score shows that the classifier can form a more correct set of selected positives.

III. Related Work

A. Other Uses of Browser Fingerprinting

There exist dual authentication mechanisms for browser fingerprinting [8], [17]. After a user has logged in on a service, a browser fingerprint will be continuously collected during the user's session. A sudden change in the fingerprint might imply that the session has been hijacked. If a fingerprint changes on a longer term, a user might be logging in from an unknown device and additional authentication might be required.

B. Thwarting Browser Fingerprinting

There are several techniques and/or tools that attempt to impede fingerprinting in some way. This section lists those prevention techniques.

1) Disable functionality: NoScript and the Tor Browser can disable scripts like JS and plugins by default until the user accepts the execution of them [18], [19]. While this is an effective way of defence, they leave the unpleasant task of deciding whether the code is actually desirable for the user. Also, the majority of users may lack the required knowledge in order to make the right decision.

2) N:1 - Many Browsers, One Configuration: In this technique, as many browsers N as possible share the same configuration in order to prevent fingerprinters to uniquely identify a single user, since N browsers share the same fingerprint. This technique is used by the Tor Browser and the Disguised Chromium Browser (DCB) [20]. Both implementations try to be as generic as possible in order to stay in the biggest anonymity group.

3) 1:N – One Browser, Many Configurations: By constantly changing the fingerprintable data shared with fingerprinters, this technique aims to break the linkability of different website visits by a single user. Implementations do this either by using a pool of real world configuration options [8], [20], or by complete randomisation of this data [21]–[23]. This happens after every HTTP request or session. A major downside of complete randomisation is that it could break legitimate functionality due to faulty values. Also, because browser technology is ever evolving, the list of settings that are known to be fingerprintable keeps growing. This requires frequent updating of randomisation implementations.

4) Blacklisting: A recently introduced feature in Firefox, Tracking Protection [24], promises to protect the user from third-party tracking. It guarantees to not only protect against third-party tracking via cookies, but also in utilising local storage, fingerprinting and etags, where etags are a caching mechanism in HTTP. This protection is achieved by blocking requests to third-parties that are in a known list. Nonetheless, this does not prevent tracking performed by the website itself. Also, as they admit, the list requires frequent updating and therefore might not block everything. However, this list of what has to be blocked, could be based on something else but a list compiled by other humans [25]. The working of the website, and thus the user experience, could also be affected by blocking false positives.

C. Detection of Browser Fingerprinting

One of the challenges in the field of browser fingerprinting is how to distinguish scripts that attempt to fingerprint from scripts that do not. There are legitimate reasons for scripts to show fingerprinting behaviour. For example, because font probing is known to be used to fingerprint browsers, a news website which probes and uses many fonts for a legitimate purpose might unintentionally get labelled as a fingerprinter. This section discusses other studies that tried to detect fingerprinting scripts and their limitations in solving this challenge.

One of the studies discussed is FPDetective, a fingerprinting detection framework which visits websites as a crawler and collects data about events that might be related to fingerprinting [26]. FPDetective classifies a JS file as a fingerprinter when it loads more than 30 system fonts, enumerates plugins or mimeTypes, detects screen and navigator properties, and sends the collected data back to a remote server. The problem with this classification is that fingerprinting scripts that happen to have all but one of these characteristics would bypass detection, resulting in false negatives. Also, FPDetective focuses on events that use JS-based font detection and Flash-based fingerprinting which covers only a small subset of the features that can be fingerprinted.

Another approach is implemented in FPGuard, which aims to detect JS objects fingerprinting, JS-based font detection, HTML5 Canvas fingerprinting, and flash-based fingerprinting at runtime [21]. Its detection mechanism counts the occurrences of suspicious calls in JS code. If the counting reaches specific threshold values, FPGuard will flag the JS code as fingerprinting. The threshold values are set based on the researchers' definition of abnormal behaviour. FPGuard can also prevent fingerprinting by randomising settings sent to a suspected tracker with a high score. Unfortunately the researchers are vague in describing their definition of abnormal behaviour, and the actual threshold values are not mentioned anywhere. Also the study does not provide the FPGuard source code. Thus, their approach is not reproducible and cannot be validated.

Both FPDetective and FPGuard show that there is a distinction between code that is used for fingerprinting and code that is not. This means that detection by JS code analysis prior to execution might be possible. However both solutions are still examples of dynamic code analysis.

Similar research utilising static code analysis can also be found. One such study attempts to measure how often certain browser fingerprinting scripts are used [27]. This is achieved by scraping the top 1000 websites in the United States and examining the scripts on these websites. The scraped scripts are only compared to three known fingerprinting scripts. Therefore only these three specific scripts can be detected. It turns out that less than 6% of examined websites use one of these three browser fingerprinting scripts. While this research does utilise static code analysis, it only tries to identify previously known fingerprinting scripts. Therefore it cannot be concluded that fingerprinting is only used on 6% of these top 1000 websites. Other fingerprinting scripts or techniques could be used on websites, without being detected by this study.

All of the mentioned studies in this section failed, in their publication, to clearly define the line between fingerprinting scripts and legitimate scripts. On the contrary, Laperdrix detailed several signs that indicate that a script is a fingerprinter [8]:

- Accessing specific functions
- Collecting a large quantity of device-specific information
- Performing numerous access to the same object or value
- Storing values in a single object
- Hashing values
- Creating an ID
- Sending information to a remote address
- Minification and Obfuscation

Since Laperdrix's thesis is extensible and written very recently, we can argue that these signs are a good source for indicators of fingerprinting. Figuring out which JS functions belong to these signs and finding those functions in the JS code analysis may detect the act of fingerprinting.

IV. Method

A. Overview

In this approach, two predefined sets of scripts are examined. The gathering of the scripts is explained in the "Gathering" phase outlined in section IV-B. After the scripts are gathered, some processing is performed to be able to better analyse the scripts. During this "Processing" phase, as explained in section IV-C, deobfuscation is performed and member expressions hidden in variables are expanded. Afterwards, the real analysis of the scripts is performed. This "Detection" of fingerprinting phase, is explained in section IV-D. In the final stage, the amount of calls that are likely to be found in fingerprinting scripts can be counted. The three phases are shown in figure 1. See Appendix C for a reference to the codebase.

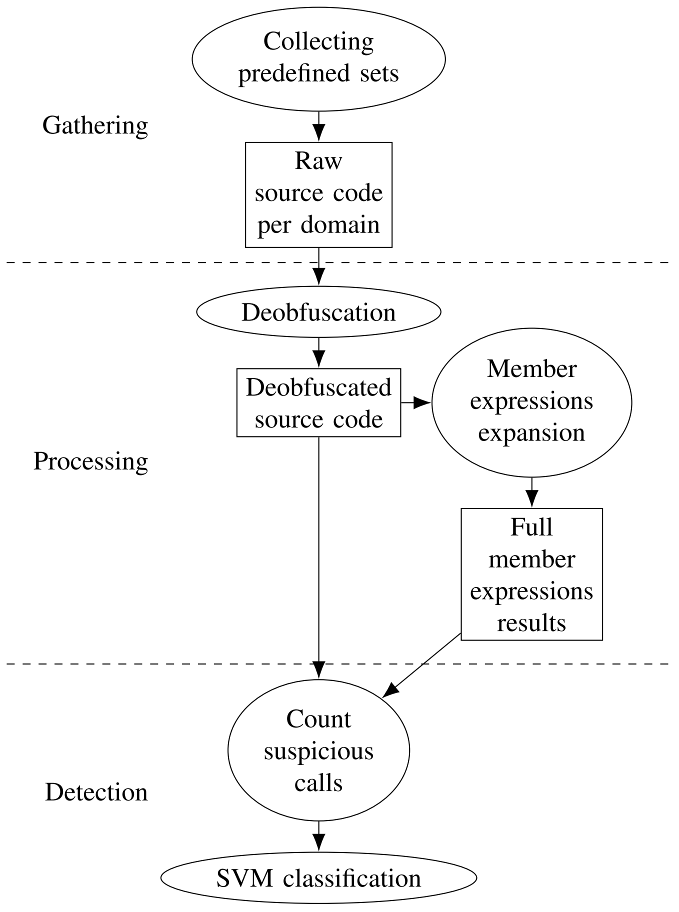
Gathering
Collecting predefined sets
Raw source code per domain
Processing
Deobfuscation
Deobfuscated source code
Member expressions expansion
Full member expressions results
Detection
Count suspicious calls
SVM classification

Fig. 1. Process of analysing JS source code for a given set of websites to find fingerprinting practices.

B. Gathering

In the first part of this process, two sets of real world JS scripts are needed: a set of fingerprinters, and a set of non-fingerprinters. Both sets are manually gathered from the Web. The set of fingerprinters are found by looking for websites of parties which are known to use browser fingerprinting. Academic sources have also been used to compile this set [8], [21], [26]. Earlier research in the detection of fingerprinting has often cited parties that fingerprint. Besides that, code used in research that aims to collect fingerprints, was also used. The non-fingerprinters are initially found by browsing the Web with several browser add-ons that detect cross-domain trackers. Finally, both sets of scripts are manually analysed in order to verify that they are indeed classified as initially expected.

C. Processing

1) Deobfuscation: Because JS is interpreted by the browser, the source code itself needs to be transferred to the client. This allows the client to observe the source code. Most often users would not examine code themselves, but code could be statically examined by the browser, extensions, plugins or more experienced users. Besides offering freedom to examine, transferring source code can also be more resource expensive.

In order to mitigate the effects of these two downsides to transferring source code, the code that is transferred is often adapted before deployment on the website. In order to lessen the size, code is often minified. As a byproduct, the code is often unreadable for humans, which can be a positive effect for companies that desire their code to not be examined. There exists a field solely focused on making source code unreadable: Obfuscation is the deliberate discombobulation of code to make it incomprehensible for static analysis or humans.

Both techniques complicate the static analysis of code. Therefore it is important to try to counter the obscurity caused by obfuscation and possibly by minification. Using JS Beautifier, the code is deobfuscated and deminified where possible [28]. The JS Beautifier repository offers several scripts, that are used in this research, to extend the beautifier to better unpack and deobfuscate scripts. JS Beautifier has been used in research to deobfuscate code for static analysis before [29], [30]. Obfuscation and deobfuscation is an ongoing arms race. Therefore it cannot be expected with full certainty that the deobfuscation will always act as desired.

Note that basic JS calls to standard objects cannot be obfuscated well. The object "*navigator*" obfuscated is still "*navigator*". Likewise, the list of plugins will always be obtained by calling the property "*plugins*". However obfuscation does hide the further meaning and workings of the source code, as well as the relation between these earlier mentioned objects and their properties.

2) Expanding Member Expressions and Deriving Variables: When trying to detect calls that are common in fingerprinting, some object properties can be accessed indirectly. This complicates the search.

The following example shows this phenomena: The object "*navigator.plugins*" is stored in the variable "*np*". Later in the code, the length of the plugins is requested by using "*np.length*". To detect fingerprinting, it is interesting to know that the number of plugins is requested. Since "*length*" is an often used property, simply noting that the length of a variable is requested, is not important. How to derive the meaning of "*np.length*" to be "*navigator.plugins.length*"? During the processing phase, the real meaning of variables from previous assignments has to be determined in order to see from what object a property is called. This allows these member expressions to be expanded.

In order to do this expansion, the code is parsed and an Abstract Syntax Tree (AST) is constructed. An AST allows source code to be represented in a structured and hierarchical order. As such, the possibility to syntactically examine to code and derive its meaning arises.

This tree can henceforth be traversed. During the traversal, all declared variables are stored in the current scope. If the initialisation or assignment of the variable is another variable or a member expression, the initialisation is stored. When a property is requested in a member expression, the previously stored variables are searched to attempt an expansion of this member expression.

```
var nav = navigator;
function fingerprint() {
  var a = nav.plugins;
  var b = a;
  var c = b.length;
  var d = nav.userAgent;
}
```

Fig. 2. Example JS code of object properties split over different variables.

An example of properties split over multiple variables is provided in figure 2. In a fingerprinting script, "*navigator.plugins.length*" and "*navigator.userAgent*" could be requested. However, a simple search in the source code for these strings will offer no result. That is because these calls are split over multiple variables. Running the member expression expansion traversal of the obtained AST, would derive the real meaning as shown in figure 3. The AST created in this example is shown in figure 6 in appendix A. Now the conclusion that "*navigator.plugins.length*" and "*navigator.userAgent*" are called can be drawn.

```
nav.plugins   is   navigator.plugins
b.length   is   navigator.plugins.length
nav.userAgent   is   navigator.userAgent
```

Fig. 3. Output of analysing the member expressions and variables in figure 2.

D. Detection

1) Count suspicious calls: Related work, section III-C, describes several signs in code which could indicate the act of fingerprinting. Accessing specific functions, which return fingerprintable information, and collecting a large quantity of device-specific information are signs which can be recognised by counting specific keywords in the scripts. This is part of static code analyses, which is the reason why these signs are included in this study's approach. To make counting possible, the processing phase makes sure that these JS calls are revealed in plain text.

A list of suspicious calls is comprised by desk research and examining existing fingerprinting solutions. The list is shown in figure B in appendix B. When comparing the occurrences of each call between the non-fingerprinters and the fingerprinters, it is possible to pick the calls which occur most differently.

This subset of calls are the most relevant and usable to classify scripts and are detailed in results, section V-A.

When a website is investigated, the number of occurrences of these fingerprinting indicators are counted. Both the deobfuscated source code and the list of expanded member expressions are used to count. This is because not every call might be a member expression that can be expanded. The result is a list of JS calls or expressions and how often these occur on a website.

2) Support Vector Machine Classification: The SVM Classifier requires two sets of scripts. As mentioned before, one collects fingerprints and the other does not. In the same manner as in background, section II-B1, the first step is to select the correct parameters. Attempting all combinations in an exhaustive search and picking the combination that provides best score, results in the best combination of parameters. Both the linear and RBF kernel are tested. The C values of 1, 10, 100 and 1000 are attempted. Where, as mentioned in background section II-B, a lower C is likely to perform better with noise. The γ values of 10^{-4}, 10^{-3}, 10^{-2}, 0.1, 0.2 and 0.5 are tried. As mentioned in background section II-B, a higher value of γ narrows the peak in the new dimension that is introduced by the RBF kernel. Too narrow a peak could result in overfitting.

When a classification is validated, a test set is kept apart and the classifier is trained with the rest of the set. In order to prevent biased results due to overfitting or sheer luck in selecting this test set, cross-validation is used. Cross-validation performs multiple iterations of fitting, with each iteration containing a different part of the complete set as test set. An average score is taken to base the score of the classifier on for a given set and parameter settings. The K-fold cross-validation used, splits the complete dataset in K parts for K iterations. Still, some test sets might only contain one of the two possible results. This phenomena is likely to occur when one set is larger than the other. Therefore a stratified K-fold is performed. A stratified K-fold roughly keeps the same ratio of positives and negatives for the train and test set as in the full dataset. Different values of K might give different results. Because of the small nature of the dataset, a higher K, and thus a higher training set, might provide better results. This cross-validation is also included in the parameter selection.

After parameter selection and fitting with the Stratified K-fold cross-validation, the F_1 score is calculated, ROC is plotted and AUC is derived from the ROC curve. Each experiment is repeated 100 times to arrive at a reliable average. These results should illustrate the accuracy and performance of the classifier.

V. Results

In this research, the fingerprinter set contains scripts from 12 domains, while the non-fingerprinter set contains scripts from 20 domains. The scripts are aggregated per domain, since functionality is often spread over multiple JS files.

A. Where fingerprinting stands out

Figure 4 illustrates the difference between fingerprinters and non-fingerprinters for the selected features. The high

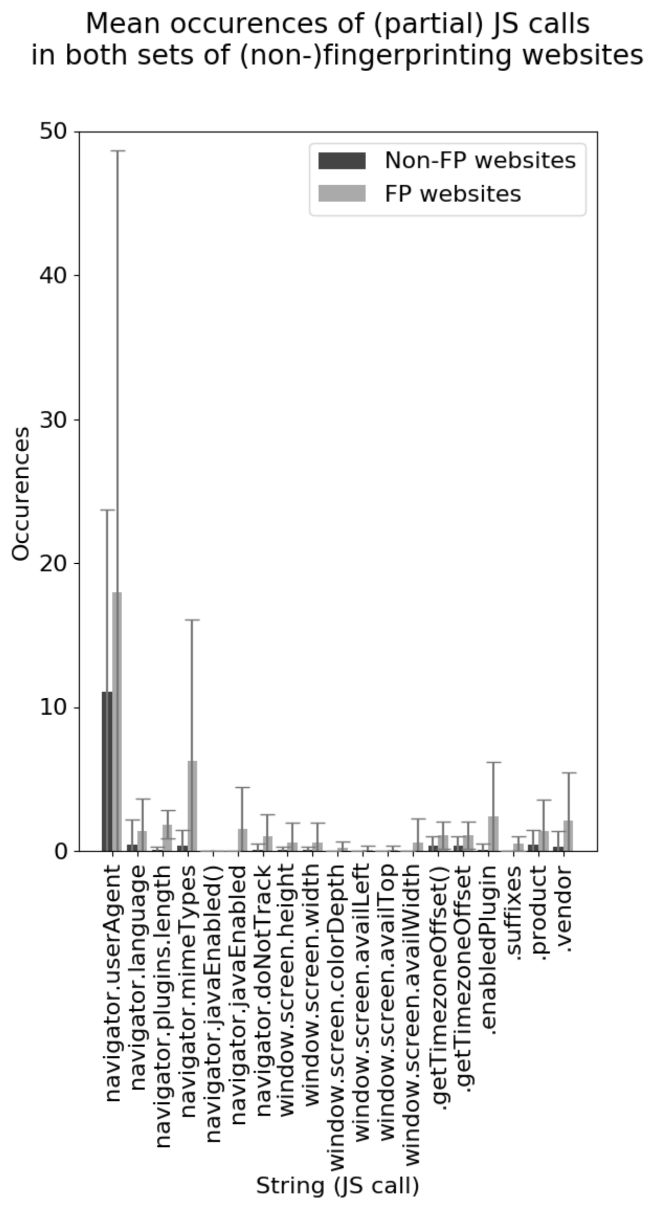

Fig. 4. The average occurrence of (partial) JS calls when comparing fingerprinters to non-fingerprinters.

standard deviation proves that simply picking one or two features, would not provide reliable results. Several JS calls are barely ever mentioned on non-fingerprinters. A call such as "*JavaEnabled*" would likely only occur for very specific applications. From the set of features gathered from earlier work, as mentioned in section IV-B, only those that occur more often in fingerprinters than in non-fingerprinters were used. For the classifier, some sub-strings are also used. In the end, 51 strings are used as features to count and classify.

B. Classification

The classifier was configured with the parameters found by the exhaustive search. The parameters are as follows: An RBF kernel, $C = 1000$ and $\gamma = 0.0001$. These parameters were

TABLE I
DIFFERENT SCORING METRICS ON 4 STRATIFIED K-FOLDS FOR BOTH SETS AND AVERAGE METRICS OF BOTH SETS. THE FINAL COLUMN SHOWS A TOTAL SIZE OF THE TEST SET.

$K = 4$	**precision**	**recall**	**F_1 score**	**size of test set**
non-fingerprinting	0.83	0.95	0.88	5
fingerprinting	0.84	0.70	0.73	3
avg/total	0.84	0.82	0.80	8

TABLE II
COMPARISON BETWEEN DIFFERENT VALUES OF K IN THE STRATIFIED K-FOLD.

K	**AUC± stdev**	**F_1 score**	**size of test set (total size is 32)**
2	0.81 ± 0.16	0.73	16
3	0.87 ± 0.15	0.78	10
4	0.90 ± 0.14	0.80	8
5	0.90 ± 0.16	0.83	6
6	0.90 ± 0.18	0.82	5
8	0.91 ± 0.23	0.84	4

used in all experiments carried out in order to come to the results in this section.

Figure 5 shows the ROC Curves for 2 and 4 stratified K-folds with the current classifier. The accuracy appears to be promising. The line is quite steep, as such, the true positive rate or recall shows that most positives are indeed classified correctly. There still is a visible false positive rate, therefore some non-fingerprinters are wrongly classified as positives.

In order to examine the results for $K = 4$ in more detail, the precision, recall and F_1 score for both sets, separate and combined, are shown in table I. This shows a precision of 0.83 for the non-fingerprinting set. This means that a ratio of 0.83 of the samples seen as non-fingerprinting, are indeed non-fingerprinting. The recall of 0.95 tells us that a ratio of 0.05 of non-fingerprinters are wrongly seen as fingerprinting. In the end, 0.70 of the fingerprinters are correctly detected.

For the detailed comparison, $K = 4$ was chosen. Other values of K are shown in table II. With the small value $K = 2$, only half of the whole set is used for training at one time. It is not surprising that such a small training set provides inferior results. The AUC and F_1 score can be seen to improve as the size of the training set increases. However, after $K = 4$, the standard deviation of the AUC also increases. This is likely due to the variation in the different test folds. When the test set is larger, each individual K-fold produces a closer average. Once the K-fold size decreases, the differences in the individual samples become more apparent. The more in-depth experiments were conducted with $K = 4$ because of the combination of the high AUC and F_1 score, along with the low standard deviation.

VI. DISCUSSION

As demonstrated by the results, some JS calls are better for detecting fingerprinters in statical analysis than others. Using this approach, new JS calls used in the future can be tested to decide whether they can be added to the list of suspicious JS calls. The performance of the SVM classifier looks promising: The AUC is 0.90 when using 4 stratified K-folds. This implies there is a substantial larger amount of true positives than false positives. In the dataset used in this study, the classifier has a fingerprinter detection rate of 0.70. Thus,

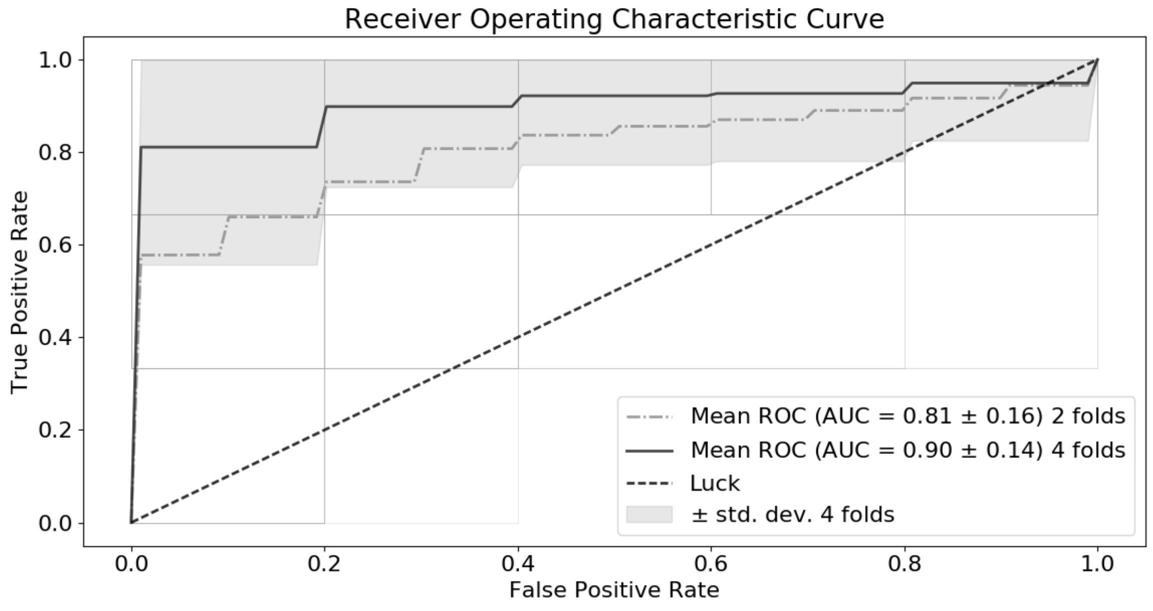

Fig. 5. ROC curve for 2 and 4 stratified K-folds. For readability, the standard deviation is only plotted for 4 stratified K-folds. The luck line could be compared to flipping a coin, one would be right 50% of the time.

most fingerprinters are indeed detected. However, 3 out of 10 are missed. Hence, the current method with this dataset, would not catch every fingerprinter. There are false positive though. 0.05 of the non-fingerprinters are classified as fingerprinting. The false positives could break the website and thus the user experience.

When decreasing the size of the test set, as seen in table II, the F_1 score increases. This is due to more samples being available for training the classifier. This shows that the dataset used in this study might be too small. Therefore, a bigger dataset is likely to improve the results. The size of the dataset is constricted due to time limitations. The manual collection proved to be a time consuming task. Another point of discussion is the possibility that the dataset is biased because the scripts are collected manually. This process is probably not completely random, and mistakes could be made in manually analysing those scripts. As a final point, this study uses the SVM algorithm as a result of the advantages described in section background, II-A. However, there might be other machine learning algorithms that could score similar or even higher.

A. *Conclusion*

There exists a distinction between JS code that fingerprints and JS code that does not. In accordance with previous research, tens of JS calls are more likely to occur in fingerprinting. It is therefore possible to use an SVM classifier to detect the scripts that perform browser fingerprinting by using these JS calls as features. The current method, though quite accurate, cannot detect everything and does show false positives. These promising results motivate future research into the use of machine learning as a detection tool for browser fingerprinting. Such detection could be used in a prevention tool, by providing a smarter method of blocking scripts.

B. *Implications*

By adding machine learning classification to the domain of detecting browser fingerprinting, this approach improves on more naive methods. It is not clear what this previous research, mentioned in section III-C related work, bases its scoring on. It might be possible to implement a system, similar to the approach discussed in this paper, for use by a blacklist provider or browser. The result might improve user privacy. Being able to block fingerprinting takes away the difficult challenge of creating a worthless browser fingerprint, as in the solutions discussed in section III-B related work.

Existing solutions are often criticised for breaking the browsing experience. Either by blocking harmless scripts or lying about important browser characteristics. Because there are false positives in the current solution, if implemented, it could also break browser characteristics. It is therefore crucial that the method is not implemented exactly as described in this paper. One solution might be the use of a lower threshold of when something is a fingerprinter, but this would likely lower the detection rate. Improvements to the current proposed method are discussed in the next section. Any enhancements might mitigate the complications caused by false positives.

On the other hand, classification of fingerprinters could also improve existing solutions. An add-on such as NoScript could be enhanced if users are aided with their decision. The same would apply for blacklist based solutions.

VII. Future Work

Currently the selected JS calls were chosen, as described in section IV-D method, by manually observing the means and deviations in occurrences of these calls from both sets. Linear SVMs allow support feature selection. There exists no such simple process with non-linear SVMs. Future research might improve on the feature selection, by comparing the performance of the classifier for different sets of features.

JS calls are not necessarily the only phenomena that show whether a script is fingerprinting [8]. Using other indications of fingerprinting alongside JS calls, might improve the classification.

The current dataset is limited by its size. As mentioned before, a larger training set hints towards better results. Future research would certainly improve on this paper, if a larger dataset could be obtained.

The SVM classifier was chosen for earlier mentioned advantages. However, there also exist other classifiers that rival SVM. Research utilising such a rival, could improve on the accuracy of detection.

If a more practical implementation is ever developed, it would be interesting to see if this could sniff out fingerprinters from the Web. It might be possible to do a small survey, where one attempts to find fingerprinters. An implementation in the form of a browser plugin, could potentially be able to block scripts that fingerprint. However, static code analysis will demand resources. Research into the performance of static code analysis would provide a valuable insight into how the user would be affected. Any improvements to the accuracy of the proposed method, will also aid in any future practical implementation.

In this paper, only classification with static code analysis is investigated. Machine learning might also improve on earlier detection solutions that utilise dynamic code analysis. These methods were usually implemented for surveys, attempting to grasp the breadth of browser fingerprinting on the Web.

References

[1] Puglisi, Silvia and Rebollo-Monedero, David and Forné, Jordi, "On web user tracking of browsing patterns for personalised advertising," *International Journal of Parallel, Emergent and Distributed Systems*, 1–20, 2017.

[2] S. Vedantam, M. Penman, *This Is Your Brain On Uber*, https://www.npr.org/2016/05/17/478266839/this-is-your-brain-on-uber, Visited on 2018-01-11.

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

[4] Yen, Ting-Fang and Xie, Yinglian and Yu, Fang and Yu, Roger Peng and Abadi, Martin, "Host Fingerprinting and Tracking on the Web: Privacy and Security Implications.," in *NDSS*, 2012.

[5] Electronic Frontier Foundation, *Panopticlick 3.0*, https://panopticlick.eff.org/, Visited on 2018-01-10.

[6] Eckersley, Peter, "How unique is your web browser?" In *Privacy Enhancing Technologies*, Springer, vol. 6205, 2010, 1–18.

[7] Laperdrix, Pierre and Rudametkin, Walter and Baudry, Benoit, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *Security and Privacy (SP), 2016 IEEE Symposium on*, IEEE, 2016, 878–894.

[8] Laperdrix, Pierre, "Browser Fingerprinting: Exploring Device Diversity to Augment Authentication and Build Client-Side Countermeasures," PhD thesis, INSA Rennes, 2017.

[9] *Clear, enable, and manage cookies in Chrome*, https://support.google.com/chrome/answer/95647?hl=en, Visited on 2018-01-22.

[10] *Enable and disable cookies that websites use to track your preferences*, https://support.mozilla.org/en-US/kb/enable-and-disable-cookies-website-preferences, Visited on 2018-01-22.

[11] *Manage cookies and website data*, https://support.apple.com/guide/safari/manage-cookies-and-website-data-sfri11471/mac, Visited on 2018-01-22.

[12] *Lightbeam for Firefox*, https://www.mozilla.org/en-US/lightbeam/, Visited on 2018-01-19.

[13] Gunn, Steve R and others, "Support vector machines for classification and regression," *ISIS technical report*, vol. 14, no. 1, pp. 5–16, 1998.

[14] scikit-learn, *Support Vector Machines*, http://scikit-learn.org/stable/modules/classes.html#module-sklearn.svm, Visited on 2018-02-01.

[15] Vert, Jean-Philippe and Tsuda, Koji and Schölkopf, Bernhard, "A primer on kernel methods," *Kernel methods in computational biology*, vol. 47, pp. 35–70, 2004.

[16] Drucker, Harris and Wu, Donghui and Vapnik, Vladimir N, "Support vector machines for spam categorization," *IEEE Transactions on Neural networks*, vol. 10, no. 5, pp. 1048–1054, 1999.

[17] Spooren, Jan and Preuveneers, Davy and Joosen, Wouter, "Mobile device fingerprinting considered harmful for risk-based authentication," in *Proceedings of the Eighth European Workshop on System Security*, ACM, 2015, 6.

[18] Giorgio Maone, *NoScript Security Suite*, https://addons.mozilla.org/nl/firefox/addon/noscript/, Visited on 2018-01-10.

[19] The Tor Project, Inc, *Tor Project*, https://www.torproject.org/, Visited on 2018-01-10.

[20] Baumann, Peter and Katzenbeisser, Stefan and Stopczynski, Martin and Tews, Erik, "Disguised chromium browser: Robust browser, flash and canvas fingerprinting protection," in *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, ACM, 2016, pp. 37–46.

[21] FaizKhademi, Amin and Zulkernine, Mohammad and Weldemariam, Komminist, "FPGuard: Detection and prevention of browser fingerprinting," in *IFIP Annual Conference on Data and Applications Security and Privacy*, Springer, 2015, 293–308.

[22] Nikiforakis, Nick and Joosen, Wouter and Livshits, Benjamin, "Privaricator: Deceiving fingerprinters with little white lies," in *Proceedings of the 24th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2015, 820–830.

[23] K. Boda., *FireGloves*, https://fingerprint.pet-portal.eu/?menu=6, Visited on 2018-01-10.

[24] Kontaxis, Georgios and Chew, Monica, "Tracking protection in firefox for privacy and performance," *arXiv preprint arXiv:1506.04104*, 2015.

[25] Yu, Zhonghao and Macbeth, Sam and Modi, Konark and Pujol, Josep M, "Tracking the trackers," in *Proceedings of the 25th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2016, 121–132.

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

[27] Rausch, Michael and Good, Nathan and Hoofnagle, Chris Jay, "Searching for Indicators of Device Fingerprinting in the JavaScript Code of Popular Websites," in *Proceedings, Midewest Instruction and Computing Symposium*, 2014.

[28] *JS Beautifier*, https://github.com/beautify-web/js-beautify, Visited on 2018-01-24.

[29] Zhao, Rui and Yue, Chuan and Sun, Kun, "A security analysis of two commercial browser and cloud based password managers," in *Social Computing (SocialCom), 2013 International Conference on*, IEEE, 2013, pp. 448–453.

[30] Sanchez-Rola, Iskander and Balzarotti, Davide and Santos, Igor, "The Onions Have Eyes: A Comprehensive Structure and Privacy Analysis of Tor Hidden Services," in *Proceedings of the 26th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2017, pp. 1251–1260.

Appendix A
Abstract Syntax Tree Example

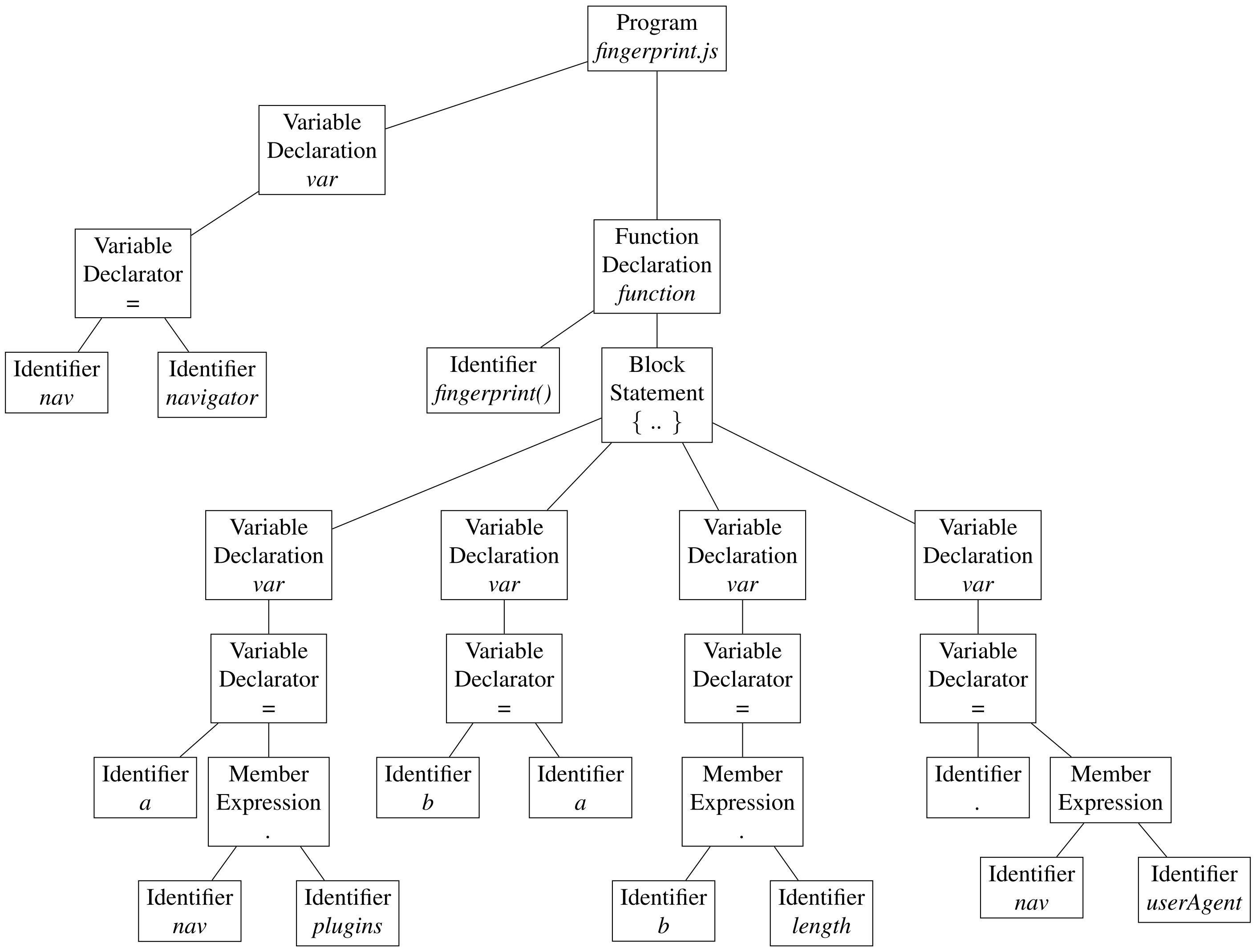

Fig. 6. The Abstract Syntax Tree of *fingerprint.js* as described in figure 2.

Appendix B
Common calls used for fingerprinting

This is the full list of features considered. The selection is explained in section IV-D1 method.

```
navigator.userAgent
navigator.appCodeName
navigator.product
navigator.productSub
navigator.vendor
navigator.vendorSub
navigator.onLine
navigator.appVersion
navigator.language
navigator.plugins.name
navigator.plugins.filename
navigator.plugins.description
navigator.plugins.length
navigator.mimeTypes
navigator.cookieEnabled()
navigator.cookieEnabled
navigator.javaEnabled()
navigator.javaEnabled
navigator.mimeTypes.enabledPlugin
navigator.mimeTypes.description
navigator.mimeTypes.suffixes
navigator.mimeTypes.type
navigator.doNotTrack
window.screen.horizontalDPI
window.screen.verticalDPI
window.screen.height
window.screen.width
window.screen.colorDepth
window.screen.pixelDepth
window.screen.availLeft
window.screen.availTop
window.screen.availHeight
window.screen.availWidth
Date().getTimezoneOffset()
Date().getTimezoneOffset
```

Appendix C
Codebase

The source code of the method proposed in this paper is open-source. A Git repository of the source code is available on: https://github.com/Timvanz/static-javascript-fingerprint-classification.

Please note that the code was designed with this particular project in mind. Feel free to contact the authors for questions that might arise.