JSRevealer: A Robust Malicious JavaScript Detector against Obfuscation

Kunlun Ren*, Weizhong Qiang*§, Yueming Wu‡, Yi Zhou*, Deqing Zou*§, Hai Jin†§
* National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab
Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security
School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China
† National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
‡Nanyang Technological University, Singapore
§ Jinyinhu Laboratory, Wuhan, China
Email: {kunlunren, wzqiang, yi_zhou, deqingzou, hjin}@hust.edu.cn, wuyueming21@gmail.com

***Abstract*—Due to the convenience and popularity of Web applications, they have become a prime target for attackers. As the main programming language for Web applications, many methods have been proposed for detecting malicious JavaScript, among which static analysis-based methods play an important role because of their high effectiveness and efficiency. However, obfuscation techniques are commonly used in JavaScript, which makes the features extracted by static analysis contain many useless and disguised features, leading to many false positives and false negatives in detection results. In this paper, we propose a novel method to find out the essential features related to the semantics of JavaScript code. Specifically, we develop *JSRevealer*, a robust, effective, scalable, and interpretable detector for malicious JavaScript. To test the capabilities of *JSRevealer*, we conduct comparative experiments with four other state-of-the-art malicious JavaScript detection tools. The experimental results show that *JSRevealer* has an average F1 of 84.8% on the data obfuscated by different obfuscators, which is 21.6%, 22.3%, 18.7%, and 22.9% higher than the tools *CUJO*, *ZOZZLE*, *JAST*, and *JSTAP*, respectively. Moreover, the detection results of *JSRevealer* can be interpreted, which can provide meaningful insights for further security research.**

***Index Terms*—Web Security, JavaScript Obfuscation, Malicious JavaScript, Robustness.**

I. Introduction

The Web is still the dominant software platform and the primary tool for billions of users worldwide to interact with the Internet. Since it is so prevalent, it naturally attracts attackers to leverage it for their own illegal purposes. Particularly, JavaScript is abused to perform various attacks [1]–[4] such as drive-by malware, malicious code targeting browser vulnerability, malicious browser extension, cryptojacking, browser-based phishing, and web skimming. Moreover, browser fingerprinting and tracking also have the potential to violate users' sensitive information. In fact, these attacks can have a very widespread impact. For example, cryptojacking, a new mechanism for mining cryptocurrency without the users' consent, is estimated to affect over 10 million web users every month [5]. Due to the widespread and harmful nature of these attacks, detecting malicious JavaScript code is a critical matter.

There have been proposed many malicious JavaScript detection methods, which can be classified into two main categories: static analysis based and dynamic analysis based. Dynamic analysis can reveal the behavior of malicious code more clearly [1], [6]–[10], but the identifiable nature of the analysis environment results in malicious code being able to evade detection through inspection of the environment. Moreover, the overhead of dynamic analysis is too high. So it is not suitable for large-scale analysis. Static analysis, on the other hand, consumes fewer resources, is simple and fast to detect malicious code, and is more cost-effective [11]–[26]. With the emergence of more and more new malicious JavaScript code, expensive manual analysis has prompted static detection tools to leverage machine learning techniques, through which good results are achieved [11], [12], [20]–[26]. However, static analysis is susceptible to obfuscation. JavaScript obfuscation is a series of code transformations that transform easy-to-read JavaScript code into a version that is extremely difficult to understand and reverse engineer. Obfuscation is now commonly used in JavaScript, both benign and malicious scripts [27]. In general, benign scripts use obfuscation to protect intellectual property, while malicious code uses obfuscation to evade detection. Previous works [12], [20], [21] briefly discussed the effects of obfuscation on their methods, but no detailed and quantitative experiments were done to analyze the effects of obfuscation on their methods. Whereas, from the features they extract, it is inevitable that these methods are significantly affected by obfuscation.

Corresponding Author: Weizhong Qiang. This work is supported in part by the National Natural Science Foundation of China (Grant No. 62272181).

In this paper, we propose a novel malicious JavaScript detection method that is robust against obfuscation. We dive into the nature of benign and malicious based on the idea of splitting and regrouping. The code is divided into fine-grained representations, and then the most important and essential features are obtained by regrouping these representations. This poses two challenges:

1) *How to divide the code into fine-grained representations and keep the complete semantic information for each representation?*
2) *How to remove confusing features from these representations and obtain the most important and essential features?*

To address the first challenge, we first represent the code of JavaScript as an *abstract syntax tree* (AST) and then add data flow information to the AST to supplement the semantic information of the code. We call this representation **enhanced AST**. To obtain more fine-grained representations, we traverse the enhanced AST and divide it into a number of paths. After collecting the paths, we perform path embedding to obtain continuous distribution vector representations and weights of them. In this way, the original AST can be transformed into more fine-grained representations with code semantics.

To tackle the second challenge, given the vectors of all paths, we first perform outlier detection to remove the ones that are irrelevant or even interfere with the behavior analysis of the code. The output of this phase is some essential path vectors. After obtaining these vectors, we cluster the vectors of benign and malicious samples separately and then remove the clusters with high similarity. The remaining clusters are used as features. In this way, the behaviors of JavaScript are abstracted into more representative and robust features, which are less susceptible to obfuscation.

We develop a prototype system namely *JSRevealer* and evaluate it on a massive dataset including over 40,000 malicious samples and over 210,000 benign samples. Specifically, we conduct a comprehensive evaluation of *JSRevealer* in four aspects: effectiveness, robustness, interpretability, and scalability. In terms of effectiveness, *JSRevealer* achieves an F1 score of 99.4% on unobfuscated samples. As for robustness, when detecting samples obfuscated by four commonly used obfuscation tools (*JavaScript-Obfuscator* [28], *Jfogs* [29], *JSObfu* [30], and *Jshaman* [31]), the average F1 of *JSRevealer* is 84.8%, while the other four comparative tools (*e.g., CUJO* [11], *ZOZZLE* [12], *JAST* [20], and *JSTAP* [21]) are only 63.2%, 62.5%, 66.1%, and 61.9%, respectively. Regarding interpretability, after analyzing the five most important features, we find that benign samples focus on the implementation of functionalities, while malicious samples focus on the manipulation of data. In terms of scalability, *JSRevealer* requires an average of around 0.6 seconds to detect a JavaScript file. Such result suggests that *JSRevealer* is suitable for large-scale detection of malicious JavaScript.

In summary, the main contributions of this paper are as follows:

- We propose *JSRevealer*, a novel approach using the idea of splitting and regrouping to transform the JavaScript code into more abstract and essential features. Based on such features, *JSRevealer* is the first malicious JavaScript detector that can resist obfuscation and have interpretability.
- We perform comprehensive evaluations of *JSRevealer*. The experimental results show that *JSRevealer* can achieve better performance than the other four state-of-the-art malicious JavaScript detection tools (*e.g., CUJO* [11], *ZOZZLE* [12], *JAST* [20], and *JSTAP* [21]).

The remainder of this paper is organized as follows. Section II introduces the background of malicious JavaScript and obfuscation techniques. Section III describes our approach in detail. Section IV presents our experimental results. Section V discusses the inspiration and some limitations of our approach. Section VI introduces some related works. Section VII concludes this paper.

II. Background

A. Malicious JavaScript

The Web is one of the most popular platforms for billions of people to interact with the Internet, and JavaScript is one of its core technologies used to provide interactive features. JavaScript is used by more than 90% of websites according to statistics [32]. Since JavaScript is so pervasive, it provides attackers with the opportunity to conduct high-impact attacks by exploiting the huge attack surface offered by JavaScript. Just like many other web technologies, JavaScript has long been used by attackers to execute attacks. The antivirus industry and academia have discovered many kinds of attacks using JavaScript [8]. At this moment, many users are still suffering threats from a large number of malicious JavaScript codes. Below are some of the most prevalent attacks using JavaScript today:

- *Browser Exploit*: Browser exploit leverages web browser vulnerabilities to breach web browser security. The web browser is the key component of the computer and the main entry point for users to access the Web. If a user visits a malicious website with a vulnerable browser, browser exploit can use the vulnerability to perform various attacks such as personal information stealing and malware spreading.
- *Web Skimming*: Web Skimming extracts data from a filled HTML form that the user has completed by injecting malicious code into Web sites. This poses an enormous threat to users' important information such as bank card information.
- *Cryptojacking*: Cryptojacking is the unauthorized use of other people's computational resources to mine cryptocurrency. Many websites are infected with cryptojackers today. Malicious JavaScript hosts unethical and insecure websites and steals CPU computing resources from website visitors to mine cryptocurrencies.

Here we list only a small number of attack types using JavaScript. From these attacks, we can get a glimpse of the

threat posed by malicious JavaScript, which can cause harm to users' devices, information, etc. Moreover, now attackers are using obfuscation techniques to hide malicious code and making it much more difficult to analyze and detect.

B. Obfuscation Technique

There are various obfuscation techniques applied in the wild. Below are several common obfuscation techniques, which can give us a clearer picture of exactly what obfuscation does to the code:

- *Variable obfuscation* randomly turns meaningful variables, methods, and constant names into meaningless gibberish-like strings such as single characters or hexadecimal strings to reduce code readability.
- *String obfuscation* arrays strings and stores them centrally with MD5 or Base64 encryption so that no plaintext strings appear in the code, thus avoiding the need to locate the entry point using a global search for strings.
- *Property encryption* transforms the properties of JavaScript objects cryptographically, hiding the invocation relationships between the code.
- *Control flow flattening* disrupts the original code execution flow of functions and function call relationships, making the code logic chaotic and disorderly.
- *Dead code injection* randomly inserts useless dead code or dead functions into the code to further clutter the code.
- *Debugging protection* checks the current runtime environment and adds some forced debugger statements based on debugger statements to make it difficult to execute JavaScript code in debug mode.
- *Polymorphic mutation* makes JavaScript code automatically mutates itself every time it is called, changing it into a completely different code than before, i.e. the function remains the same but the form of the code changes, thus eliminating the code from being dynamically analyzed and debugged.

According to the work of Moog et al. [27], most Alexa Top 10K sites [33] contain at least one obfuscated script, and obfuscation is commonly used by both benign and malicious scripts. Benign scripts tend to apply minification. Over 60% of the scripts from the websites apply minification. However, there are also some benign samples applying multiple obfuscations. There are about 6% of benign scripts using variable obfuscation. About 3% of benign scripts apply string obfuscation. The percentage of benign scripts that use other obfuscation techniques is less than 3%. Malicious scripts tend to combine many obfuscation techniques. 25%-27% of malicious scripts use variable obfuscation. 17%-21% of malicious scripts use string obfuscation. Other obfuscation techniques are used in malicious scripts with a probability of 5%-10%. So we can not simply use the detection of obfuscation to classify content as malicious. Note that obfuscation only changes the appearance of code, not its semantics or function, that is obfuscation does not change the essence of the code. If we can capture the essence of what the code achieves, it will be less difficult to analyze the obfuscated code.

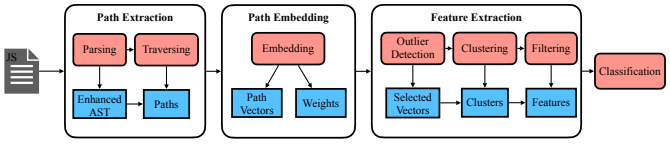


Fig. 1: Architecture of JSRevealer

III. Approach

In this section, we introduce the design of our robust malicious JavaScript detector, *JSRevealer*.

A. Overview

Figure 1 illustrates the architecture of *JSRevealer*. *JSRevealer* consists of four components, which are path extraction, path embedding, feature extraction, and classification. First, path extraction parses JavaScript files into ASTs, adds data flow information as enhanced ASTs, and then traverses those enhanced ASTs to obtain the paths. Next, in the path embedding, we use neural networks and attention mechanisms to get the embeddings of the paths, namely path vectors, and the corresponding weights. After that, in the feature extraction stage, we perform outlier detection on those vectors with weights. Then the selected vectors are clustered, and the features are obtained by filtering the clusters. Finally, we use these features for learning and classification.

In the following, we describe the details of each component in turn.

```
function testcase(){
  var timeZoneMinutes = new Date().getTimezoneOffset()*(-1);
  var date,dateStr;
  try{
    if(timeZoneMinutes > 0){
        data = new Date(1970, 0, 100000001,0,0+timeZoneMinutes+60,0,1);
        dataStr = date.toISOString();
        return false;
    }else{
        date = new Date(1970,0,100000001,0,0,0,1);
        dateStr = date.toISOString();
        return false;
    }
  }catch(e){
    return e instanceof RangeError;
  }
}
```

Listing 1: A real JavaScript code example

B. Path Extraction

Obfuscation changes the appearance of the code, but not its semantics. We believe that if the method is affected by obfuscation, it is because too much attention is given to representational features. We focus on abstract features that are related to the intrinsic semantics of the code so that the approach using these features is not easily affected by obfuscation. The general idea of obtaining abstract features is splitting and regrouping.

First, we split the code into more fine-grained units. The AST path is a good abstract representation proven by previous works [34], [35]. However, the AST contains only the syntactic structure of the code and misses semantic information, which does not help us to find the abstract features of the intrinsic semantics. To make the fine-grained units contain semantic

information, we first use *Esprima* [36] to parse the JavaScript files into AST and then add data flow information to the AST. We refer to such an AST as an **enhanced AST**. A data dependency edge is added between statements that contain the same variable. This representation captures the data flow between the different components so that the semantic information of the code can be reflected in the units, that is, the paths we extract next.

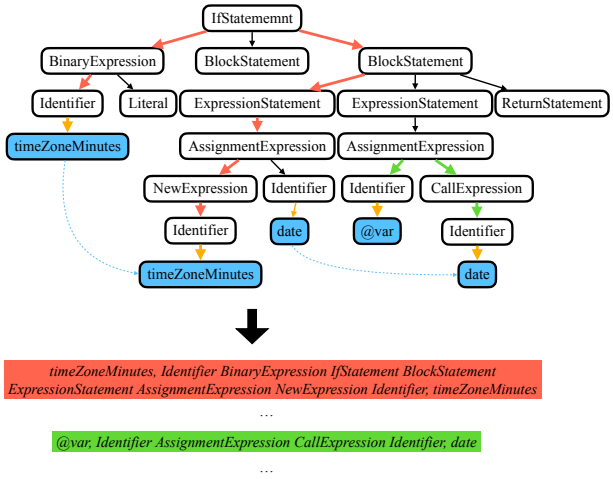


Fig. 2: Enhanced AST corresponding to line 5 to 12 of the code in Listing 1

Then we traverse enhanced ASTs to obtain paths. The path here is similar to the path-context Alon et al. [34] defined, i.e., a triple $< x_s, n_1 n_2 ... n_k, x_t >$, where n_1 and n_k are leaves of the AST, x_s and x_t are the values associated with n_1 and n_k, $n_2 ... n_{k-1}$ is the sequence of AST nodes between the two leaves. Specifically, for the syntactic unit with data dependency, we keep the value of that syntactic unit. In this way, x_s or x_t in the triple of the path is a specific value, for example, a variable name. The two paths with data dependency will have the same value in their triplets, and the vectors obtained in the embedding process will be closer. For the syntactic unit with data dependency, x_s or x_t here is an indicator, *@var_int* for integer type variables, *@var_str* for string type variables, and so on. Figure 2 shows the enhanced AST generated from line 5 to 12 of the code in Listing 1. We add a data dependency edge (the blue dot line) to the two leaves that have data dependency (a program statement refers to the data of a preceding statement). The value of the leaves, *timeZoneMinutes*, is preserved. The variable *dateStr* has no data dependencies with other components of the program, so *@var_str* is used here to indicate it.

The number of paths obtained will be huge and the computational overhead will be unacceptable if we extract all the paths naively. To limit the number of extracted paths, we put a limit on the maximum length and maximum width of a path as the previous works did. Maximum length is the maximal value of k. Maximum width is the maximal difference in index between two child nodes of the same intermediate node. We set these values to 12 and 4 empirically. The rationality has been discussed by Alon et al. [34] in terms of locality, sparsity, and performance.

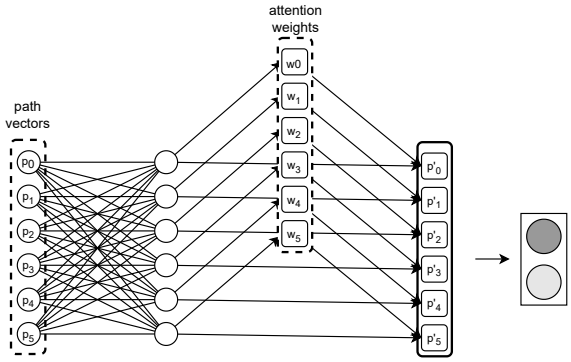


Fig. 3: A schematic of our neural network for embedding

C. Path Embedding

Different from the previous methods [20], [21], we do not extract features from the path directly. The information that the path exhibits is not intuitive and the relationships between paths are not clear. To represent paths better, the neural network is used for *path embedding*. The continuous distribution vector representations facilitate analysis using data-driven automated methods.

We build a model based on the attention mechanism, which consists mainly of a fully connected layer and attention (as shown in Figure 3). The paths $P = \{p_1, p_2, ..., p_n\}$ extracted from the script are fed into the model, and each path is first embedded into a d-dimensional vector after passing through the fully connected layer.

$$p'_i = tanh(W \cdot p_i) \tag{1}$$

where $p_i \in \mathbb{R}^{|P|}$ is the initial vector representing $i-th$ path, p'_i is the output of the fully connected layer, that is, the embedding of the path, and $W \in \mathbb{R}^{d \times |P|}$ is the learned weight matrix.

Then the attention mechanism is used to calculate the weights of path vector.

$$\alpha_i = \frac{exp({p'_i}^T \cdot \boldsymbol{\alpha})}{\sum_{j=1}^{n} exp({p'_j}^T \cdot \boldsymbol{\alpha})} \tag{2}$$

where α_i is the attention weight of p'_i and $\boldsymbol{\alpha}$ is the attention vector, which is initialized randomly.

Finally, we train this model with the labels of scripts.

$$\boldsymbol{v} = \sum_{i=1}^{n} \alpha_i \cdot p'_i \tag{3}$$

$$y' = softmax(W \cdot \boldsymbol{v}) \tag{4}$$

$$Loss = cross - entropy(y, y') \tag{5}$$

where $\boldsymbol{v}$ is the aggregated vector, which represents the whole script, y is the label of the script, which is binary here, y' is the probability vector that denotes the probability that the script should be tagged to each label, and we use cross-entropy loss function to get the loss.

Note that here we train this model with 5,000 additional data, including 2,500 benign samples and 2,500 malicious samples, using the labels of the samples, namely benign and malicious, as the ground truth. We choose 300 as the path embedding size of d (the length of path vector) and train the model for 100 epochs. When we get a trained model, the paths from an unseen script can be fed into the model as same as the training step. We take the output vectors of the fully connected layer and the weights calculated by the attention mechanism. These vectors are the embeddings of the paths, and the corresponding weights reflect the importance of each path in the script.

D. Feature Extraction

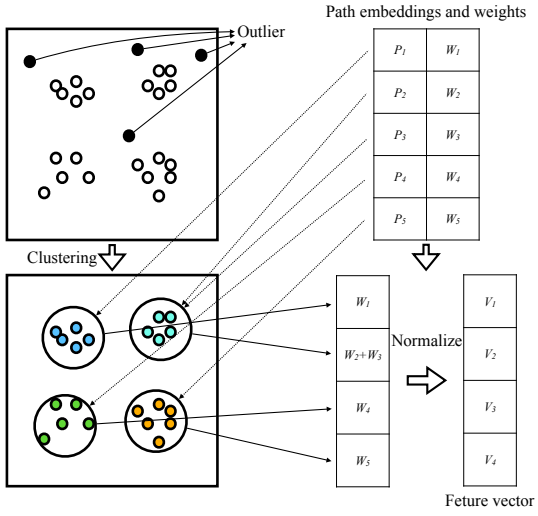


Fig. 4: A schematic for feature extraction

Our core insight is to find more essential and abstract features that are less likely to be disturbed by obfuscation. We use a clustering algorithm to cluster these vectors to reveal the hidden groupings of these paths once the paths are embedded as vectors. Using the obtained clusters as features should better indicate the essential semantics of the code and be more resistant to code transformations such as obfuscation.

Figure 4 shows a schematic of our feature extraction. There is bound to be some noise in the paths we get, such as some statements in the program that make little sense. For better clustering results, outlier removal is performed before clustering. There are many outlier detection algorithms. Selecting a suitable outlier detection algorithm is to some extent a matter of luck. Models are difficult to evaluate because the target dataset is often unlabeled and there is no universal evaluation function. Using a data-driven approach to model selection is a good idea. We use *MetaOD* [37] proposed by Zhao et al. [38] to select our outlier detection model. *MetaOD*, based on mata-learning [39], is designed to automatically select a good outlier detection method and its hyperparameters on a new dataset. The optimal outlier detection model on our dataset returned by *MetaOD* is *Fast Angle-Based Outlier Detection* using the approximation (*FastABOD*). *FastABOD* is an outlier detection method that is based on the principle of calculating the variance of the angle formed by each sample and all other samples, where the variance of the outlier is small. Leveraging an implementation of *FastABOD* in a Python library, *PyOD* [40], we remove the vectors that are identified as outliers from all the path vectors.

Then we perform clustering on the remaining vectors. Theoretically, for these continuously distributed vectors, vectors corresponding to paths with similar semantics have close distances. The commonly used center-of-mass-based clustering algorithm is suitable for the task of clustering on these vectors. For better performance than K-Means, we choose Bisecting K-Means to perform clustering. Bisecting K-Means algorithm is a modification of the K-Means algorithm, mainly to solve the problem of uncertainty in the clustering results caused by the randomness of choosing the initial center of mass in the K-means algorithm. Utilizing the implementation in a Python library, *scikit-learn* [41], we cluster the path vectors obtained in the benign samples and ones in the malicious samples separately. Then the groups with high overlap are removed. The remaining groups are used as features to identify benign and malicious samples. For the choice of clustering K values, we combine the elbow method and our classification task to experiment with different K values. Empirically, the optimal K value for benign samples is 11 and 10 for malicious samples. There are no similar clusters to be removed, that is, we end up with 21 clusters, which is described in detail in Section IV-B.

It can be explained that each feature represents a class of behavior of the code when we select the clusters obtained by clustering as features. As shown in Figure 4, we use the importance of the behavior in the code, that is, the weights of paths obtained in path embedding, as the value of the feature, rather than the binary value of whether the path occurs or not. Specifically, for a target script, we get the vector and weight corresponding to each path after path extraction and embedding, and this weight indicates the path's importance in the script. For all the paths in this script, we judge which cluster they belong to in turn. If a path belongs to a certain cluster, the weight of that path vector is added to the corresponding feature value. Finally, the obtained feature vectors are normalized by leveraging min-max normalization. The equation is as follows:

$$V' = \frac{V - min(V)}{max(V) - min(V)} \tag{6}$$

where V is the feature vector and V' is the normalized feature vector.

E. Classification

The last component of *JSRevealer* is a machine learning classifier. The feature vectors are used to train our classifier, or targets for classification by the trained classifier. We empirically evaluate several machine learning algorithms (*Support Vector Machine* (*SVM*), *logistic regression*, *decision tree*, *Gaussian naive Bayes*, and *random forest*), and choose *random forest*, which is described in detail in Section IV-B.

IV. Evaluation

In this section, we describe the results of our evaluation of *JSRevealer*. We aim to answer the following *Research Questions* (RQs):

- *How does JSRevealer perform on detecting obfuscated malicious JavaScript?*
- *How does JSRevealer perform compared to other state-of-the-art approaches?*
- *How is the interpretability of JSRevealer?*
- *How is the runtime overhead of JSRevealer?*

A. Experimental Setup

1) Dataset: Our dataset contains data from different sources. Table I shows the detail of the dataset. The malicious samples in the dataset consist of the malware collection of Hynek Petrak [42], exploit kits from GeeksOnSecurity [43], and the additional samples from VirusTotal [44]. The benign samples come from the 150k JavaScript Dataset published by Raychev et al. [45], consisting of 150,000 JavaScript files, and the scripts crawled from Alexa Top-10k websites, consisting of over 60,000 scripts.

Even though previous studies have proposed some approaches to detect obfuscation [27], [46], we are not sure which of these scripts in the original dataset are obfuscated and in what way the obfuscated scripts are obfuscated with these approaches. But our subsequent experiments need to know which scripts are obfuscated in which way with certainty. To achieve this, the obfuscated samples in our test set are re-obfuscated by the obfuscators below.

TABLE I: Dataset

Class	Source	#JS
Malicious	Hynek Petrak	39,450
	GeeksOnSecurity	1,370
	VirusTotal	1778
Benign	150k JavaScript Dataset	150,000
	Alexa Top-10k	65,203

2) JavaScript Obfuscation Tools: We evaluate *JSRevealer* against four of the most commonly used obfuscators:

- *JavaScript-Obfuscator* [28] is a powerful and free obfuscator for JavaScript, containing a variety of features, such as variable renaming and control flow flattening.
- *Jfogs* [29] is a JavaScript obfuscator that focuses on removing function call identifiers and parameters.
- *JSObfu* [30] is a JavaScript obfuscator written in Ruby, which randomizes and removes easily-signaturable string constants as much as possible.
- *Jshaman* [31] is a professional JavaScript code obfuscation and encryption platform, providing professional JavaScript obfuscation and JavaScript encryption services.

3) JavaScript Malware Detectors: We compare *JSRevealer* with four state-of-art, static, learning-based JavaScript malware detectors:

- *CUJO* uses n-grams features from both static and dynamic analysis to detect malware. Since *JSRevealer* is static, we compare *JSRevealer* only with CUJO's static part. We use the re-implementation of CUJO provided by Fass et al. [11] in our experiments.
- *ZOZZLE* detects malware based on the hierarchical features of ASTs. In our comparison, we use the *ZOZZLE* re-implemented by Fass et al. [47].
- *JAST* extracts n-grams features from ASTs to detect malicious JavaScript. We directly use the system available on GitHub [48].
- *JSTAP* extends the detection capability of lexical and AST-based pipelines by also leveraging control and data flow information. It extracts n-gram features or combines them with the name information of variables. We consider *JSTAP*'s PDG code abstraction with the n-grams feature in our comparison. We use their implementation available on GitHub [49].

4) Classifier Training: We use the following protocol to train and evaluate *JSRevealer* and four detectors. To build a balanced model, we randomly select 20,000 samples from benign and malicious samples, respectively. Then we split them into training and validation sets containing 75% and 25% of the samples to train these detectors. The remaining samples are used to evaluate the performance of these detectors. The results are obtained by repeating the procedure five times and then averaging.

Note that the ratio of benign to malicious in our test set is 1:1. This is different from the distribution in reality, where there are more benign samples. However, we compare detectors on multiple metrics and we highlight FPR and FNR, which are unaffected by the ratio of the two types of samples in the test set. So the difference does not affect the reliability of our experimental conclusions.

We conduct all our experiments on a server with Ubuntu 18.04.1, an Intel Xeon Gold 6234 CPU @ 3.30GHz, NVIDIA Quadro RTX 5000 GPU, and 32G RAM.

B. RQ1: Performance

It is difficult to get the optimal value of K for clustering. Here we use the *elbow method* to guide us to find a more appropriate K value. The core metric of the elbow method is SSE (*sum of the squared errors*), which is the clustering error of all samples, and represents the clustering effect, and the core idea is that the intra-cluster distance decreases as the

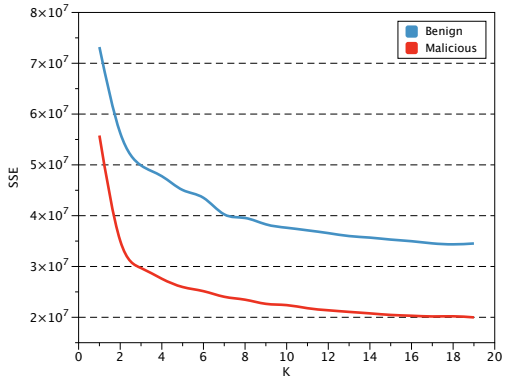


Fig. 5: SSE values corresponding to different K in benign and malicious samples

TABLE II: Performance of different machine learning algorithms

Method	Accuracy (%)	F1 (%)	FPR (%)	FNR (%)
SVM	98.9	98.7	1.3	0.9
Logistic Regression	98.4	98.3	2.0	1.0
Gaussian Naive Bayes	98.4	98.2	2.1	1.0
Decision Tree	99.0	98.8	1.2	0.9
Random Forest	**99.4**	**99.4**	**0.3**	**0.8**

value of K increases in the process of changing from small to large, but there is no significant decrease in the intra-cluster distance when the optimal solution of K is obtained. Figure 5 shows the SSE values corresponding to different K values in benign and malicious samples. We can see that the elbow value in the benign sample is around 7 and the elbow value in the malicious sample is around 4.

We use the K values 7 and 4 to evaluate several common machine learning methods, including SVM, logistic regression, decision tree, Gaussian naive Bayes, and random forest. Table II shows the results of these methods trained and tested on JavaScript files without obfuscation. We can see that these methods perform similarly, with random forest performing best. Since random forest is also able to provide some interpretability, we choose it in all our other experiments to build our classifier.

But the K values 7 and 4 are not necessarily optimal for detection on obfuscated data. We also perform experiments with K values around these two values, using obfuscated samples to test the *JSRevealer*'s performance. Table III shows the average F1 of *JSRevealer* using different K values on code obfuscated by four obfuscators. It can be seen that *JSRevealer* performs best when the clustering K value is taken to be 11 in benign samples and 10 in malicious samples. In the following experiments, we use 11 and 10 as the clustering K values in benign and malicious samples, respectively.

To evaluate the effectiveness of *JSRevealer*, we test *JSRevealer* on the dataset obfuscated by four obfuscators described in Section IV-A2. Table IV shows the results of *JSRevealer* to classify the JavaScript code obfuscated by each obfuscator. The first row gives the results of *JSRevealer* classifying the

TABLE III: Average F1-measure (%) of *JSRevealer* using different clusters on code obfuscated by four obfuscators

Clusters from malicious samples	Clusters from benign samples						
	6	7	8	9	10	11	12
3	84.8	84.6	84.8	84.7	84.8	84.8	84.7
4	84.5	84.6	84.6	84.3	84.3	85.4	84.9
5	84.6	84.7	84.2	85.0	84.5	84.7	84.3
6	84.1	84.4	84.3	84.5	84.3	85.4	84.7
7	84.3	84.5	84.4	85.9	84.6	85.7	84.5
8	84.4	84.5	84.8	84.1	84.8	84.6	85.2
9	85.1	85.1	85.1	85.3	85.3	85.4	85.6
10	85.1	85.4	85.3	85.2	85.7	**86.9**	85.0
11	84.2	86.9	84.9	84.9	85.5	86.4	85.6
12	86.6	84.5	85.4	85.5	85.6	84.9	85.0

scripts without obfuscation, which serves as the baseline. *JSRevealer* performs well in all metrics on the dataset without obfuscation, with 99.4% accuracy and F1, and only 0.3% FPR and 0.8% FNR. We aim to have good results on the obfuscated data as well, but the performance will definitely somewhat degrade compared to the unobfuscated data. Rows two through five show the results of *JSRevealer* tested on the data obfuscated by obfuscators. *JSRevealer* shows decreases in all metrics. *Jshaman* affected *JSRevealer* the least, with only a 5.2% decrease in accuracy compared to baseline, only a 7.5% increase in FPR, and a 3.1% increase in FNR. This is probably because we only use the basic version of *Jshaman*, which mainly uses variable obfuscation techniques, resulting in a weaker obfuscation compared to other obfuscators. The accuracy of *JSRevealer* on the data obfuscated by *JavaScript-Obfuscator* and *Jfogs* are 86.7% and 83.3%, respectively. However, their different characteristics have a different impact on *JSRevealer*. On the code obfuscated by *JavaScript-Obfuscator*, the FPR of *JSRevealer* reaches 22.2%, while the FNR is only 5.6%. In the contrast, the FPR is only 4.7%, while the FNR reaches 28.1% of the code obfuscated by *Jfogs*. *JSObfu* has the strongest impact on *JSRevealer*, with an accuracy of only 73.6%. Both the FPR and FNR of *JSRevealer* increase significantly compared to the baseline, by 29.9% and 22.3%, respectively.

TABLE IV: Performance of *JSRevealer* using enhanced AST and regular AST on code obfuscated by different obfuscators

Obfuscator	Representation	Accuracy (%)	F1 (%)	FPR (%)	FNR (%)
Baseline (No obfuscation)	Enhanced AST	99.4	99.4	0.3	0.8
	Regular AST	76.1	78.5	41.6	1.9
JavaScript-Obfuscator	Enhanced AST	86.7	88.4	22.2	5.6
	Regular AST	68.5	75.4	56.5	10.1
Jfogs	Enhanced AST	83.3	81.5	4.7	28.1
	Regular AST	61.1	72.1	77.5	2.5
JSObfu	Enhanced AST	73.6	75.4	30.2	23.1
	Regular AST	76.3	80.8	44.6	4.6
Jshaman	Enhanced AST	94.2	94.2	7.8	3.9
	Regular AST	63.6	72.3	68.0	4.2
Average on obfuscated samples	Enhanced AST	84.5	84.8	16.2	15.2
	Regular AST	67.4	75.2	61.7	5.4

But in general, *JSRevealer* exceeds 73% accuracy and F1 on various obfuscated data and keeps the decline within 24%

compared to the baseline. On average, the accuracy is 84.5% and F1 is 84.9%. The FPR and FNR are relatively constant, with an average increase of 15.9% and 14.3% compared to the baseline, respectively.

To demonstrate the necessity of the enhanced AST, we evaluate the performance of JSRevealer using regular AST. We replace the enhanced AST used in JSRevealer with the regular AST and keep the rest of the parts the same. Note that we obtain different values of K than using enhanced AST. The K values we obtain using the elbow method are 5 on the benign samples and 4 on the malicious samples. Then as using the enhanced AST, we evaluate K values around these two values on the obfuscated samples. Finally, we take 5 and 6 for the K values.

From Table IV, we can see that *JSRevealer* using regular AST exhibits high FPRs on both unobfuscated and obfuscated samples. The performance on each obfuscator is poorer than using the enhanced AST. The only exception is *JSObfu*, where using regular AST has 2.7% higher accuracy and 5.4% higher F1 than using enhanced AST on the samples it obfuscates. However, its FPR is up to 44.6%. On average, the FPR of using regular AST is up to 61.7%, which is unacceptable in practice.

We argue that the path extracted directly from the AST does not contain semantic information about the code, so further abstraction such as embedding and clustering is meaningless. It does not make the detector gain the ability to distinguish between benign and malicious samples, which is demonstrated here by the tendency to classify various types of samples as malicious scripts.

Summary: *JSRevealer* can achieve good detection performance on both unobfuscated and obfuscated JavaScript code, in which enhanced AST plays an important role.

C. RQ2: Comparison with Other Techniques

In this section, we compare *JSRevealer* with four state-of-art malicious JavaScript detectors.

TABLE V: Accuracy (%) of *JSRevealer* and other detectors on code obfuscated by different obfuscators

Detector	Baseline	*JavaScript-Obfuscator*	*Jfogs*	*JSObfu*	*Jshaman*
CUJO	77.4	52.6	50.3	51.2	51.4
ZOZZLE	98.0	71.5	77.8	36.9	74.7
JAST	97.9	80.9	59.4	67.1	88.0
JSTAP	99.1	70.4	54.1	**75.6**	**98.8**
JSRevealer	**99.4**	**86.7**	**83.3**	73.6	94.2

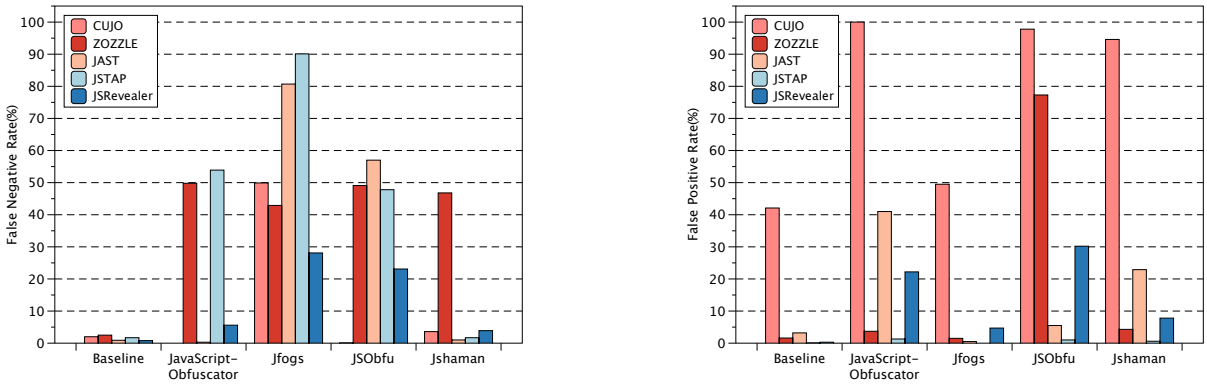
Table V shows the accuracy of training and testing *JSRevealer* and four comparative detectors on the unobfuscated samples and the samples obfuscated by different obfuscators. On the scripts obfuscated by *JavaScript-Obfuscator* and *Jfogs*, the accuracy of *JSRevealer* is higher than all four detectors, exceeding them by 5.8%-36.1% and 5.5%-30.0%. On the code obfuscated by *JSObfu* and *Jshaman*, the accuracy of *JSRevealer* is slightly lower than *JSTAP*, by 2.0% and 4.6%, and higher than the other three detectors, by 6.5%-36.7% and 6.2%-42.8%, respectively. Overall, the accuracy of the *JSRevealer* is higher than that of the four compared detectors on various types of samples. The only exception is a lower accuracy than *JSTAP* on the samples obfuscated by *JSObfu* and *Jshaman*. However, this is due to the significant classification bias of *JSTAP*, which means that it will be biased to identify samples as a certain class.

Next, we discuss the FNR and FPR metrics. The specific performance of the detectors is very different. Figure 6 shows the FNRs and FPRs of *JSRevealer* and four detectors on the different obfuscated samples. *CUJO* mainly shows a significant increase in FPR on obfuscated samples. The FPRs of *CUJO* increase by 57.9%, 55.7%, and 52.5% on the data obfuscated by *JavaScript-Obfuscator*, *JSObfu*, and *Jshaman*, respectively. Such performance may be due to the fact that the features extracted by *CUJO* are token n-grams obtained from lexical analysis. These obfuscators will disrupt the original token ordering of the code. Most of malicious samples in the original dataset as the training set also use obfuscation, which makes it easy for *CUJO* to extract features indicating maliciousness from the benign samples obfuscated by these three obfuscators for testing. *CUJO*'s performance on samples obfuscated by *Jfogs* is an exception. The FPR is 49.5% and the FNR is 49.9%, both close to 50%. This suggests that *CUJO* is unable to distinguish between benign and malicious samples obfuscated by *Jfogs*. This may be because *Jfogs* uses a similar structure to remove function call identifiers and parameters and use the string *fog* instead, making the features extracted by *CUJO* all similar.

ZOZZLE shows a completely different performance, with a significant increase in FNR on obfuscated samples. Compared to the baseline, the FNRs of *ZOZZLE* increase by 47.3%, 40.4%, 46.6%, and 44.3% on the samples obfuscated by *JavaScript-Obfuscator*, *Jfogs*, *JSObfu*, and *Jshaman*, respectively. The feature *ZOZZLE* extracts consists of AST context and the text of the AST node. Each of these obfuscators can break this combination, thus allowing malicious samples to escape *ZOZZLE*'s detection.

JAST's FNRs significantly increase on the samples obfuscated by *Jfogs* and *JSObfu*, by 79.8% and 56.1%, respectively, while FPRs significantly increase on the samples obfuscated by *JavaScript-Obfuscator* and *Jshaman*, by 37.8% and 19.7%, respectively. *JAST* uses n-grams of AST syntactic units as features. Each of these four obfuscators can change the structure of the code's AST, introducing many different syntactic unit n-grams. These changes may predispose *JAST* to classify obfuscated samples as either benign or malicious, suggesting that *JAST* is more vulnerable.

JSTAP shows a similar trend with *ZOZZLE*, mainly a significant increase in FNR. *JSTAP*'s FNRs increase 52.2%, 88.4%, and 46.1% on the code obfuscated by *JavaScript-Obfuscator*, *Jfogs*, and *JSObfu*, respectively. Except *JSTAP* performs well against *Jshaman*, not much worse than baseline. *JSTAP* extracts n-gram features from PDG of the code. It extracts a larger number of n-grams, and when the malicious



(a) False Negative Rate

(b) False Positive Rate

Fig. 6: False negative rate and false positive rate of *JSRevealer* and other detectors on code obfuscated by different obfuscators

samples are obfuscated, the obvious malicious features may be drowned in other features, thus allowing malicious samples to evade detection. Since we use the basic version of *Jshaman*, it has less impact on the structure of the code's PDG, thus *JSTAP* has better performance on the data it obfuscates.

In contrast, *JSRevealer* keeps the decrease in FNRs and FPRs within 30% on all types of obfuscated samples compared to the baseline. This indicates that the features *JSRevealer* extracts are not easily affected by obfuscation. Although the FNR or FPR is higher than a certain detector on a certain type of obfuscated samples, *JSRevealer* has the most stable performance when we combine the four types of obfuscated data. *JSRevealer* does not show very serious errors on all four types of obfuscated data, while all four detectors as comparisons do, which is unacceptable in practice.

TABLE VI: F1-measure (%) of *JSRevealer* and other detectors on code obfuscated by different obfuscators

Detector	Baseline	*JavaScript-Obfuscator*	*Jfogs*	*JSObfu*	*Jshaman*
CUJO	80.8	69.0	49.8	67.2	66.7
ZOZZLE	97.9	65.4	72.0	44.8	67.6
JAST	98.0	84.9	32.2	58.2	89.1
JSTAP	99.1	62.6	18.0	68.1	**98.8**
JSRevealer	**99.4**	**88.4**	**81.5**	**75.4**	94.2

We further discuss F1 scores of *JSRevealer* and other detectors (as Table VI shows), which show the comprehensive performance of these approaches. On the data obfuscated by *JSObfu*, *JSTAP* has a higher accuracy than *JSRevealer*, but it is important to see that it has a very high FNR of 47.8%. In terms of F1, *JSRevealer* is 7.3% higher than *JSTAP* and 8.2%-30.6% higher than the other three detectors. On the data obfuscated by *JavaScript-Obfuscator* and *Jfogs*, *JSRevealer* is 3.5%-25.8% and 9.5%-63.5% higher than the four detectors. The only exception is that *JSTAP*'s F1 is 4.6% higher than *JSRevealer*'s on the code obfuscated by *Jshaman*. This is probably due to the fact, as we describe above, that *Jshaman* only uses the technique of variable obfuscation and *JSTAP* uses PDG n-grams as features, which results in *JSTAP* being less affected.

Overall, *JSRevealer*'s comprehensive performance is superior to the compared detectors. Figure 7 shows the average performance of *JSRevealer* and other detectors on code obfuscated by different obfuscators. The average F1 of *JSRevealer* is significantly higher than the compared detectors, 21.6%, 22.3%, 18.7%, and 22.9% higher than *CUJO*, *ZOZZLE*, *JAST*, and *JSTAP*, respectively.

Summary: On the obfuscated samples, *JSRevealer* detection performance is significantly better than four of the most influential detectors.

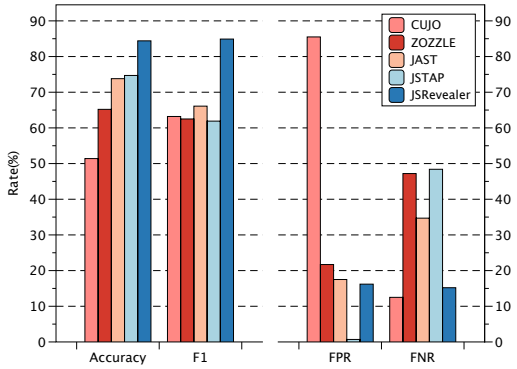


Fig. 7: Average performance of *JSRevealer* and other detectors on code obfuscated by different obfuscators

D. RQ3: Analysis of Features

Our method has a more robust performance by using the features we extract. The essence of the feature extraction is to use a data-driven approach to automatically delineate the categorization of semantic units. These categories provide a distinctive perspective on the different patterns exhibited by benign and malicious code. We choose the five most important features, or clusters, based on random forests. We

TABLE VII: The central paths and importances corresponding to the five most important features

Importance	Central paths
0.133	*[controls, Identifier VariableDeclarator VariableDeclaration BlockStatement VariableDeclaration VariableDeclarator ConditionalExpression MemberExpression Identifier,options]*
0.225	*[@var_str, Identifier MemberExpression AssignmentExpression FunctionExpression BlockStatement ReturnStatement CallExpression CallExpression Identifier,item]*
0.103	*[@var_str, Identifier CallExpression IfStatement BlockStatement IfStatement BlockStatement IfStatement BinaryExpression CallExpression Literal,@var_str]*
0.202	*[@var_int, Literal MemberExpression BinaryExpression MemberExpression MemberExpression Literal,@var_int]*
0.093	*[@var_str, Literal AssignmentExpression ExpressionStatement BlockStatement IfStatement BlockStatement ExpressionStatement AssignmentExpression Literal,@var_int]*

store all the paths obtained in the training set as a corpus and use the index to represent each path. The paths and the corresponding embeddings are one-to-one. We additionally store the index of a path while obtaining its embedding. So for each embedding, we can get the corresponding path with the traditional AST representation based on the index. We obtain the paths corresponding to these cluster centers from the stored indexes. Table VII shows details of the central paths and importances corresponding to the five most important features.

The first, second, and third are obtained by clustering benign samples, and the fourth and fifth are obtained from malicious samples. The values at the beginning and end of the first path are "*controls*" and "*options*", respectively, and the specific values indicate data dependencies with other paths. These two words are often found in JavaScript files in the form of "*options.controls*". The middle path contains multiple *VariableDeclaration*, representing a part of the configuration of various variables. It is often found when setting up and implementing multimedia functions for web pages. The "*@var_int*" in the second path indicates an integer-type variable name, and the "*@var_str*" below indicates a string-type variable name. The middle of the path is *FuntionExpression* and *BlockStatement*. We assume that the second path represents the units related to the overall structure of functions. The third path contains multiple *IfStatement* and *CallExpression*, and the beginning and end are *Identifier* "*@var_str*" and *Literal* "*@var_str*", respectively, representing variable names of string type and value of string type. We think it represents the units associated with different function call cases. In contrast, the clustering centers obtained from malicious samples are significantly different. The fourth path has *BinaryExpression* in the middle, and *MemberExpression* and the value of integer type on either side, indicating the operation of integer values. We consider the fourth path to represent the units related to basic data manipulation. The fifth path has the *IfStratement* and *BlockStatement* in the middle. On either side, there is a string-type value assignment and an integer-type value assignment. We assume that it represents the assignment and fetches operations for different cases.

In summary, we can see that benign samples focus on the implementation of functions, while malicious samples focus on the manipulation of data, probably because malicious samples are mainly interested in the sensitive data of users. This essentially explains the difference between benign and malicious samples and provides further insight into the nature of both benign and malicious samples.

Summary: *JSRevealer* has interpretability and the features we extract can reflect the nature of benign and malicious JavaScript code.

E. RQ4: Runtime Overhead

In this part, we evaluate the time overhead of *JSRevealer*. The experimental results are obtained by training and testing *JSRevealer* five times and then averaging the time. In particular, the average file size in our dataset is 62 KB. Table VIII presents the average time overhead and standard deviation per module on **one** file. While path embedding, feature generation, and classification are based on machine learning and deep learning. Many of the libraries used to implement them are optimized for large-scale data processing. The standard deviations of file-consuming time for these parts cannot be evaluated here, so they are not included here.

The path extraction module includes enhanced AST generation and path traversal. The enhanced AST generation is time-consuming compared to other modules, while the path traversal is more time-consuming, resulting in the path extraction taking much longer time. Though we use depth and width pruning to reduce the traversal time consumption, the number of paths is still quite large. The standard deviation of the time consumption in path extraction is also quite large. This is because some files are so large that enhanced AST generation and path traversal on them can take several times or even tens of times longer. In path embedding module, it requires additional files for pre-training as Section III-C describes. Since we use a very simple model, the time overhead here is small, with an average pre-training time of 22.546 ms per file. The time overhead of embedding is even smaller, only 11.66 ms. Feature generation consists of outlier detection and clustering. Outlier detection takes the longest time, 396.471 ms per file. This is related to the outlier detection method we choose. The time overhead of different outlier detection methods can vary significantly, and the time overhead here is within an acceptable range. In an actual deployment, it is possible to change to an outlier detection method with less time overhead as needed. In contrast, clustering has a small time overhead, only 24.243 ms per file. As for classification,

the training and classifying phases both have outstanding time performance, only 0.235 ms and 0.143 ms, respectively.

The path extraction and feature generation modules have relatively high time overheads. However, in practical applications, feature generation only needs to be run once. The time that is consumed to detect one file includes the time overheads of path extraction, embedding in the path embedding module, and classifying in the classification module, that is, the average time to detect one file is 582 ms. Considering that the average size per file is 62 KB, the time performance is acceptable for accomplishing scalable detection.

Summary: *JSRevealer* has relatively good time performance and can meet the needs of large-scale detection.

TABLE VIII: *JSRevealer*'s run-time per module

Modules	Period	Average time consumed per file (ms)	Standard Deviation (ms)
Path extraction	Enhanced AST	221.278	822.920
	Path traversal	348.537	1048.058
Path embedding	Pre-training	22.546	-
	Embedding	11.660	-
Feature generation	Outlier detection	396.471	-
	Clustering	24.243	-
Classification	Training	0.235	-
	Classifying	0.143	-

V. Discussion and Limitations

A. Discussion

Previous works have briefly discussed the effects of obfuscation in their papers, but there are no quantitative experiments to show exactly what the effects would be. Our experiments above illustrate that their methods are very susceptible to obfuscation, with substantially lower accuracy on obfuscated samples. Although malicious scripts may not employ the same obfuscation tools we use in reality, these experimental results show that malicious scripts can easily evade detection through these obfuscations, which already pose a potentially significant threat to web users. And when malicious scripts use more targeted obfuscation, they are less likely to be detected by these detectors. The experimental results above confirm that the detector we propose, *JSRevealer*, does have good resistance to obfuscation. We get the features by fine-grained partitioning of the code and then clustering, i.e., splitting and then regrouping. We argue that such characteristics are abstract and more essential, which are less susceptible to obfuscation. To explain further, it is about appearance and essence. The obfuscation tools now in common use only change the appearance, not its semantics, that is, the essence. Finding features that better reflect the nature of benign versus malicious will mitigate the effects of obfuscation naturally, resulting in a more robust approach. However, our work targets the binary classification of benign and malicious samples and does not distinguish between different JavaScript attacks. Our future work will add a JavaScript malware family component, and will also further optimize detection for different types of JavaScript attacks.

Additionally, there may be some problematic vectors with low weights, and it may be unavoidable for *JSRevealer* to ignore them. However, JSRevealer identifying scripts is not dependent on a certain vector. All path vectors of a script can roughly satisfy the condition that the more they reflect the semantics of the script, the higher the weight. So these omitted low-weight vectors have little impact on *JSRevealer*.

B. Limitations

As with all learning-based methods, our approach is dependent on the dataset. If some malicious samples are very different from those in the training set, it is also difficult for our method to detect them. For example, if the distance of the vector of a test sample from the dataset centroid is longer than the longest distance of the vectors of samples in the dataset from the dataset centroid, such a sample may be beyond the capability of the trained model. This can only be solved by increasing the number and variety of samples in the training set as much as possible. However, since we extract more abstract features, our method is more difficult to be targeted compared to the previous methods. It is possible to focus the improvements only on improving the quality of the dataset.

In our experiments, we first select a value using the elbow method and then search for an appropriate value K around this value. The specific value K depends on the following classification task and we cannot guarantee its optimality. We consider incorporating more methods for selecting K values in future work, such as Silhouette Coefficient and Gap Statistic, and then combining multiple methods to select a more appropriate K that is not dependent on the downstream task.

Although our method is less affected by obfuscation compared to other methods, it still cannot detect some obfuscated samples correctly. Our approach does not completely eliminate the threat of malicious samples evading detection through obfuscation. The features we extract do not identify the essential difference between benign and malicious very clearly. In future improvements, the essential features of these abstractions can be further clarified in combination with other static and dynamic analysis techniques.

Moreover, our approach is certainly not resistant to all obfuscation techniques. For example, Romano et al. [50] propose Wobfuscator, which leverages WebAssembly to evade static JavaScript malware detection. For malware that uses WebAssembly, *JSRevealer* is likely to fail to detect it because important code segments are missing. Since *JSRevealer* is based on static analysis of the script's own code, it is not capable of detecting malicious behaviors that leverage the external import of JavaScript, either. Additionally, *JSRevealer* dives into the semantics of the code based on enhanced AST to obtain more abstract and more relevant features to the nature of benign and malicious. If more sophisticated obfuscation strategies cause deep damage to the syntactic structure and data flow of the code, it may be unavoidable that *JSRevealer*

will be affected. For example, compared to the performance on other obfuscated samples, *JSRevealer* has higher FPR and FNR on the samples obfuscated by *JSObfu*. *JSObfu* takes an iterative obfuscation technique (the number of iterations we use is three), which may be able to corrupt the structure of enhanced AST more deeply than other obfuscators, thus causing *JSRevealer* to extract more corrupted features on such obfuscated samples. In future work, we consider fusing multiple code representations such as *Data Flow Graph* to make *JSRevealer* also work well on scripts that use very complex obfuscation strategies.

VI. Related works

Up to now, many methods have been proposed to detect malicious JavaScript, which can be roughly divided into two categories: dynamic analysis-based and static analysis-based.

Methods based on dynamic analysis: Dynamic analysis-based approaches leverage information from program execution to detect malicious behavior. Cova et al. [1] presented an approach that combines anomaly detection with emulation to automatically detect malicious JavaScript code. Kolbitsch et al. [6] proposed a JavaScript multiexecution virtual machine, ROZZLE, to explore multiple execution paths within a single execution, thus exposing malicious behavior. Invernizzi et al. [7] presented EVILSEED, which uses an initial seed of known, malicious web pages to automatically generate search engine queries to identify other malicious pages. Xue et al. [8] used *Deterministic Finite Automaton* (DFA) to abstract and summarize common behaviors of malicious JavaScript of the same attack type. Kim et al. [9] proposed J-FORCE, which explores all possible execution paths by mutating the outcomes of branch predicates to detect malicious behaviors. Sarker et al. [10] investigated the nature of JavaScript obfuscation through its concealing effect on JavaScript browser API features.

Methods based on static analysis: Some works use lexical, syntactic, and semantic information extracted from JavaScript code to identify malicious scripts. Curtsinger et al. [12] proposed ZOZZLE, using hierarchical features of the JavaScript AST to detect malware. Laskov et al. [14] presented a technique to detect JavaScript-bearing malicious PDF documents based on lexical analysis. Wang et al. [16] implemented JDSC to detect JavaScript malware using features of lexical analysis, program structures, and risky function calls. Seshagiri et al. [17] proposed AMA to detect malicious code through static code analysis of web pages. Fass et al. [20] presented JAST, which uses the extraction of features from the AST to detect malicious JavaScript. Later they proposed JStap [21], which extends the detection capability of existing lexical and AST-based methods by also leveraging CFG and PDG of the code. Alazab et al. [51] employed several features and machine-learning techniques to detect obfuscation in JavaScript and then classify the code as benign or malicious.

Some work uses the characteristics of specific attacks to extract targeted features. Rieck et al. [11] presented CUJO for the detection of drive-by-download attacks. They extracted n-gram features from both static and dynamic analysis. Canali et al. [13] implemented Prophiler, using static analysis features including features derived from the HTML contents of web pages, the associated JavaScript code, and corresponding URLs to detect malicious web pages. Xu et al. [15] proposed JStill to defend against obfuscated malicious JavaScript code, based on the static analysis of function invocation. Stock et al. [18] presented KIZZLE for finding exploit kits by generating anti-virus signatures. Kar et al. [19] presented an approach to detect injection attacks by modeling SQL queries as a graph of tokens and using the centrality measure of nodes as features.

There are other methods that use deep learning to conduct feature learning [22]–[26]. Wang et al. [22] presented a deep learning-based method to extract features from JavaScript code directly. Ndichu et al. [26] used Doc2Vec to conduct feature learning for AST of JavaScript code. Fang et al. [23] relied on Bi-LSTM networks and syntactic unit sequences from AST to detect malicious JavaScript. Later they presented the method using the program dependency graph and graph neural network [25]. Huang et al. [24] introduced Word2Vec and two *Bidirectional Long Short-Term Memory* (Bi-LSTM) layers to conduct feature learning and leveraged TextCNN for classification.

While some works claim that their methods are able to detect obfuscated malicious code [15], [21], [23]–[26], [51], it is not clear whether the code that is correctly identified by their methods is still correctly identified after obfuscation. The goal of our approach is that the code will still be recognized properly after the transformation of obfuscation, which is the biggest advantage over other approaches.

VII. Conclusion

Obfuscation techniques are now commonly used in JavaScript, which will undoubtedly affect the existing detectors based on static analysis. In this paper, we propose a more robust method against obfuscation, which uses more abstract and essential features to detect malicious JavaScript. Experimental results show that our method not only performs well on unobfuscated data but also achieves relatively good performance on data obfuscated by four commonly used obfuscation tools. Meanwhile, compared with the other four most influential detectors, our method performs significantly better on the obfuscated data. Our proposed method for extracting features provides a novel perspective for understanding benign and malicious. We believe it better reflects the nature of benign and malicious, making the detector less susceptible to obfuscation.

References

[1] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*, 2010, pp. 281–290.

[2] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, 2018, pp. 1714–1730.

[3] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee, "Understanding malvertising through ad-injecting browser extensions," in *Proceedings of the 24th International Conference on World Wide Web (WWW'15)*, 2015, pp. 1286–1295.

[4] M. Alsharnouby, F. Alaca, and S. Chiasson, "Why phishing still works: User strategies for combating phishing attacks," *International Journal of Human-Computer Studies*, vol. 82, pp. 69–82, 2015.

[5] G. Hong, Z. Yang, S. Yang, L. Zhang, Y. Nan, Z. Zhang, M. Yang, Y. Zhang, Z. Qian, and H. Duan, "How you get shot in the back: A systematical study about cryptojacking in the real world," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, 2018, pp. 1701–1713.

[6] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP'12)*, 2012, pp. 443–457.

[7] L. Invernizzi, P. M. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna, "Evilseed: A guided approach to finding malicious web pages," in *Proceedings of the 33rd IEEE symposium on Security and Privacy (SP'12)*, 2012, pp. 428–442.

[8] Y. Xue, J. Wang, Y. Liu, H. Xiao, J. Sun, and M. Chandramohan, "Detection and classification of malicious javascript via attack behavior modelling," in *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA'15)*, 2015, pp. 48–59.

[9] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-force: Forced execution on javascript," in *Proceedings of the 26th International Conference on World Wide Web (WWW'17)*, 2017, pp. 897–906.

[10] S. Sarker, J. Jueckstock, and A. Kapravelos, "Hiding in plain site: Detecting javascript obfuscation through concealed browser API usage," in *Proceedings of the ACM Internet Measurement Conference (IMC'20)*, 2020, pp. 648–661.

[11] K. Rieck, T. Krueger, and A. Dewald, "Cujo: efficient detection and prevention of drive-by-download attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*, 2010, pp. 31–39.

[12] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "ZOZZLE: fast and precise in-browser javascript malware detection," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[13] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th International Conference on World Wide Web (WWW'11)*, 2011, pp. 197–206.

[14] P. Laskov and N. Šrndić, "Static detection of malicious javascript-bearing PDF documents," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*, 2011, pp. 373–382.

[15] W. Xu, F. Zhang, and S. Zhu, "Jstill: mostly static detection of obfuscated malicious javascript code," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY'13)*, 2013, pp. 117–128.

[16] J. Wang, Y. Xue, Y. Liu, and T. H. Tan, "JSDC: A hybrid approach for javascript malware detection and classification," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '15)*, 2015, pp. 109–120.

[17] P. Seshagiri, A. Vazhayil, and P. Sriram, "Ama: static code analysis of web page for the detection of malicious scripts," *Procedia Computer Science*, vol. 93, pp. 768–773, 2016.

[18] B. Stock, B. Livshits, and B. Zorn, "Kizzle: A signature compiler for detecting exploit kits," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*, 2016, pp. 455–466.

[19] D. Kar, S. Panigrahi, and S. Sundararajan, "Sqligot: Detecting SQL injection attacks using graph of tokens and SVM," *Computers & Security*, vol. 60, pp. 206–225, 2016.

[20] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "Jast: Fully syntactic detection of malicious (obfuscated) javascript," in *Proceedings of the 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'18)*, 2018, pp. 303–325.

[21] A. Fass, M. Backes, and B. Stock, "Jstap: a static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*, 2019, pp. 257–269.

[22] Y. Wang, W.-d. Cai, and P.-c. Wei, "A deep learning approach for detecting malicious javascript code," *Security and Communication Networks*, vol. 9, no. 11, pp. 1520–1534, 2016.

[23] Y. Fang, C. Huang, Y. Su, and Y. Qiu, "Detecting malicious javascript code based on semantic analysis," *Computers & Security*, vol. 93, p. 101764, 2020.

[24] Y. Huang, T. Li, L. Zhang, B. Li, and X. Liu, "Jscontana: Malicious javascript detection using adaptable context analysis and key feature extraction," *Computers & Security*, vol. 104, p. 102218, 2021.

[25] Y. Fang, C. Huang, M. Zeng, Z. Zhao, and C. Huang, "Jstrong: Malicious javascript detection based on code semantic representation and graph neural network," *Computers & Security*, vol. 118, p. 102715, 2022.

[26] S. Ndichu, S. Kim, S. Ozawa, T. Misu, and K. Makishima, "A machine learning approach to detection of javascript-based attacks using AST features and paragraph vectors," *Applied Soft Computing*, vol. 84, p. 105721, 2019.

[27] M. Moog, M. Demmel, M. Backes, and A. Fass, "Statically detecting javascript obfuscation and minification techniques in the wild," in *Proceeding of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'21)*, 2021, pp. 569–580.

[28] (2022) javascript-obfuscator: A powerful obfuscator for javascript and node.js. [Online]. Available: https://github.com/javascript-obfuscator/javascript-obfuscator/

[29] (2022) jfogs. [Online]. Available: https://github.com/zswang/jfogs/

[30] (2022) Jsobfu: Obfuscate javascript (beyond repair) with ruby. [Online]. Available: https://github.com/rapid7/jsobfu/

[31] (2022) Jshaman. [Online]. Available: https://www.jshaman.com/

[32] W3Techs. (2022) Usage statistics of javascript as client-side programming language on websites. [Online]. Available: https://w3techs.com/technologies/details/cp-javascript/

[33] (2022) Alexa top websites. [Online]. Available: https://www.expireddomains.net/alexa-top-websites/

[34] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.

[35] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[36] (2022) Esprima. [Online]. Available: https://esprima.org/

[37] (2022) Metaod: Automating outlier detection via meta-learning. [Online]. Available: https://github.com/yzhao062/MetaOD/

[38] Y. Zhao, R. Rossi, and L. Akoglu, "Automating outlier detection via meta-learning," *arXiv preprint arXiv:2009.10606*, 2020.

[39] C. Lemke, M. Budka, and B. Gabrys, "Metalearning: a survey of trends and technologies," *Artificial Intelligence Review*, vol. 44, no. 1, pp. 117–130, 2015.

[40] (2022) Pyod: A comprehensive and scalable python library for outlier detection. [Online]. Available: https://github.com/yzhao062/pyod/

[41] (2022) scikit-learn: Machine learning in python. [Online]. Available: https://scikit-learn.org/stable/

[42] (2022) Javascript malware collection. [Online]. Available: https://github.com/HynekPetrak/javascript-malware-collection/

[43] (2022) Malicious javascript dataset. [Online]. Available: https://github.com/geeksonsecurity/js-malicious-dataset/

[44] (2022) VirusTotal. [Online]. Available: https://www.virustotal.com/

[45] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," *ACM Sigplan Notices*, vol. 51, no. 1, pp. 761–774, 2016.

[46] P. Skolka, C.-A. Staicu, and M. Pradel, "Anything to hide? studying minified and obfuscated code in the web," in *Proceedings of the 28th International Conference on World Wide Web (WWW'19)*, 2019, pp. 1735–1746.

[47] (2022) syntactic-jsdetector. [Online]. Available: https://github.com/Aurore54F/syntactic-jsdetector/

[48] (2022) Jast - js ast-based analysis. [Online]. Available: https://github.com/Aurore54F/JaSt/

[49] (2022) Jstap: A static pre-filter for malicious javascript detection. [Online]. Available: https://github.com/Aurore54F/JStap/

[50] A. Romano, D. Lehmann, M. Pradel, and W. Wang, "Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP'22)*, 2022, pp. 1574–1589.

[51] A. Alazab, A. Khraisat, M. Alazab, and S. Singh, "Detection of obfuscated malicious javascript code," *Future Internet*, vol. 14, no. 8, p. 217, 2022.