e-Informatica Software Engineering Journal, Volume 8, Issue 1, 2014, pages: 65–78, DOI 10.5277/e-Inf140105

Malicious JavaScript Detection by Features Extraction

Gerardo Canfora*, Francesco Mercaldo*, Corrado Aaron Visaggio*
*Department of Engineering, University of Sannio
canfora@unisannio.it, fmercaldo@unisannio.it, visaggio@unisannio.it

Abstract

In recent years, JavaScript-based attacks have become one of the most common and successful types of attack. Existing techniques for detecting malicious JavaScripts could fail for different reasons. Some techniques are tailored on specific kinds of attacks, and are ineffective for others. Some other techniques require costly computational resources to be implemented. Other techniques could be circumvented with evasion methods. This paper proposes a method for detecting malicious JavaScript code based on five features that capture different characteristics of a script: execution time, external referenced domains and calls to JavaScript functions. Mixing different types of features could result in a more effective detection technique, and overcome the limitations of existing tools created for identifying malicious JavaScript. The experimentation carried out suggests that a combination of these features is able to successfully detect malicious JavaScript code (in the best cases we obtained a precision of 0.979 and a recall of 0.978).

1. Introduction

JavaScript [1] is a scripting language usually embedded in web pages with the aim of creating interactive HTML pages. When a browser downloads a page, it parses, compiles, and executes the script. As with other mobile code schemes, malicious JavaScript programs can take advantage of the fact that they are executed in a foreign environment that contains private and valuable information. As an example, a U.K. researcher developed a technique based on JavaScript timing attacks for stealing information from the victim machine and from the sites the victim visits during the attack [2]. JavaScript code is used by attackers for exploiting vulnerabilities in the user's browser, browser's plugins, or for tricking the victim into clicking on a link hosted by a malicious host. One of the most widespread attacks accomplished with malicious JavaScript is drive-by-download [3, 4], consisting of downloading (and running) malware on the victim's machine. Another example of JavaScript-based attack is represented by scripts that abuse systems resources, such as opening windows that never close or creating a large number of pop-up windows [5].

JavaScript can be exploited for accomplishing web based attacks also with emerging web technologies and standards. As an example, this is happening with Web Workers [6], a technology recently introduced in HTML 5. A Web Worker is a JavaScript process that can perform computational tasks, and send and receive messages to the main process or to other workers. A Web Worker differs from a worker thread in Java or Python in a fundamental aspect of the design: there is no sharing of the state. Web Workers were designed to execute portions of JavaScript code asynchronously, without affecting the performance of the web page. The operations performed by Web Workers are therefore transparent from the point of view of the user who remains unaware of what is happening in the background.

The literature offers many techniques to detect malicious JavaScripts, but all of them show some limitations. Some existing detection solutions leverage previous knowledge about malware, so they could be very effective against well-known attacks, but they are ineffective against zero-day attacks [7]. Another limitation of many detectors of malicious JavaScript code is that they are designed for recognizing specific kinds of attack, thus for circumventing them, attackers usually mix up different attack's types [7]. This paper proposes a method to detect malicious JavaScript that consists of extracting five features from the web page under analysis (WUA in the remaining of the paper), and using them for building a classifier. The main contribution of this method is that the proposed features are independent of the technology used and the attack implemented. So it should be robust against zero-day attacks and JavaScripts which combine different types of attacks.

1.1. Assumptions and Research Questions

The features have been defined on the basis of three assumptions. One assumption is that a malicious website could require more resources than a trusted one. This could be due to the need to iterate several attempts of attacks until at least one succeeds, to executing botnets functions, or to examining and scanning machine resources. Based on this assumption, two features have been identified. The first feature (*avgExecTime*) computes the average execution time of a JavaScript function. As discussed in [8, 9], the malware is expected to be more resource-consuming than a trusted application. The second feature (*maxExecTime*) computes the maximum execution time of JavaScript function.

The second assumption is that a malicious web page generally calls a limited number of JavaScript functions to perform an attack. This could have different justifications, i.e. a malicious code could perform the same type of attacks over and over again with the aim of maximizing the probability of success: this may mean that a reduced number of functions is called many times. Conversely, a benign JavaScript usually exploits more functions to implement the business logic of a web application [10]. One feature has been defined on this assumption (*funcCalls*) that counts the number of function calls done by each JavaScript.

The third assumption is that a JavaScript function can make use of malicious URLs for many purposes, i.e. performing drive-by download attacks or sending data stolen from the victim's machine. The fourth feature (*totalUrl*) counts the total number of the URLs into a JavaScript function, while the fifth feature (*extUrl*) computes the percentage of URLs outside the domain of the WUA.

We build a classifier by using these five features in order to distinguish malicious web applications from trusted ones; the classifier runs six classification algorithms.

The paper poses two research questions:

- RQ1: can the five features be used for discriminating malicious from trusted web pages?
- RQ2: does a combination of the features exist that is more effective than a single feature to distinguish malicious web pages from trusted ones?

The paper proceeds as follows: next section discusses related work; the following section illustrates the proposed method; the fourth section discusses the evaluation, and, finally, conclusions are drawn in the last section.

2. Related Work

A number of approaches have been proposed in the literature to detect malicious web pages. Traditional anti-virus tools use static signatures to match patterns that are commonly found in malicious scripts [11]. As a countermeasure, complex obfuscation techniques have been devised in order to hide malicious code to detectors that scan the code for extracting the signature. Blacklisting of malicious URLs and IPs [7] requires that the user trusts the blacklist provider and entails high costs for management of database, especially for guaranteeing the dependability of the information provided. Malicious websites, in

fact, change frequently the IP addresses especially when they are blacklisted.

Others approaches have been proposed for observing, analysing, and detecting JavaScript attacks in the wild, for example, using high-interaction honeypots [12–14] and low-interaction honeypots [15–17]. High-interaction honey-clients assess the system integrity, by searching for changes to the registry entries, and to the network connections, alteration of the file system, and suspect usage of physical resources. This category of honey-clients is effective, but entails high computational costs: they have to load and run the web application for analysing it, and nowadays websites contain a large number of heavy components. Furthermore, high-interaction honey-clients are ineffective with time-based attacks, and most honey-clients' IPs are blacklisted in the deep web, or they can be identified by an attacker employing CAPTCHAs [7].

Low-interaction honey-clients reproduce automatically the interaction of a human user with the website, within a sandbox. These tools compare the execution trace of the WUA with a sample of signatures: this makes this technique to fail against zero-day attacks.

Different systems have been proposed for off-line analysis of JavaScript code [3, 18–20]. While all these approaches are successful with regard to the malicious code detection, they suffer from a serious weakness: they require a significant time to perform the analysis, which makes them inadequate for protecting users at run-time. Dewald [21] proposes an approach based on a sandbox to analyse JavaScript code by merging different approaches: static analysis of source code, searching forbidden IFrames and dynamic analysis of JavaScript code's behaviour.

Concurrently to these offline approaches, several authors focused on the detection of specific attack types, such as heap-spraying attacks [22, 23] and drive-by downloads [24]. These approaches search for symptoms of certain attacks, for example the presence of shell-code in JavaScript strings. Of course, the main limitation is that such approaches cannot be used for all the threats.

Recent work has combined JavaScript analysis with machine learning techniques for deriving automatic defences. Most notably are the learning-based detection systems Cujo [25], Zozzle [26], and IceShield [27]. They classified malware by using different features, respectively: q-grams from the execution of Javascript, context's attributes obtained from AST and some DOM tree's characteristics. Revolver [28] aims at finding high similarity between the WUA and a sample of known signatures. The authors extract and compare the AST structures of the two JavaScripts. Blanc et al. [29] make use of AST fingerprints for characterizing obfuscating transformations found in malicious JavaScripts. The main limitation of this technique is the high false negatives rate due to the quasi similar subtrees.

Clone detection is a direction explored by some researchers [2, 30], consisting on finding similarity among WUA and known JavaScript fragments. This technique can be effective in many cases but not all, because some attacks can be completely original.

Wang et al. [31] propose a method for blocking JavaScript extensions by intercepting Cross-Platform Component Object Model calls. This method is based on the recognition of patterns of malicious calls; misclassification could occur with this technique so innocent JavaScript extensions could be signaled as malicious. Barua et al. [32] also faced the problem of protecting browsers from JavaScript injections of malicious code by transforming the original and legitimate code with a key. By this way, the injected code is not recognized after the deciphering process and thus detected. This method is applicable only to the code injection attacks. Sayed et al. [33] deal with the problem of detecting sensitive information leakage performed by malicious JavaScript. Their approach relies on a dynamic taint analysis of the web page which identifies those parts of the information flow that could be indicators of a data theft. This method does not apply to those attacks which do not entail sensitive data exfiltration. Schutt et al. [34] propose a method for early identification of threats within javascripts at runtime, by building a classifier which uses the

events produced by the code as features. A relevant weakness of the method is represented by evasion techniques, described by authors in the paper, which are able to decrease the performance of the classification. Tripp et al. [35] substitute concrete values with some specific properties of the document object. This allows for a preliminary analysis of threats within the JavaScript. The method seems to not solve the problem of code injection. Xu and colleagues [36] propose a method which captures some essential characteristics of obfuscated malicious code, based on the analysis of function invocation. The method demonstrated to be effective, but the main limitation is its purpose: it just detects obfuscated (malicious) JavaScripts, but does not recognize other kinds of threats.

Cova et al. [3] make use of a set of features to identify malicious JavaScript including the number and target of redirections, the browser personality and history-based differences, the ratio of string definition and string uses, the number of dynamic code executions and the length of dynamically evaluated code. They proposed an approach based on an anomaly detection system; our approach is similar but different because uses the classification.

Wang et al. [37] combine static analysis and program execution to obtain a call graph using the abstract syntax tree. This could be very effective with attacks that reproduce other attacks (this practice is very common among inexperienced attackers, known also as "script-kiddies") but it is ineffective with zero-day attacks.

Yue et al. [38] focus on two types of insecure practices: insecure JavaScript inclusion and insecure JavaScript dynamic generation. Their work is a measurement study focusing on the counting of URLs, as well as on the counting of the eval() and the document.write() functions.

Techniques known as language-based sandboxing [33, 39–42] aimed at isolating the untrusted JavaScript content from the original webpage. BrowserShield [43], FBJS from Facebook [44], Caja from Google [45], and ADsafe which is widely used by Yahoo [39], are examples of this technique. It is very effective when coping with widget and mashup webpages, but it fails if the web page contains embedded malicious code. A relevant limitation of this technique is that third parties' developers are forced to use the Software Development Kits delivered by sandboxes' producers.

Ismail et al. [46] developed a method which detects XSS attacks with a proxy that analyses the HTTP traffic exchanged between the client (web browser) and the web application. This approach has two main limitations. Firstly, it only detects reflected XSS, also known as non-persistent XSS, where the attack is performed through a single request and response. Second, the proxy is a possible bottleneck for performance as it has to analysing all the requests and responses transmitted between the client and the server. In [47], Kirda et al. propose a web proxy that analyses dynamically generated links in web pages and compares those links with a set of filtering rules for deciding if they are trusted or not. The authors leverage a set of heuristics to generate filtering rules and then leave the user to allow or disallow suspicious links. A drawback of this approach if that involving users might negatively affect their browsing experience.

3. The Proposed Method

Our method extracts three classes of features from a web application: JavaScript execution time, calls to JavaScript functions and URLs referred by the WUA's JavaScript.

To gather the required information we use:

1. dynamic analysis, for collecting information about the execution time of JavaScript code within the WUA and the called functions;
2. static analysis, to identifying all the URLs referred in the WUA within and outside the scope of the JavaScript.

The first feature computes the average execution time required by JavaScript function:

$$avgExecTime = \frac{1}{n}\sum_{k=1}^{n} t_i$$

where: t_i is the execution time of the i-th JavaScript function, and n is the number of JavaScript functions in the WUA.

The second feature computes the maximum execution time of all JavaScript functions:

$$maxExecTime = \max(t_i)$$

where t_i is the execution time of the i-th JavaScript function in the WUA.

The third feature computes the number of functions calls made by the JavaScript code:

$$funcCalls = \sum_{i=1}^{n} c_i$$

where n is the is the number of JavaScript functions in the WUA, and c_i is the number of calls for the i-th function.

The fourth feature computes the total number of URLs retrieved in a web page:

$$totalUrl = \sum_{i=0}^{m} u_i$$

where: u_i is the number of times the i-th url is called by a JavaScript function and m is the number of different urls referenced within the WUA.

The fifth feature computes the percentage of external URLs referenced within the JavaScript:

$$extUrl = \frac{\sum_{k=0}^{j} u_k}{\sum_{i=0}^{m} u_i} * 100$$

where: u_k is the number of times the k-th url is called by a JavaScript function, for j different external URLs referenced within the JavaScript, while u_i is the number of times the i-th url is called by a JavaScript function, for m total URLs referenced within the JavaScript.

We used these features for building several classifiers. Specifically, six different algorithms were run for the classification, by using the Weka suite [48]: J48, LADTree, NBTree, RandomForest, RandomTree and RepTree.

3.1. Implementation

The features extracted from the WUA by dynamic analysis were:
- execution time;
- calls to javascript function;
- number of function calls made by the javascript code.

The features extracted from the WUA by static analysis were:
- number of URLs retrieved in the WUA;
- URLs referenced within the WUA.

The dynamic features were captured with Chrome developer [49], a publicly available tool for profiling Web Applications. Each WUA was opened with a Chrome browser for a fixed time of 50 seconds, and the Chrome developer tool performed a default exploration of the WUA, mimicking user interaction and collecting the data with the Mouse and Keyboard Recorder tool [50], a software able to record all mouse and keyboard actions, and then repeat all the actions accurately.

The static analysis aimed at capturing all the URLs referenced in the JavaScript files included in the WUA. URLs were recognized through regular expressions: when an URL was found, it was compared with the domain of the WUA: if the URL's domain was different from the WUA's domain, it was tagged as an external URL.

We have created a script to automate the data extraction process. The script takes as input a list of URLs to analyse and perform the following steps:
- step 1: start the Chrome browser;
- step 2: start the Chrome Dev Tools on the panel Profiles;
- step 3: start the tool for profiling;
- step 4: confirm the inserted URL as parameter to the browser and waiting for the time required to collect profiling data;
- step 5: stop profiling;
- step 6: save data profiling in the file system;
- step 7: close Chrome;
- step 8: start the Java program to parse profiling saved;
- step 9: extract the set of dynamic features;
- step 10: save source code of the WUA;

– step 11: extract the set of static features;
– step 12: save the values of the features extracted into a database.

The dynamic features of the WUA are extracted from the log obtained with the profiling.

4. Experimentation

The aim of the experimentation is to evaluate the effectiveness of the proposed features, expressed through the research questions RQ1 and RQ2.

The experimental sample included a set of 5000 websites classified as "malicious", while the control sample included a set of 5000 websites classified as "trusted".

The trusted samples includes URLs belonging to a number of categories, in order to make the results of experimentation independent of the type of web-site: Audio-video, Banking, Cooking, E-commerce, Education, Gardening, Government, Medical, Search Engines, News, Newspapers, Shopping, Sport News, Weather.

As done by other authors [33] the trusted URLs were retrieved from the repository "Alexa" [51], which is an index of the most visited websites. For the analysis, the top ranked websites for each category were selected, which were mostly official websites of well-known organizations. In order to have a stronger guarantee that the websites were not phishing websites or did not contain threats, we submitted the URLs to a web-based engine, VirusTotal [52], which checks the reliability of the web sites, by using anti-malware software and by searching the web site URLs and IPs in different blacklists of well-known antivirus companies.

The "malicious" sample was built from the repository hpHosts [53], which provides a classification of websites containing threats sorted by the type of the malicious attack they perform. Similarly to the trusted sample, websites belonging to different threat's type were chosen, in order to make the results of the analysis independent of the type of threat. We retrieved URLs from various categories: sites engaged in malware distribution, in selling fraudulent applications, in the use of misleading marketing tactics and browser hijacking, and sites engaged in the exploitation of browser and OS vulnerabilities. For each URL belonging to the two samples, we extracted the five features defined in section 3.

Two kinds of analysis were performed on data: hypothesis testing and classification. The test of hypothesis was aimed at understanding whether the two samples show a statistically significant difference for the five features. The features that yield the most relevant differences between the two samples were then used for the classification.

We tested the following null hypothesis:

$\mathbf{H_0}$: malware and trusted websites have similar values of the proposed features.

The $\mathbf{H_0}$ states that, given the i-th feature f_i, if f_{iT} denotes the value of the feature f_i measured on a trusted web site, and f_{iM} denoted the value of the same feature measured on a malicious web site:

$$\sigma(f_{iT}) = \sigma(f_{iM}) \text{ for } i = 1, \ldots, 5$$

being $\sigma(f_i)$ the means of the (control or experimental) sample for the feature f_i.

The null hypothesis was tested with Mann-Whitney (with the p-level fixed to 0.05) and with Kolmogorov-Smirnov Test (with the p-level fixed to 0.05). Two different tests of hypotheses were performed in order to have a stronger internal validity since the purpose is to establish that the two samples (trusted and malicious websites) do not belong to the same distribution.

The classification analysis was aimed at assessing whether the features where able to correctly classify malicious and trusted WUA. Six algorithms of classification were used: J48, LadTree, NBTree, RandomForest, RandomTree, RepTree. Similarly to hypothesis testing, different algorithms for classification were used for strengthening the internal validity.

These algorithms were first applied to each of the five features and then to the groups of features. As a matter of fact, in many cases a classification is more effective if based on groups of features rather than a single feature.

4.1. Analysis of Data

Figure 1 illustrates the boxplots of each feature. Features *avgExecTime*, *maxExecTime* and *funcCalls* exhibit a greater gap between the distributions of the two samples.

Features *totalUrl*, and *extUrl* do not exhibit an evident difference between trusted and malicious samples. We recall here that *totalUrl* counts the total number of URLs in the JavaScript, while *extUrl* is the percentage of URLs outside the WUA domain contained in the script. A possible reason why these two features are similar for both the samples is that trusted websites may include external URLs due to external banners or to external legal functions and components that the JavaScript needs for execution (images, flash animation, functions of other websites that the author of the WUA needs to recall). Using external resources in a malicious JavaScript is not so uncommon: examples are drive by download and session hijacking. External resources can be used when the attacker injects a malicious web page into a benign website and needs to lead the website user to click on a malicious link (which can not be part of the benign injected website). We expect that extending this analysis to the complete WUA (not limited to JavaScript code) could produce different results: this goal will be included in the future work.

On the contrary, features *avgExecTime*, *maxExecTime* and *funcCalls* seem to be more effective in distinguishing malicious from trusted websites, which supports our assumptions.

Malware requires more execution time than trusted script code because of many reasons (*avgExecTime*, *maxExecTime*). Malware may require more computational time for performing many attempts of the attack till it succeeds. Examples may be: complete memory scanning, alteration of parameters, and resources occupation.

Some kinds of malware aim at obtaining the control of the victim machine and the command centre, once infected the victim, could occupy computational resources of the victim for sending and executing remote commands. Furthermore, some other kinds of malware could require time because they activate secondary tasks like downloading and running additional malware, as in the case of drive-by-download.

The feature *funcCalls* suggests that trusted websites have a larger number of functions called

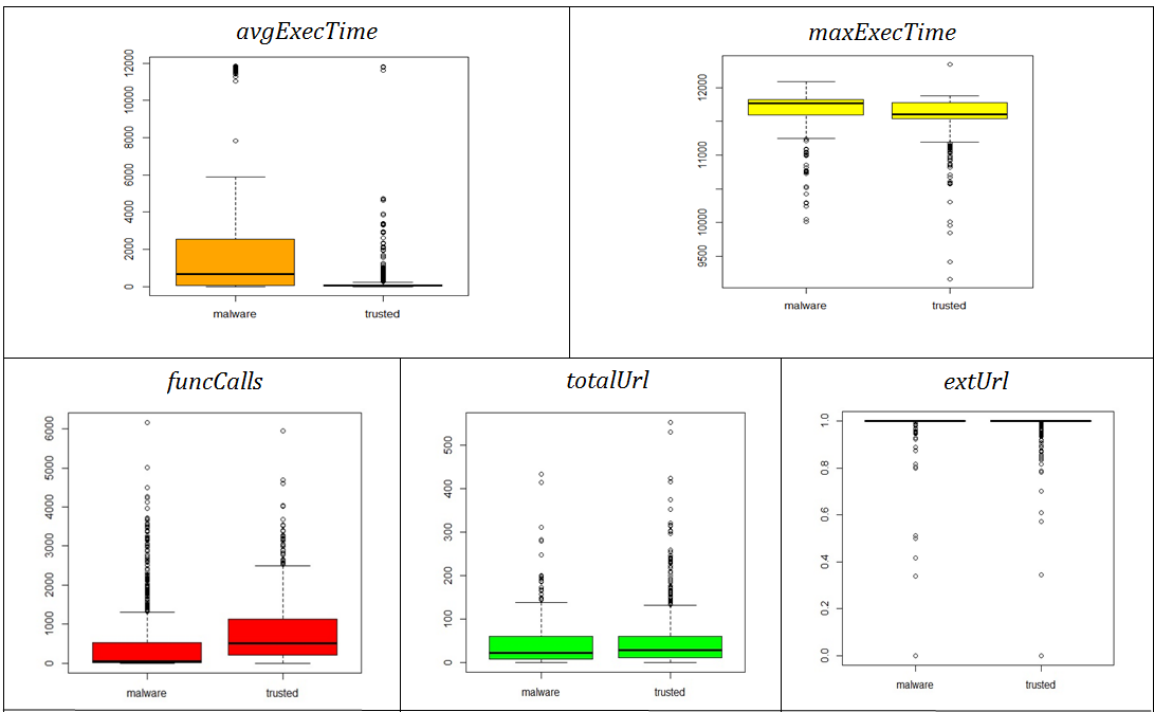

Figure 1. Boxplots of features

or function-calls. Our hypothesis for explaining this finding is that trusted websites need calling many functions for executing the business logic of the website, like data field controls, third party functions such as digital payment, elaborations of user inputs, and so on. On the contrary, malicious websites have the only goal to perform the attack, so they are poor of business functions with the only exception for the payload to execute. Instead, they perform their attack at regular intervals; for this reason malicious WUAs show a higher value of *avgExecTime* and *maxExecTime* with respect to the trusted ones.

In order to optimize the client-server interaction, the trusted website could have many functions but usually with a low computational time, in order to avoid impacting on the website usability. This allows, for example, performing controls, such as data input validation, on the client side and sending to the server only valid data.

The hypothesis test produced evidence that the features have different distributions in the control and experimental sample, as shown in Table 1.

Summing up, the null hypothesis can be rejected for the features *avgExecTime*, *maxExecTime*, *funcCalls*, *totalUrl* and *extUrl*.

With regard to classification, the training set T consisted of a set of labelled web applications (WUA, l) where the label $l \in \{trusted, malicious\}$. For each WUA we built a feature vector $F \in R^y$, where y is the number of the features used in training phase ($1 \leq y \leq 5$). To answer to RQ1 we performed five different classifications each with a single feature ($y = 1$), while for RQ2 we performed three classifications with $2 \leq y \leq 5$).

We used k-fold cross-validation: the dataset was randomly partitioned into k subsets of data. A single subsets of data was retained as the validation data for testing the model, while the remaining $k - 1$ subsets was used as training data. We repeated the process k times, each of the k subsets of data was used once as validation data. To obtain a single estimate we computed the average of the k results from the folds.

Specifically, we performed a 10-fold cross validation. Results are shown in Table 2. The rows represent the features, while the columns represent the values of the three metrics used to evaluate the classification results (precision, recall and roc-area) for the recognition of malware and trusted samples. The *Recall* has been computed as the proportion of examples that were assigned to class X, among all examples that truly belong to the class, i.e. how much part of the class was captured. The *Recall* is defined as:

$$Recall = \frac{tp}{tp + fn}$$

where tp indicates the number of true positives and fn is the number of false negatives.

The Precision has been computed as the proportion of the examples that truly belong to class X among all those which were assigned to the class, i.e.:

$$Precision = \frac{tp}{tp + fp}$$

where fp indicates the number of false positives.

The Roc Area is the area under the ROC curve (AUC), it is defined as the probability that a randomly chosen positive instance is ranked above randomly chosen negative one. The classification analysis with the single features suggests several considerations.

With regards to the recall:

– generally the classification of malicious websites is more precise than the classification of trusted websites.

Table 1. Results of the test of the null hypothesis H_0

Variable	Mann-Whitney	Kolmogorov-Smirnov
avgExecTime	0.000000	p<.001
maxExecTime	0.000000	p<.001
funcCalls	0.000000	p<.001
totalUrl	0.000000	p<.001
extUrl	0.002233	p<.001

Table 2. Precision, Recall and RocArea obtained by classifying Malicious and Trusted dataset, using the single features of the model, with the algorithms J48, LadTree, NBTree, RandomForest, RandomTree and RepTree.

Features	**Algorithm**	**Precision**		**Recall**		**RocArea**	
		Malware	Trusted	Malware	Trusted	Malware	Trusted
avgExecTime	J48	0.872	0.688	0.597	0.898	0.741	0.741
	LADTree	0.836	0.691	0.606	0.881	0.789	0.789
	NBTree	0.872	0.686	0.59	0.895	0.771	0.771
	RandomForest	0.744	0.758	0.759	0.744	0.762	0.762
	RandomTree	0.773	0.762	0.762	0.763	0.766	0.766
	RepTree	0.971	0.735	0.704	0.819	0.725	0.725
maxExecTime	J48	0.657	0.635	0.606	0.684	0.675	0.675
	LADTree	0.638	0.663	0.69	0.606	0.691	0.691
	NBTree	0.672	0.634	0.587	0.713	0.654	0.654
	RandomForest	0.683	0.703	0.718	0.667	0.775	0.775
	RandomTree	0.678	0.708	0.731	0.653	0.782	0.782
	RepTree	0.663	0.686	0.706	0.641	0.724	0.724
funcCalls	J48	0.928	0.677	0.582	0.876	0.722	0.722
	LADTree	0.816	0.678	0.587	0.868	0.75	0.75
	NBTree	0.824	0.677	0.582	0.896	0.727	0.727
	RandomForest	0.784	0.719	0.629	0.772	0.672	0.672
	RandomTree	0.782	0.683	0.646	0.765	0.696	0.696
	RepTree	0.763	0.675	0.686	0.788	0.787	0.787
totalUrl	J48	0.615	0.552	0.381	0.762	0.603	0.603
	LADTree	0.566	0.555	0.511	0.609	0.6	0.6
	NBTree	0.607	0.533	0.284	0.717	0.565	0.565
	RandomForest	0.624	0.653	0.689	0.585	0.691	0.691
	RandomTree	0.619	0.655	0.7	0.57	0.691	0.691
	RepTree	0.617	0.609	0.595	0.631	0.66	0.66
extUrl	J48	0.514	0.7	0.993	0.061	0.527	0.527
	LADTree	0.51	0.704	0.972	0.066	0.512	0.512
	NBTree	0.514	0.716	0.992	0.062	0.527	0.527
	RandomForest	0.513	0.513	0.992	0.061	0.532	0.532
	RandomTree	0.514	0.787	0.993	0.061	0.527	0.527
	RepTree	0.514	0.597	0.593	0.561	0.527	0.527

This could be due to the fact that some trusted websites have values for the features comparable with the ones measured for malicious ones. This is evident by looking at the boxplots (figure 1), which show an area of overlapping between the boxplots of the trusted and malicious websites. The problem is that some trusted websites could have values comparable with the malware while others do not. As a matter of fact, some trusted WUAs can contain more business functions than other ones, and require more client machine resources, and so on. This depends on the specific business goals of each trusted website. And, consequently, on the type and numbers of the functions that must be implemented for supporting the business goals. Except for *funcCalls*, the trusted websites' sample include a greater number of outliers than the malware sample, which is the main cause of the misclassifications of trusted websites and supports our explanation.

– the feature *extUrl* is the best in terms of recall regarding the malicious websites; in fact, its value is 0.993 using the algorithms of classification J48 and RandomTree. This feature is able to reduce the false negatives in malicious detection because external URLs are commonly used by malicious websites, for the reasons previously discussed.

– regarding the recall inherent the recognition of the trusted websites, the best feature is *avgExecTime* (recall is 0.898 with the J48 classification algorithm). This confirms the conjecture that malicious scripts tend to be more resource demanding than trusted ones.

With regards to the precision:

– the features *avgExecTime* and *funcCalls* are the best for the detection of the malicious JavaScript, with values, respectively of 0.971 (with the algorithm RepTree) and 0.928 (with the algorithm J48). This strengthens the conjecture that trusted websites make use of less computational time and a larger number of functions than malicious websites.
– the precision in the classification of sites categorized as trusted shows the maximum value 0.787 (classification of the feature *extUrl* with the algorithm RandomTree). This value is largely unsatisfactory, and it will be improved by using combinations of features, as discussed later in this section.

With regards to the roc area:

– the performances of all the algorithms are pretty the same for malware and trusted applications.
– the feature *avgExecTime* presents the maximum roc-area value equal to 0.789 with LADTree algorithm. Reasons have been discussed previously, even if it cannot be considered a good value.

In order to make the classification more effective, we run the classification algorithms by using groups of features. The first group includes the features *avgExecTime* and *funcCalls*, while the second includes *avgExecTime*, *funcCalls*, and *extUrl*. Finally, the last group is made up of all the five features extracted. The groups were made on the basis of the classification results of individual features, in order to improve both the precision and the recall of the classification.

avgExecTime and *funcCalls* were the best in class, so we grouped them together. In particular, these features were grouped together in order to obtain the maximum precision value for detecting malicious web applications.

avgExecTime, *funcCalls*, and *extUrl* were grouped together in order to obtain the maximum precision value in the detection of trusted applications. We excluded *maxExecTime* and *totalUrl* from the second phase of classification, because they produced the worst results in the first phase of classification.

The classification of the groups of features confirms (shown in Table 3) our expectations. The first set of features, *avgExecTime* and *funcCalls*, presents the maximum precision regarding malicious websites, corresponding to 0.982 with the classification algorithm J48, while in the detection of the trusted web sites the precision is 0.841 with the classification algorithm REPTree. Compared to the individual features we have therefore an improvement, in fact *avgExecTime* had a precision of 0.971 while *funcCalls* showed a precision 0.928 in the recognition of malware websites. The recall for malicious websites is 0.873 with the classification algorithm J48, while for trusted sites it is 0.897 with the classification algorithm NBTree. With respect to the recognition of malicious websites we have registered an improvement, as with individual features the obtained values were respectively 0.762 (*avgExecTime*) and 0.686 (*funcCalls*). With respect to the trusted websites, the situation is pretty similar, as the values of single features were 0.898 (*avgExecTime*) and 0.896 (*funcCalls*), i.e. slightly greater.

The second group (*avgExecTime*, *funcCalls*, *extUrl*) is very close to the first group (two values are slightly higher and two are slightly lower), but precision and recall are higher than the second group.

We can conclude that the best classification is based on

– *avgExecTime*, *funcCalls*, i.e. the average execution time (*avgExecTime*) and the cumulative number of function calls done by each portion of JavaScript code (*funcCalls*);
– *avgExecTime*, *funcCalls*, *extUrl*, i.e. the set of the features of the first group classified along with the percentage of external domain URLs that do not belong to the Web Application's domain (*extUrl*).

Table 3. Precision, Recall and RocArea obtained by classifying Malicious and Trusted dataset, using the three groups of features of the model, with the algorithms J48, LadTree, NBTree, RandomForest, RandomTree and RepTree

Features	**Algorithm**	**Precision**		**Recall**		**RocArea**	
		Malware	Trusted	Malware	Trusted	Malware	Trusted
	J48	0.982	0.823	0.873	0.88	0.888	0.888
	LADTree	0.87	0.801	0.784	0.873	0.872	0.857
avgExecTime	NBTree	0.848	0.686	0.59	0.897	0.779	0.779
funcCalls	RandomForest	0.84	0.825	0.815	0.797	0.985	0.985
	RandomTree	0.824	0.818	0.779	0.768	0.977	0.977
	RepTree	0.871	0.841	0.824	0.856	0.913	0.913
	J48	0.873	0.842	0.835	0.879	0.885	0.885
	LADTree	0.86	0.801	0.784	0.873	0.857	0.857
avgExecTime	NBTree	0.848	0.69	0.599	0.893	0.789	0.789
funcCalls	RandomForest	0.969	0.978	0.978	0.969	0.985	0.985
extUrl	RandomTree	0.97	0.979	0.98	0.97	0.979	0.979
	RepTree	0.867	0.852	0.849	0.87	0.918	0.918
	J48	0.875	0.878	0.879	0.874	0.922	0.922
avgExecTime	LADTree	0.858	0.804	0.788	0.87	0.858	0.858
maxExecTime	NBTree	0.847	0.72	0.657	0.881	0.827	0.827
funcCalls	RandomForest	0.979	0.984	0.985	0.979	0.992	0.992
totalUrl	RandomTree	0.982	0.978	0.978	0.982	0.98	0.98
extUrl	RepTree	0.877	0.871	0.87	0.879	0.927	0.927

Although the proposed features show to be effective in detecting malicious javascript, misclassification occurs however. The explanation maybe the following: each feature represents an indicator of the possibility that the JavaScript is malicious, rather than offering the certainty. The fact that in average a malicious JavaScript requires a longer execution time (as shown by boxplots) does not mean that all the benign JavaScripts require a small execution time (as outliers in boxplots show). Many payloads contained in malicious javascript entail a long time to be executed, but also some business logic of benign javascripts may require long time to be executed. For instance, a benign javascript may contain a multimedia file. The same explanation applies to justify the presence of misclassification for all the other features. Benign files could have a smaller fragmentation because they have a simpler business logic or because of the style of the programmer who has written the code.

Finally, the number of the external URLs may be high in a benign websites for several reasons: the benign websites make use of many resources or services hosted in other websites, or it has many advertisement links in its pages.

5. Conclusion and Future Work

In this paper we propose a method for detecting malicious websites that uses a classification based on five features.

Current detection's techniques usually fail against zero-day attacks and websites that merge several techniques. The proposed method should overcome these limitations, since its independent of the implementation of the attack and the type of the attack.

The selected features, combining static and dynamic analysis, respectively compute the average and maximum execution time of a JavaScript function, the number of functions invoked by the JavaScript code, and finally the number and the percentage of the URLs contained in the JavaScript code, but that are outside the domain of the WUA.

The analysis of data collected by analysing a sample of 5000 trusted and 5000 untrusted websites demonstrated that considering groups of features for classifications, rather than single features, produces better performances. As matter of fact the group (*avgExecTime* and *funcCalls*) and the group (*avgExecTime*, *funcCalls*, *extUrl*) produce high values of precision and recall, both

for the recognition of malicious websites, and for trusted websites. Regarding to the second group (*avgExecTime*, *funcCalls*, *extUrl*), the precision is 0.979 for malware websites and 0.969 for trusted ones. The recall is 0.978 for malware websites and 0.969 for trusted ones.

In summary, the two groups of features seem effective for detecting current malicious JavaScripts.

Possible evasion techniques that attackers can assume against this detection method are the following.

Concerning *avgExecTime* and *funcCalls*, the attacker should reduce the time of scripts execution and improve the fragmentation of the code. The first workaround is very difficult to implement, because the large amount of time is often a needed condition of the attacks performed. Improving the fragmentation of code is possible, but as the attacker should produce a number of functions similar to a typical trusted website, the required effort could make very expensive the development of the malicious website, and this could be discouraging. As shown in the boxplot (figure 1), the gap to fill is rather large. Our opinion is that *extUrl* is the weakest feature and so the easiest to evade, but it must be considered that in the group of features (*avgExecTime*, *funcCalls*, *extUrl*) the strength of the other ones may compensate its weakness.

Obfuscation is an evasion technique that could be effective especially with regards to *funcCalls*, *totalUrl* and *extUrl* features; future works will address this problem by studying: i) the impact of obfuscated JavaScript on the classification performances of our method; and ii) de-obfuscation methods to precisely calculate these features. Many benign websites may make use of external libraries which are highly time-consuming. In a future work we will investigate the possibility to recognize these time-consuming external libraries in order to exclude them from the computation of the feature. Additionally we plan to enforce the reliability of our findings by extending the experimentation to a larger sample, in order to enforce the external validity. Another improvement of our method consists of extending the search of URLs to the complete WUA, and not limiting it to the JavaScript scope.

References

[1] D. Flanagan, *JavaScript: The Definitive Guide*, 4th ed. O'Reilly Media, 2001. [Online]. http://shop.oreilly.com/product/9780596000486.do

[2] "Javascript and timing attacks used to steal browser data," Blackhat 2013, last visit 19th June 2014. [Online]. http://threatpost.com/JavaScript-and-timing-attacks-used-to-steal-browser-data/101559

[3] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proc. of the International World Wide Web Conference (WWW)*, 2010, pp. 281–290.

[4] C. Eilers, *HTML5 Security.* Developer Press, 2013.

[5] O. Hallaraker and G. Vigna, "Detecting malicious JavaScript code in mozilla," in *Proceedings of the 10th IEEE International Conference of Engineering of Complex Computer System*, 2005, pp. 85–94.

[6] "Web workers, W3C candidate recommendation," 2012, last visit 19th June 2014. [Online]. http://www.w3.org/TR/workers/

[7] B. Eshete, "Effective analysis, characterization, and detection of malicious web page," in *Proceedings of the 22nd International Conference on World Wide Web companion.* International World Wide Web Conferences Steering Committee, 2013, pp. 355–360.

[8] L. Martignoni, R. Paleari, and D. Bruschi, "A framework for behavior-based malware analysis in the cloud," in *Proceedings of the 5th International Conference on Information Systems Security*, 2009, pp. 178–192.

[9] M. F. Zolkipli and A. Jantan, "An approach for malware behavior identification and classification," in *Proceedings of International Conference of Computer Research and Development*, 2011.

[10] C. Ardito, P. Buono, D. Caivano, M. Costabile, and R. Lanzilotti, "Investigating and promoting UX practice in industry: An experimental study," *International Journal of Human-Computer Studies*, Vol. 72, No. 6, 2014, pp. 542–551.

[11] "ClamAV. Clam antivirus." last visit 19th June 2014. [Online]. http://clamav.net

[12] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iFRAMEs point to us," in *Proc. of USENIX Security Symposium*, 2008.

[13] C. Seifert and R. Steenson, "Capture honeypot client (capture hpc)," Victoria University of Wellington, NZ, 2006. [Online]. https://projects.honeynet.org/capture-hpc

[14] Y. M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowsk, S. Chen, and S. T. King, "Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities," in *Proc. Of Network and Distributed System Security Symposium (NDSS)*, 2006.

[15] A. Büscher, M. Meier, and R. Benzmüller, "Throwing a monkeywrench into web attackers plans," in *Proc. Of Communications and Multimedia Security (CMS)*, 2010, pp. 28–39.

[16] A. Ikinci, T. Holz, and F. Freiling, "Monkey-spider: Detecting malicious websites with low-interaction honeyclients," in *Proc. of Conference "Sicherheit, Schutz und Zuverl'assigkeit (SICHERHEIT)*, 2008, pp. 891–898.

[17] J. Nazario, "A virtual client honeypot," in *Proc. Of USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[18] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in *Proc. of the International World Wide Web Conference (WWW)*, 2011, pp. 197–206.

[19] S. Karanth, S. Laxman, P. Naldurg, R. Venkatesan, J. Lambert, and J. Shin, "Zdvue: Prioritization of JavaScript attacks to discover new vulnerabilities," in *Proceedings of the Fourth ACM Workshop on Artificial Intelligence and Security (AISEC 2011)*, 2011, pp. 637–652.

[20] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," Microsoft Research, Tech. Rep. MSR-TR-2011-94, 2011. [Online]. http://research.microsoft.com/pubs/152601/rozzle-tr-10-25-2011.pdf

[21] A. Dewald, T. Holz, and F. Freiling, "ADSandbox: sandboxing JavaScript to fight malicious websites," in *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*, 2010, pp. 1859–1864.

[22] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *In Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009, pp. 88–106.

[23] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: A defense against heap-spraying code injection attacks," in *Proc. of USENIX Security Symposium*, 2009.

[24] L. Lu, V. Yegneswaran, P. A. Porras, and W. Lee, "Blade: An attack-agnostic approach for preventing drive-by malware infections," in *Proc. of Conference on Computer and Communications Security (CCS)*, 2010, pp. 440–450.

[25] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient detection and prevention of drive-by-download attacks," in *26th Annual Computer Security Applications Conference (ACSAC)*, 2010, pp. 31–39.

[26] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser JavaScript malware detection," in *Proc. of USENIX Security Symposium*, 2010, pp. 3–3.

[27] M. Heiderich, T. Frosch, and T. Holz, "Iceshield: Detection and mitigiation of malicious websites with a frozen dom," in *Proceedings of Recent Adances in Intrusion Detection (RAID)*, 2011, pp. 281–300.

[28] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegle, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware," in *Proceedings of the 22nd USENIX conference on Security*, 2013, pp. 637–652.

[29] G. Blanc, D. Miyamoto, M. Akiyama, and Y. Kadobayashi, "Characterizing obfuscated JavaScript using abstract syntax trees: Experimenting with malicious scripts," in *Proceedings of International Conference of Advanced Information Networking and Applications Workshops*, 2012.

[30] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," School of Computing Queen's University at Kingston, Ontario, TR 2007-541, 2007.

[31] P. Wang, L. Wang, J. Xiang, P. Liu, N. Gao, and J. Jing, "MJBlocker: A lightweight and run-time malicious JavaScript extensions blocker," in *Proceedings of International Conference on Software Security and Reliability*, 2013.

[32] A. Barua, M. Zulkernine, and K. Weldemariam, "Protecting web browser extension from JavaScript injection attacks," in *Proceedings of International Conference of Complex Computer Systems*, 2013.

[33] B. Sayed, I. Traore, and A. Abdelhalim, "Detection and mitigation of malicious JavaScript using information flow control," in *Proceedings of Twelfth Annual Conference on Privacy, Security and Trust (PST)*, 2014.

[34] K. Schutt, M. Kloft, A. Bikadorov, and K. Rieck, "Early detection of malicious behaviour in JavaScript code," in *Proceedings of AISec 2012*,

2012.

[35] O. Tripp, P. Ferrara, and M. Pistoia, "Hybrid security analysis of web JavaScript code via dynamic partial evaluation," in *Proceedings of International Symposium on Software Testing and Analysis*, 2014.

[36] W. Xu, F. Zhang, and S. Zhu, "JStill: Mostly static detection of obfuscated malicious JavaScript code," in *Proceedings of International Conference on Data and Application Security and Privacy*, 2013.

[37] Q. Wang, J. Zhou, Y. Chen, Y. Zhang, and J.Zhao, "Extracting URLs from JavaScript via program analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 627–630.

[38] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 961–970.

[39] J. Politz, S. Eliopoulos, A. Guha, and S. Krishnamurthi, "Adsafety: Type-based verification of JavaScript sasndboxing," in *Proceedings of the 20th USENIX conference on Security*, 2011.

[40] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript," in *ECOOP 2010-Object-Oriented*, 2011, pp. 1–25.

[41] M. Finifter, J. Weinberger, and A. Barth, "Preventing capability leaks in secure JavaScript substes," in *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[42] A. Taly, U. Erlingsson, J. Mitchell, M. Miller, and J. Nagra, "Automated analysis of security-critical JavaScript apis," in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 363–379.

[43] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: Vulnerability-driven filtering of dynamic html," *ACM Transactions on the Web*, Vol. 1, No. 3, 2007.

[44] "Facebook SDK for JavaScript," last visit 13th October 2014. [Online]. https://developers.facebook.com/docs/javascript

[45] "Google Caja," last visit 13th October 2014. [Online]. https://developers.google.com/caja/

[46] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability," in *Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, Vol. 2, 2014.

[47] E. Kirda, C.Kruegel, G. Vigna, and N. Jovanic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, pp. 330–337.

[48] "Weka 3: Data mining software in Java," last visit 19th June 2014. [Online]. http://www.cs.waikato.ac.nz/ml/weka/

[49] "Chrome DevTools overview," last visit 19th June 2014. [Online]. https://developers.google.com/chrome-developer-tools/

[50] "Robot Soft - mouse and keyboard recorder," last visit 13th October 2014. [Online]. http://www.robot-soft.com/

[51] "Actionable analytics for the web," last visit 19th June 2014. [Online]. http://www.alexa.com/

[52] "VirusTotal," last visit 19th June 2014. [Online]. https://www.virustotal.com/

[53] "hpHosts onliine," last visit 19th June 2014. [Online]. http://www.hosts-file.net/