JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript

Aurore Fass[1](✉), Robert P. Krawczyk[2], Michael Backes[3], and Ben Stock[3]

[1] CISPA, Saarland University, Saarbrücken, Germany,
Saarland Informatics Campus,
`aurore.fass@cispa.saarland`,
[2] German Federal Office for Information Security (BSI), Bonn, Germany,
`robert.krawczyk@bsi.bund.de`,
[3] CISPA Helmholtz Center i.G., Saarbrücken, Germany,
Saarland Informatics Campus,
`{backes, stock}@cispa.saarland`

Abstract. JavaScript is a browser scripting language initially created to enhance the interactivity of web sites and to improve their user-friendliness. However, as it offloads the work to the user's browser, it can be used to engage in malicious activities such as Crypto Mining, Drive-by Download attacks, or redirections to web sites hosting malicious software. Given the prevalence of such nefarious scripts, the anti-virus industry has increased the focus on their detection. The attackers, in turn, make increasing use of obfuscation techniques, so as to hinder analysis and the creation of corresponding signatures. Yet these malicious samples share syntactic similarities at an abstract level, which enables to bypass obfuscation and detect even unknown malware variants.
In this paper, we present JaSt, a low-overhead solution that combines the extraction of features from the abstract syntax tree with a random forest classifier to detect malicious JavaScript instances. It is based on a frequency analysis of specific patterns, which are either predictive of benign or of malicious samples. Even though the analysis is entirely static, it yields a high detection accuracy of almost 99.5% and has a low false-negative rate of 0.54%.

1 Introduction

Information Technology is constantly under threat with the amount of newly found malware increasing permanently: over 250,000 new malicious programs are registered every day [11]. Moreover, our Internet-driven world enables malware to rapidly infect victims everywhere, anytime (e.g., Mirai [26], NotPetya [27]). Currently, the most vicious attacks are the so-called crypto trojans (e.g., WannaCry [28]), which often use JavaScript as a payload in the first stage of the infection of the victim's computer. This plethora of new attacks renders manual analysis impractical: defenders remedy this situation by automating the analysis of potentially malicious code. As a consequence, new alternatives based on machine learning algorithms are being explored to obtain a better understanding of complex data collected from various systems, thereby automatically detecting and analyzing new malicious variants [16, 19, 30, 31].

Many malware families use methods of script obfuscation to evade detection by classical anti-virus signatures and to impose additional hurdles to manual analysis. As a result, analysis tools and techniques constantly need improvement to be able to recognize these obfuscated patterns and to mitigate the threats. A possible approach to detect malicious obfuscated JavaScript relies on lexical or syntactic analyses, which enable an elimination of the artificial noise, e.g., introduced by identifier renaming, created by the attacker while using these evasion methods. While at a textual level, an accurate detection of malicious documents can be foiled by the use of obfuscation, programmatic and structural constructs can still be identified. Therefore, using the way in which lexical (e.g., keywords, identifiers, operators) or syntactic (e.g., statements, expressions, declarations) units are arranged in a given JavaScript file provides valuable insight to capture the salient properties of the code and hence to identify specific and recurrent malicious patterns. Approaches that use lexical [18, 21, 29] or syntactic units derived from the Abstract Syntax Tree (AST) [7, 9, 17] to analyze new variants of malicious code have already been proposed. We choose a syntactic approach over the lexical one, as the AST contains more information than the lexical units, allowing us to leverage grammar information for an improved analysis.

In this paper, we present an advanced method based on an AST-level analysis to automatically classify JavaScript samples containing obfuscated code. This implementation responds to the following challenges: resilience to common obfuscation transformations, practical applicability, and robustness against previously presented pollution attacks. We address these challenges by proposing a methodology to learn and recognize specific patterns either typical of benign or of malicious JavaScript documents. The key elements of JaSt are the following:

– *Fully Static AST-Based Analysis:* Our system benefits from the AST to extract syntactic features from JavaScript files. Being entirely static, it is also able to analyze samples whose behavior is time- or environment-dependent [25].
– *Extraction of N-Grams Feature:* Using the syntactic features, patterns of length *n*, namely n-grams, are built and their frequency is analyzed. We find that these differ significantly between benign and malicious samples, allowing us to distinguish them. This approach is resistant to common obfuscation transformations since the intermediate representation used is close to the semantics of the code.
– *Accurate Detection of Malicious JavaScript:* Based on our n-gram approach, applying off-the-shelf supervised machine learning tools can be used to reliably differentiate benign from malicious JavaScript files.
– *Comprehensive Evaluation:* We evaluated our system in terms of detection accuracy, false-positive and false-negative rates, temporal stability, and performance on an extensive dataset composed of 105,305 current and unique JavaScript samples found in the wild: 85,059 malicious and 20,246 benign. It makes accurate predictions (with a detection accuracy of almost 99.50% on our sample set) and has a low false-negative rate of 0.54% which is more than ten times less than other state-of-the-art systems.

The remaining paper is organized as follows. The implementation of JaSt is described and justified in Section 2. The detection results as well as the throughput are analyzed in Section 3 and further discussed in Section 4. Finally, Section 5 presents some related work while Section 6 concludes the paper.

2 Methodology

The architecture of our JavaScript detection system consists of a feature-extraction part and learning components, as shown in Figure 1. First, a static analysis of JavaScript documents is performed, extracting in particular syntactic units. Then, substrings of length *n*, namely n-grams, are produced and their frequency is used as input to the learning components. These components are used to train a classifier or update an existing model with the aim of distinguishing benign from malicious JavaScript samples. In the following section, we discuss the details of each stage in turn.

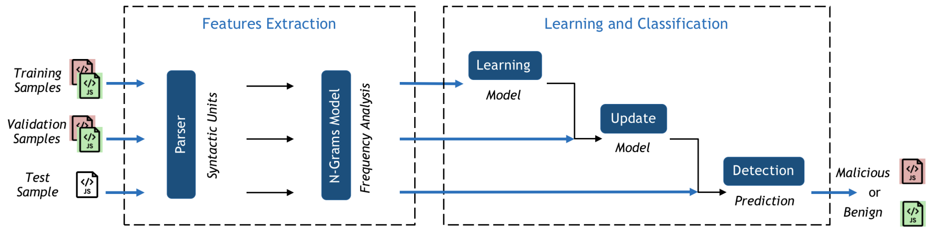


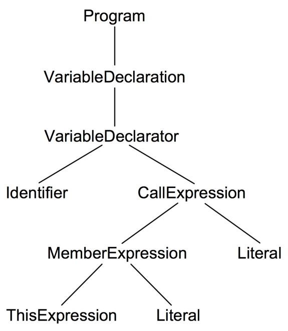
Fig. 1: Schematic depiction of JaSt

2.1 Syntactic Analysis of JavaScript Documents

The choice of a syntactic analysis to detect malicious JavaScript instances is motivated by its resilience to common obfuscation transformations. At a textual level, an accurate discrimination between malicious and benign documents can be foiled by this evasion method, but programmatic and structural constructs are still identifiable. Moreover, this parsing process provides a certain level of code abstraction, ignoring, for example, the variable names to consider them as *Identifier*, skipping blank spaces or comments. Therefore, using the way syntactic units (e.g., statements, expressions, declarations) are arranged in a given JavaScript file provides valuable insight to capture the salient properties of the code and hence to identify specific and recurrent malicious (or benign) patterns. The syntactic analysis is performed by the state-of-the-art open source JavaScript parser Esprima [10], which takes a valid JavaScript sample as input, produces an ordered tree describing the syntactic structure of the program (also known as Abstract Syntax Tree (AST)) and traverses it depth-first post-order, before extracting the corresponding syntactic units. Figure 2 illustrates the parsing process, where the malicious entity from Figure 2a is transformed into an AST (Figure 2b), whose traversal gives a sequence of syntactic units (Figure 2c).

```
var Euur1V = this [ "l9D" ] ("ev#333399al")
```

(a) Malicious JavaScript example from [30]



(b) AST produced from (a)

Description	Symbolic name	Value
Identifier	Identifier	15
ThisExpression	Expression	0
Literal	Literal	3
MemberExpression	Expression	0
Literal	Literal	3
CallExpression	Expression	0
VariableDeclarator	Declarator	12
VariableDeclaration	Declaration	4
Program	Program	8

(c) Syntactic units extracted from (b)

Fig. 2: Extraction of AST-based units from a JavaScript sample

Overall the Esprima parser can produce 69 different syntactic entities ranging from *FunctionDeclaration* to *ImportDefaultSpecifier*. For performance reasons, a simplification of the list of syntactic units returned by the parser is performed. It consists in grouping together elements with the same abstract syntactic meaning (e.g., *FunctionDeclaration* and *VariableDeclaration* are both referred to as *Declaration*, while *ForStatement* and *WhileStatement* are both referred to as *Statement*), also considering one-element families if they could not be grouped with other entities (e.g., *Identifier*, *Program*). It enables a reduction of the number of different units from 69 to 19, while still preserving their syntactic meaning, as each element is analyzed within its context, using n-grams feature.

2.2 N-Grams Model

To identify specific patterns in JavaScript documents, a fixed-length window of *n* symbols is moved over each syntactic unit previously extracted, so as to get every subsequence of length *n*, namely n-grams, at each position. As shown in the literature, this is a generic and effective means for modeling reports [20–22, 29, 32–34].

The use of n-grams feature enables a representation of how these syntactic units were originally arranged in the analyzed JavaScript file. Therefore, reports sharing several n-grams with the same frequency present similarities with one another, while reports with different n-grams have a more dissimilar content. As a consequence, analyzing the frequency of these short patterns provides valuable insights to determine if the sample is either benign or malicious. To be able to compare the frequency of all n-grams appearing in several JavaScript inputs, a vector space is constructed such that each n-gram is associated with one dimension, while its corresponding frequency is stored at this position in the vector *R*. For this mapping process, not all possible n-grams are considered, so as to limit the size of the vector space (initially of 19^n

Table 1: Comparison between the number of all possible n-grams versus the number of n-grams extracted

N-grams	#All possible n-grams	#N-grams considered
N = 1	19	17
N = 2	361	114
N = 3	6,859	570
N = 4	130,321	2,457
N = 5	2,476,099	8,025

according to the length n of n-grams chosen), which has a direct impact on the performance. Besides, not all n-gram combinations make sense, e.g., as the root of the AST, the *Program* unit can only be present once. Therefore, a set S containing n-grams preselected based on their number of occurrences in our dataset is created. For this selection, the suitability criteria defined by Wressnegger et al. [34] were considered: the *perturbation*, as the expected ratio of n-grams in a benign sample that are not part of the training data, and the *density*, as the ratio of the number of unique n-grams in the dataset to the number of all possible n-grams. In particular, we aimed at significantly reducing the *density* of our dataset, while keeping an extremely low *perturbation* so as to limit the number of false-positives induced by unknown n-grams in benign data. This was achieved by only considering the n-grams appearing in our dataset (knowing that currently unknown n-grams can extend the list of n-grams currently considered), which enables a feature reduction of more than 90% for 3-, 4-, and 5-grams, as shown in Table 1.

Formally, the previous vector R is defined using the set S of n-grams considered and the set $S' = (x_i)_{i \in [\![1, |S'|]\!]}$ of JavaScript samples to be analyzed such as:

$$R^{\mathrm{T}} = \{{r_1}^{\mathrm{T}}, \ldots, {r_{|S'|}}^{\mathrm{T}}\}$$
$$\text{knowing that } \forall i \in [\![1, |S'|]\!], r_i = \phi(x_i)$$
$$\text{and } \phi : x_i \longrightarrow (\phi_n(x_i))_{n \in S}$$
$$\text{with } \phi_n(x_i) \text{ the frequency of the n-gram } n \text{ in the report } x_i \quad .$$

As a consequence, the ϕ function maps a JavaScript file x_i to the vector space $\mathbb{R}^{|S|}$ such that all dimensions associated with the n-grams contained in the set S are set to their frequency. To avoid an implicit bias on the length of the reports, the frequencies are normalized, such that: $\forall i \in [\![1, |S'|]\!], ||r_i|| = ||\phi(x_i)|| = 1$. The frequency vector R (defined as $R^{\mathrm{T}} = (r_i)_{i \in [\![1, |S'|]\!]}{}^{\mathrm{T}}$) is then used as input to the learning components.

2.3 Learning and Classification

The learning-based detection completes the design of our system. Before predicting if a given JavaScript sample is either benign or malicious, the classifier has to be trained on a representative, up-to-date, and balanced set of both benign and malicious JavaScript files. Therefore, a model is initially built using the vectorial representation $R^{\mathrm{T}} = (r_i)_{i \in [\![1, |S'|]\!]}{}^{\mathrm{T}}$, presented in Section 2.2, of the files S' to be classified. This vector is furthermore used to update an old model with newer JavaScript samples, without having to train the classifier from scratch again.

We empirically evaluated different off-the-shelf classifiers (Bernoulli naive Bayes, multinomial naive Bayes, support vector machine (SVM), and random forest) and determined that random forest yielded the best results. Contrary to the two naive Bayes algorithms, random forest does assume independence between the attributes, leading to a higher detection accuracy. It is a meta estimator which combines the predictions of several decision trees on various sub-samples of the dataset: for each tree predictor, an input is entered at the top and as it traverses down the tree, the data is bucketed into smaller and smaller sets. Therefore the whole forest provides predictions more accurate than those of a single tree [31] and controls overfitting, as it classifies an unknown instance according to the decision of the majority of the tree predictors [4].

JaSt is implemented in Python and its Scikit-learn implementation of random forest is used to classify unknown data [23]. This Python module integrates a collection of state-of-the-art tools and machine learning algorithms for data mining and data analysis, and provides highly extensible implementations controlled by several parameters (e.g., number of trees in a forest, number of features considered). To optimize the predictions of our learning-based detection, the tuple of hyperparameters yielding an optimal model (that minimizes a predefined loss function on an independent dataset) has been determined. Our independent dataset, which was provided by the German Federal Office for Information Security (BSI) and labeled according to the protocol described in Section 3.1, contains 17,500 unique benign samples, and as many malicious ones. This way, it has a balanced distribution and avoids any overfitting. First, random search has been performed with 5-fold cross-validation on this dataset, sampling a fixed number of parameter settings from the specified distributions [3]. This method enabled us to narrow down the range of possibilities for each hyperparameter and in a second step, to concentrate the search on a lower number of tuples. Indeed, grid search has been used on the previous results, exhaustively testing all the resulting combinations with cross-validation, to tune the optimal set of hyperparameters.

The tuple of hyperparameters yielding an optimal model on our independent dataset is presented hereafter. As far as features are concerned, 4-grams have been selected because the length four provided the best trade-off between false-positives and false-negatives. As for the forest, it contains 500 trees having each time a maximum depth of 50 nodes. When looking for the best node's split, $\lceil \sqrt{2,457} \rceil = 50$ features are considered and the Gini criterion is used to measure the quality of a split, based on the Gini impurity [8]. These hyperparameters have been selected because their combination leads to the best trade-off between performance and accuracy.

3 Comprehensive Evaluation

In this section, we outline the results of our extensive evaluation. The success of our learning-based approach comes from the previous well-considered tuple of hyperparameters, and a high-quality dataset. This was confirmed by the high detection accuracy obtained on several unknown datasets. Based on the accuracy of our system's predictions, a study of the temporal evolution of JavaScript files over one year was performed. JaSt was also compared to state-of-the-art approaches where its extremely low false-negative rate is without precedent. We also evaluated JaSt in terms of runtime performance.

3.1 Experimental Datasets

The experimental evaluation of our approach rests on an extensive dataset mainly provided by the German Federal Office for Information Security (BSI). This dataset, which comprises 105,305 unique (based on their SHA1 hash) JavaScript samples, is in particular composed of 20,246 benign and 85,059 malicious JavaScript files (Table 2) between 100 bytes and 1 megabyte, with a total size of more than 3.6 gigabytes. Our malicious samples mainly correspond to JavaScript extracted from emails, one of the most common and effective way to spread JScript-Loader, knowing that a double-click on the attachment is by default sufficient to execute it on Windows hosts, leading to e.g., drive-by download or ransomware attacks. JavaScript as infection vector is particularly relevant and powerful here since it is especially prone to obfuscation and therefore enables the attackers to build a unique copy of the malicious attachment for each recipient, foiling classical anti-virus signatures. These samples have been labeled as malicious based on a score obtained after having been tested by twenty different anti-virus systems, the malware scanner of the BSI, and a runtime-based analysis. As for the benign files, they were extracted among others from Microsoft products (e.g., Microsoft Exchange 2016 and Microsoft Team Foundation Server 2017), the majority of which are obfuscated, which enabled us to ensure that JaSt does not confound obfuscation and maliciousness, but leverages grammar information for an accurate distinction between benign and malicious inputs. For our dataset to be more up-to-date and representative of the JavaScript distribution found in the wild, we also included some open source games written in JavaScript, web frameworks and the source code of Atom [1], tested either using the previous protocol or directly downloaded from the developers' web page. These extra samples extend our dataset with some new, sometimes unusual or specific (e.g., games) coding styles, which shows again that our system does not confound unseen nor unusual syntactic structures with maliciousness. Reasons for not including any web JavaScript extracted from HTML documents in this dataset are discussed in Section 4.2.

3.2 Detection Performance

In our first experiment, we studied the detection performance of JaSt in terms of true-positive and true-negative rates (correct classification of the samples, either as benign or as malicious), false-positive and false-negative rates (misclassification of the samples, malicious instead of benign, or the opposite), and overall detection accuracy. The experimental protocol is the following: 3,500 unique JavaScript files were each time randomly extracted from the email dataset (malicious) and Microsoft dataset (benign), and were used to build a balanced model. The remaining samples were considered

Table 2: JavaScript dataset description

JS type	Creation	#JS	Label	Obfuscated
Emails	2017-2018	85,059	Malicious	y
Microsoft	2015-2018	17,668	Benign	y
Games	N/A	2,007	Benign	n
Web frameworks	N/A	434	Benign	N/A
Atom	2011-2018	137	Benign	n

Table 3: Detection accuracy of JaSt

JS type	#Misclassified	#Correctly classified	Detection accuracy
Emails	443	81,116	99.46%
Microsoft	71	14,097	99.50%
Games	10	1,997	99.52%
Web frameworks	4	430	99.03%
Atom	1	136	98.98%
Average benign	86	16,660	99.48%

unknown and were used to measure the detection performance. We repeated this procedure five times and the averaged results are shown in Table 3. JaSt was able to correctly classify 99.48% of our benign dataset, while still detecting 99.46% of the malicious email samples. As both these benign and malicious files were, for the most part, obfuscated, this demonstrates the resilience of our system to this specific form of evasion. More importantly, it shows that JaSt does not confound obfuscation with maliciousness, and plain text with benign inputs, but could use differences between benign and malicious obfuscation at a syntactic level to distinguish benign obfuscated from malicious obfuscated files. Indeed, while the former is used to protect code privacy and intellectual property, the latter aims at hiding its malicious purpose without specific regard to the performance. Furthermore, our system offers a very high true-negative rate for the web frameworks, the source code of JavaScript games and of Atom, even though these sample families were not present in the training set. The possible transfer of an email-based model to web samples is further discussed in Section 4.2.

Both the false-positive (0.52%) and the false-negative (0.54%) rates are very low for JaSt indicating that, based on a frequency analysis of their 4-grams, our classifier is able to make an accurate distinction between benign and malicious samples almost 99.5% of the time. We achieved this optimal trade-off between the false-positives and the false-negatives by using Youden's J statistic, where J is defined as [24, 38]:

$$J = \text{sensitivity} + \text{specificity} - 1 = TPR - FPR$$

This index corresponds to the area beneath the curve subtended by a single operating point. Its maximum value is used as a criterion for selecting the optimal cut-off point between false-positives and false-negatives. In our case, Youden's index was determined with 5-fold cross validation on the independent dataset presented in Section 2.3. The value we got is 0.29, which means that a sample will be considered malicious if the probability of it being malicious is above 0.29, according to our random forest classifier. Figure 3 presents the evolution of the detection performance when the value of Youden's index varies between 0.19 and 0.95. The best trade-off between false-positive and false-negative rates is obtained for 0.29, while the maximum detection accuracy is obtained for a threshold of 0.25. The value 0.23 is equally interesting, as it represents a reduction of the sharp false-positive rate decline while retaining a low false-negative rate and an extremely high detection accuracy of 99.49%. As for trading an extremely low false-positive rate for a higher false-negative rate, it downgrades the overall detection accuracy extremely rapidly. For the rest of the paper, if no other indication is given, we consider an index of 0.25.

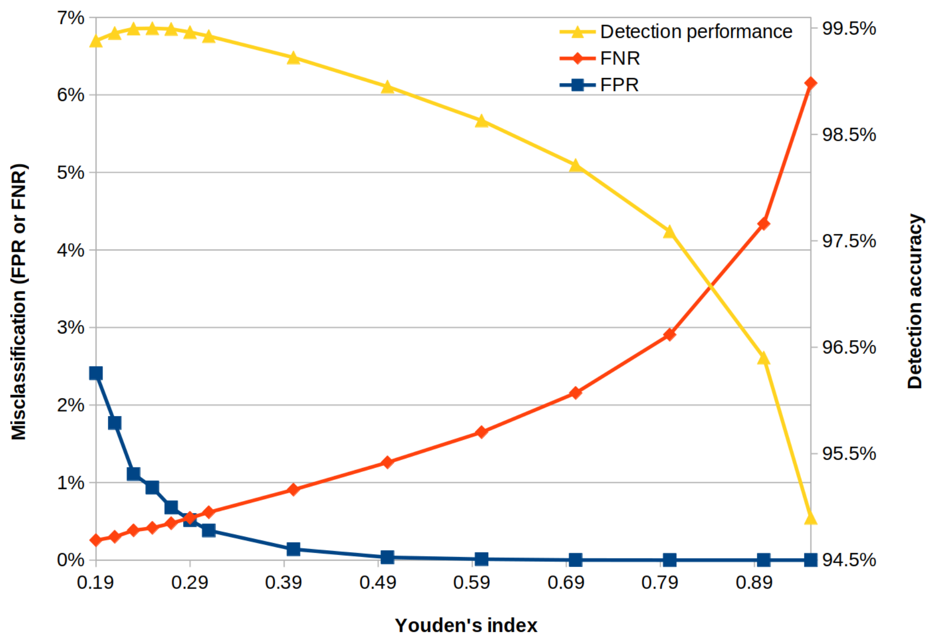


Fig. 3: Detection performance, depending on Youden's index

3.3 JavaScript Temporal Evolution

In our second experiment, we focused on the temporal evolution of malicious email-JavaScript received from January 2017 to January 2018. For each month, two steps were performed:

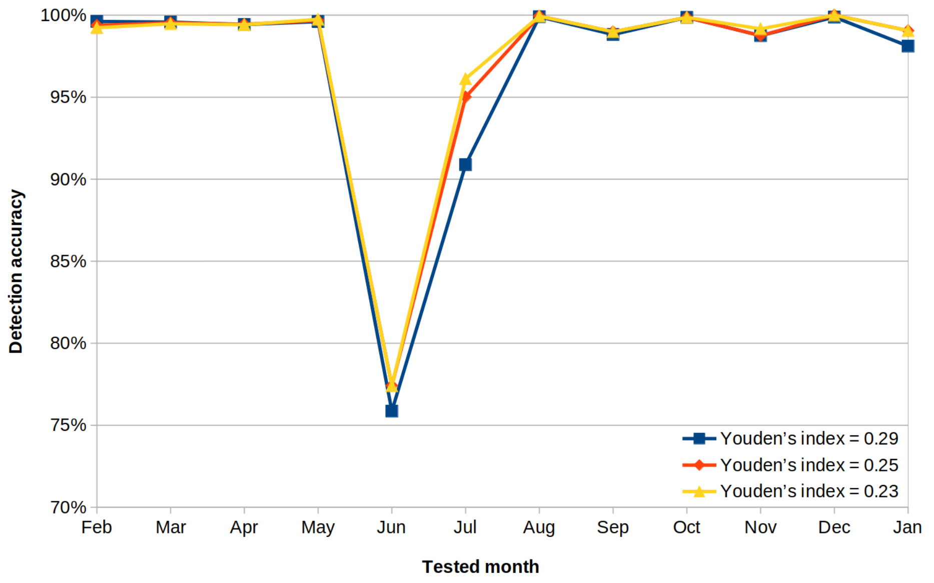


Fig. 4: Temporal evolution of the detection accuracy, depending on Youden's index

– The labels of the JavaScript samples collected on the current month were considered unknown. The model built in the previous months (the first model being created in January 2017) was used to classify these JavaScript instances;
– The true labels (obtained from other sources such as AV) of the JavaScript samples collected on the current month were used to build a new model, including all the samples of the previous months.

As a consequence, the samples from January 2018 were classified with a model initially built in January 2017 and extended each month, until December 2017 inclusive, with new and up-to-date malicious as well as benign JavaScript instances.

Figure 4 shows the performance of the random forest classifier in terms of detection accuracy –defined by the proportion of samples correctly classified, either as benign or as malicious– for three different thresholds. The prediction decline in June and to a lesser extent in July only depends on malicious JavaScript misclassifications, the mean false-positive rate over the whole period being 0.21%. The decrease gets more important when the threshold increases, which makes sense as it represents the probability cut-off to consider that a sample is malicious. As a comparison, we replaced the complete relearning of a model each month, by an update function adding 100 new trees to the forest built in the previous months. As both experiments presented the same decline in June and July, we concentrated our analysis on these two months to understand where the major changes at a sample level originated from. Indeed, a manual inspection of the samples from June and July, combined with the use of JSInspect [14] –a project built on the AST to detect structurally similar code– confirmed that the misclassifications came from several big JavaScript waves, each wave (also referred to as family) containing samples with the exact same AST-based struc-

Table 4: Insights into the malicious samples collected between 2017 and 2018

Months	#Malicious	#Malicious big waves[4]	Part of a big wave	Part of a FN big wave
Feb	4,894	8	66.92%	0%
Mar	4,838	4	28%	0%
Apr	4,883	4	30.04%	0%
May	4,922	4	44.64%	0%
Jun	4,987	6	73.73%	39.74%
Jul	4,831	6	53.88%	7.32%
Aug	6,536	6	64.35%	0%
Sep	592	0	0%	0%
Oct	3,610	1	8.8%	0%
Nov	120	0	0%	0%
Dec	419	0	0%	0%
Jan	53	0	0%	0%

ture. As a matter of fact, the attackers abused obfuscation to send a unique copy of the malicious JavaScript email attachment to each recipient. In this specific case, they only randomized the function and variable names for each JavaScript file they produced: since their SHA1 hash is different, we did not consider them as duplicate, but they are identical at the AST level (variable/function names are represented by an *Identifier* node). In June, we can notice in particular the appearance of four such misclassified waves with respectively 213, 355, 578, and 1,049 files in them. If one sample of one of the previous families is misclassified, so is the entire wave, which yielded in our case a high number of false-negatives. A similar phenomenon was observed in July, where two waves respectively containing 107 and 354 samples were received and misclassified. These six specific waves were admittedly composed of malicious samples only, but the classifier labeled each one of them as benign with a probability over 78%. We could have observed the inverse phenomenon, where one sample of a malicious wave would have been recognized as malicious, and therefore the entire wave too (as a wave only contains samples with the exact same AST-based structure), which would have yielded a high number of true-positives (depending on the wave's size). Table 4 indicates that it is globally not the case since the biggest malicious families were rather found in June and July. Besides, we only had big wave (over 300 samples) misclassifications in June and July, the other waves were being correctly flagged as malicious with a probability over 50% (in general even over 75%).

As a second experiment, Figure 5 presents the evolution of the detection accuracy when the month used to build the initial model varies, the rest of the experiment staying the same. As previously, and for the reasons mentioned before, a decline was observed in June and July. In particular, the differences in terms of detection accuracy in June between a model first built in April and the other ones, or between the different

[4] We define a *big wave* as a wave containing more than 300 syntactically similar samples

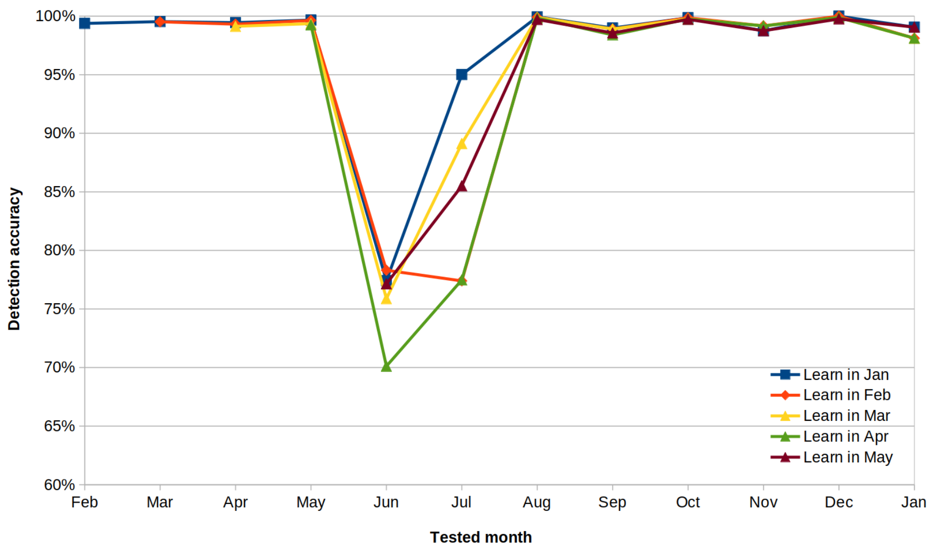


Fig. 5: Temporal evolution of the detection accuracy, depending on the training month

models in July, highlight the presence of these big JavaScript waves (through correct detections or on the contrary misclassifications).

This way, June and July both misclassified several big waves of syntactically similar JavaScript samples, with in particular a wave containing more than 1,000 samples, therefore accounting for more than a fifth of the samples collected in June. Even though it yielded a high number of false-negatives, this is avoidable with a model more representative of the distribution found in the wild. As a matter of fact, another sampling of the *exact same* JavaScript files gave in Section 3.2 a false-negative rate of 0.54% (threshold: 0.29, with an equally low false-positive rate of 0.52%). It was not obtained by chance, especially since the training set was significantly smaller, but through models containing more entropy in their randomly-selected samples, in comparison to a few syntactically similar JavaScript waves.

3.4 Comparison with Other Approaches

Table 5 presents a quantitative comparison with closely related work. Cujo and PJScan both used lexical units combined with an n-gram analysis to detect malicious JavaScript respectively embedded in web pages and in PDF documents. As for Zozzle, it used features extracted from the AST to identify malicious JavaScript samples. We discuss their approach further in Section 5. Table 5 shows that JaSt is heavily optimized to detect malicious JavaScript instances with its low false-negative rate of 0.54% (threshold 0.29), which is between 10 and 28 times lower than the other tools proposed thus far. Compared to Cujo and Zozzle, who also used the results of a dynamic analysis to detect malicious JavaScript samples more accurately, our approach outperforms these concepts. Like the majority of anti-virus systems, they rather traded a low false-

Table 5: Accuracy comparison with closely related work

Project	FP rate	FN rate	Static	Dynamic
JaSt	5.2E-3	5.4E-3	✓	-
Cujo [29]	2.0E-5	5.6E-2	✓	✓
PJScan [21]	1.6E-1	1.5E-1	✓	-
Zozzle [7]	3.1E-6	9.2E-2	✓	✓

positive rate for a higher false-negative rate. Indeed, as indicated by Curtsinger et al., given the number of URLs on the web, a false-positive rate of 5% is considered acceptable for static analysis tools, but rates even 100 times lower are not acceptable for in-browser detection. Besides, the low false-positive rate of Zozzle has to be taken with a grain of salt, since this tool rather aims at detecting benign samples. For this reason, it was tested on more benign than malicious files (1.2 million versus a few thousand) which lowered the false-positive rate at the expense of the false-negatives. JaSt on the other hand is designed to detect *malicious* JavaScript files, while still retaining a low false-positive rate of 0.52%.

As mentioned in Section 3.2, Youden's index can furthermore be used to shift false-positive and false-negative rates, according to the system's use case and dataset. Therefore, to perform further comparisons with Cujo and Zozzle, we increased the value of Youden's index, so as to lower our false-positive rate. With a threshold of 0.7, our system already had a lower false-positive rate than Cujo's, while retaining a lower false-negative rate (Table 6a). To ensure that these results were not coming from a lack of benign JavaScript samples, we extracted 119,233 unique benign JavaScript files from Alexa top 10k web sites and classified them as previously (Table 6b). Our model did not have such a low false-positive rate on these samples as before since we were using an email-based model to classify web-JavaScript (this concept is further discussed in Section 4.2). Nevertheless, with a Youden's index of 0.8, the false-positive rate of JaSt on the Alexa dataset was lower than Cujo's, while still retaining a lower false-negative rate. As for Zozzle, a threshold of 0.8 on our dataset provided both a better false-positive and a better false-negative rate. As previously, we also performed this comparison on the samples extracted from Alexa top 10k. With a threshold of 0.9, we had a false-positive rate of 6.71E-4% (standing for 0.8 false-positive, averaged over 5 runs) –admittedly a little superior to Zozzle's– but still a lower false-negative rate (Table 6).

Several parameters are responsible for the higher detection accuracy of JaSt compared to Cujo and Zozzle. First, we did not trade a very low false-positive rate for a higher false-negative rate, which enables our system to accurately detect benign samples with an accuracy of 99.48% and 99.46% for malicious ones. As indicated in Figure 3, an extremely low false-positive rate significantly degrades the classifier's accuracy. Besides, maximizing the detection accuracy also corresponds to the better trade-off between false-positives and false-negatives. Furthermore, the choice of our random

Table 6: Accuracy comparison according to Youden's index

Threshold	FP rate	FN rate
0.7	1.19E-5	2.16E-2
0.8	0	2.91E-2
0.9	0	4.34E-2

(a) With our dataset

Threshold	FP rate	FN rate
0.7	1.26E-4	-
0.8	1.68E-5	-
0.9	6.71E-6	-

(b) With samples from Alexa top 10k

forest classifier has an impact on the detection performance, since it performed better than Bernoulli naive Bayes and SVM –respectively chosen for Zozzle and for Cujo– on our dataset. Last but not least, our syntactic analysis also has an impact on the detection accuracy e.g., Cujo is based on a lexical analysis, which does not perform as well as an AST-based one, because lexical units lack context information.

3.5 Run-Time Performance

The run-time performance of our system was tested on a commodity PC with a quad-core Intel(R) Core(TM) i3-2120 CPU at 3.30GHz and 8GB of RAM. The experiments have each time been performed 5 times, on 5 different sets randomly selected. The processing time for all stages of our method on 500 unique JavaScript samples (half of which are benign, the other half being malicious), representing 14.9 megabytes, is shown in Table 7. The most time-consuming operation corresponds to the parsing of the JavaScript files with the open source tool Esprima, written in JavaScript, which accounts for more than 85% of the overall detection time. In comparison, the production of all 4-grams and the creation of a vector of 2,457 dimensions, containing the frequency of each of the previous 4-grams, is quite fast (12.72% of the time). As for the performance of the random forest classifier, it mainly depends on the throughput of Python Scikit-learn algorithms and represents in average less than 1% of the processing time. It includes building a forest of 500 trees (*c.f.* Section 2.3) using the previous samples, updating the previous model by adding 100 new trees to the forest –which is more than 4 times faster than creating the original model– and testing the model on unknown samples, the most important part of the detection system and also clearly the fastest. In total, JaSt extracted the syntactic units of 500 JavaScript samples, constructed 2,457 different 4-grams and computed each frequency for all input files (representing 1,228,500 frequencies), built a model based on the previous frequency vector and updated it, before using it to classify 500 unknown JavaScript documents in less than 2 minutes. Compared to PJScan –implemented in C with its own C library to classify JavaScript entities–, which can analyze a PDF document in 0.0032 seconds, our approach is slower. In compensation, the accuracy of our predictions is significantly better (Section 3.4), which primes as the throughput could always be improved by parallelization for a deployment in the wild.

To have a closer look into the performance of our system, we used the models built in Section 3.2 to classify our different datasets. Table 8 presents the throughput of JaSt

Table 7: Processing time for 500 JavaScript samples for different stages of our system

	Parser	N-grams analysis	Learner	Updater	Classifier	Total
Total	96.55s	14.43s	1.79s	0.41s	0.26s	113.43s
Percentage	85.12%	12.72%	1.57%	0.36%	0.23%	100%

Table 8: Throughput characteristics of JaSt

	#Samples	Time	Size	Files per second	Throughput
Emails	85,059	20,247.08 s	2.5 GB	4.20	0.12 MB/s
Microsoft	17,668	6,049.21 s	1.1 GB	2.33	0.18 MB/s
Games	2,007	370.52 s	50.9 MB	5.45	0.14 MB/s
Web frameworks	434	94.39 s	13.8 MB	4.65	0.15 MB/s
Atom	137	30.70 s	5.4 MB	5.70	0.18 MB/s
All files	105,305	26,791.90s	3.67 GB	3.93	0.14 MB/s

depending on the type of JavaScript file to be detected. The run-time performance is closely related to the number of samples to be analyzed: bigger datasets will have a lower per-file-throughput as smaller ones, since the 4-grams frequencies of all files have to be kept in the buffer before being used for classification purpose. On average, JaSt analyzes 0.14 MB/s, which is comparable to the 0.2 MB/s of Zozzle for the same amount of features as ours, while still retaining a higher detection accuracy (99.46% compared to 99.20% for Zozzle).

4 Discussion

In this section, we first examine the limitations of our learning-based approach, focusing on evasion techniques that might be used by an attacker. We then discuss to what extent an email-based model is able to detect web samples, such as exploit kits provided by Kafeine DNC[5] or JavaScript extracted from Alexa[6].

4.1 Limitations and Further Evasion Techniques

All learning-based malware detection tools will fail to detect some attacks, such as malicious instances not containing any of the features present in the training set. As a matter of fact, machine learning does not always take into account the concept of uncertainty involved in the prediction task [2], and relies in particular on statistical

[5] Malware don't need Coffee, `https://malware.dontneedcoffee.com`

[6] Alexa top sites, `http://www.alexa.com/topsites`

assumptions about the distribution of the training data to construct models, which are then used for future analyses. As a consequence, adversaries could also exploit these limitations to disrupt the analysis process, not to mention engaging malicious activities that could fail to be detected [13]. Our approach is resilient to attacks benefitting from the adaptive aspect of machine learning to design training data that will cause the learning system to produce models misclassifying future inputs, as the system never uses unknown input data as a training set. Attackers could rather try to manipulate a malicious sample to find a variant, preserving its maliciousness, but which would be classified as benign [15, 36]. Evasion is made somewhat more difficult because of the absence of a classification score. In particular, our system does preserve a lower false-negative rate than Zozzle when confronted to the Jshield samples [6] (80.14% versus 36.7%), admittedly tailored to avoid Zozzle's detection. These malicious files have been polluted by an injection of benign features, as it decreases their maliciousness by statistically reducing the impact of their malicious features. To foil JaSt, an attacker could rather inject benign functions into a malicious file, which would not change its functionality but would statistically reduce our system's maliciousness rating. Even though it is highly effective (from our 200 malicious samples modified by a transplantation of 100 random benign functions, none of them were detected), we did not have any such false-negatives in our dataset. This would furthermore be easy to detect and avoid, e.g., with dead code elimination or even statically with Esprima, comparing *FunctionDeclaration*'s *Identifiers* with *CallExpression*'s *Identifiers* since these functions are never called. Another attack on JaSt might consist of inserting a malicious sample into a –preferably significantly bigger– benign one. Even if it accounted for around a fourth of our false-negatives, it barely represents 0.1% of our malicious dataset, the adversaries rather relying on obfuscation to hide their malicious purpose. A defense against further adversarial attacks could be to combine the predictions of several classifiers (if the throughput is not a constraint). Another possibility consists in adding some parameters to our evaluation system, like the number of (different) nodes in the AST or the amount of different n-grams used, since it has a direct impact on the frequency analysis whom an attacker might try to foil.

4.2 Extension of an Email-Based Model to Detect Web Samples

Section 3.2 presents the detection performance of JaSt after having been trained and tested with malicious emails (and Microsoft samples for the benign part). As this model has yielded good predictions on web frameworks, the source code of Atom and also on the peculiar coding style of JavaScript games, even though these families were not represented in the model, it is worth looking at an extension of an email-based model to detect other types of JavaScript. In this experiment, we used the five email-based models, constructed previously, to classify inline JavaScript extracted from malicious HTML email-attachments, exploit kits from 2010 to 2017, and Alexa top 10k web pages. For Alexa, we also extracted third-party scripts and considered that all scripts were benign. While this is arguable in theory, JavaScript extracted from the first layer of the ten thousand web sites with the highest ranking provided us in practice enough confidence for this experiment. Although it has been showed that these web sites could host malicious advertisements, our JavaScript extraction process, which relies on statically parsing the web page with Python and extracting *script* and *src* tags, protected us from these elements generated dynamically. Figure 6 presents the detection accuracy (in terms of either true-positive or true-negative rates) on the previous samples. An HTML page or an exploit kit were considered benign if all the JavaScript snippets they contained were classified as benign. If one malicious JavaScript sample was detected, the whole page was labeled as malicious.

JaSt was able to detect 82.31% of the 13,595 malicious web JavaScript, which shows a certain similarity at the 4-grams level between email-JavaScript and web-JavaScript. Further insights into the false-negatives indicated that 14.37% of the previous samples had been *correctly* classified as benign. As a matter of fact, a manual inspection of 80 exploit kits showed that in 21,25% of the cases, the malicious part was *not* embedded in JavaScript samples. Instead, the attack vector was either contained in an SWF bundle or the exploit kit merely included a resource trying to exploit an existing flaw without any scripting code at all. Another issue is related to the quality of JavaScript samples: while analyzing 110 malicious email attachments and exploit kits, we discovered that some files were broken and could therefore not be parsed. In 3.64% of the cases, the *malicious* part could not be parsed, therefore not analyzed. We note that this means that in an attack, this code would not have been executed. As a consequence, when considering only HTML documents and exploit kits which could be entirely parsed and whose malicious behavior was included into a JavaScript snippet, we got a true-positive rate after treatment of 85,18%, which represents an improvement of 3.36%. While malicious email and web samples present some similarities, they also have syntactic differences, which prevented JaSt to provide as much confidence in the detection of malicious web-JavaScript as for email-inputs. As a matter of fact, the former tends to contain less malicious patterns and rather have comments at regular intervals, benign snippets next to the malicious part, and a different form of obfuscation than malicious email-JavaScript. While the latter aims at providing a unique copy for each recipient and therefore abuses *variable and function name randomization*, *data obfuscation* and *encoding obfuscation* [35], malicious web-JavaScript rather tend to identify software vulnerabilities in client machines and exploit them to upload and execute malicious code on the client side. For this purpose, the attackers preferably use *variable and function name randomization* and neatly package their code, which can slightly degrade JaSt efficiency.

As for the detection of benign JavaScript extracted from Alexa top 10k, JaSt detected 46.11% of them. Instead of grouping all JavaScript snippets of a web page together and labeling the web site as benign if all samples were recognized as benign, we performed a second experiment. We collected every JavaScript snippet of Alexa top 10k (between 100 bytes and 1 megabyte, so as to have JavaScript code with enough features to be representative of the benign or of the malicious class, without downgrading the performance with too big a size) and considered them one after the other. In total, we extracted 119,125 JavaScript samples and got a true-negative rate of 92.79% (averaged over 5 runs) –which is much more acceptable than previously, based on an exclusive non-web-model– therefore a false-positive rate of 7.21%. If we consider that an Alexa top 10k web page contains *n* JavaScript snippets, we could expect a false-positive rate of $100 - 7.21^{n}\%$. In average it contains 16 snippets, therefore the probability of getting a false-positive is 69.82%, even higher than the 46.11% we found, since an average value does not give any information regarding the data distribution. However, we envision that JaSt would not operate on an unfiltered set of all Web pages, but rather use greyware which already showed some indication of maliciousness (e.g., by instantiating an ActiveX object). This approach was also used by Kizzle [30].

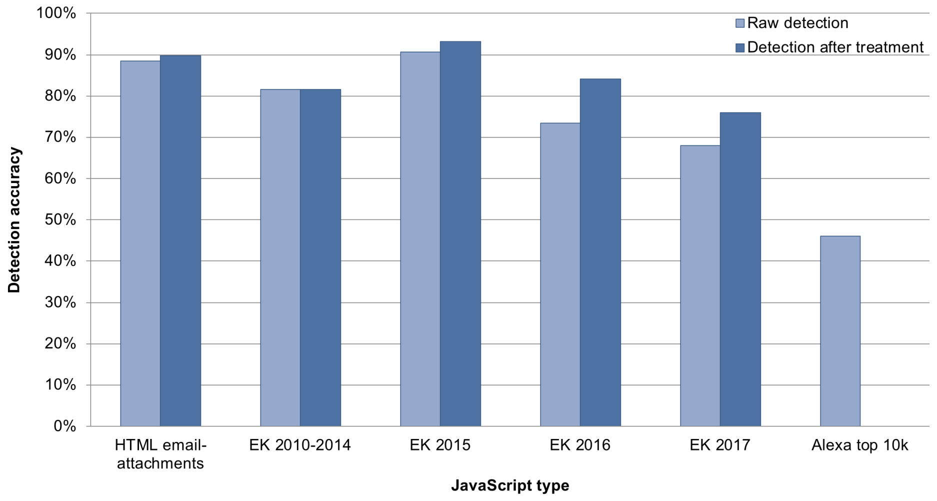


Fig. 6: Detection accuracy of web JavaScript based on an email-model

We chose not to include any web-JavaScript extracted from HTML documents for the training and evaluation part of this paper, but rather discuss the extension of an email-based model to detect web samples for three reasons. First, we did not have any ground truth regarding the position of the malicious entity in the malicious HTML file, which would have required a systematic analysis of our 13,595 snippets to detect and use only the *malicious* JavaScript samples to train our classifier with. For this reason, we decided to exclude them from the evaluation part and instead chose to flag any HTML documents containing at least one malicious JavaScript snippet as malicious. For symmetry purpose, we applied the same treatment to benign HTML files, which thereby reduced the number of benign scripts in our dataset. Last but not least, splitting email and web evaluation was a way to show that the syntax-based features can be core in classifying JavaScript: no matter the obfuscation used for hiding their functionality, malicious JavaScript do not necessarily hide their true function.

5 Related Work

In the literature, several systems used essential differences in lexical, syntactic, or other structural properties of JavaScript files to analyze them.

Lexical analysis Several approaches benefitted from lexical units and an SVM classifier to distinguish benign from malicious instances. In particular, Rieck et al. developed Cujo [29], a system combining static and dynamic analyses for the automatic detection and prevention of drive-by download attacks. Embedded in a web proxy, it transparently inspects web pages, extracting generic features based on an n-gram analysis of JavaScript lexical units, and implementing a learning-based detection. With PJScan, Laskov et al. [21] also combined the extraction of lexical units with an n-gram analysis to detect malicious PDF documents. Contrary to Cujo, the learning phase was performed only on malicious samples, the idea being to build a model of normality and label the files not respecting this model as benign. Beyond pure JavaScript detection, Kar et al. [18] showed with SQLiGoT that a lexical analysis could also be performed to detect SQL injections. This system normalizes SQL queries into sequences of tokens and generates a weighed graph representing the interactions between the previous tokens, before training a classifier to identify malicious nodes.

Syntactic units derived from the AST Some other systems rather made use of some essential features extracted from the AST to analyze JavaScript instances. For example, Curtsinger et al. implemented Zozzle [7], a mostly static JavaScript malware detector deployed in the browser. It combines the extraction of features from the AST, as well as the corresponding JavaScript text, with a Bayesian classification system to identify syntax elements highly predictive of malware. To address the issue of obfuscation, Zozzle is integrated with the browser's JavaScript engine to collect and process the code created at runtime. A naive Bayes classification algorithm was also used by Hao et al. [9] to analyze JavaScript code by benefitting from extended API symbol features by means of the AST. Beyond a pure malware analysis, Kapravelos et al. [17] rather chose to detect JavaScript samples, which were an evolution of known malicious files, or modified malicious instances, now tailored to be recognized as benign (evasion process). For this purpose, they developed Revolver to automatically detect evasive behavior in malicious JavaScript files, by using the AST as well as a dynamic analysis to identify similarities between JavaScript samples. Besides detecting malicious instances directly, the JavaScript AST can effectively be used to identify a programmer (for plagiarism purpose, or to indirectly detect potentially malicious files, based on the writing skills of a known malware author). For this application, Wisse et al. [33] extracted structural features from the AST, used n-grams to describe the coding style of an author and their frequency analysis to recognize the programmer. As for dissimulation techniques, Kaplan et al. [16] quantified with NoFus the fact that obfuscation does not imply maliciousness: this static and automatic classifier can indeed distinguish obfuscated and non-obfuscated JavaScript files, using a Bayesian classifier over the AST. More generally, Yamaguchi et al. [37] used ASTs to identify zero-day vulnerabilities. Indeed, they guided the search for new exploits by extrapolating known vulnerabilities using structural patterns extracted from the ASTs, which enabled them to find similar flaws in other projects.

Other detection or clustering tools Lexical and syntactic analyses aside, additional tools, benefitting from other features, can be found in the wild. Essential differences in structural properties between benign and malicious files can also be a way for detecting malicious PDF documents, for example, as explained by Šrndic et al. [31]. Kizzle from Stock et al. [30] also aimed at clustering malware samples with a special focus on exploit kits. This malware signature compiler benefits from the fact that the attackers reuse code while delivering it in various kits. Then Kolbitsch et al. implemented Rozzle [19], a JavaScript virtual machine exploring multiple execution paths in parallel to detect environment specific malware. In practice, it imitates multiple browser and environment configurations while dynamically crawling to detect malware. EvilSeed [12] is an approach designed by Invernizzi et al. to efficiently search the web for pages that are likely to be malicious. Its starts from an initial seed of known malicious web pages to identify other malicious ones by similarity or relation to the seed. Finally, Canali et al. [5] had also been working on a faster collection of malicious web pages with Prophiler. This filter quickly discards benign pages based on HTML-derived lexical features, the JavaScript AST, and an URL analysis.

6 Conclusion

Many malicious JavaScript samples today are obfuscated to hinder the analysis and creation of signatures. To countermand this, in this paper we proposed JaSt, a fully static AST-based analysis to automatically detect malicious JavaScript instances. The key elements of this approach are: (a) an extraction of the syntactic units contained in the files to be classified; (b) a frequency analysis of the n-grams built upon the previous features; (c) the construction of a random forest using the previous frequencies as input to either build a model or classify unknown samples and (d) the evaluation of JaSt on an extensive, up-to-date and balanced JavaScript dataset. In practice, our approach yields extremely accurate predictions (almost 99.50% of correct classification results) and has an outstanding false-negative rate of 0.54%, especially since the system is *entirely* static. Despite this high detection performance, JaSt is also quite fast with a mean throughput of 0.14 megabyte per second, while considering more than 2,400 different features, either predictive of malicious or of benign samples. Besides, this selection of abstract patterns enables our system to be resistant to common obfuscation transformations without having to execute some code. As a consequence, it cannot be foiled by malware variants whose behavior are time- or environment dependent.

To be long-time effective, JaSt has to adapt to new JavaScript instances. This adaptation process is achieved by extending the set of n-grams feature used with new, up-to-date JavaScript samples. As a matter of fact, an analysis of JavaScript instances over a few months showed that building a model each month to detect malicious variants in the current month is only effective if the training set contains enough files representative of the distribution found in the wild, and is not simply a compilation of different JavaScript waves received in the past few months. Last but not least we showed in this paper that the same benign and malicious patterns are partially found in different JavaScript families. As a consequence, a model constructed with malicious emails and benign frameworks can even be used to classify exploit kits, JavaScript games, or other web-JavaScript inputs, highlighting the similarity between different classes of malicious JavaScript.

Acknowledgments

This work would not have been possible without the help of the German Federal Office for Information Security and Kafeine DNC which provided us with materials for our experiments. We would also like to thank the anonymous reviewers of this paper for their well-appreciated feedback. This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345).

References

1. Atom: Atom the hackable text editor for the 21st Century. In: `https://atom.io`. Accessed on 2018-02-21
2. Backes, M., Nauman, M.: LUNA: Quantifying and Leveraging Uncertainty in Android Malware Analysis through Bayesian Machine Learning. In: Euro S&P (2017)
3. Bergstra, J., Bengio, Y.: Random Search for Hyper-parameter Optimization. In: Journal of Machine Learning Research (2012)
4. Breiman, L.: Random Forests. In: Machine Learning (2001)
5. Canali, D., Cova, M., Vigna, G., Kruegel, C.: Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In: International Conference on World Wide Web (2011)
6. Cao, Y., Pan, X., Chen, Y., Zhuge, J.: JShield: Towards Real-time and Vulnerability-based Detection of Polluted Drive-by Download Attacks. In: Annual Computer Security Applications Conference (ACSAC) (2014)
7. Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: Zozzle: Fast and Precise In-Browser JavaScript Malware Detection. In: USENIX (2011)
8. Gastwirth, J.L.: The Estimation of the Lorenz Curve and Gini Index. In: Review of Economics and Statistics (1972)
9. Hao, Y., Liang, H., Zhang, D., Zhao, Q., Cui, B.: JavaScript Malicious Codes Analysis Based on Naive Bayes Classification. In: International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (2014)
10. Hidayat, A.: ECMAScript Parsing Infrastructure for Multipurpose Analysis. In: `http://esprima.org`. Accessed on 2017-04-05
11. AV-TEST - The Independent IT-Security Institute: New malware. In: `https://www.av-test.org/en/statistics/malware`. Accessed on 2018-02-01
12. Invernizzi, L., Benvenuti, S., Cova, M., Comparetti, P.M., Kruegel, C., Vigna, G.: EvilSeed: A Guided Approach to Finding Malicious Web Pages. In: S&P (2012)
13. Joseph, A.D., Laskov, P., Roli, F., Tygar, J.D., Nelson, B.: Machine Learning Methods for Computer Security. In: Dagstuhl Manifestos (2013)
14. Jules, D.S.: JSinspect Detect copy-pasted and structurally similar code. In: `https://github.com/danielstjules/jsinspect`. Accessed on 2018-02-19
15. Kantchelian, A., Tygar, J.D., Joseph, A.D.: Evasion and Hardening of Tree Ensemble Classifiers. In: International Conference on Machine Learning (2016)
16. Kaplan, S., Livshits, B., Zorn, B., Siefert, C., Curtsinger, C.: "NoFus: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code". In: Microsoft Research Technical Report (2011)
17. Kapravelos, A., Shoshitaishvili, Y., Cova, M., Krügel and Giovanni Vigna, C.: Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In: USENIX (2013)
18. Kar, D., Panigrahi, S., Sundararajan, S.: SQLiGot: Detecting SQL Injections Attacks using Graph of Tokens and SVM. In: Computers & Security (2016)

19. Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: De-cloaking Internet Malware. In: S&P (2012)
20. Kolter, J.Z., Maloof, M.A.: Learning to Detect and Classify Malicious Executables in the Wild. In: Journal of Machine Learning Research (2006)
21. Laskov, P., Šrndić, N.: Static Detection of Malicious JavaScript-Bearing PDF Documents. In: Annual Computer Security Applications Conference (ACSAC) (2011)
22. Likarish, P., Jung, E., Jo, I.: Obfuscated Malicious Javascript Detection using Classification Techniques. In: International Conference on Malicious and Unwanted Software (MALWARE) (2009)
23. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research 12, 2825–2830 (2011)
24. Powers, D.M.W.: Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. In: Journal of Machine Learning Technologies (2011)
25. Rao, V., Hande, K.: A Comparative Study of Static, Dynamic and Hybrid analysis Techniques for Android Malware Detection. In: International Journal of Engineering Development and Research (IJEDR) (2017)
26. Symantec Security Response: Mirai: what you need to know about the botnet behind recent major DDoS attacks. In: `https://www.symantec.com/connect/blogs/mirai-what-you-need-know-about-botnet-behind-recent-major-ddos-attacks`. Accessed on 2018-02-14
27. Symantec Security Response: Petya ransomware outbreak: Here is what you need to know. In: `https://www.symantec.com/blogs/threat-intelligence/petya-ransomware-wiper`. Accessed on 2018-02-14
28. Symantec Security Response: What you need to know about the WannaCry Ransomware. In: `https://www.symantec.com/blogs/threat-intelligence/wannacry-ransomware-attack`. Accessed on 2018-02-14
29. Rieck, K., Krueger, T., Dewald, A.: Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In: Annual Computer Security Applications Conference (ACSAC) (2010)
30. Stock, B., Livshits, B., Zorn, B.: Kizzle: A Signature Compiler for Detecting Exploit Kits. In: Dependable Systems and Networks (DSN) (2016)
31. Šrndić, N., Laskov, P.: Detection of Malicious PDF Files Based on Hierarchical Document Structure. In: NDSS (2013)
32. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: International Conference on Recent Advances in Intrusion Detection (RAID) (2006)
33. Wisse, W., Veenman, C.J.: Scripting DNA: Identifying the JavaScript Programmer. In: Digital Investigation (2015)
34. Wressnegger, C., Schwenk, G., Arp, D., Rieck, K.: A Close Look on n-Grams in Intrusion Detection: Anomaly Detection vs. Classification. In: ACM workshop on Artificial intelligence and security (AISec) (2013)
35. Xu, W., Zhang, F., Zhu, S.: The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study. In: International Conference on Malicious and Unwanted Software (MALWARE) (2012)
36. Xu, W., Qi, Y., Evans, D.: Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In: NDSS (2016)
37. Yamaguchi, F., Lottmann, M., Rieck, K.: Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In: Annual Computer Security Applications Conference (ACSAC) (2012)
38. Youden, W.J.: Index for Rating Diagnostic Tests. In: Cancer (1950)