Detecting Obfuscated JavaScripts using Machine Learning

Simon Aebersold*, Krzysztof Kryszczuk*, Sergio Paganoni†, Bernhard Tellenbach*, Timothy Trowbridge*
*Zurich University of Applied Sciences, Switzerland
†GovCERT.ch, Reporting and Analysis Centre for Information Assurance MELANI

***Abstract*—JavaScript is a common attack vector for attacking browsers, browser plug-ins, email clients and other JavaScript enabled applications. Malicious JavaScripts redirect victims to exploit kits, probe for known vulnerabilities to select a fitting exploit or manipulate the Document Object Model (DOM) of a web page in a harmful way. Malicious JavaScript code is often obfuscated in order to make it hard to detect using signature-based approaches. Since the only other reason to use obfuscation is to protect intellectual property, the share of scripts which are both benign and obfuscated is quite low, and could easily be captured with a whitelist. A detector that can reliably detect obfuscated JavaScripts would therefore be a valuable tool in fighting malicious JavaScripts. In this paper, we present a method for automatic detection of obfuscated JavaScript using a machine-learning approach. Using a dataset of regular, minified and obfuscated samples from a content delivery network and the Alexa top 500 websites, we show that it is possible to distinguish between obfuscated and non-obfuscated scripts with precision and recall around 99%. We also introduce a novel set of features, which help detect obfuscation in JavaScripts. Our results presented here shed additional light on the problem of distinguishing between malicious and benign scripts.**

***Index Terms*—Computer security; Machine learning; Pattern analysis; Classification algorithms; JavaScript, Random Forest; Malicious**

I. Introduction

JavaScript is omnipresent on the web. Almost all websites make use of it and there are a lot of other applications, such as Portable Document Format (PDF) forms or HyperText Markup Language (HTML) e-mails, where JavaScript plays an important role. This strong dependence creates an attack opportunity for individuals looking for an entry point into a victims system. The main functionalities of a malicious JavaScript are reconnaissance and exploitation, and cross-site scripting (XSS) vulnerabilities in web applications.

JavaScript exploit kits belong to the first category of functionality and typically contain code for identification of the victim's browser and its plug-ins. Most of the malicious JavaScripts are obfuscated in order to hide what they are doing and to evade detection by signature based security systems. Since the only other reason to use obfuscation is to protect intellectual property, the share of benign obfuscated scripts is quite low and could probably be captured with a whitelist. A detector that can reliably detect obfuscated JavaScripts would therefore be a valuable tool in fighting malicious JavaScripts.

The most common method to address the problem of malicious JavaScripts is having malware analysts write rules for anti-virus or intrusion detection systems that identify common patterns in obfuscated (or non-obfuscated) malicious scripts. While signature based detection is good at detecting known malware, it often fails to detect it when obfuscation is used to alter the features captured by the signature [1]. Furthermore, keeping up with the attackers and their obfuscation techniques is a time consuming task. This is why a lot of research effort is put into alternative solutions to identify/classify malicious JavaScripts. One area is to automate at least parts of the manual analysis required to identify whether or not a script is malicious and to craft suitable signatures. JSDetox [2] and Wepawet [3] are two solutions that help with the dynamic analysis of JavaScript samples.

Likarish *et al.* [4] take another approach and apply machine learning algorithms to detect obfuscated malicious JavaScript samples. The authors use a set of 15 features like the number of strings in the script or the percentage of white-space that are largely independent from the language and JavaScript semantics. The results from their comparison of four machine learning classifiers (naive bays, ADTree, SVM and RIPPER) are very promising: the precision and recall (see III-E for a definition) of the SVM classifier is 92% and 74.2%. But since their study originates from 2009, it is unclear how recent trends like the minification of JavaScripts (see II-A) would impact on their results.

A more recent study from Kaplan *et al.* [5] addresses the problem of detecting obfuscated scripts using a Bayesian classifier. They refute the assumption made by previous publications that obfuscated scripts are mostly malicious and advertise their solution as filter for projects where users can submit applications to a software repository such as a browser extension gallery for browsers like Google Chrome or Firefox. Also techniques such as AdSafe [6], severely restrict what is allowed JavaScript and what not to simplify analysis.

Wang *et al.* [7] propose another machine learning based solution to separate malicious and benign JavaScript. They compare the performance of ADTree, NaiveBayes and SVM machine learning classifiers using a set of 27 features of which some are similar to those of Likarish *et al.* [4]. Their results suggest a significant improvement over the work of Likarish *et al.*

In this paper, we present a method for automatic detection of obfuscated JavaScript using a machine-learning approach. We confirm results from other researchers that using approaches based on machine learning, it is possible to distinguish between obfuscated and non-obfuscated scripts with precision and recall above 95%. Our results complement previous research in that they expose a substantial challenge to obtain those good results. Our results suggest that it is difficult to train detectors to be robust versus changes in the way obfuscation is done. If there are no samples of scripts obfuscated with a specific obfuscation tool or method, detection rates drop

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-475-6

significantly.

Today's JavaScript is mostly minified code. The second contribution of this paper is an investigation whether minification has an impacts on the detection of obfuscated JS using machine learning techniques. Finally, we shed additional light on the problem of distinguishing between malicious and benign scripts using a custom database. In our experiments we could not confirm the promising results from previously published research, where similar features and machine learning approach was taken.

The rest of the paper is organized as follows. Section II briefly explains the different JavaScript classes, which we aimed to separate in this work. In Section III, we discuss the machine-learning approach adopted in the presented work, as well as the discriminatory features and classifiers we used. Section IV presents our results, followed by a discussion and conclusions in Section V.

II. Syntactic and functional varieties of JavaScript

The client-side JavaScript for JavaScript-enabled applications can be attributed to one of the following four classes: regular, minified, obfuscated and malicious. The regular class contains the scripts as they have been written by their developers. These scripts are typically easy to read and understand by human beings. Obfuscation and minification are code modifications that change the syntax but not the functionality of the code. In this work, we refer to different syntactic and functional varieties of JavaScript as *classes*.

A. Minification

Since the introduction of the YUI Compressor [8] and other minification tools, more and more JavaScript in the Internet is minified. It is considered good practice to concatenate and minify JavaScript files to arrive at smaller file sizes and fewer requests. Minification removes spaces, line breaks and renames functions and variables to obtain a more compact version of the script. While this makes the scripts harder to read and understand for a human, the program flow remains the same.

B. Obfuscation

In contrast to minifiers, obfuscation tools do modify the program flow with the goal to make it hard to understand while keeping the original functionality. Many obfuscation techniques exist. For example, encoding obfuscation encodes strings using hexadecimal character encoding or Unicode encoding to make strings harder to read. Other obfuscation steps involve hiding code in data to execute it later using the eval JavaScript function. The following listing shows a sample use of the latter technique:

```
1 var a = "ale";
2 a += "rt(";
3 a += "'hello'";
4 a += ");";
5 eval(a);
```

Listing 1. A simple example for data obfuscation

C. Malicious vs benign

The dichotomy benign/malicious is of functional rather than syntactic nature. In contrast to the regular, minified and obfuscated class, scripts in the malicious class can have a regular form or make use of minifiers or obfuscators. This makes it difficult to detect those scripts using features that focus on differentiating the first three classes only. Previous work sometimes conflates obfuscation with maliciousness. In this work and in prior art (see [5]), it is explicitly stated that neither all obfuscated code is malicious nor is all malicious code obfuscated. Although formally speaking malicious JavaScript does not have to be obfuscated, in practice, it usually is.

III. Machine learning approach to JavaScript classification

In order to evaluate the feasibility and accuracy of distinguishing between different classes of JavaScript, we adopted a classical machine learning approach. We collected a database containing a number of instances representing each of the classes of interest, i.e., regular, minified, obfuscated, benign and malicious. For each of the samples in the database we extracted a set of discriminatory features, which we list in Table II below. The extracted features form fixed-length feature vectors, which in turn are used for training and evaluation of classifiers.

A. Data Set

Our dataset consists of data from three different sources: (1) the complete list of JavaScripts available from the *jsDelivr* content delivery network, (2) the *Alexa Top 500* websites and (3) a set of malicious JavaScript samples from the Swiss Reporting and Analysis Centre for Information Assurance *MELANI*.

jsDeliver: contains a large number of JavaScript libraries and files in both a regular and a minified version. Since the files are subject to manual review and approval and should not make use of obfuscation, we used the regular versions of the files as a basis for our evaluation. After a preprocessing step including rule-based filtering, de-duplication as well as manual sampling to check and make sure that the assumed properties (minified, obfuscated or malicious) are met, we generated three additional file sets from these files. For the first set, we processed the files with uglifyjs [9], the most popular JavaScript minifier to obtain a minified version of them. uglifyjs works by extracting an abstract syntax tree (AST) from the JavaScript source and then transforming it to an optimized (smaller) one. For the second and third set, we used the Dean Edwards' Packer [10] and javascriptobfuscator.com [11] to create obfuscated versions of these files. Note that the second and third set can also be considered to be minified. The two obfuscators remove whitespaces and make the scripts more compact. Scripts that are first minified and then obfuscated look similar or are the same as when obfuscation is applied only. Applying obfuscation and then minification might lead to partial

de-obfuscation (e.g., decoding of encoded strings) and is therefor unlikely to be used in practice.

Alexa Top 500: To have a more comprehensible representation of actual scripts found on websites [12], we created a set of files consisting of the JavaScripts found on the Alexa Top 500 websites [13]. To extract the scripts from these websites, we parsed them with BeautifulSoup [14] and extracted all scripts that were either inlined (e.g., `<script>alert("foo");</script>`) or referenced via external files (e.g., `<script type="text/javascript" src="filename.js" ></script>`). Since we make no assumption about the properties of these files other than that they are non-malicious, no preprocessing was performed except for de-duplication.

MELANI: The fileset from MELANI contains only malicious samples. Most of the malicious samples in the set are either JS droppers used in drive-by-download attacks or Exploit Kits for exploiting vulnerabilities in browser plugins. Most samples are at least partially obfuscated and seem to make use of different obfuscation techniques and tools.

B. Preprocessing

Since a manual inspection of a random subset of the non-minified files downloaded from jsDeliver showed that 10% of them were minified nevertheless, we preprocessed them by applying the following simple heuristic:

- Remove files with less than 5 lines
- Remove files if less than 1% of all characters are spaces.
- Remove files where more than 10% of all lines are longer than 1000 characters).

While we did not inspect all of the removed files (around 10% of the files), a manual inspection of a subset of them showed no false positives. We did not find false negatives in the files that were not removed. It must be noted, however, that the above heuristic may not necessary be valid for other JavaScript datasets. If the data source contains a large number of small JavaScript snippets, the first rule might prove problematic.

In a next step, we used DoubleKiller [15] to remove all duplicate files. DoubleKiller compares files based on file name, size, modification date and content (CRC32). After preprocessing and de-duplication of the jsDeliver fileset a total of 4218 unique JavaScript files were left. After de-duplication of the Alexa Top 500 fileset, a total of 9459 files remained.

C. Feature Selection

For our experiments reported in this paper, we selected a set of 21 features derived from manual inspection, related work ([4], [16]) and analysis of the histograms of candidate features. For example, observations showed that obfuscated scripts often make use of encodings using hexadecimal or Unicode characters (F17) and often remove white spaces (F8). Furthermore, some rely on splitting a job in a lot of functions (F14) and almost all use a lot of strings (F7) and

TABLE I
DATA COLLECTIONS

Collection	Properties	#Files
jsDelivr.com	regular	4218
jsDelivr.com	minified (uglifyjs)	4218
jsDelivr.com	obfuscated (Dean Edwards Packer)	4218
jsDelivr.com	obfuscated (javascriptobfuscator.com)	4218
Alexa Top 500	unknown	9459
MELANI	malicious, obfuscated	132

are lacking comments (F9). An example of a comparison of feature distributions across classes is shown in Figure 1. Here, it can be noted that if a script has 70% or more of its characters in strings, this is a strong indication that the file is obfuscated or malicious.

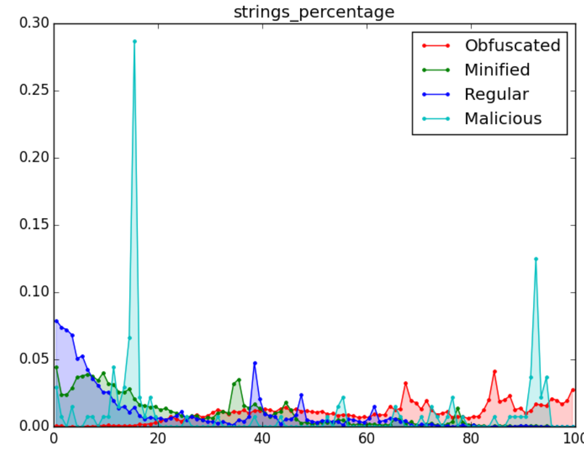

Fig. 1. Histogram for feature F7: The share of scripts that have x% of their characters in strings.

Table II lists the discriminatory features we used for training and evaluation of the classifiers in the reported experiments. These features are complemented with 26 features reflecting the frequency of 26 different JavaScript keywords: break, case, catch, class, continue, do, else, false, finally, for, if, instanceof, new, null, return, switch, this, throw, true, try, typeof, var, while, toString, valueOf and undefined. While the present set yielded promising results in our experiments, further investigations are required to determine an optimal set of classification features for the problem. The features labeled as 'new' in Table II are a novel contribution of the present paper. The special JavaScript elements used in feature F15 are elements often used and renamed (to conceal their use) in obfuscated or malicious scripts. This includes the following functions, objects and prototypes:

- **Functions**: eval, unescape, String.fromCharCode, String.charCodeAt
- **Objects**: window, document
- **Prototypes**: string, array

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-475-6

TABLE II
DISCRIMINATORY FEATURES

Feature	Description	Used in:
F1	total number of lines	[4]
F2	avg. # of chars per line	[4]
F3	# chars in script	[4]
F4	% of lines ¿1000 chars	new
F5	Shannon entropy of the file	[16]
F6	avg. string length	[4]
F7	share of chars belonging to a string	new
F8	share of space characters	[4]
F9	share of chars belonging to a comment	[4]
F10	# of `eval` calls divided by F3	new
F11	avg. # of chars per function body	new
F12	share of chars belonging to a function body	new
F13	avg. # of arguments per function	[4]
F14	# of function definitions divided by F3	new
F15	# of special JavaScript elements divided by F3	new
F16	# of renamed special JavaScript elements divided by F3	new
F17	share of encoded characters (for example `\u0123` or `\x61`)	[4]
F18	share of backslash characters	new
F19	share of pipe characters	new
F20	# of array accesses using dot or bracket syntax divided by F3	new

D. Feature Extraction

To extract the above features, we implemented a Node.js application traversing the abstract syntax tree (AST) generated by Esprima [17], a JavaScript parser compatible with Mozilla's SpiderMonkey Parser API [18]. Traversing the AST and extracting all of the above features for an average JavaScript library with around 20K characters takes around 330ms.

E. Machine Learning

The extracted set of feature vectors was used to train and evaluate three different classifiers:

- **Linear Discriminant Analysis (LDA)** Due to it's simple design it avoids function overfitting. This means it is still possible to overtrain this classifier, but only by training insufficient amount of data.
- **Random Forest (RF)** Random Forest uses a tree based approach. In comparison to decision trees it is less prone to overfitting because it selects a random subset of features to build multiple decision tress.
- **Support Vector Machine (SVM)** SVM works by searching a hyperplane in a feature space that separates labels/-classes. We use a radial basis function kernel (RBF) with parameters γ=0.03, C=8.0 to capture non linearities.

For exhaustive details on the used classifiers, the reader is referred to [19]. We used scikit-learn [20], which contains the implementation of the above mentioned classifiers. We performed following steps per experiment:

1) Normalization of the data using the StandardScaler of scikit-learn
2) Random partitioning of the data set to be evaluated into training and testing subsets. With one exception (see IV-A), 60% of the data are used for training, 40% for testing.
3) Training and testing of the classifiers. The partitioning and the training and testing is done 10 times (10-fold cross validation).
4) The results from the 10 rounds are averaged and the standard deviation is calculated.

We report the (p)recision, (r)ecall, (f1)-score and (s)upport for each considered class and considered classifier. Precision is the number of true positives divided by the number of true positives and false positives. Recall is the number of true positives divided by the number of true positives and the number of false negatives. The F1 Score conveys the balance between precision and recall and is equal to $2 * \frac{precision * recall}{precision + recall}$. Finally, support is the total number of scripts tested for a specific label. Note that since we used 10-fold cross-validation, this number is the sum of the scripts tested in the 10 runs.

IV. Results

In this section, we present how the partitioning of the dataset impacts on classification accuracy and how well the classifiers are able to distinguish obfuscated from non-obfuscated scripts. Finally, we show our results from experiments where the classifiers had to distinguish malicious from benign samples.

A. Partitioning and Accuracy

To check the impact of the size of the training set on the observed accuracy, we first tested all splits from 1% to 99% of test versus training data and calculated the share of scripts whose label is predicted correctly. Figure 2 shows the impact of the split of training and test data on the observed accuracy. The x-axis shows the test dataset size in %, the y-axis shows the accuracy in %. The training dataset size is equal to 100% minus the test dataset size. Over 95% of the scripts are labeled correctly for all classifiers if at least 15% of the data vectors is used for training. At 60% training data, 2 out of 3 classifiers reach a plateau in accuracy, hence we used 60% of our data for training.

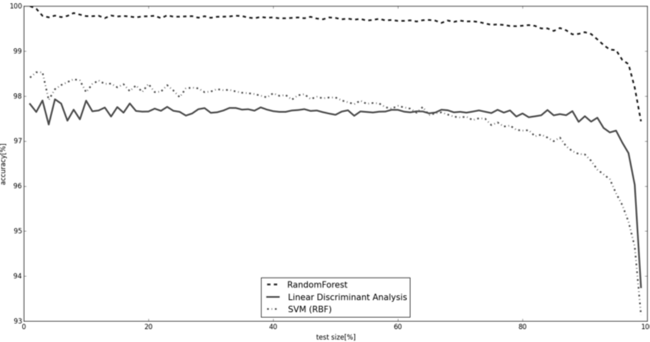

Fig. 2. Impact of the split of training and test data on the observed accuracy.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-475-6

B. Obfuscated vs. Non-Obfuscated

To test the performance of the three machine learning classifiers, we conducted the following three experiments. For the first experiment, we used the **jsDelivr.com** data set and labeled the regular and minified files as non-obfuscated, and we labeled the files processed with one of the obfuscator tools as obfuscated. Figure 3a shows the classification results after 10-fold cross-validation and a split of 60%/40% for training and test datasets. On average, per cross-validation round, only 0.60% non-obfuscated scripts out of 3377 were classified as obfuscated and only 1.67% obfuscated scripts out of 3366 were classified as non-obfuscated.

In order to verify the performance of the classifiers trained with the jsDelivr.com dataset when deployed to classify a broad set of scripts from various sources, we used the JavaScripts found in the **Alexa Top 500** websites as the evaluation dataset. Figure 3b shows the results of this experiment. 99.4% of the scripts were labeled non-obfuscated and 0.6% as obfuscated. A manual inspection of the 52, 17 and 140 scripts for the SVM, random forest and LDA classifier revealed that about 30% of these scripts are true positives whilst 70% are false positives. The false negative (FN) rate is not measurable without manually classifying all the Alexa scripts. A manual inspection of a random sample of 50 of the scripts labeled as non-obfuscated did not contain obfuscated scripts. Based on this limited sample inspection we assume the database contains a negligible number of obfuscated scripts.

Figure 3 shows the results for the third experiment where we trained the classifiers with the full jsDelivr.com dataset and then tested them with the MELANI dataset. Since all of the scripts in the MELANI dataset are obfuscated and therefore have the correct label, not false positives are possible resulting in a precision of 100%. However, there are quite a lot of false negatives which results in low (for SVM) or even very low (for RF/LDA) recall.

		SVM	RF	LDA
Non Obfuscated	p	99.35%	99.81%	99.17%
	r	99.40%	99.97%	99.38%
	f1	98.87%	98.89%	99.27%
	s	33770	33770	33770
Obfuscated	p	99.40%	99.97%	99.37%
	r	98.33%	99.80%	99.16%
	f1	98.86%	98.89%	99.26%
	s	33660	33660	33660

(a) Classification results for the **jsDelivr.com** data set with 10-fold cross-validation and a split of 60%/40% for training and test data.

		SVM	RF	LDA
Non Obfuscated	p	100.00%	100.00%	100.00%
	r	99.45%	99.82%	98.52%
	f1	99.72%	98.91%	99.25%
	s	9459	9459	9459

(b) Classification results for the **Alexa Top 500** data set for the classifiers trained with the jsDelivr.com data set, based on the assumption that it contains no obfuscated scripts.

		SVM	RF	LDA
Obfuscated	p	100.00%	100.00%	100.00%
	r	60.61%	19.70%	22.73%
	f1	75.47%	32.91%	37.04%
	s	132	132	132

(c) Classification results for the **MELANI** data set when the classifiers are trained with the jsDelivr.com data set. All scripts in the MELANI data set are obfuscated but with different obfuscation techniques and tools than used in the jsDelivr.com data set.

Fig. 3. Performance of the classifiers with respect to distinguishing between obfuscated and non-obfuscated scripts, for different test data sets.

C. Malicious vs. Benign

In the previous experiments, we focused on detecting obfuscation rather then maliciousness. Because our ultimate goal is to be able to distinguish between benign and malicious JavaScript, we trained and evaluated the classifiers to distinguish between benign and malicious scripts as well. The MELANI data set with malicious files is not large or representative enough for this result to be statistically significant. However, the results provide an indication whether the set of features and the machine learning approach can be used to detect malicious JavaScript. For this experiment, the jsDeliver.com and MELANI data sets were both serving as training and test data using a 10-fold cross-validation using a 60%/40% split. Figure 4b shows the results of this experiment. Less than one benign script has been classified as malicious per round and therefore precision and recall for benign samples is high.

Figure 4 contains the results of our last experiment where we deployed a classifier trained with the jsDelivr.com and MELANI data sets to classify the Alexa Top 500 dataset. Since all of the scripts in this data set are labeled as benign, the precision is 100%. The recall of above 99% shows, that only a few of the Alexa Top 500 script were labeled as malicious. A verification of theses scripts did not reveal any malicious content.

V. Discussion and Conclusions

The results presented in Section IV-B give a summary of the evaluation of a machine-learning approach to distinguishing obfuscated vs. non-obfuscated JavaScripts. The results show that if the training dataset contains a representative set of samples of a particular type of obfuscation, it is likely to be reliably detected in the testing phase. One intriguing question presents itself: is the classifier learning a particular syntactic structure of a particular type of obfuscator, or do different types of obfuscation share some inherent characteristics that can be captured in the learning phase. The data in Figure 3b containing the results obtained for the Alexa 500 dataset suggest that the obfuscation techniques may share certain common syntactic characteristics. A more detailed analysis is required to identify the root cause and to improve classification

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-475-6

		SVM	RF	LDA
Benign	p	99.87%	99.84%	99.59%
	r	99.98%	100.00%	99.58%
	f1	99.93%	99.92%	99.59%
	s	67414	67414	67414
Malicious	p	97.65%	99.76%	48.63%
	r	83.88%	79.22%	49.08%
	f1	90.25%	88.31%	48.85%
	s	546	546	546

(a) Classification results for the combination of **jsDelivr.com** and the **MELANI** data set with 10-fold cross-validation and a split of 60%/40% for training and test data.

		SVM	RF	LDA
Benign	p	100.00%	100.00%	100.00%
	r	99.65%	99.97%	99.26%
	f1	99.83%	99.98%	99.63%
	s	9459	9459	9459

(b) Classification results for the **Alexa Top 500** dataset, when the classifiers are trained with the combined **jsDelivr.com** and **MELANI** datasets.

Fig. 4. Performance of the classifiers with respect to distinguishing malicious vs. benign scripts, for different test data sets.

results. However, low recall for the MELANI dataset 3 clearly suggests that there might be limits that could be challenging to overcome if custom obfuscation strategies or tools are used.

Our results presented in this paper suggest that it may be feasible to detect malicious JavaScripts. As the numbers reported in Figure 4 indicate, the precision and recall on the task of discriminating between malicious and benign scripts is high. These results, however, must be approached with caution. Since the database of malicious scripts was limited and much smaller than that of benign ones, it is not evident that the classifier is capturing the actual syntactic characteristics correlated with the malicious behavior. It is possible that the mere syntactic structure of the scripts in the MELANI database is sufficiently different from that of the jsDelivr.com to allow an accurate classification. As mentioned in the Introduction, the malicious script behavior is due to the script's functionality and not syntax per se. An in-depth analysis of the link between the functionality and syntax of a malicious code must be performed in order to deliver conclusive results.

We envision to continue our efforts towards understanding of the problem of automatic detection of malicious JavaScript code by collecting more representative datasets. Given such a dataset, an analysis of the statistical distribution of syntactic features and their dependence on the malicious code behavior will be studied. Consequently, we intent do develop a dedicated set of classification features insensitive to the type of obfuscation, which will allow for automatic detection of malicious JavaScript.

REFERENCES

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

[2] sven_t, "JSDetox," last accessed on 2016-01-30. [Online]. Available: http://www.relentless-coding.com/projects/jsdetox

[3] "Wepawet," last accessed on 2016-01-30. [Online]. Available: http://wepawet.iseclab.org/

[4] P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques," in Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on, Oct 2009, pp. 47–54.

[5] S. Kaplan, B. Livshits, B. Zorn, C. Siefert, and C. Curtsinger, ""nofus: Automatically detecting" + string.fromcharcode (32)+" obfuscated".tolowercase()+" javascript code," Microsoft Research, 2011.

[6] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "Adsafety: Type-based verification of javascript sandboxing," CoRR, vol. abs/1506.07813, 2015. [Online]. Available: http://arxiv.org/abs/1506.07813

[7] W.-H. Wang, Y.-J. LV, H.-B. Chen, and Z.-L. Fang, "A static malicious javascript detection using svm," in Proceedings of the International Conference on Computer Science and Electronics Engineering, vol. 40, 2013, pp. 21–30.

[8] J. Lecomte, "Introducing the YUI Compressor," last accessed on 2016-01-30. [Online]. Available: http://www.julienlecomte.net/blog/2007/08/13/introducing-the-yui-compressor/

[9] M. Bazon, "UglifyJS," last accessed on 2016-01-30. [Online]. Available: http://lisperator.net/uglifyjs/

[10] D. Edwards, "Dean Edwards Packer," last accessed on 2016-01-30. [Online]. Available: http://dean.edwards.name/packer/

[11] "JavaScript Obfuscator," last accessed on 2016-01-30. [Online]. Available: http://javascriptobfuscator.com/

[12] P. Likarish and E. Jung, "A targeted web crawling for building malicious javascript collection," in Proceedings of the ACM first international workshop on Data-intensive software management and mining. ACM, 2009, pp. 23–26.

[13] "Alexa Top 500 Global Sites," last accessed on 2016-01-30. [Online]. Available: http://www.alexa.com/topsites

[14] L. Richardson, "Beautiful Soup," last accessed on 2016-01-30. [Online]. Available: http://www.crummy.com/software/BeautifulSoup/

[15] "DoubleKiller," last accessed on 2016-01-30. [Online]. Available: http://www.bigbangenterprises.de/en/doublekiller/

[16] B.-I. Kim, C.-T. Im, and H.-C. Jung, "Suspicious malicious web site detection with strength analysis of a javascript obfuscation," International Journal of Advanced Science and Technology, vol. 26, 2011, pp. 19–32.

[17] A. Hidayat, "Esprima JavaScript Parser," last accessed on 2016-01-30. [Online]. Available: http://esprima.org/

[18] "SpiderMonkey Parser API," last accessed on 2016-01-30. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API

[19] R. O. Duda, P. E. Hart, and D. G. Stork, Pattern Classification, 2nd Edition, 2001.

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