
# JStap: A Static Pre-Filter for Malicious JavaScript Detection

Aurore Fass, Michael Backes, and Ben Stock
CISPA Helmholtz Center for Information Security
{aurore.fass,backes,stock}@cispa.saarland

## ABSTRACT

Given the success of the Web platform, attackers have abused its main programming language, namely JavaScript, to mount different types of attacks on their victims. Due to the large volume of such malicious scripts, detection systems rely on static analyses to quickly process the vast majority of samples. These static approaches are not infallible though and lead to misclassifications. Also, they lack semantic information to go beyond purely syntactic approaches. In this paper, we propose JStap, a modular static JavaScript detection system, which extends the detection capability of existing lexical and AST-based pipelines by also leveraging control and data flow information. Our detector is composed of ten modules, including five different ways of abstracting code, with differing levels of context and semantic information, and two ways of extracting features. Based on the frequency of these specific patterns, we train a random forest classifier for each module. In practice, JStap outperforms existing systems, which we reimplemented and tested on our dataset totaling over 270,000 samples. To improve the detection, we also combine the predictions of several modules. A first layer of unanimous voting classifies 93% of our dataset with an accuracy of 99.73%, while a second layer–based on an alternative modules' combination–labels another 6.5% of our initial dataset with an accuracy over 99%. This way, JStap can be used as a precise pre-filter, meaning that it would only need to forward less than 1% of samples to additional analyses. For reproducibility and direct deployability of our modules, we make our system publicly available.[1]

## CCS CONCEPTS

• **Security and privacy** → *Web application security*; *Malware and its mitigation.*

## KEYWORDS

Web Security, Malicious JavaScript, Data-Flow, Control-Flow, AST

**ACM Reference Format:**
Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359789.3359813

[1] https://github.com/Aurore54F/JStap


*ACSAC '19, December 9–13, 2019, San Juan, PR, USA*

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359813

## 1 INTRODUCTION

The Web has become the most popular software platform, used by billions of people every day. Given its popularity, it naturally attracts the interest of malicious actors, which try to leverage the Web as a vector for attacking their victim's computer. Specifically, attackers abuse JavaScript to exploit bugs in the browser, probe systems for vulnerabilities before injecting malicious content [9, 24], e.g., targeting Flash plugins, or mine cryptocurrencies without user's consent [33]. Due to this plethora of attacks, the antivirus industry has increased the focus on the detection of such nefarious scripts. As a response, attackers use obfuscation techniques [56], which foils techniques directly relying on content matching (e.g., traditional antivirus signatures) and imposes additional hurdles to manual analysis. Nevertheless, abstracting the code on a lexical (e.g., keywords, identifiers) or syntactic level (e.g., statement or expression nodes extracted from the AST (Abstract Syntax Tree)) enables to collect specific and recurrent features, either typical of malicious or of benign intent. This way, machine learning-based detectors can leverage such static features for an accurate malicious JavaScript detection [12, 17, 45].

Due to their speed and accuracy, static systems are particularly relevant to quickly discard benign samples, leaving only those few which are likely malicious for costly manual analysis or dynamic components [9]; at the same, this implies that they must be accurate to neither waste expensive resources nor let malicious files through. Such static approaches are not infallible though. In particular, different lexical and AST-based detectors tend to yield (different) false negatives and false positives [12, 17, 45]. At the same time, such static systems merely consider the syntax of the analyzed files, i.e., how the lexical units (tokens) are arranged, or traverse the AST to extract syntactic units. However, they do not retain semantic information, such as control or data flow. This means that while they take the *syntactic order* of code into account, they do not leverage the *semantic order* of the code's logic.

In this paper, we extend the detection capability of existing lexical and AST-based pipelines to pre-filter JavaScript samples, by augmenting such approaches with control and data flow information. This way, we have a higher overall detection rate than existing systems, while also limiting the number of samples forwarded to more costly analyses. To this end and contrary to purely lexical and syntactic systems, we also consider semantic information in our abstract code representations, by means of control and data flow. In particular, the Control Flow Graph (CFG) takes into account the flow of control between statements. Besides flow of control, the PDG (Program Dependency Graph) also considers the dependencies

between statements, meaning that, e.g., dead-code would not be linked to the actual functionality of the programs by such flows. On the contrary, ASTs and lexical units mainly represent the arbitrary sequencing choices made by the programmers. Specifically, we present JStap, a modular JavaScript static classification system for which the user can choose the level of the analysis, namely lexical, AST, CFG, PDG with data flow only and PDG with data and control flow. Similarly, the user may also choose to combine several schemes. For a better overall detection accuracy and to limit the number of features overlapping between the different components, JStap can either analyze these features using an n-gram approach or combine them with variable's name information. We refer to the resulting combinations of these two ways of analyzing features extracted from one of the five considered code representations as ten different *modules*. Due to the static character and high detection accuracy of each of JStap's module separately, we envision that combining several modules–with different code representations, with more or less semantic information–can be used as a pre-filtering step, sending only samples with conflicting labels to further analysis. This way, JStap can help avoid unnecessary invocation of costly dynamic analyses.

Our implementation responds to the following challenges: resilience to common obfuscation transformations, practical applicability and high accuracy in terms of JavaScript classification, and robustness against malware attempting to evade detection. We address these challenges by proposing a methodology to build and traverse the AST, CFG, and PDG before extracting and leveraging specific features from these graphs, also considering lexical units, for an effective and reliable malicious JavaScript detection. The key elements of JStap are the following:

- *Fully Static JavaScript Analysis* — Besides extracting lexical units, our system also leverages the AST produced by Esprima [23] to build a CFG and a PDG, also representing the control and/or data flow between the nodes.

- *Features Extraction* — We traverse the previous static structures, extracting and combining tokens' or nodes' information before selecting features typical of benign or malicious samples.

- *Accurate JavaScript Classification* — JStap considers the frequency of the features previously extracted to build a random forest model and accurately classify unknown JavaScript samples.

We evaluate our system on an extensive dataset totaling over 270,000 samples, including over 130,000 unique malicious JavaScript samples and over 140,000 unique benign scripts. We focus on the true-positive and true-negative rates of each JStap module separately, the best one having a detection rate of 99.44%, which is significantly higher than closely related work implementations, which we trained and tested on our dataset. To make even more accurate predictions, we envision that a combination of JStap modules could be used as a pre-filtering step, before sending only samples with conflicting labels to more costly dynamic components or manual analysis. In this configuration, we have a detection accuracy of 99.73% on 93% of our dataset, for which the selected modules can make a unanimous decision. For the remaining 7% of the samples, we can classify them with a second layer of unanimous voting, from different modules, and with an accuracy still over 99%; meaning that less than 1% of our initial dataset is sent to more costly analyses.

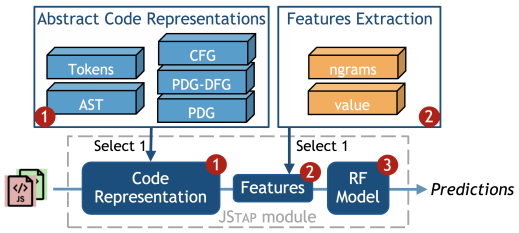


**Figure 1: Architecture of JStap with focus on one module**

For reproducibility and direct deployability of our modules, we make our system publicly available. Specifically, we release as open-source software our code to a) build the different data structures we used (e.g., CFG, PDG), b) train a random forest classifier on JavaScript samples (including the features selection process with $\chi^2$) and c) classify (unknown) JavaScript inputs. More details can be found at https://github.com/Aurore54F/JStap.

## 2 METHODOLOGY

JStap is composed of several modules, which can run independently or combined, to accurately detect malicious JavaScript inputs. The architecture of each module consists of an abstract code representation (stage 1 of Figure 1), a feature-extractor (stage 2) and learning components (stage 3). First, we perform a static analysis of JavaScript samples, leveraging the Abstract Syntax Tree (AST) to build the Control Flow Graph (CFG) and Program Dependency Graph (PDG). Then, we traverse the graphs by following the control and/or data flow to extract syntactic units, whose combination still carries the initial control or data flow semantics. We also consider lexical units and syntactic units extracted from the AST to extend our approach with node context information, since control and data flow only link statement nodes together. In particular, we combine the previous units by groups of $n$, to build n-gram features. At the same time, and independently of the prior approach, we also combine the previous units with variable's name information. In both cases, we use the frequency of the extracted features as input to learning components, which distinguish benign from malicious JavaScript samples. In the following sections, we discuss the details of each stage in turn.

### 2.1 Abstract Code Representations

The choice of a static analysis to detect malicious JavaScript instances is motivated by its speed, reliability, and code coverage. In particular, we can leverage different levels of code abstraction, with more or less semantic information, to identify recurrent programmatic and structural constructs specific to malicious or benign reports. In particular, a lexical analysis directly processes the code, one word after the other. On the contrary, an AST-based analysis takes into account the grammar, thereby the syntactic structure of the program. As for the CFG, it adds some semantic information to the analysis, as it takes into account the conditions that have to be met for a specific program's path to be taken. Finally, the PDG adds more semantics by also considering the order in which statements have to be executed. This way, each code representation processes

**Table 1: Lexical units extracted from the code of Listing 1**

| Token | Value | Token | Value | Token | Value |
|---|---|---|---|---|---|
| Identifier | x | Numeric | 1 | Punctuator | ) |
| Punctuator | . | Punctuator | ; | Punctuator | { |
| Keyword | if | Keyword | if | Identifier | d |
| Punctuator | = | Punctuator | ( | Punctuator | = |
| Numeric | 1 | Identifier | x | Identifier | y |
| Punctuator | ; | Punctuator | . | Punctuator | ; |
| Keyword | var | Keyword | if | Punctuator | } |
| Identifier | y | Punctuator | == | | |
| Punctuator | = | Numeric | 1 | | |

JavaScript at a different static level. Thereby, they can be combined to represent the different code's properties more accurately.

*2.1.1 Lexical Units Extraction.* First, we perform a lexical analysis of JavaScript with the tokenizer Esprima [23], which builds an abstract representation of the code. This way, the source code is linearly converted into a list of abstract symbols representing lexical units (e.g., `Keyword`, `Identifier`). Still, this technique uses neither the context in which a given word appears nor the overall syntactic structure of the snippet it analyses; therefore it is, e.g., unable to infer that a traditionally reserved word (*if*) is not always used as a `Keyword`, but can be used as an `Identifier` (Table 1, line 3).

*2.1.2 Abstract Syntax Tree (AST).* Contrary to the previous tokens, the AST describes the syntactic structure of an input sample, as it rests upon the JavaScript grammar [15]. In particular, we use the parser Esprima [23], which can produce up to 69 different syntactic units, referred to as nodes. Inner nodes represent *operators* such as `VariableDeclaration`, `AssignmentExpression` or `IfStatement`, while the leaf nodes are *operands*, e.g., `Identifier`, `Literal` or `EmptyStatement`. As an illustration, Figure 2 shows the Esprima AST obtained from the code snippet of Listing 1 (for legibility reasons, the variables' names and values appear in the paper's graphical representations, but they are not part of the graphs). This time, the construct *x.if* is recognized as a `MemberExpression` with *x* and *if* being correctly labeled as `Identifier`. Still, the AST only retains information about the nesting of programming constructs to form the source code but does not contain any semantic information such as control or data flow.

*2.1.3 Control Flow Graph (CFG).* Contrary to the AST, the CFG allows to reason about the conditions that have to be met for specific program's paths to be taken. To this end, statements (predicates and non-predicates) are represented by nodes that are connected by labeled and directed edges to represent flow of control.

Since the Esprima AST comprises not only statements but also non-statement and still non-terminal nodes, as shown in Figure 2, we construct the CFG over the previous AST (we refer to the extension of the AST with control flow edges as *CFG*) so as not to lose the relationships between nodes, which are both non-statement and non terminal. We use the JavaScript grammar [15] to determine which nodes are statements and which ones are not. Nevertheless, we consider that a `SwitchCase` and a `ConditionalExpression` are both statements, in order to indicate the conditional flow of control originating from these two nodes. Then, we traverse the AST depth-first pre-order and define two labels to link statement

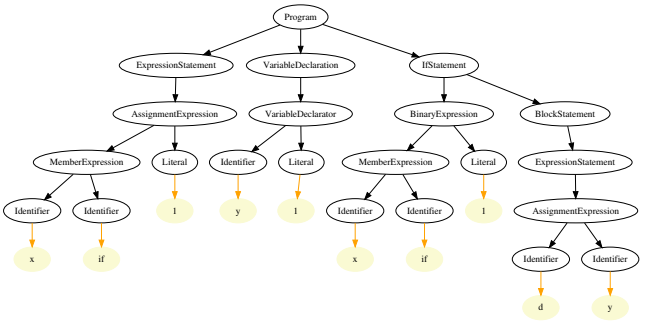


**Figure 2: AST corresponding to the code of Listing 1**

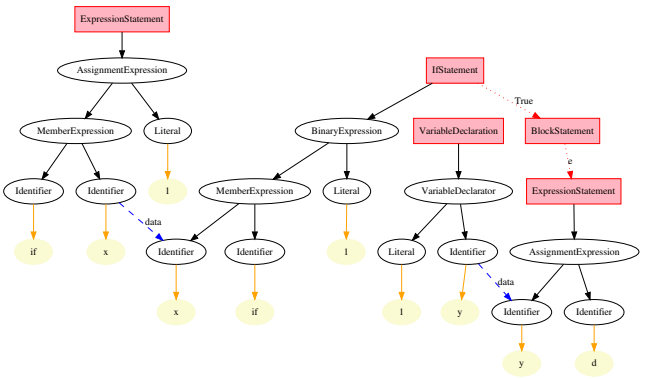


**Figure 3: AST of Listing 1 extended with control flow (red dotted edges) and data flow (blue dashed edges)**

```
1 x.if = 1;
2 var y = 1;
3 if (x.if == 1) {d = y;}
```

**Listing 1: JavaScript code example**

nodes with a control flow dependency. The label *e* is used for edges originating from non-predicate statements, while edges originating from predicates are labeled with a boolean, standing for the value the predicate has to evaluate to, for this path in the graph to be chosen. Contrary to the AST of Figure 2, Figure 3 (considering only the control flow edges) shows an execution path difference when the *if* condition is true, and when it is not. Still, the CFG does not enable to reason about the order in which statements are executed.

*2.1.4 Program Dependency Graph (PDG).* To this end, we build a PDG [20] by adding data flow information to the previous CFG. We connect statements with a directed data dependency edge iff a variable (also including object and function) defined or modified at the source node is used at the destination node, taking into account its reaching definition. Since this code representation captures the data and control flow between the different program components, it is not influenced by arbitrary sequencing choices made by the programmer. Contrary to the AST of Figure 2, Figure 3 indicates the order in which statements from Listing 1 should be executed (for legibility reasons, we drew the data dependencies between

leaf nodes instead of their corresponding nearest statement nodes), e.g., as shown by the data flow, lines 1 and 2 are executed before line 3; still, we could swap lines 1 and 2 without altering the code semantics.

In particular, our PDG implementation respects JavaScript's scoping rules, makes the distinction between function declarations–a standalone construct defining named function variables–and function expressions–functions that are part of larger expressions–and handles lexical scoping. Also, we connect the function call nodes to the corresponding function definition nodes with a data dependency, thus defining the PDG at the program level [57].

## 2.2 Features Extraction

Once JStap built abstract code representations to analyze JavaScript samples, we extract lexical units and traverse the different graphs to collect syntactic units. Subsequently and for each code representation, we consider (independently) n-gram features and the combination of the extracted units with their corresponding node's value (variable's name). Finally, learning components take the frequency of such features as input for the classification process.

*2.2.1 Graph Traversal.* As far as the lexical analysis is concerned, we already extracted lexical units (tokens) in Section 2.1.1. For the AST, CFG and PDG, we need to traverse each graph by following its specific edges to extract the name of each node (referred to as a syntactic unit). Specifically, a depth-first pre-order traversal of Figure 2 gives the following syntactic units: `ExpressionStatement`, `AssignmentExpression`, `MemberExpression`, [...] `Identifier` (the `Program` node just represents the root and does not have any syntactic meaning). For the CFG, we also traverse the AST but only store nodes linked by a control flow edge (i.e., the *e*, *True* and *False* labels), e.g., on Figure 3: `IfStatement`, `BlockStatement` and `ExpressionStatement`. In practice, considering only statement nodes is not informative enough to distinguish benign from malicious JavaScript inputs though (due to them linking the same statements with one another, cf. Section 3.2.1). To add more context information, we also traverse the sub-AST of each node with a control flow *once*. For example, in Figure 3, JStap reports the `IfStatement` node and traverses its sub-AST before following the control flow and traversing the `BlockStatement`, then the `ExpressionStatement` nodes. This time, we do not traverse their corresponding sub-ASTs, as we already did it, while handling the `IfStatement` node.[2] Finally, the process is similar for the PDG, with consideration of the data flow. In the following, we use the term *PDG-DFG* (for *Data Flow Graph*) to refer to this traversal only along data flow edges, and the term *PDG* to refer to the PDG traversal through the data flow edges, followed by a second traversal along the control flow edges.

*2.2.2 Features Analysis.* For the five previous abstract code representations, namely tokens, AST, CFG, PDG-DFG, and PDG, we (independently) consider n-gram features and the combination of the extracted units with their corresponding node values to build features. JStap therefore contains ten modules, with five different static code analysis levels, and two ways of representing features extracted from these different code representations.

[2]At the end, we retain the following units: `IfStatement, BinaryExpression, MemberExpression, Identifier, Identifier, Literal, BlockStatement, ExpressionStatement, AssignmentExpression, Identifier, Identifier, BlockStatement, ExpressionStatement`

**Table 2: Number of relevant features per module**

| | Tokens | AST | CFG | PDG-DFG | PDG |
|---|---|---|---|---|---|
| `ngrams` | 602 | 11,050 | 18,105 | 17,997 | 24,706 |
| `value` | 24,912 | 45,159 | 36,961 | 45,566 | 46,375 |

*N-Gram Features.* To identify specific patterns in JavaScript documents, in the first scenario, we move a fixed-length window of $n$ symbols over each lexical or syntactic unit previously extracted, to get every sub-sequence of length $n$ (n-grams) at each position. For example, the first 2-grams of Table 1 are: `(Identifier, Punctuator)`, `(Punctuator, Keyword)` and `(Keyword, Punctuator)`. The use of n-grams feature enables a representation of how the lexical and syntactic units were originally arranged in the analyzed JavaScript files, and is an effective means for abstracting the files [32, 34, 36, 53, 54]. Thus, we build n-grams upon the lexical units and the features previously extracted from the AST, CFG, PDG-DFG and PDG (Section 2.2.1). We empirically evaluated different $n$ values, and selected $n = 4$, which provides the best trade-off between detection accuracy and run-time performance. In the following, we use the keyword `ngrams` to refer to the 4-gram features we built as described above.

*Node Value Features.* In the second scenario, we do not use n-gram features, but combine each lexical unit with their corresponding value (as presented in Table 1) and each syntactic unit extracted from the AST, CFG, PDG-DFG, and PDG with their corresponding `Identifier/Literal` value. For example, the first features of Figure 2 are (`ExpressionStatement`, x) and (`AssignmentExpression`, x). In the following, we use the keyword `value` to refer to the features combining lexical or syntactic units with their corresponding value, as described above.

*2.2.3 Features Space.* JavaScript samples sharing several features with the same frequency present similarities with one another, while files with different features have a more dissimilar content. Hence, analyzing the frequency of the features previously extracted (`ngrams` and `value`, Section 2.2.2) is an indicator to accurately determine if a given input is either benign or malicious.

To compare the frequency of the features appearing in several JavaScript files, we construct a vector space such that each feature is associated with one consistent dimension, and its corresponding frequency is stored at this position in the vector. To limit the size of the vector space, which has a direct impact on the performance, we use the $\chi^2$ test to check for correlation. We select only features for which $\chi^2 \geq 6.63$, meaning that feature's presence and script classification are correlated with a confidence of 99% [52]. Table 2 presents the number of features considered for each of the ten JStap modules based on our training set, which we describe in Section 3.1.3. For the `ngrams` variant, there are more statistically representative features when the complexity of the code representation structure increases, as complex graph structures lead here to more edges. This also holds for the `value` approach, except for the CFG traversal, for which we both have fewer representative

features and fewer features in general than for the AST or PDG. We assume that it comes from benign and malicious actors using more similar variables name in statements with a control flow than in other statements. This is confirmed to some extent in Section 3.2.2, where this approach does not perform as well as the other ones. Finally, we store the frequency of each feature in Compressed Sparse Row (CSR matrix) [10] to efficiently represent non-zero values.

## 2.3 Learning and Classification

The learning-based detection completes the design of our system. We first build and leverage the CSR matrix of a representative and balanced set of both benign and malicious JavaScript files to train our classifier, as presented in Section 3.1.3. We empirically evaluated several off-the-shelf systems (Bernoulli naive Bayes, multinomial naive Bayes, Support Vector Machine (SVM), and random forest), and selected random forest, which provided the most reliable detection results, with the best true-positive and true-negative rates.

# 3 COMPREHENSIVE EVALUATION

In this section, we outline the results of our evaluation of JStap. In particular, we leverage a high-quality dataset from various sources, totaling over 270,000 unique samples. First, we study and justify the detection rate of all ten modules of JStap, comparing them with one another and analyzing their high(er) detection performance, before comparing our implementations with closely-related work, and explaining why our systems perform better. Finally, we analyze the detection accuracy of a detector combining the predictions of three JStap modules, before evaluating the overall run-time performance.

## 3.1 Experimental Protocol

The experimental evaluation of our approach rests upon two extensive datasets, with a total size over 6.2 GB. The first one contains 131,448 SHA1-unique malicious JavaScript samples and the second one 141,768 unique benign files. We used these two datasets to both train and test our random forest classifier on.

*3.1.1 Malicious Dataset.* Our malicious dataset (Table 3) is a collection of samples mainly provided by the German Federal Office for Information Security (BSI) [8]. These samples have been labeled as malicious based on a score provided by the combination of antivirus systems, malware scanners, and a dynamic analysis. To reduce possible similarities between samples from the same source, we got the malware collection of Hynek Petrak (Hynek) [43], exploit kits from Kafeine DNC (DNC) [27] and GeeksOnSecurity (GoS) [21], and additional samples from VirusTotal [50]. This way, our malicious dataset contains different samples performing various activities. For example, we have JScript-loaders leading to, e.g., drive-by download or ransomware attacks, and exploit kits (e.g., Blackhole, Donxref, RIG) targeting vulnerabilities in old versions of Java, Adobe Flash or Adobe Reader plugins, also trying to exploit old browsers versions. Most of these samples are obfuscated, e.g., through string manipulation, dynamic arrays, encoding obfuscation [56]. Even though the samples are labeled by their sources, in some cases, we extracted JavaScript from HTML documents and thereby had to ensure that the maliciousness lay in the script and was not, e.g., contained in an SWF bundle. For this purpose, we manually analyzed our 19,942 extracted JavaScript samples, 15,475 of which are malicious (we discarded the other samples, which are also not represented in Table 3). Since our analysis is entirely static, it provides a complete code coverage based on the proportion of source code analyzed. In turn, it is unable to consider dynamically generated JavaScript.[3] To this end, we parsed each malicious file with Esprima and automatically inlined all code passed through `eval` (for invocations with static strings). Thereby, we increased the code coverage of JStap on 1,868 unique scripts, as we did not merely consider a `CallExpression` node with a fixed string parameter anymore, but the code contained in the string, possibly (depending on JStap's selected module) along with its control and/or data flow. Also, 1,094 samples used conditional compilation [39], which Esprima parses as a large comment. Thus, we automatically replaced this construct with the corresponding code for the parser to produce the actual ASTs of such scripts.

**Table 3: Malicious JavaScript dataset**

| Source | #JS | Creation | Obfuscated |
|---|---|---|---|
| BSI | 83,361 | 2017-2018 | y |
| Hynek | 29,558 | 2015-17 | y |
| DNC | 12,982 | 2014-18 | y |
| GoS | 2,491 | 2017 | y |
| VirusTotal | 3,056 | 2018 | y |
| Total | 131,448 | 2014-18 | y |

**Table 4: Benign JavaScript dataset**

| Source | #JS | Collection | Obfuscated |
|---|---|---|---|
| Tranco-10k | 122,910 | 2019 | N/A |
| Microsoft | 16,271 | 2015-18 | y |
| Games | 1,992 | N/A | n |
| Web frameworks | 427 | N/A | N/A |
| Atom | 168 | 2011-18 | n |
| Total | 141,768 | - | - |

*3.1.2 Benign Dataset.* As for the benign dataset (Table 4), we used Chromium to visit the start pages of Tranco top 10,000 websites [35].[4] For each visited web page, we waited for the load of the page and observed the site for one second, to also collect dynamically generated scripts. In particular, we stored all inline scripts from the same document in one file–keeping the order in which they are executed–and consider all external scripts separately. This way, we obtain 122,910 unique JavaScript files. Given the fact that we extracted JavaScript from the start pages of high-profile websites, we assume them to be benign. Based on a study from Skolka et al. [48], over 30% of first-party scripts are either obfuscated or minified and over 55% of third-party scripts. In addition, we consider benign JavaScript from Microsoft products,[5] the majority of which are also obfuscated, which enables us to ensure that JStap does not confound obfuscation with maliciousness. As we also own malicious scripts from Microsoft (i.e., JScript-loaders), we are not introducing a bias in our dataset, even if Microsoft uses custom obfuscation methods.

[3] We discuss some possible drawbacks induced by the static analysis in Section 4.1
[4] Even though we *crawled* Tranco top 10,000 in 2019, the scripts were not necessarily written in 2019, such that our malicious dataset is not older than our benign dataset
[5] Microsoft Exchange 2016 and Microsoft Team Foundation Server 2017

Finally, we collected benign JavaScript from open source games, web frameworks, and Atom [2]. As these samples may contain new or specific (e.g., games) coding styles, we show that JStap does not confound unknown or unusual structures with maliciousness either.

*3.1.3 Classifier Training.* For the next sections, we built machine learning models using the following protocol. First, we randomly selected 10,000 JavaScript samples from our malicious dataset. We deemed our malicious training set to be representative of the distribution found in the wild due to our multiple malware sources and random selection from an initial malicious pool with over 130,000 samples. For the benign part, we also randomly selected 10,000 samples to build a balanced model. As previously, we assume our benign training set to be representative of the distribution found in the wild through our multiple sources, with different coding styles (e.g., games). In the following sections, we consider that the remaining samples are unknown and use them to evaluate the performance of the different detectors. In addition, we extracted all features present in our training set, before randomly selecting 5,000 new unique malicious and as many benign samples, to check on this validation dataset which of the previous features are correlated with the classification, using the $\chi^2$ test described in Section 2.2.3. In the remainder of this paper, we consider only these features.

We specifically chose to assemble balanced datasets, even though in reality, benign webpages outnumber malicious ones. With Tesseract [42], Pendlebury et al. argue that using unrealistic assumptions about the ratio of benign samples to malware in the data can lead to inflated detection results. In our case, it is not an issue, since we specifically chose metrics to evaluate the detection accuracy of JStap on *both* benign *and* malicious samples, and not merely a score to rate the proportion of correct predictions of our modules (cf. Section 3.2). Finally, to limit any statistical effects from randomized datasets, we repeated the previous procedure five times and averaged the detection results. Contrary to 5-fold cross-validation, we explore more ways of partitioning data, such that each sample is not necessarily tested only once.

## 3.2 JStap's Detection Performance

First, we compare the detection performance of JStap, in terms of true-positive (correct classification of a malicious script as malicious) and true-negative (correct classification of a benign input as benign) rates, depending on the level of the analysis, i.e., tokens, AST, CFG, PDG-DFG, or PDG. We make, in particular, the distinction between the `ngrams` and the `value` approach (Section 2.2.2). Specifically, we chose to compare the accuracy of the different modules over their true-positive (TPR) and true-negative rates (TNR), and not AUC [18] or F-measure, so as to evaluate how well they can detect benign *and* malicious inputs. For this reason, AUC and F-measure would be heavily biased by the composition of our test sets (proportion of benign and malicious samples), while we aim at having a more realistic estimation of our modules' accuracy both on benign and malicious samples. Then, we conclude on the predictions' accuracy of JStap's modules, before justifying why they make such accurate predictions.

*3.2.1* `ngrams` *Features.* In the first scenario, we consider the `ngrams` approach. As Figure 4 shows, both the true-positive (TPR) and true-negative rates (TNR) of JStap stay constant across our five analyses. Specifically, the TPR ranges from 98.73% (tokens) to 99.22% (CFG), making the CFG the most reliable malicious JavaScript detector in this configuration. As for the TNR, it ranges from 99.34% (PDG) to 99.62% (AST), meaning that the AST detects benign JavaScript best. In terms of overall detection rate, defined as the proportion of samples correctly classified, the AST performs best with an accuracy around 99.38%, whereas the token-based approach performs worst with a detection rate of 99.08%, while CFG, PDG-DFG, and PDG have similar detection rates between 99.27% and 99.28%.

As mentioned in Section 2.1.1, the lexical level of code abstraction does not use the context (in terms of syntactic structure) in which a given token occurs (e.g., `IfStatement`, `ForStatement`, `VariableDeclaration`), but merely processes JavaScript inputs one word after the other. For example, the following two JavaScript snippets `for(i = 0; i < 5; i++)` and `if(i == 1) j = 2;k--;` are composed of exactly the same tokens, namely `Keyword, Punctuator, Identifier, Punctuator, Numeric, Punctuator, Identifier, Punctuator, Numeric, Punctuator, Identifier, Punctuator, Punctuator`, while performing different actions. On the contrary, the AST-based analysis leverages the JavaScript grammar, which provides more insight than an analysis purely based on tokens, and makes the distinction, e.g., between the previous *for* and *if* constructs, hence a better detection accuracy.

Even though the AST-based approach performs better overall, the CFG, PDG-DFG, and PDG also are reliable. Still, we observe that the AST code representation may be slightly more informative to distinguish benign from malicious JavaScript than the control and data flow. We ran the same experiments where the CFG, PDG-DFG, and PDG only followed the control and/or data dependency, without also traversing the sub-AST corresponding to nodes with such a control/data flow (Section 2.2.1). The TPR stays relatively similar to the results from Figure 4, between 98.87% (PDG-DFG) and 99.33% (CFG), but the TNR decreased between 94.92% (PDG-DFG) and 95.50% (PDG). As the control and data flow are represented only between statement nodes, these nodes are less representative of benign or malicious intent than the AST structure. Specifically, we extracted the five features most representative of malicious or benign intent, for all five `ngrams` modules, according to the corresponding random forest models [47]. Since `Identifier` nodes are *always* part of each of these five most important features, we highlight the importance, in terms of predictions' accuracy, of not just considering statement nodes (which, thereby, do not include `Identifier`). Thus, adding AST information into the CFG, PDG-DFG, and PDG improved their detection rates up to the AST standards. Still, these three approaches may inherently be limited if there is no control or data flow present in the considered files. Out of the 253,216 samples[6] we classified, the CFG could handle on average 231,490.8 of them (91.4%), the PDG-DFG 233,484 (92.2%) and PDG 237,415.4 (93.8%), while the token- and AST-based analyses classified them all. Nevertheless, due to the possibility of combining several modules (Section 3.4), JStap can still classify such samples.

[6] We excluded the samples used to train the model

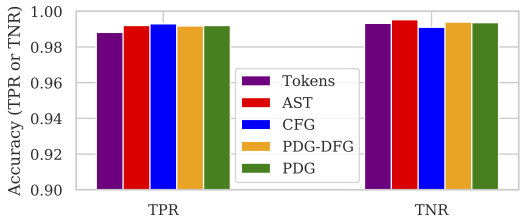


**Figure 4: Accuracy comparison with the `ngrams` approach**

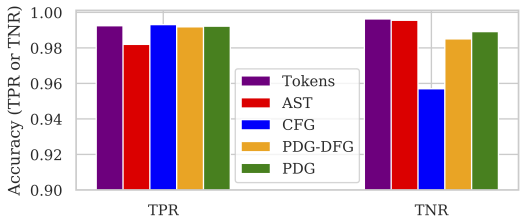


**Figure 5: Accuracy comparison with the `value` approach**

*3.2.2* `value` *Features.* In the second scenario, we consider the `value` approach. Contrary to the previous `ngrams` variant, the TPR and TNR are less constant across our five analyses (Figure 5), given that considering node values increases the number of different features. In particular, the TPR ranges from 98.44% (AST) to 99.23% (CFG). Even though the CFG performs best to detect malicious JavaScript, it performs worse to accurately label benign samples, with a TNR of 95.87% compared to 99.67% for the token-based approach. The overall detection rates across the five analyses also are more sparse than with the `ngrams` approach: from 97.55% for the CFG to 99.44% for the lexical analysis, while AST, PDG-DFG, and PDG have similar detection rates between 98.9% and 99.1%.

This time and contrary to Section 3.2.1, the lexical level of code abstraction leverages context information, since the `value` approach takes the value of each token into consideration (Section 2.2.2). Thus, the *for* and *if* code snippet from the `ngrams` approach in Section 3.2.1 would have a different representation, which contributes–for the reasons mentioned previously–to a better overall detection accuracy. In particular, each token has a value by construction, while only the `Identifier` and `Literal` nodes have one in the graph representation. For this purpose, we mapped the non-identifier and non-literal nodes to their nearest `Identifier`/`Literal` child, if any (on average, only 2.8 samples did not have any `Identifier` nor `Literal` nodes [41], representing 0.001% of our dataset). As a consequence, the same value is used by several nodes and may not always be informative, even though it is significant w.r.t. $\chi^2$ (Section 2.2.3). Besides, the syntactic analyses do not benefit from the JavaScript grammar anymore, as each feature is analyzed independently (compared to an `ngrams` analysis previously). As mentioned in Section 3.2.1, the context information was mainly responsible for the high detection results; therefore, the lexical analysis now performs best. To overcome the lack of context, we tried to combine the current `value` approach with an n-gram analysis, by combining pairs of *(unit, value)* $n$ times, but the TNR dropped to 80%. As a matter of fact, the features got too specific to one file and could not be generalized over the whole dataset anymore. Last but not least, we assume that the CFG approach does not perform as well as the other ones, since benign and malicious developers may tend to use similar names for nodes with control flow, as suggested by the small number of features compared to the AST or PDG (Section 2.2.3).

*3.2.3 Predictions' Accuracy Summary of JStap's Modules.* To sum up, each of the ten JStap modules could correctly classify our JavaScript collection with an accuracy over 97.6%, eight modules of which had an accuracy over 99%. For the `ngrams` approach, the AST performs best, mainly due to the context information brought by the combination of syntactic units. Similarly, the `value` lexical module performs best thanks to the context information brought by the tokens' values. Nevertheless, the CFG, PDG-DFG, and PDG also are very accurate ways of detecting malicious JavaScript and add all the more semantic information into the considered features.

*3.2.4 Most Important Features for Classification.* To accurately distinguish benign from malicious JavaScript inputs over 97.6% of the time, JStap leverages differences between benign and malicious samples at several abstract levels (e.g., AST, CFG). Specifically, using the way in which given lexical and syntactic units are arranged in JavaScript files, along with their frequency, provides valuable insight to capture the salient properties of the code and identify recurrent patterns, specific to malicious or benign intent. For the ten JStap's modules, we extracted the five features most representative of malicious or benign intent, according to the corresponding random forest models [47]. For example, the most representative feature for the `ngrams` approach and for the AST, CFG, PDG-DFG and PDG levels is the following: `[MemberExpression, MemberExpression, Identifier, Identifier]`, which is in line with the tokens' most representative feature, namely `[Punctuator, Identifier, Punctuator, Identifier]`, and represents an element of the form *a.b.c*. We assume that this construct is rather typical of benign samples, such as jQuery, which define several objects with multiple properties, while our malicious files rather store data inside simpler variables or tables. For instance, the fourth most representative feature of the tokens `value` module is the `Punctuator "+"`, which might point to the string splitting/string concatenation data obfuscation form [56], massively used in malicious samples to evade, e.g., signatures-based detection, while benign inputs might rather tend to avoid it, du to the resulting performance downgrade. Similarly, the fifth most important feature of the AST `value` module is `(NewExpression, 'Array')`, which may this time point to the obfuscation technique where strings are fetched from a global array.

All in all, malicious JavaScript samples try to hide their maliciousness by using different obfuscation techniques, which leave specific and recognizable traces in the source code. While benign documents may also be obfuscated to protect code privacy and intellectual property, they have more concerns about the performance, and therefore use different techniques. For this reason, we also assume that malicious code is so different from benign inputs that the natural evolution of the code experienced over a few years should not change the detection results. Therefore, we consider that even if our benign (Table 4) and malicious (Table 3) datasets have been collected over a few years, it does not introduce a bias in

our experiments. Still, we discuss to what extent an attacker could make benign and malicious features similar in Section 4.2.

## 3.3 Analysis of Closely Related Detectors

Several systems already combined differences at a lexical or an AST level with off-the-shelf supervised machine learning algorithms to distinguish malicious from benign JavaScript. In this section, we focus on Cujo [45], JaSt [17] and Zozzle [12], as they are–to the best of our knowledge–the most closely related works to our token- and AST-based approaches. After explaining the better overall detection rates of JStap compared to the previous systems, we focus on combining their predictions.

*3.3.1 Presentation of Cujo, Zozzle and JaSt.* In 2010, Rieck et al. developed Cujo [45], which builds n-gram features from JavaScript lexical units, before using an SVM classifier for an accurate malware detection. As the system is not open source, we contacted the authors who pointed us to the tokenizer they initially used [44] and encouraged us to use the `HashingVectorizer` from Scikit-learn [46] to map the extracted features to a corresponding vector space. In the original implementation, Cujo also leverages an enhanced version of ADSandbox [14], which executes the code associated with a webpage within the JavaScript interpreter SpiderMonkey [40]. We contacted Dewald et al., who informed us that ADSandbox is neither maintained nor running anymore. Since we specifically focus on *static* JavaScript detectors in this paper, we consider only the static part of Cujo. Also, we assume that our reimplementation is functionally equivalent to the original one, and for reproducibility, we make this system publicly available at https://github.com/Aurore54F/lexical-jsdetector.

Curtsinger et al. implemented Zozzle [12], which combines the extraction of features from the AST, as well as their corresponding node value, with a Bayesian classification system to detect malicious JavaScript. We approached the authors and asked for their code or inputs, but did not get any response. Thus, we reimplemented the system with automatic features selection, 1-level features, and naive Bayes, based on the information from the paper. Similarly to Cujo, Zozzle also has a dynamic part, to first hook into the JavaScript engine of a browser to get the deobfuscated version of the code. As previously, we reimplemented the static part of the tool and make it publicly available at https://github.com/Aurore54F/syntactic-jsdetector.

Last but not least, with JaSt, Fass et al. [17] leveraged n-grams from an AST traversal to detect malicious JavaScript. As the system is open source [3], we directly used it for the comparisons.

*3.3.2 Benefits of JStap's Lexical and AST-Based Modules.* Conceptually the `ngrams` module of JStap, working at the tokens level, is identical to Cujo. In contrast, we rely on Esprima for tokenization, use 4-grams instead of 3-grams, do not consider all features, but select them with a $\chi^2$ test, and use a different classifier (random forest). For Zozzle, the `value` module of JStap, working at the AST level is conceptually equivalent. Still, we consider all nodes from the AST (and not only expressions and variable declarations), a different confidence for the $\chi^2$ test, and random forest instead of naive Bayes. As for JaSt, it is conceptually identical to the `ngrams` module of JStap, working at the AST level. Still, we do not simplify the syntactic units returned by the parser but perform a $\chi^2$ test to reduce the size of our feature space.

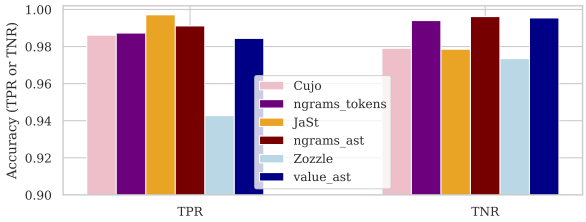


**Figure 6: Accuracy comparison between related work and our improved corresponding implementations**

*3.3.3 Comparison with Cujo, Zozzle and JaSt.* Overall, the three corresponding modules of JStap have a better detection rate compared to Cujo, Zozzle, and JaSt (Figure 6). Specifically, JStap has a higher TPR than Cujo (98.73% compared to 98.61%) and a higher TNR (99.4% and 97.9%), meaning that we classify 2,051 files more accurately than Cujo. Our implementation performs better due to the differences in the implementation mentioned in Section 3.3.2. In particular, 4-grams performed better than 3-grams and random forest better than SVM during our hyper-parameters selection process (Section 2.2.2, Section 2.3). Also, we hypothesize that Cujo performed differently than in its original paper [45] (with a FPR of 2.0E-3% and 5.6% FNR) mainly due to our malicious dataset, comprising 131,448 samples from different sources, compared to 609 for Cujo. This way, our reimplementation recognizes more malicious JavaScript than initially, but to the detriment of benign samples.

We observe a similar trend for Zozzle, which has a significantly lower TPR (94.27% and 98.44%) and TNR (97.35% to 99.54%) than the corresponding JStap's module. As before, we mentioned the differences in the implementation in Section 3.3.2. We also assume that Zozzle performs differently than in its original paper [12] (with a FPR of 3.1E-4% and 9.2% FNR) due to our malicious dataset. Specifically, they considered only 919 malicious samples and clearly stated in 2011 that "relatively few identifier-renaming schemes [were] being employed by attackers", which is not the case anymore, where malicious samples are heavily obfuscated (as observed during the manual analysis from Section 3.1.1). While it might be unfair to consider only the static parts of Cujo and Zozzle to compare them with the corresponding JStap's modules–as their accuracy might also stem from their dynamic components–we are focussing here on comparing several *static* analysis systems, working at different abstract levels (and we did not have the original systems to check the added value, or not, of their dynamic components).

Finally, JaSt has a slightly higher TPR than JStap (99.71% and 99.11%) but in compensation a significantly lower TNR (97.86% and 99.62%), meaning that we classify on average 1,592.8 files more accurately than JaSt. We believe that JStap has a higher overall detection accuracy than JaSt mainly due to us not simplifying the syntactic units returned by the parser. As Fass et al. grouped units with the same abstract meaning they considered, e.g., `ForStatement` and `IfStatement` as a *Statement* node, therefore losing context information, as explained in Section 3.2.1. As we used the open source JaSt version, we assume that our dataset, which is bigger

**Table 5: Analysis of the detection accuracy when Cujo, JaSt and Zozzle made different predictions**

| Approach | TPR | TNR | Accuracy | Approach | TPR | TNR | Accuracy |
|---|---|---|---|---|---|---|---|
| ngrams_tokens | 0.87 | 0.94 | 0.91 | value_tokens | 0.89 | 0.96 | 0.93 |
| ngrams_ast | 0.89 | 0.96 | 0.93 | value_ast | 0.84 | 0.96 | 0.9 |
| ngrams_cfg | 0.91 | 0.93 | 0.92 | value_cfg | 0.9 | 0.7 | 0.81 |
| ngrams_pdg-dfg | 0.9 | 0.92 | 0.91 | value_pdg-dfg | 0.88 | 0.88 | 0.89 |
| ngrams_pdg | 0.9 | 0.93 | 0.92 | value_pdg | 0.88 | 0.9 | 0.89 |
| Cujo | 0.78 | 0.64 | 0.71 | Zozzle | 0.14 | 0.66 | 0.42 |
| JaSt | 0.97 | 0.73 | 0.84 | | | | |

and contains more diverse JavaScript than in the original paper [17], is responsible for the different rates we got compared to the paper's (5.2E-3 FPR and 5.4E-3 FNR), which is in line with the assumptions we made for Cujo and Zozzle.

*3.3.4 Combination of Related Work Predictions.* Next, we studied the detection accuracy of JStap's different modules on samples for which Cujo, Zozzle and JaSt made different predictions. Due to their different classification results, these samples may be trying to evade detection. Specifically, the three related work classifiers considered made different predictions for 17,178.6 samples (6.78% of our dataset[7]), 7,943.4 of which are malicious.

Table 5 presents the detection accuracy of all JStap's modules, and of Cujo, Zozzle and JaSt, on such samples. First, our `ngrams` approach at the tokens level performs better than Cujo also in this configuration, with both a significantly higher TPR (87% compared to 78%) and TNR (94% and 64%). Similarly, we outperform Zozzle by correctly classifying over twice as many samples with JStap's `value` AST-based module (overall detection accuracy of 90% compared to 42%). Still, these results have to be taken with a grain of salt, as we tested the classifiers on samples likely to try to evade detection. As a matter of fact, in Section 3.3.3, Zozzle did not perform as well as Cujo and JaSt. In particular, it reported almost 7,000 false-negatives (FNR of 5.7%) compared to 348 for JaSt and 1,675 for Cujo. Therefore, and out of the 7,943.4 malicious samples considered here, at least 5,300 are initial false negatives from Zozzle, meaning that its TPR could not be over 33%. Finally, and as previously, JaSt has a higher TPR than our `ngrams` AST-based approach (97% compared to 89%), but at the same time significantly fewer true-negatives (73% compared to 96%), meaning that JStap has a higher overall detection accuracy, classifying 1,550 files more accurately than JaSt.

As for the remaining JStap's modules, they are also impacted by these samples likely to be evasive, with a mean accuracy between 81% (`value` CFG, otherwise from 89%) and 93% (`value` tokens), compared to over 97.55% (`value` CFG) and up to 99.44% (`value` tokens) in Section 3.2 on a standard dataset. Still, all JStap's modules significantly outperform Cujo, Zozzle and JaSt.

## 3.4 Combining Modules for a Higher Accuracy

JStap is a modular JavaScript static classification system for which the user can choose the type of analysis (`ngrams` or `value`), as well as its level (tokens, AST, CFG, PDG-DFG and PDG). Even though all approaches (except `value` CFG) have an overall detection accuracy between 98.9% and 99.44% (Section 3.2), thereby outperforming related-work detectors trained and tested on the same datasets (Section 3.3), they can still be combined for an even better detection rate. In the following sections, we discuss the JStap's modules we combined, as well as the detection accuracy on the resulting combination, using majority predictions voting, before focussing on the detectors' confidence for a given prediction.

*3.4.1 Selection of JStap's Modules for Predictions Combination.* For the combination process, we chose the `value` token- and `ngrams` AST-based approaches, which perform particularly well (Section 3.2) and use different features that do not overlap. As a matter of fact, the former leverages the lexical structure of a JavaScript file and combines each extracted token with its corresponding value, while the latter rests upon the AST traversal and an n-gram combination of the traversed nodes, for an accurate malicious JavaScript detection. As we need an odd detectors' number to perform majority voting, we selected a third one. The PDG `value` approach complements the previous two systems, as it also uses new features, which do not overlap with the previous ones. On the contrary, choosing the CFG, PDG-DFG or PDG `ngrams` option would have overlapped with the AST `ngrams` approach, while the PDG `value` module has different features, due to the consideration of the nodes' value. Also, we wanted to strengthen our system with *both* control and data flow information, hence the choice for the PDG.

*3.4.2 Predictions With Majority Voting.* We perform a combination of the three selected systems (`ngrams` AST, `value` tokens and `value` PDG), by choosing the prediction with the most votes, for a given JavaScript input. Such a combination presents both a high TPR of 99.2% and a TNR of 99.7%, representing an accuracy of 99.46%. Still, when we considered each module separately in Section 3.2, we had an approaching accuracy for the `value` tokens approach (best module) with a detection rate of 99.44%. This means that combining modules leads to a detection of 36 additional samples (0.015% of our dataset), which we do not see as a major improvement.

Nevertheless, we also leveraged the combination of these three modules to classify the 17,178.6 samples for which Cujo, Zozzle and JaSt made different predictions (Section 3.3.4). This time, we retain an accuracy of 93.47%, which is, again, better than the `ngrams` AST-based and `value` token-based approaches (Table 5), which performed best in this configuration. In particular, we detect on average 47.9 extra samples with the combination of modules than with the `value` tokens approach, which is 1.3 times more samples than in our standard dataset, where we have almost 15 times more samples. Therefore, combining modules brings a real added value when classifying samples likely to be evasive. Similarly, this combination process also recognizes 121.9 more samples than the `ngrams` AST approach. In particular, the `value` token-based approach correctly classified 74 extra samples compared to the `ngrams` AST variant (0.43% of our evasive dataset), while only classifying 0.07% more of our standard dataset, meaning that the difference in terms of detection rate between these two modules tends to increase on evasive samples. Therefore, combining JStap modules always perform better than each module separately, in particular on evasive samples. In the case of such evasive samples, some modules may struggle to classify them correctly, while combining modules significantly limits the proportion of samples evading our system.

[7]We consider here only the samples, which are not in the model, thus 253,216

**Table 6: JSTAP's modules predictions combination**

| Approach | TPR (%) | TNR (%) | Accuracy (%) |
|---|---|---|---|
| Same predictions from the 3 modules | 99.55 | 99.9 | 99.73 |
| Majority voting on remaining (likely evasive) | 86.9 | 98.16 | 96.02 |

*3.4.3 Confidence of the Combined Predictions.* Last but not least, we focus on the JavaScript samples for which all three of our combined modules made the same predictions, and on the contrary, those for which they had different classification results. On average, `ngrams` AST, `value` tokens and `value` PDG labeled 234,875.8 JavaScript inputs the same way (92.76%). On these samples specifically, they have both an extremely high TPR of 99.55% and TNR of 99.9% (standing for an overall detection accuracy of 99.73%, Table 6), meaning that their predictions are more trustworthy than on the whole dataset, which we expected since the modules perform better combined than separately. Finally, we classified the remaining 18,340.2 samples (over 80% of which are benign), which can also be seen as samples that may try to evade detection (similarly to Section 3.3.4). Still, we retain a high TNR of 98.16% with the majority voting system (Table 6) on such samples, meaning that we accurately detect malicious JavaScript over 98% of the time. In turn, we have a TPR of 86.9%. All in all, we retain over 96% accuracy on samples for which our combined detectors predict conflicting labels, which we consider to still be relatively high.[8] Nevertheless, the overall detection accuracy of JSTAP should not be evaluated only on such samples, but on our whole dataset (also containing these samples), where we retain an accuracy of almost 99.5% (Section 3.4.2).

## 3.5 Run-Time Performance

We tested JSTAP's run-time performance on several CPUs, each Intel(R) Xeon(R) Platinum 8160 CPU at 2.10GHz. Even though we parallelized our implementation to generate the results for this paper, the run-time of our system was tested on one CPU only. Table 7 presents the average, median, minimum, and maximum duration to generate each of our considered code representations. The tokenizing and parsing with Esprima [23] are relatively fast, with an average time of respectively 17 and 35 ms per file. The most time-consuming operation is the PDG generation, which highly depends on the AST size, since we have to traverse it, pushing and popping the variables encountered all the way down to the leaves. Once we generated all the PDGs of all the files from our dataset, we stored them so as not to have to produce them for each module *again*. Therefore, we did not take into consideration the PDGs (and tokens, for comparison purpose) generation time in Table 8.

This table presents the duration times to generate the features considered by each module, for *one* file. The last two columns stand for the run-time to leverage the previous features to build a model (averaged for one file) and to classify one unknown input. In overall, more complicated code representations (e.g., PDG, CFG compared to tokens or AST) lead to a higher overhead, since we follow more edges in the graphs and consider more features. The `value` approach also is slower than the `ngrams` one, as we fetch a value for each unit, thereby traversing sub-ASTs down to the leaves.

Specifically, classifying a JavaScript sample with the `ngrams` tokens module takes on average 19 ms for the features generation (including tokens production) and, 0.71 ms for the classification. For the `value` AST-based approach, it takes 112 ms to produce features, with an AST previously generated, and 1.4 ms for the classification. Based on the number of features each module considers (Table 2) and an average size of 23 KB per file, we consider the overhead to be reasonable. Also, JSTAP is fully parallelized to leverage all available CPU cores for a faster analysis for a deployment in the wild.

[8] We further discuss this point in Section 4.3

**Table 7: Run-time to generate JSTAP's code representations**

| Code representations | Mean (ms) | Median (ms) | Min (ms) | Max (s) |
|---|---|---|---|---|
| Tokenizer | 16.894 | 9.0 | 0.0 | 0.175 |
| Parser | 34.921 | 19.0 | 1.0 | 0.311 |
| AST from parser | 97.711 | 11.487 | 0.038 | 4.103 |
| CFG from AST | 39.085 | 4.635 | 0.004 | 1.114 |
| PDG from CFG | 369.49 | 8.71 | 0.125 | 27.27 |

**Table 8: JSTAP's run-time per module**

| Modules | Mean (ms) | Median (ms) | Min (ms) | Max (s) | Learner (ms) | Classifier (ms) |
|---|---|---|---|---|---|---|
| `ngrams_tokens` | 2.344 | 1.42 | 0.65 | 0.203 | 0.162 | 0.715 |
| `ngrams_ast` | 9.683 | 2.592 | 0.635 | 0.722 | 0.19 | 1.427 |
| `ngrams_cfg` | 18.288 | 3.781 | 0.762 | 0.778 | 0.252 | 1.667 |
| `ngrams_pdg-dfg` | 19.412 | 3.736 | 0.723 | 1.111 | 0.228 | 2.685 |
| `ngrams_pdg` | 34.745 | 5.544 | 0.799 | 1.243 | 0.241 | 2.763 |
| `value_tokens` | 13.251 | 3.743 | 0.947 | 1.397 | 0.187 | 1.127 |
| `value_ast` | 112.036 | 11.131 | 1.085 | 86.619 | 0.227 | 1.37 |
| `value_cfg` | 129.77 | 12.138 | 0.875 | 207.255 | 0.187 | 1.174 |
| `value_pdg-dfg` | 101.83 | 9.707 | 0.99 | 107.432 | 0.195 | 1.279 |
| `value_pdg` | 216.895 | 21.44 | 1.003 | 247.253 | 0.173 | 1.311 |

# 4 DISCUSSION

In this section, we first analyze the limitations JSTAP might have, focussing on the *static* analysis of JavaScript. We then discuss techniques that might evade our system in theory but are not specifically used in practice. Finally, we introduce new strategies to classify more JavaScript instances accurately.

## 4.1 Limitations

JSTAP is based on a static analysis of JavaScript to build both the control and data flow of a given script. Therefore, it provides a complete code coverage based on the proportion of source code analyzed. In turn, it is subject to the traditional flaws induced by the high dynamic of the language [1, 19, 25, 26]. Specifically, JavaScript models inheritance with prototype chaining [38], where properties can be added or removed during the execution, and property names may be dynamically computed. Also, JavaScript can generate code at run-time, e.g., with the *eval* function, a dynamically constructed string can be interpreted as a program fragment and executed in the current scope. Still, to partially mitigate this limitation, we automatically (and statically) rewrote *eval* calls to a string into the corresponding code that would have been generated dynamically. This way, we increased JSTAP's code coverage by not merely considering a `CallExpression` node anymore, but the actual content of the string along with possible control/data flow (Section 3.1.1).

Still, our approach would not work on *eval* calls with a variable as parameter or nested *evals*. Nevertheless, JStap aims at working directly at the *code level* to detect malicious JavaScript, by analyzing the traces left in the syntax of malicious files, e.g., due to the specific malicious obfuscation techniques used by attackers. As long as all the code is not dynamically generated, which we encountered only for conditional compilation and solved by automatically generating the actual code (Section 3.1.1), JStap will leverage the existing code to classify the JavaScript inputs considered.

## 4.2 Evasion Techniques

All learning-based malware detection systems will fail to detect some attacks, e.g., if the considered malicious instances do not contain any features present in the training set, given that machine learning relies on statistical assumptions about the distribution of the training data to classify unknown inputs [4]. Therefore, adversaries could modify their malicious samples by adding benign features (not to mention copy their malicious file into a significantly bigger benign one), to statistically increase the proportion of benign features in a malicious file and have a more benign-looking structure [28]. This way, they would mislead our detector into classifying the considered sample as benign. Even though related work effectively added, deleted or replaced specific features of a given file [13, 22, 51, 55] and injected malicious content into bigger benign samples [37], we observed very few such samples. Specifically and out of the 19,942 malicious samples we manually analyzed, we found such evasion techniques less than 50 times. For this reason, we believe that malicious actors rather use obfuscation to hide their attack. The rarity of such malicious samples could also be a limitation of our dataset. In this case, it would mean that our malware providers did not detect these samples, which should not happen after a dynamic analysis.

Another class of attacks against JStap are samples with the same structure but different ground truths. In particular, Fass et al. showed with HideNoSeek [16] that malicious samples can be rewritten, so that they have exactly the same AST as an existing benign file. Yet, because their variables have different values, they perform distinct actions after execution. While this attack would by construction impact our token- and AST-based modules, we believe that our PDG-DFG module might be able to recognize such samples, because of the specific changes induced by the attack at the data flow level.

Nevertheless, as presented in Section 3.4, our system makes more accurate predictions when we combine the labels given by several modules. This fact also holds for samples which might be trying to evade detection (for example when several modules classified them differently). In this specific case, we suggest to use JStap as a pre-filtering system before sending samples for which the modules predicted conflicting labels to more costly dynamic components (Section 4.3). This way, JStap is more resilient to evasive samples than any of its modules alone.

## 4.3 Improving the Detection With Pre-Filtering Layers

To detect malicious JavaScript, we specifically chose a static approach, which is by construction fast, while still making accurate predictions. Dynamic detectors may perform better, in particular, if they visit all possible execution paths [30, 31], but at the same time, they are more costly, e.g., they require specific instrumentations, they introduce overhead inherently depending on the code's execution, also the necessary amount of time to observe a malicious behavior is not defined [51]. Besides, such analyses can be defeated if the sample notices that it is running in a sandboxed environment [6, 7]. To this end and to maximize the detection accuracy while at the same time minimizing the run-time performance, we rather envision that JStap could be used to pre-filter JavaScript samples, sending, e.g., only those with conflicting labels to much slower dynamic components. In the context of Section 3.4.3, the 18,340.2 samples (on average) for which `ngrams` AST, `value` tokens and `value` PDG made different predictions, could be sent to such components. For this purpose, we could also consider a second pre-filtering layer to limit the number of inputs to be executed in a sandboxed environment. Similar to the combination of Cujo [45], JaSt [17] and Zozzle [12], we combined `ngrams` tokens, `ngrams` AST and `value` AST to classify the previous 18,340.2 samples. These three detectors predicted the same labels for 16,469.4 of them, with an accuracy over 99%, meaning that only the resulting 1,870.8 could be sent to dynamic components. Naturally, it depends on the reliability a user would like to have. Still, out of our 253,216 sample set, JStap correctly classified 234,875.8 instances (92.76%) with an accuracy of 99.73% in a first pre-filtering step (Section 3.4.3). Then, it correctly labeled 16,469.4 additional samples (6.5% of the initial dataset) with an accuracy over 99% in a second pre-filtering step, meaning that only 0.74% of the original dataset would be outsourced to more costly components, while having a detection accuracy significantly over 99% for the majority of the considered samples.

# 5 RELATED WORK

JStap is a modular malicious JavaScript detector, which goes beyond leveraging purely lexical and syntactic information for an accurate classification. As a matter of fact, we also consider semantic information, such as control and/or data flow. Also, we envision to combine the predictions of several JStap modules and send only samples with conflicting labels to more costly dynamic components.

## 5.1 Lexical Analysis

In the literature, several approaches have been proposed to detect malicious JavaScript inputs by means of lexical analysis. Specifically, Rieck et al. developed Cujo [45], which combines n-gram features from JavaScript lexical units with dynamic code features, before using an SVM classifier for an accurate malware detection. Similarly, Laskov et al. implemented PJScan [34], which combines n-grams built upon lexical features with a model of normality to detect malicious PDF documents. Stock et al. also used tokens to implement Kizzle [49], a malware signature compiler focussing on exploit kits. Beyond JavaScript detection, Kar et al. [29] leveraged a lexical analysis to detect SQL injections.

## 5.2 Syntactic Units Extracted From the AST

To analyze JavaScript inputs, other systems leverage the AST. In particular, Curtsinger et al. implemented Zozzle [12], combining

the extraction of features from the AST and their corresponding node value, with a Bayesian classifier to detect malicious JavaScript. For this purpose, Fass et al. [17] proposed JaSt, which leverages n-gram features from an AST traversal. Beyond a purely static analysis, Kapravelos et al. [28] presented *Revolver*, which uses the AST to identify similarities between JavaScript inputs. If its dynamic detector labels similar files differently, they are reported as evasive.

### 5.3 PDG for Security Analysis

JStap can also be compared to systems using ASTs or PDGs for vulnerability detections. For example, Yamaguchi et al. extrapolated known vulnerabilities using structural patterns from the AST to find similar flaws in other projects [58]. They also leveraged the combination of AST, CFG, and PDG to mine more source code [57]. This combination was also used by Backes et al. to identify different types of web application vulnerabilities [5].

### 5.4 Dynamic Detectors

Lexical and syntactic analyses aside, additional approaches may be effective to detect malicious JavaScript. With JSAND, Cova et al. [11] combine anomaly detection with emulation to identify malicious JavaScript by emulating its behavior and comparing it to benign established profiles. Kolbitsch et al. implemented Rozzle [31], which imitates multiple browser and environment configurations to explore various execution paths to detect malicious JavaScript dynamically. Similarly, J-Force [30] also forces the JavaScript execution engine to test all execution paths systematically. We envision that JStap could be combined with such dynamic detectors to classify samples that have a conflicting label. Regarding filtering systems, EvilSeed from Invernizzi et al. [24] searches the web for pages likely to be malicious, by similarity detection and relation to an initial set of malicious seeds. Canali et al. [9] also worked on a faster collection of malicious web pages with Prophiler, which discards benign pages based on HTML-derived lexical features, the JavaScript AST, and an URL-based analysis.

## 6 CONCLUSION

Attackers tend to obfuscate malicious JavaScript to hinder the analysis and the creation of signatures. Still, these specific evasion techniques tend to leave recurrent traces in the source code of malware, thus contributing to their detection by systems leveraging features from the source code (at a lexical or syntactic level). Also, and due to their usage of static features, such systems cannot be foiled by malware variants whose behavior are time- or environment-dependent. In this paper, we proposed and built JStap, a modular system that can work at a lexical level, but also on the AST, CFG and PDG representations, to automatically, statically and accurately detect malicious JavaScript. In particular, we leverage random forest classifiers, in combination with different semantic and syntactic representations of the JavaScript samples, to classify them.

We evaluated our system on an extensive, up-to-date, and balanced JavaScript dataset of both benign and malicious samples, totaling over 270,000 unique scripts. In practice, our different modules are all, and independently, very accurate, the best one yielding accurate predictions 99.44% of the time, with a low false-positive rate of 0.33% and 0.8% false-negatives. For JStap to make more accurate predictions, we combined the predictions of three modules working at different levels, to leverage different aspects of the samples for classification. To this end, we envision that this combination of modules could be used as a pre-filtering step, before sending samples with conflicting labels to more costly follow-up analysis. In this scenario, we could classify almost 93% of our dataset with a detection accuracy of 99.73% and 6.5% of our dataset with an accuracy still over 99%, meaning that less than 1% of our samples would require additional scrutiny.

## ACKNOWLEDGMENTS

We would like to thank the German Federal Office for Information Security (BSI), VirusTotal and Kafeine DNC, which provided us with malicious JavaScript samples for our experiments. We would also like to thank the anonymous reviewers of this paper for their well-appreciated feedback. Special thanks also go to Dennis Salzmann for having reimplemented Cujo. Furthermore, we gratefully acknowledge the help in terms of feedback or inspiring discussion of Christian Rossow, Konrad Rieck, Marius Steffens, and Pierre Laperdrix. 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] Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*.
[2] AtomEditor. [n.d.]. Atom: a hackable text editor for the 21st Century. In: https://atom.io. Accessed on 2019-06-05.
[3] Aurore54F. [n.d.]. JaSt - JS AST-Based Analysis. In: https://github.com/Aurore54F/JaSt. Accessed on 2019-05-24.
[4] Michael Backes and Mohammad Nauman. 2017. LUNA: Quantifying and Leveraging Uncertainty in Android Malware Analysis through Bayesian Machine Learning. In *Euro S&P*.
[5] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *S&P*.
[6] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2010. Efficient Detection of Split Personalities in Malware. In *NDSS*.
[7] Michael Brengel, Michael Backes, and Christian Rossow. 2016. Detecting Hardware-Assisted Virtualization. In *DIMVA*.
[8] BSI. [n.d.]. German Federal Office for Information Security (BSI). In: https://www.bsi.bund.de/EN. Accessed on 2019-07-18.
[9] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. 2011. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In *International Conference on World Wide Web (WWW)*.
[10] The SciPy community. [n.d.]. scipy.sparse.csr_matrix. In: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html#scipy.sparse.csr_matrix. Accessed on 2019-06-05.
[11] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *International Conference on World Wide Web (WWW)*.
[12] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2011. Zozzle: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security*.
[13] Hung Dang, Yue Huang, and Ee-Chien Chang. 2017. Evading Classifiers by Morphing in the Dark. In *CCS*.
[14] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. 2010. ADSandbox: Sandboxing JavaScript to Fight Malicious Websites. In *ACM Symposium on Applied Computing (SAC)*.
[15] Ecma International. [n.d.]. ECMAScript 2018 Language Specification (ECMA-262, 9th edition, June 2018). In: https://www.ecma-international.org/ecma-262/9.0. Accessed on 2019-06-04.
[16] Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *CCS*.

[17] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *DIMVA*.

[18] Tom Fawcett. 2006. An Introduction to ROC Analysis. *Pattern Recogn. Lett.* (2006).

[19] Asger Feldthaus and Anders Møller. 2013. Semi-Automatic Rename Refactoring for JavaScript. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[20] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1987).

[21] GeeksOnSecurity. [n.d.]. Malicious Javascript Dataset. In: https://github.com/geeksonsecurity/js-malicious-dataset. Accessed on 2019-04-22.

[22] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial Perturbations Against Deep Neural Networks for Malware Classification. In *European Symposium on Research in Computer Security*.

[23] Ariya Hidayat. [n.d.]. ECMAScript Parsing Infrastructure for Multipurpose Analysis. In: http://esprima.org. Accessed on 2019-06-04.

[24] Luca Invernizzi, Stefano Benvenuti, Marco Cova, Paolo Milani Comparetti, Christopher Kruegel, and Giovanni Vigna. 2012. EvilSeed: A Guided Approach to Finding Malicious Web Pages. In *S&P*.

[25] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[26] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *International Symposium on Static Analysis (SAS)*.

[27] Kafeine. [n.d.]. MDNC - Malware don't need coffee. In: https://malware.dontneedcoffee.com. Accessed on 2019-04-22.

[28] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, and Christopher Krügel and Giovanni Vigna. 2013. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *USENIX Security*.

[29] Debabrata Kar, Suvasini Panigrahi, and Srikanth Sundararajan. 2016. SQLiGot: Detecting SQL Injections Attacks using Graph of Tokens and SVM. In *Computers & Security*.

[30] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-Force: Forced Execution on JavaScript. In *WWW*.

[31] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking Internet Malware. In *S&P*.

[32] J. Zico Kolter and Marcus A. Maloof. 2006. Learning to Detect and Classify Malicious Executables in the Wild. In *Journal of Machine Learning Research*.

[33] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense. In *CSS*.

[34] Pavel Laskov and Nedim Šrndić. 2011. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *ACSAC*.

[35] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *NDSS*.

[36] Peter Likarish, Eunjin Jung, and Insoon Jo. 2009. Obfuscated Malicious JavaScript Detection Using Classification Techniques. In *International Conference on Malicious and Unwanted Software (MALWARE)*.

[37] Davide Maiorca, Igino Corona, and Giorgio Giacinto. 2013. Looking at the Bag is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection. In *ASIACCS*.

[38] Mozilla Developer Network. [n.d.]. Inheritance and the prototype chain. In: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain. Accessed on 2019-06-04.

[39] Mozilla Developer Network. [n.d.]. JavaScript Conditional Compilation: cc_on. In: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Microsoft_Extensions/at-cc-on. Accessed on 2019-06-04.

[40] Mozilla Developer Network. [n.d.]. SpiderMonkey. In: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. Accessed on 2019-09-10.

[41] Patricio Palladino. [n.d.]. Non alphanumeric JavaScript. In: http://patriciopalladino.com/blog/2012/08/09/non-alphanumeric-javascript.html. Accessed on 2019-06-10.

[42] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *USENIX Security Symposium*.

[43] Hynek Petrak. [n.d.]. Javascript Malware Collection. In: https://github.com/HynekPetrak/javascript-malware-collection. Accessed on 2019-04-22.

[44] Konrad Rieck. [n.d.]. Jassi: A Simple and Robust JavaScript Lexer. In: https://github.com/rieck/jassi. Accessed on 2019-05-24.

[45] Konrad Rieck, Tammo Krueger, and Andreas Dewald. 2010. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In *ACSAC*.

[46] scikit-learn developers. [n.d.]. Scikit-learn: HashingVectorizer. In: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html. Accessed on 2019-06-05.

[47] scikit-learn developers. [n.d.]. sklearn.ensemble.RandomForestClassifier. In: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.feature_importances_. Accessed on 2019-06-12.

[48] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *The World Wide Web Conference (WWW)*.

[49] Ben Stock, Benjamin Livshits, and Benjamin Zorn. 2016. Kizzle: A Signature Compiler for Detecting Exploit Kits. In *Dependable Systems and Networks (DSN)*.

[50] VirusTotal. [n.d.]. VirusTotal - Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community. In: https://www.virustotal.com. Accessed on 2019-04-22.

[51] Nedim Šrndić and Pavel Laskov. 2013. Detection of Malicious PDF Files Based on Hierarchical Document Structure. In *NDSS*.

[52] Edwin B. Wilson and Margaret M. Hilferty. 1931. The Distribution of Chi-Squared. *National Academy of Sciences of the United States of America* (1931).

[53] Wilco Wisse and Cor J. Veenman. 2015. Scripting DNA: Identifying the JavaScript Programmer. In *Digital Investigation*.

[54] Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. 2013. A Close Look on n-Grams in Intrusion Detection: Anomaly Detection vs. Classification. In *ACM workshop on Artificial intelligence and security (AISec)*.

[55] Weilin Xu, Yanjun Qi, and David Evans. 2016. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In *NDSS*.

[56] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study. In *International Conference on Malicious and Unwanted Software (MALWARE)*.

[57] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *S&P*.

[58] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *ACSAC*.