Detecting Obfuscated JavaScripts from Known and Unknown Obfuscators using Machine Learning

Bernhard Tellenbach
Zurich University of Applied Sciences
Switzerland
Email: tebe@zhaw.ch

Sergio Paganoni
SecureSafe / DSwiss AG
Zurich, Switzerland
Email: sergio.paganoni@dswiss.com

Marc Rennhard
Zurich University of Applied Sciences
Switzerland
Email: rema@zhaw.ch

***Abstract*—JavaScript is a common attack vector to probe for known vulnerabilities to select a fitting exploit or to manipulate the Document Object Model (DOM) of a web page in a harmful way. The JavaScripts used in such attacks are often obfuscated to make them hard to detect using signature-based approaches. On the other hand, since the only legitimate reason to obfuscate a script is to protect intellectual property, there are not many scripts that are both benign and obfuscated. A detector that can reliably detect obfuscated JavaScripts would therefore be a valuable tool in fighting JavaScript based attacks. In this paper, we compare the performance of nine different classifiers with respect to correctly classifying obfuscated and non-obfuscated scripts. For our experiments, we use a data set of regular, minified, and obfuscated samples from jsDeliver and the Alexa top 5000 websites and a set of malicious samples from MELANI. We find that the best of these classifiers, the boosted decision tree classifier, performs very well to correctly classify obfuscated and non-obfuscated scripts with precision and recall rates of around 99 percent. The boosted decision tree classifier is then used to assess how well this approach can cope with scripts obfuscated by an obfuscator not present in our training set. The results show that while it may work for some obfuscators, it is still critical to have as many different obfuscators in the training set as possible. Finally, we describe the results from experiments to classify malicious obfuscated scripts when no such scripts are included in the training set. Depending on the set of features used, it is possible to detect about half of those scripts, even though those samples do not seem to use any of the obfuscators used in our training set.**

***Index Terms*—Machine learning; Classification algorithms; JavaScript; Obfuscated; 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 attack opportunities for individuals by using malicious JavaScripts, which may provide them with an entry point into a victim's system. The main functionalities of a malicious JavaScript are reconnaissance, exploitation, and cross-site scripting (XSS) vulnerabilities in web applications.

The JavaScripts used in such attacks are often obfuscated to make them hard to detect using signature-based approaches. On the other hand, the only legitimate reason to obfuscate a script is to protect intellectual property. Our evaluation of the prevalence of such scripts on the Alexa top 5000 home pages suggests that this is fairly uncommon. One reason for this might be that a lot of JavaScript code on these pages is code from JavaScript libraries that are available to the public anyway. If it is indeed the case that there are not too many scripts that are both benign and obfuscated, it should be easy to capture these with a whitelist. A detector that can reliably detect obfuscated JavaScript code would then be a valuable tool in fighting malicious JavaScript based attacks. But even if there would be a lot of obfuscated benign JavaScript code, a detector could play an important role in that it helps to filtering such scripts and feed them to a component that performs a more in-depth analysis.

The most common method to address the problem of malicious JavaScripts is having malware analysts write rules for anti-malware 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. 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. See Section V for details.

In this paper, we analyze and extend our approach to automatically detect obfuscated JavaScripts using machine learning presented in [1]. Our paper makes the following three contributions. First, we make use of the cloud based Microsoft Azure Machine Learning Studio [2] to quickly create, train, evaluate and deploy predictive models of nine different classifiers and to do an analysis of what the most descriptive features are. The top performing one, the boosted decision tree classifier, was not among the three classifiers tested in [1]. Second, using the boosted decision tree classifier we perform a comprehensive analysis of how well this approach can cope with scripts obfuscated by an obfuscator not present in our training set. As pointed out in [1], such an analysis would be quite desirable since malicious JavaScripts are likely to make

use of a custom obfuscation approaches. Finally, we describe the results from experiments to classify *malicious* obfuscated scripts when no such scripts are included in the training set.

For our experiments, we use the same data set as in [1] with two important modifications. First, we use the scripts from the Alexa top 5000 instead of the top 500 websites and second, we perform a rigorous preprocessing to get rid of scripts that are identical or almost the same as other scripts in the data set (see Section III). Not doing this could produce results that are better than they should be if there are scripts that appear on almost all of the pages (e.g., Google Analytics scripts).

The rest of the paper is organized as follows. Section II briefly explains the different JavaScript classes that we distinguish in this work. In Section III, we discuss our data set, the preprocessing steps performed, feature selection and the machine learning methodology and tools. Section IV presents our results, followed by a discussion of results IV. Section V discusses related works and Section VI concludes the paper.

II. Syntactic and functional varieties of JavaScript

Client-side JavaScript for JavaScript-enabled applications can be attributed to one of the following four classes: regular, minified, obfuscated, and malicious. Note that only regular, minified, and obfuscated are disjoint classes and that we distinguish only obfuscated, non-obfuscated, and malicious JavaScripts in the reminder of this paper.

A. Regular JavaScripts

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.

B. Minified JavaScripts

Minified scripts are more compact versions of the regular scripts. To achieve this, minifiers such as the YUI Compressor [3] remove space and new line characters that only exist to make the code easier to read for humans. Some of the minifiers do also rename functions and variables to get rid of long function or variable names. While this makes the scripts harder to read and understand for a human, the program flow stays the same. Minification main purpose is to reduce bandwidth usage when loading JavaScripts.

C. Obfuscated JavaScripts

Obfuscation tools keep the original functionality of the code but modify the program flow with the goal to make it hard to understand. 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 (code unfolding). The following listing shows a simple example of the latter technique:

```
var a = "ale";
a += "rt(";
a += "'hello'";
a += ");";
eval(a);
```

Listing 1. A simple example of code unfolding

Note that the obfuscated files can also be considered minified. The 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 only obfuscation is applied. Applying obfuscation and then minification might lead to partial de-obfuscation (e.g., decoding of encoded strings) and is therefore unlikely to be used in practice.

D. Malicious JavaScripts

Whether or not a JavaScript is malicious is a question of its semantics and not of its syntax. Hence, a malicious JavaScript could be a regular, minified or obfuscated one. Previous work sometimes conflates obfuscation with maliciousness. In this work and in prior art (see [4]), it is explicitly stated that neither is all obfuscated code malicious nor is all malicious code obfuscated. However, in practice, it appears that at least for now, most malicious scripts are indeed obfuscated, as all of the recent malicious JavaScripts samples we collected in the wild were obfuscated.

III. Machine learning approach to JavaScript classification

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

A. Data Set

Our data set consists of data from three different sources: (1) the complete list of JavaScripts available from the *jsDelivr* content delivery network, (2) the *Alexa Top 5000* 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 regular and minified form. We use the regular form of the files as a basis for our evaluation.

Alexa Top 5000: To have a more comprehensible representation of actual scripts found on websites [5], we downloaded the JavaScripts found on the Alexa Top 5000 home pages [6]. To extract the scripts from these websites, we parsed them with BeautifulSoup [7]

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>`).

MELANI: The fileset from MELANI contains only malicious samples. Most of the malicious samples in the set are either JS droppers used in malspam campaigns or Exploit Kits (EK) resources for exploiting vulnerabilities in browser plugins. All samples are at least partially obfuscated and seem to make use of different obfuscation techniques and tools. The composition of the malicious data set is shown in Table I.

TABLE I. MALICIOUS DATA SET COMPOSITION

Name	Description	Count
CrimePack EK	Landing pages and iFrame injection code	2001
JS-Droppers	Malicious samples from different malspam campaigns	419
Angler EK	Landing pages	168
RIG EK	Landing pages	60
Misc	Different samples from other EKs (Nuclear Pack, Phoenix, BlackHole)	58

For our evaluation, we make the following three assumptions about the files from jsDelivr and the Alexa Top 5000 home pages: these files are non-malicious, non-minified and non-obfuscated.

Assuming that there are no malicious scripts in the files downloaded from the top 5000 home pages should be quite safe. The same is true for the files from jsDelivr since they are subject to manual review and approval. Nevertheless, we checked the scripts with Windows Defender and in contrast to the set of well-known malicious JavaScripts, Windows Defender did not raise any alarm.

The second assumption that these scripts are not minified, is very unlikely to hold since making use of minified scripts has become quite popular. In order to make this assumption hold, a preprocessing step is required to remove scripts that are not minified from the data set. Only then we have a clean starting point for the generation of the seven additional file sets (see III-B for details).

The last assumption about the absence of obfuscated scripts should hold for the jsDelivr data set since these scripts are subject to manual review and approval. It should also be true for the Alexa Top 5000 data set because there is little reason that home pages contain JavaScripts that need to be protected by obfuscating them. To check whether this is true, we inspected a random subset of about 150 scripts and found none that was obfuscated. Furthermore, we inspected those scripts that are later reported to be obfuscated by our classifier (supposedly false-positives) and found that from the 173 files only 15 were indeed obfuscated. However, when considering our results in relation to the presence or absence of a specific obfuscator in the data set, we cannot be sure that the Alexa Top 5000 data set does not contain scripts obfuscated by an obfuscator whose characteristics (as captured by our feature vector) are very different from the characteristics of the other obfuscators. Note that even if this were the case, it would not invalidate our results but confirm our findings concerning the presence or absence of a specific obfuscator.

In summary, after the preprocessing step, which removes minified scripts and does some additional sanitation of the dataset (see III-B for details), our data set should have the assumed properties and contain regular, non-obfuscated and non-malicious JavaScript files only.

Based on this set of files, we generated seven additional sets of files. For the first set, we processed the files with uglifyjs [8], 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 to seventh set, we used six different JavaScript ofuscators:

- **javascriptobfuscator.com standard:** To use this commercial obfuscator [9], we wrote a C# application that queries its web API with the default settings plus the parameters MoveStrings, MoveMembers, ReplaceNames. The version used was the one online on the 28th of July 2016.
- **javascriptobfuscator.com advanced:** Since the two parameters DeepObfuscation and EncryptStrings change the way the resulting scripts look like significantly, we added them to the configuration from above to create another file set.
- **javascript-obfuscator:** This obfuscator is advertised as free offline alternative to [9]. We used version 0.6.1 in its default configuration.
- **jfogs:** This is a javascript obfuscator [10] developped by zswang. We used version 0.0.15 in its default configuration.
- **jsobfu:** This is the obfuscator [11] used by the Metasploit penetration testing software to obfuscate JavaScript payloads. We used its default configuration with one iteration.
- **closure:** The Closure Compiler [12] has not been developed to obfuscate JavaScripts but to make them download and run faster. Nevertheless, it makes most JavaScripts that contain more than a few lines of code hard to read and understand even when JavaScript beautifiers are applied to them. Scripts with a few lines of code are often left unchanged. That is why the set of scripts obfuscated with this tool is smaller than the others. We obfuscated only scripts that are at least 1500 characters long. We used version 20160822 with option *–compilation_level SIMPLE*.

The reason why we did not use the old but well-known Dean Edwards' Packer [13] from [1] is that it may create parsable but semantically incorrect JavaScripts. For example, in some cases, this obfuscator removed entire parts of the

script because it uses regular expressions instead of a parser to identify multi-line comments correctly.

B. Preprocessing

The preprocessing of the files downloaded from jsDelivr and the Alexa Top 5000 home pages is divided into the following three steps:

1) Removal of duplicate and similar files
2) Removal of minified files
3) Removal of files that cannot be parsed

The preprocessing starts with a total of 42378 files downloaded from Alexa Top 5000 home pages and 4224 files from the jsDelivr data set used in [1].

The removal of duplicate and similar files in the first preprocessing step is performed using the tool ssdeep [14], a program for computing fuzzy hashes. ssdeep can match inputs that have homologies. It outputs a score about how similar two files are. We remove 13234 (Alexa) and 587 (jsDelivr) files that had a score of 90 or higher with 75 of them having a score equal or higher to 99. Not removing such files could produce results that are better than they should be if the same script appears in the training and the testing set. The impact is even worse if the same or a slightly modified script appears not just twice but multiple times in the training and testing data sets.

In the next step, we remove minified files downloaded from the Alexa Top 5000 home pages using the following heuristics:

- Remove files with fewer 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).

14490, approximately half of the remaining files were minified and therefore removed. Note that in [1], this heuristic has also be applied to the jsDelivr files even though they should be non-minified. A manual inspection of a random subset of supposedly non-minified files showed that around 10% of them were minified.

It is important to point out that by doing this, we get rid of small scripts (fewer than 5 lines), which is likely to make classification of such scripts difficult. This limitation could be used to split an obfuscated script into multiple parts and (probably) circumvent detection. As a countermeasure, one would have to detect such behavior.

The third preprocessing step removes any of the remaining original jsDelivr and Alexa Top 5000 scripts, where the parsing of the script, or of one of its transformed versions (minified, obfuscated), failed. After this step, the data set contains the number of samples listed in Table II. Overall, there are 101974 samples. Note that since the closure compiler does not perform well on small files (no obfuscation), we are only obfuscating samples with more then 1500 chars. Therefore, the number of samples reported there is significantly smaller than for the other obfuscators.

TABLE II. DATA COLLECTIONS

Collection	Properties	#Samples
jsDelivr.com	regular	3403
jsDelivr.com	minified (uglifyjs)	3403
jsDelivr.com	obfuscated (closure)	2004
jsDelivr.com	obfuscated (javascript-obfuscator)	3403
jsDelivr.com	obfuscated (javascriptobfuscator.com standard)	3403
jsDelivr.com	obfuscated (javascriptobfuscator.com advanced)	3403
jsDelivr.com	obfuscated (jfogs)	3403
jsDelivr.com	obfuscated (jsobfu)	3403
Alexa Top 5000	unknown / potentially non-obfuscated	9519
Alexa Top 5000	minified (uglifyjs)	9512
Alexa Top 5000	obfuscated (closure)	6825
Alexa Top 5000	obfuscated (javascript-obfuscator)	9519
Alexa Top 5000	obfuscated (javascriptobfuscator.com standard)	9516
Alexa Top 5000	obfuscated (javascriptobfuscator.com advanced)	9516
Alexa Top 5000	obfuscated (jfogs)	9519
Alexa Top 5000	obfuscated (jsobfu)	9517
MELANI	malicious and obfuscated (see Table I)	2706

C. Feature Selection

For our experiments reported in this paper, we selected a set of 45 features derived from manual inspection, related work [15], [16], and analysis of the histograms of candidate features. For example, observations showed that obfuscated scripts often make use of encodings using hexadecimal, Base64 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 are lacking comments (F9).

Table III lists the discriminatory features we used for training and evaluation of the classifiers in the reported experiments. These features are complemented with 25 features reflecting the frequency of 25 different JavaScript keywords: *break*, *case*, *catch*, *continue*, *do*, *else*, *false*, *finally*, *for*, *if*, *instanceof*, *new*, *null*, *return*, *switch*, *this*, *throw*, *true*, *try*, *typeof*, *var*, *while*, *toString*, *valueOf* and *undefined*. The rationale behind the selection of these keywords is that if control flow obfuscation [17] is used, the frequency of these keywords might differ significantly.

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 III 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, object

TABLE III. DISCRIMINATORY FEATURES

Feature	Description	Used in:
F1	total number of lines	[15]
F2	avg. # of chars per line	[15]
F3	# chars in script	[15]
F4	% of lines >1000 chars	new
F5	Shannon entropy of the file	[16]
F6	avg. string length	[15]
F7	share of chars belonging to a string	new
F8	share of space characters	[15]
F9	share of chars belonging to a comment	[15]
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	[15]
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 (e.g., `\u0123` or `\x61`)	[15]
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
F21-F45	frequency of 25 common JavaScript keywords	new

D. Feature Extraction

To extract the above features, we implemented a Node.js application traversing the abstract syntax tree (AST) generated by Esprima [18], a JavaScript parser compatible with Mozilla's SpiderMonkey Parser API [19].

E. Machine Learning

To train and evaluate the machine learning algorithms, we decided to use Azure Machine Learning [2] (Azure ML) instead of a more traditional local approach. Azure ML is a cloud-based predictive analytics service that makes it possible to quickly create, train, evaluate, and deploy predictive models as analytics solutions. To design and run the experiments, we used Azure Machine Learning Studio, which provides an efficiently usable collaborative drag-and-drop tool.

Azure ML offers different classification algorithms [20]. Given the flexibility of the cloud-based service, we trained and evaluated several of them:

- Averaged Perceptron (AP)
- Bayes Point Machine (BPM)
- Boosted Decision Tree (BDT)
- Decision Forrest (DF)
- Decision Jungle (DJ)
- Locally-Deep Support Vector Machine (LDSVM)
- Logistic Regression (LR)
- Neural Network (NN)
- Support Vector Machine (SVM)

For a quick introduction into these classifiers and a comparison of their advantages and disadvantages, the Azure ML documentation [21] provides a concise overview. For more details about some of these algorithms, the reader is referred to [22].

For each experiment that we performed, the steps in the following list were carried out. These steps guarantee a sound machine learning approach with clear separation of testing data and training data.

1) Normalization of the data in the case of SVM-based classifiers using the Azure ML default normalizer (with the other classifiers, normalization is not required).
2) Partitioning of the the data into a *testing set*, a *training set*, and a *validation set*.
   a) First, the testing set is constructed. The samples to include in this set depends on the experiment (see Section IV).
   b) The remaining data is randomly partitioned into a training set and a validation set, using a split of 60%/40%.
   c) We always use stratified partitioning, which guarantees that the data in each set is representative for the entire data set.
3) Training of the classifier using the training set and optimizing it using the validation set.
4) Assessing the performance of the fully-trained classifier using the testing set.

For each script in the testing set, classification works as follows: The classifier computes a probability $p \in [0, 1]$ that states how likely the script is obfuscated. The probability is then mapped to the discrete labels *obfuscated* if $p \geq t$ and *non-obfuscated* if $p < t$, where t is the *threshold*. We set the threshold always to 0.5 to make the different experiments comparable. In practice, this threshold can be used to fine-tune the classification: Setting it to a higher value (e.g., 0.8) increases the probability that a script labeled as obfuscated is truly obfuscated (true positive), but also implies a higher rate of false negatives (obfuscated scripts falsely labeled as non-obfuscated). Conversely, setting it to a value below 0.5 increase true negatives at the cost of more false positives.

For each experiment, 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. High precision (close to 1) means that most scripts labeled as obfuscated are indeed obfuscated. Recall is the number of true positives divided by the number of true positives and false negatives. High recall (close to 1) means that most of the obfuscated scripts are indeed labeled correctly as obfuscated without missing many of them. The F1 score conveys the balance between precision and recall, is computed as $2 * \frac{precision * recall}{precision + recall}$ and should ideally be close to 1. Finally, support is the total number of scripts tested for a specific label.

IV. RESULTS

In this section, we present the results of the three main experiments we performed. First, we show the performance of all nine different classifiers with respect to correctly classifying

obfuscated and non-obfuscated scripts. The best of these classifiers is used in the further experiments. Next, we demonstrate how well this classifier is capable of correctly classifying scripts that were obfuscated with an unknown obfuscator, i.e., an obfuscator that was not used for any of the scripts in the training set. Finally, we describe the results from experiments to classify malicious obfuscated scripts when no such scripts are included in the training set.

A. Obfuscated vs. Non-Obfuscated

In the first series of experiments, we analyzed the performance of the classifiers with respect to correctly classifying obfuscated and non-obfuscated scripts. We used the entire data set (see Table II) and labeled the regular and minified files as non-obfuscated, the files processed with one of the obfuscator tools as obfuscated, and the malicious files also as obfuscated. 30% of all scripts are used in the training set and the other 70% are used for the training and validation sets, using a split of 60%/40%.

In the first experiment, all 45 features as described in Section III-C were used. All nine classifiers listed in Section III-E were trained and optimized using the training and validation sets and evaluated using the testing set. Table IV shows the results. The upper half shows the performance to classify non-obfuscated script correctly while the lower half shows the same for the obfuscated scripts. It can be seen that the best results can be achieved using a boosted decision tree classifier. With this classifier, only 80 of 7752 non-obfuscated scripts were classified as obfuscated (false positive rate of 1.03%) and only 73 of 22842 obfuscated scripts were classified as non-obfuscated (false negative rate of 0.32%). Overall, boosted decision tree was the only classifier that achieved F1-scores above 99% for both classifying obfuscated and non-obfuscated scripts.

At the bottom end with respect to classification performance, there are the averaged perceptron, logistic regression, and support vector machine classifiers. All three of them performed quite poorly. The explanation is that all of them are linear models (the support vector machine classifier in Azure ML only supports a linear kernel), which is apparently not well suited to classify obfuscated and non-obfuscated scripts.

Next, we performed the same experiment as above but instead of using all features, we only used the features that are most descriptive for correct classification. One advantage of using fewer features is that it reduces the time and memory requirements to train a classifier, but as we will see later, it has additional benefits when trying to classify scripts that are obfuscated with an unknown obfuscator. To determine the most descriptive features, we used Pearson's correlation. For each feature, Pearson's correlation returns a value that describes the strength of the correlation with the label of the scripts.

Table V lists the 20 most descriptive features based on Pearson's correlation, in descending order. The rightmost column shows the value of the Pearson's correlation and the column 'Feature' references the corresponding feature in Table III if the feature is also included in that table. The table contains several interesting findings. First of all, by comparing Table V with Table III, we can see that only five of the features that were described in previous works [15], [16] are among the 20 most descriptive features while 15 of them are new features that were introduced by us. Also, it appears that quite simple features such as the frequencies of some JavaScript keywords are well suited to distinguish between obfuscated and non-obfuscated scripts, as nine of them made it into the list.

Table VI shows the performance of all nine classifiers when using only the 20 most descriptive features listed in Table V instead of all features. It can be seen that in general, the performance is a little lower compared to Table IV, but the difference is small in most cases. For instance, in the case of the boosted decision tree classifier, the F1-scores were reduced by 1.12% and 0.42% resulting in 97.89% and 99.25%. This allows two conclusions: First, using only the 20 most descriptive features instead of all 45 features does not reduce classification performance significantly. Second, as 15 of the features in Table V are newly introduced features and only five of them have been used in previous works, the newly added features provide a significant improvement to classify the scripts.

To justify that using the 20 most descriptive features is a reasonable choice, we analyzed the performance when using the most descriptive 5, 10, 15,. . . , 40 features with the boosted decision tree classifier. Figure 1 shows the F1-scores depending on the number of used features. As expected, using fewer features results in lower performance while using more features increases the performance, getting closer and closer to the performance when using all features. In addition, Figure 1 shows that using 20 features is a good compromise between computational requirements during training and performance of the trained classifier because on the one hand, using 20 features provides a substantial improvement compared to using only 15 features and on the other hand, using more than 20 features only provides small further benefits.

As the number of malicious scripts in the data set is small compared to the others, it is important to have a more detailed look at the classification performance of these scripts. Table VII depicts the results when evaluating only the malicious scripts in the testing set. As all malicious scripts are labeled obfuscated, the figure only contains results to classify obfuscated scripts correctly. For the same reason, false positives cannot occur, which implies a precision of 100%. To assess the results, it is therefore best to use the recall value and comparing this value with the ones in Table IV and VI. Doing this, it can be seen that the recall value of malicious scripts is about 1% lower than of the other scripts. However, both recall values are still above 98%, which clearly shows that classifying malicious scripts still works well despite the relatively low fraction of malicious scripts in the data set.

TABLE IV. PERFORMANCE OF THE CLASSIFIERS TO CLASSIFY NON-OBFUSCATED AND OBFUSCATED SCRIPTS, USING ALL FEATURES

		AP	BPM	BDT	DF	DJ	LDSVM	LR	NN	SVM
Non	p	80.46%	92.44%	99.06%	98.50%	97.93%	93.53%	78.31%	95.64%	81.65%
Obfuscated	r	66.31%	78.03%	98.97%	98.14%	98.10%	88.40%	68.28%	90.02%	66.82%
	F1	72.70%	84.63%	99.01%	98.32%	98.02%	90.89%	72.95%	92.74%	73.50%
	s	7752	7752	7752	7752	7752	7752	7752	7752	7752
Obfuscated	p	89.21%	92.61%	99.65%	99.37%	99.36%	96.14%	89.68%	96.68%	89.39%
	r	94.54%	97.73%	99.68%	99.49%	99.30%	97.92%	93.58%	98.61%	94.90%
	F1	91.80%	95.10%	99.67%	99.43%	99.33%	97.02%	91.59%	97.63%	92.07%
	s	22842	22842	22842	22842	22842	22842	22842	22842	22842

TABLE V. 20 MOST PREDICTIVE DISCRIMINATORY FEATURES

Feature	Description	Used in	Corr.
F18	share of backslash characters	new	0.238
F9	share of chars belonging to a comment	[15]	0.236
	frequency of keyword if	new	0.233
F15	# of special JavaScript elements divided by F3	new	0.221
F4	% of lines >1000 chars	new	0.219
	frequency of keyword false	new	0.209
F17	share of encoded characters (e.g., `\u0123` or `\x61`)	[15]	0.208
F8	share of space characters	[15]	0.203
	frequency of keyword true	new	0.194
F20	# of array accesses using dot or bracket syntax divided by F3	new	0.160
F12	share of chars belonging to a function body	new	0.158
	frequency of keyword return	new	0.139
	frequency of keyword var	new	0.133
F7	share of chars belonging to a string	new	0.119
	frequency of keyword toString	new	0.112
F5	Shannon entropy of the file	[16]	0.106
F2	avg. # of chars per line	[15]	0.102
	frequency of keyword this	new	0.084
	frequency of keyword else	new	0.081
	frequency of keyword null	new	0.081

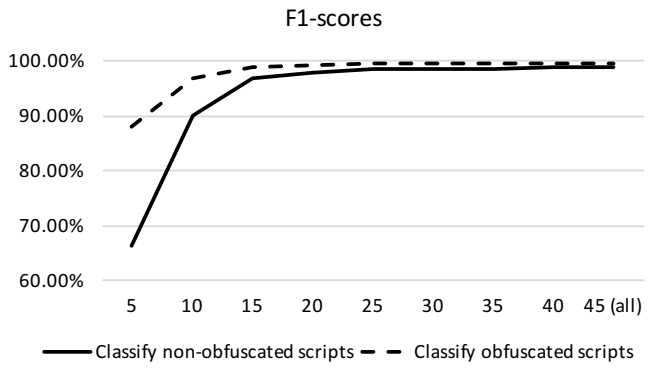

Figure 1. Performance of the Boosted Decision Tree classifier to classify non-obfuscated and obfuscated scripts, depending on the used number of most descriptive features.

To conclude this first series of experiments, we summarize the most relevant findings:

- Using our data set, classification between obfuscated and non-obfuscated scripts works well. The best classifier, boosted decision tree, yields F1-scores above 99% when using all 45 features.
- Using only the 20 most descriptive features, classification performance gets lower. However, the boosted decision tree classifier still achieves F1-scores close to and above 98% with the benefit of reduced time and memory requirements to train the classifier.
- Compared to the features used in previous works, the newly added features provide a significant improvement to classify the scripts.
- Even though the number of malicious scripts in the data set is relatively small, classifying them delivers only slightly lower performance as with the other scripts.

In the remainder of the paper, we will focus on the boosted decision tree classifier, as this has demonstrated to be the best classifier to classify obfuscated and non-obfuscated scripts.

B. Detecting Unknown Obfuscators

In the experiments performed above, all three sets (training set, validation set, and testing set) included scripts obfuscated with all different obfuscators that are used in our data set. This implies that the trained classifier 'knows' about all obfuscators and as a result, the evaluation using the testing set exhibited good classification performance. In reality, however, new obfuscators will be developed and used and ideally, the classifier should also perform well in classifying scripts that are obfuscated with such a new obfuscator.

To evaluate the performance to classify obfuscated scripts that were obfuscated using an unknown obfuscator, we first excluded the malicious scripts from the data set, which guarantees we are using a well-defined set of obfuscators. Then, we took the scripts that are obfuscated with a specific obfuscator (say obfuscator $Obf_{unknown}$) and put them into a testing set 1. From the remaining scripts, we put 30% into a testing set 2 and the rest was split into a training set and a validation set, using a split of 60%/40%. Note that this means that the scripts obfuscated with obfuscator $Obf_{unknown}$ are only present in testing set 1 and not included in any of the other sets. Training and validation sets were then used to train a boosted decision tree classifier and the trained classifier was evaluated

TABLE VI. PERFORMANCE OF THE CLASSIFIERS TO CLASSIFY NON-OBFUSCATED AND OBFUSCATED SCRIPTS, USING THE 20 MOST DESCRIPTIVE FEATURES

		AP	BPM	BDT	DF	DJ	LDSVM	LR	NN	SVM
Non	p	80.78%	74.43%	98.36%	97.35%	96.93%	94.51%	77.19%	95.65%	82.35%
Obfuscated	r	65.16%	63.97%	97.42%	95.18%	94.34%	85.13%	66.38%	89.33%	64.04%
	F1	72.13%	68.80%	97.89%	96.25%	95.61%	89.58%	71.38%	92.38%	72.05%
	s	7752	7752	7752	7752	7752	7752	7752	7752	7752
Obfuscated	p	88.90%	88.33%	99.13%	98.37%	98.10%	95.12%	89.11%	96.46%	88.65%
	r	94.74%	92.54%	99.45%	99.12%	98.98%	98.32%	93.34%	98.62%	95.34%
	F1	91.73%	90.39%	99.25%	98.75%	98.54%	96.69%	91.18%	97.53%	91.87%
	s	22842	22842	22842	22842	22842	22842	22842	22842	22842

TABLE VII. PERFORMANCE OF THE BDT CLASSIFIER TO CORRECTLY CLASSIFY MALICIOUS SCRIPTS AS OBFUSCATED

		BDT (all features)	BDT (20 features)
Obfuscated	p	100.00%	100.00%
	r	98.40%	98.52%
	F1	99.19%	99.25%
	s	811	811

using both testing sets. Classifying the scripts in testing set 2 should work well as it includes only obfuscators that are also included in the training set. Classifying the scripts in testing set 1 shows the performance of the classifier to classify scripts that were obfuscated with the unknown obfuscator $\text{Obf}_{\text{unknown}}$.

Table VIII shows the results. Each column contains the results when one specific obfuscator was excluded from the training and validation sets. The lower part with the results of evaluating training set 2 shows that classifying non-obfuscated scripts and scripts that were obfuscated with known obfuscators performs similar as in Table IV, which corresponds to the expected result. More interesting is the evaluation of training set 1 in the upper part of Table VIII, which shows the performance to detect scripts obfuscated with the excluded obfuscator. Just like in Table VII, false positives cannot occur, so the precision is always 100% and we use the recall value to assess the performance. The results vary greatly depending on the excluded obfuscator. Scripts obfuscated with closure or jfogs can hardly be detected (recall $<$1%) while those obfuscated with javascript-obfuscator and javascriptobfuscator.com advanced can be detected quite well (recall 76.93% and 99.80%). Scripts obfuscated with javascriptobfuscator.com standard and jsobfu are also hard to detect with recall values of 18.22% and 39.88%.

These results imply that some obfuscators are more similar than others. For example, scripts obfuscated with javascriptobfuscator.com advanced result in code that – with respect to the discriminatory features – is similar to the output of one or more of the other obfuscators. On the other hand, scripts obfuscated with jfogs must be very different from all other obfuscated scripts, as nearly none of them could be correctly classified as obfuscated. The results also imply that one should include many different obfuscators into the training and validation sets so the classifier can learn many different kinds of obfuscation techniques, which increases the probability that scripts obfuscated with unknown obfuscators can be detected.

In Table IX, the results of the same analysis while using only the 20 most descriptive features instead of all features is shown. Comparing the recall values of training set 1 in Tables VIII and IX, one can see that the performance is better when using only 20 features. While basically nothing changed for javascriptobfuscator.com advanced (it already had a very high recall value) and jfogs, the recall values for closure and javascriptobfuscator.com standard could be improved by about 2.5 %, for jsobfu by about 6% and for javascript-obfuscator by more than 16%.

Determining the exact reason of this increased performance requires more detailed analysis, but in general, using fewer features increases the fitting error of a trained classifier and at least with the obfuscators we used in the data set, this is beneficial for the recall value. Of course, this comes at a price: Reducing the number of features reduces the F1-scores when classifying scripts that are either non-obfuscated or obfuscated with a known obfuscator, as can be seen by comparing the evaluation results of training set 2 in Tables VIII and IX. This is not surprising and confirms what we already observed in Section IV-A.

To conclude this second series of experiments, we summarize the most relevant findings:

- It is possible to detect scripts obfuscated with an unknown obfuscator. Depending on the unknown obfuscator and the obfuscators in the training and validation sets, the recall value can range from close to 0% (closure and jfogs in our case) to more than 99% (javascriptobfuscator.com advanced in our case).
- One should include many different obfuscators into the training and validation sets so the classifier can learn many different kinds of obfuscation techniques, which increases the probability that scripts obfuscated with unknown obfuscators can be detected.
- Using only the 20 most descriptive instead of all features can increase the classification performance of scripts that are obfuscated with unknown obfuscators – at least with the obfuscators we used in our data set. On the downside, this has a slightly negative effect on classifying non-obfuscated scripts and scripts obfuscated with known obfuscators.

C. Detecting Malicious Scripts

With the results in Table VII, we demonstrated that correctly classifying malicious scripts as obfuscated if malicious scripts

TABLE VIII. PERFORMANCE OF THE BDT CLASSIFIER TO DETECT SCRIPTS OBFUSCATED WITH AN UNKNOWN OBFUSCATOR, USING ALL FEATURES.

		closure	javascript-obfuscator	javascript-obfuscator. com advanced	javascript-obfuscator. com standard	jfogs	jsobfu
Obfuscated (training set 1)	p	100.00%	100.00%	100.00%	100.00%	100.00	100.00
	r	0.05%	76.93%	99.80%	18.22%	0.24%	39.88%
	F1	0.09%	86.96%	99.90%	30.83%	0.48%	57.02%
	s	8829	12922	12919	12919	12922	12920
Non Obfuscated (training set 2)	p	99.53%	99.38%	99.34%	99.73%	99.55%	99.29%
	r	99.15%	99.03%	99.02%	99.45%	99.26%	98.99%
	F1	99.34%	99.21%	99.18%	99.59%	99.41%	99.14%
	s	7752	7752	7752	7752	7752	7752
Obfuscated (training set 2)	p	99.66%	99.59%	99.58%	99.76%	99.69%	99.57%
	r	99.81%	99.74%	99.72%	99.88%	99.81%	99.70%
	F1	99.74%	99.66%	99.65%	99.82%	99.75%	99.63%
	s	19382	18154	18155	18155	18154	18155

TABLE IX. PERFORMANCE OF THE BDT CLASSIFIER TO DETECT SCRIPTS OBFUSCATED WITH AN UNKNOWN OBFUSCATOR, USING THE 20 MOST DESCRIPTIVE FEATURES

		closure	javascript-obfuscator	javascript-obfuscator. com advanced	javascript-obfuscator. com standard	jfogs	jsobfu
Obfuscated (training set 1)	p	100.00%	100.00%	100.00%	100.00%	100.00	100.00
	r	2.59%	93.48%	99.65%	20.76%	0.22%	45.89%
	F1	5.06%	96.63%	99.83%	34.38%	0.43%	62.91%
	s	8829	12922	12919	12919	12922	12920
Non Obfuscated (training set 2)	p	98.92%	98.21%	98.11%	98.81%	99.31%	98.89%
	r	98.09%	96.72%	97.30%	97.47%	98.93%	97.90%
	F1	98.50%	97.46%	97.71%	98.14%	99.12%	98.39%
	s	7752	7752	7752	7752	7752	7752
Obfuscated (training set 2)	p	99.24%	98.61%	98.85%	98.93%	99.54%	99.11%
	r	99.57%	99.25%	99.20%	99.50%	99.71%	99.53%
	F1	99.41%	98.93%	99.03%	99.21%	99.63%	99.32%
	s	19382	18154	18155	18155	18154	18155

using the same obfuscators are also included in the training set works well. In the final experiments, we analyzed how well this works if no malicious scripts are used in the training set. Basically, these experiments are similar to the ones done in Section IV-B as the malicious scripts use different obfuscators than the ones we used to create our own obfuscated scripts in the data set.

The setting is similar to the previous experiments, but this time, testing set 1 contains all malicious (and therefore also obfuscated) scripts from the data set. Table X illustrates the results. Just like above, the evaluation of training set 2 shows that classifying non-obfuscated scripts and scripts that were obfuscated with known obfuscators performs well. With respect to classifying the malicious scripts as obfuscated, the recall value is low (16.52%) when all features are used (left column). This indicates that the obfuscation techniques used for the malicious samples are not represented well by the obfuscators in the training set. However, using only the 20 most descriptive features (right column) increases the performance substantially: The recall value raises by more than 31% to 47.71%. This confirms the finding of Section IV-B that reducing the number of features can increase the performance to detect scripts that are obfuscated with an unknown obfuscator.

To conclude this third and final series of experiments, we summarize the most relevant findings:

TABLE X. PERFORMANCE OF THE BDT CLASSIFIER TO DETECT MALICIOUS SCRIPTS, USING ALL FEATURES OR THE 20 MOST DESCRIPTIVE FEATURES.

		BDT (all features)	BDT (20 features)
Obfuscated (training set 1)	p	100.00%	100.00%
	r	16.52%	47.71%
	F1	28.35%	64.60%
	s	2706	2706
Non Obfuscated (training set 2)	p	99.38%	98.58%
	r	99.05%	97.67%
	F1	99.21%	98.12%
	s	7752	7752
Obfuscated (training set 2)	p	99.66%	99.18%
	r	98.78%	99.51%
	F1	99.72%	99.34%
	s	22031	22031

- It is possible to detect malicious obfuscated scripts that are obfuscated with an unknown obfuscator. Using all features and based on our data set, the recall value is low, though.
- Using only the 20 most descriptive substantially improves the recall value in our case. This confirms that reducing the number of features can increase the performance to detect scripts that are obfuscated with an unknown obfuscator.
- Ideally and for best possible recall value of malicious scripts, different malicious scripts should be included

into the training and validation sets as demonstrated in Table VII.

V. Related Work

In [23] Xu *et al.* study the effectivenes of traditional anti-virus/signature based approaches to detect malicious JavaScript code. They find that for their sample set, the average detection rate of 20 different anti-virus solutions is 86.4 percent. They also find that making use of additional data- and encoding-based obfuscation, the detection ratio can be lowered by around 40 and 100 percent respectively.

Likarish *et al.* [15] take an approach similar to ours. They 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 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-B) would impact on their results.

Wang *et al.* [24] 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. Some of them are similar to those of Likarish *et al.* [15]. Their results suggest a significant improvement over the work of Likarish *et al.*

Study from Kaplan *et al.* [4] 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. Similarly, *ZOZZLE*, a malicious JavaScript detection solution from Curtsinger *et al.* [25] also uses a Bayesian classifier with hierarchical features but, instead of just performing pure static detection, it has a run-time component to address JavaScript obfuscation. The component passes the unfolded JavaScript to the static classifier just before being executed.

Other solutions toward dynamic analysis, like Wepawet [26] (now discontinued), use JavaScript instrumentation to extract features and apply anomaly detection techniques on them. JSDetox [27] on the other side is a tool that uses both static and dynamic analysis capabilities in order to help analysts understand and characterize malicious JavaScript.

AdSafe [28] uses a completely different approach, it defines a simpler subset of JavaScript, which is powerful enough to perform valuable interactions, while at the same time preventing malicious or accident damage. This allows to put safe guest code (e.g., third party advertising or widgets) on a web-page defeating the common malvertising scenario.

VI. Discussion and Conclusions

In this paper, we analyzed how well a machine learning approach is suited to distinguish between obfuscated and non-obfuscated JavaScripts. To perform the analysis, we used a data set with more than 100000 scripts consisting of non-obfuscated (regular and minified) scripts, obfuscated scripts that are generated with several different obfuscators, and malicious scripts that are all obfuscated. This large data set and the broad spectrum of obfuscators strengthen the general validity of our results. To train and evaluate the different classifiers, we used 45 discriminatory features from the samples.

The results in Section IV-A show that if the training set contains a representative set of all samples (i.e., it contains obfuscated samples of all obfuscators), very good classification performance can be achieved. Of the nine classifiers we compared, the boosted decision tree classifier provided the best performance, with F1-scores above 99% when using all 45 features. When using only the 20 most descriptive features, classification performance gets lower, but it is still possible to achieve F1-scores close to and above 98%, while having the benefit of reduced time and memory requirements to train the classifier. As these 20 most descriptive features have only a small overlap with the features used in previous works but still provide nearly as good classification performance as with 45 features, it can be concluded that the newly added features provide a significant improvement to classify the scripts.

We also evaluated the performance to classify obfuscated scripts that were obfuscated using an unknown obfuscator, i.e., one that is not used by the samples in the training set. The results in Section IV-B demonstrate that it is possible to detect such scripts, but the classification performance heavily depends on both the unknown obfuscator and the obfuscators in the training set and the recall value ranges from close to 0% to more than 99%. We also observed that using only the 20 most descriptive instead of all features increases the classification performance of scripts that are obfuscated with unknown obfuscators. While determining the exact reason of this increased performance requires more detailed analysis, the most plausible reason is that using fewer features increases the fitting error of a trained classifier, which is beneficial for the recall value of samples obfuscated with an unknown obfuscator. However, for best performance, it is important to include many different obfuscators into the training set so the classifier can learn many different kinds of obfuscation techniques, which increases the probability that scripts obfuscated with unknown obfuscators can be correctly classified. The best performing classifier trained with the full data set and using all of the 45 features can be tested under the following URL: http://jsclassify.azurewebsites.net.

Finally, we analyzed the classification performance of malicious obfuscated scripts if no malicious scripts are used in the training set. The results in Section IV-C show that it was possible to correctly classify such scripts with recall values of about 16% when all features are used and 47% when

the 20 most descriptive features are used. This undermines two findings from above: Using fewer features increases the performance to detect scripts that use an unknown obfuscator and for best classification results, one should include many different malicious obfuscated scripts in the training set.

Besides showing that machine learning is a well-suited approach for classification of obfuscated and non-obfuscated JavaScripts, our work also created new questions that require more analysis. One of these questions is whether detection of JavaScripts that use unknown obfuscators can be improved by using additional or different features. This requires analyzing the obfuscated scripts that could only be classified poorly if the obfuscator was not used in the training set in more detail to understand their differences compared to scripts that are obfuscated with other obfuscators. In addition, while being able to distinguish between non-obfuscated and obfuscated scripts is already very helpful towards detecting malicious scripts because obfuscated scripts are likely candidates to be malicious, we envision to eventually being able to distinguish between malicious and benign scripts, independent of whether they are obfuscated or not. The main obstacle here is currently the data set: We have a good data set of non-obfuscated and obfuscated scripts, but the number of malicious samples is still relatively small. Getting more malicious samples is therefore the key to start a more detailed analysis about classifying malicious and benign scripts.

REFERENCES

[1] S. Aebersold, K. Kryszczuk, S. Paganoni, B. Tellenbach, and T. Trowbridge, "Detecting obfuscated javascripts using machine learning," in The 11th International Conference on Internet Monitoring and Protection (ICIMP). IARIA, May 2016.

[2] Microsoft, "Microsoft Azure Machine Learning Studio," last accessed on 2016-09-16. [Online]. Available: https://studio.azureml.net/

[3] 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/

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

[5] 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.

[6] "Alexa Top One Million Global Sites," last accessed on 2016-01-30. [Online]. Available: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

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

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

[9] CuteSoft, "JavaScriptObfuscator JavaScript obfuscator," last accessed on 2016-09-16. [Online]. Available: http://javascriptobfuscator.com/

[10] zswang (Wang Hu), "jfogs JavaScript obfuscator," last accessed on 2016-09-16. [Online]. Available: https://www.npmjs.com/package/jfogs

[11] rapid7, "jsobfu JavaScript obfuscator," last accessed on 2016-09-16. [Online]. Available: https://github.com/rapid7/jsobfu/

[12] Google, "Closure Compiler," last accessed on 2016-09-16. [Online]. Available: https://developers.google.com/closure/compiler/

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

[14] J. Kornblum, "ssdeep," last accessed on 2016-09-16. [Online]. Available: http://ssdeep.sourceforge.net/

[15] 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.

[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] W. M. Wu and Z. Y. Wang, "Technique of javascript code obfuscation based on control flow tansformations," in Computer and Information Technology, ser. Applied Mechanics and Materials, vol. 519. Trans Tech Publications, 5 2014, pp. 391–394.

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

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

[20] Microsoft, "Machine Learning / Initialize Model / Classification," last accessed on 2016-09-16. [Online]. Available: https://msdn.microsoft.com/en-us/library/azure/dn905808.aspx

[21] B. Rohrer, "How to choose algorithms for Microsoft Azure Machine Learning," last accessed on 2016-09-16. [Online]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-algorithm-choice

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

[23] 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.

[24] 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.

[25] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser javascript malware detection," in Proceedings of the 20th USENIX Conference on Security, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028070

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

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

[28] 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