An Empirical Study on the Effects of Obfuscation on Static Machine Learning-based Malicious JavaScript Detectors

Kunlun Ren[*]
Huazhong University of Science and Technology
Wuhan, China
kunlunren@hust.edu.cn

Weizhong Qiang[*][†]
Huazhong University of Science and Technology
Wuhan, China
wzqiang@hust.edu.cn

Yueming Wu[‡]
Nanyang Technological University
Singapore
wuyueming21@gmail.com

Yi Zhou[*]
Huazhong University of Science and Technology
Wuhan, China
yi_zhou@hust.edu.cn

Deqing Zou[*][†]
Huazhong University of Science and Technology
Wuhan, China
deqingzou@hust.edu.cn

Hai Jin[§][†]
Huazhong University of Science and Technology
Wuhan, China
hjin@hust.edu.cn

ABSTRACT

Machine learning is increasingly being applied to malicious JavaScript detection in response to the growing number of Web attacks and the attendant costly manual identification. In practice, to hide their malicious behaviors or protect intellectual copyrights, both malicious and benign scripts tend to obfuscate their own code before uploading. While obfuscation is beneficial, it also introduces some additional code features (*e.g.,* dead code) into the code. When machine learning is employed to learn a malicious JavaScript detector, these additional features can affect the model to make it less effective. However, there is still a lack of clear understanding of how robust existing machine learning-based detectors are on different obfuscators.

In this paper, we conduct the first empirical study to figure out how obfuscation affects machine learning detectors based on static features. Through the results, we observe several findings: 1) Obfuscation has a significant impact on the effectiveness of detectors, causing an increase both in *false negative rate* (FNR) and *false positive rate* (FPR), and the bias of obfuscation in the training set induces detectors to detect obfuscation rather than malicious behaviors. 2) The common measures such as improving the quality of the training set by adding relevant obfuscated samples and leveraging state-of-the-art deep learning models can not work well. 3) The root cause of obfuscation effects on these detectors is that feature spaces they use can only reflect shallow differences in code, not about the nature of benign and malicious, which can be easily affected by the differences brought by obfuscation. 4) Obfuscation has a similar effect on realistic detectors in VirusTotal, indicating that this is a common real-world problem.

CCS CONCEPTS

• **Security and privacy** → *Malware and its mitigation.*

KEYWORDS

web security, JavaScript obfuscation, machine learning, malicious JavaScript detector

ACM Reference Format:
Kunlun Ren, Weizhong Qiang, Yueming Wu, Yi Zhou, Deqing Zou, and Hai Jin. 2023. An Empirical Study on the Effects of Obfuscation on Static Machine Learning-based Malicious JavaScript Detectors. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598146

1 INTRODUCTION

The Web is the dominant software platform and the primary tool for billions of users worldwide to interact with the Internet. At the same time, it has naturally become one of the main targets of attacks [25, 29, 50, 69]. In recent years, the number of web attacks has doubled year after year, and the attacks tend to be diversified and decentralized [18]. JavaScript, a language used by the majority of the world's websites, is often used by attackers to target web users [26, 27, 31, 38, 44, 74]. JavaScript malicious code poses a significant security risk. Therefore, academia and industry are paying more and more attention to the detection of JavaScript malicious code.

Current methods for JavaScript malicious code detection 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 [29, 44, 47, 49, 62, 72], but the identifiable nature of the analysis environment results in malicious code being able to evade detection through inspection of the environment. Static analysis, on the other hand, consumes

[*]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
[†]Jinyinhu Laboratory, Wuhan, China
[‡]Corresponding author
[§]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, HUST, Wuhan, 430074, China

fewer resources, is simple and fast to detect malicious code, and is more cost-effective [30, 35, 36, 59, 65]. 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 [30, 33, 35, 36, 42, 59, 61, 68]. However, many malicious scripts employ obfuscation to evade static analysis [53, 64, 70]. In addition, obfuscation techniques are also used for benign scripts to protect intellectual property [53, 64], which can bias machine learning methods.

It is widely believed that obfuscation can seriously affect static analysis. However, machine learning based on features extracted by static analysis also performs well on obfuscated samples [33, 35, 36, 42, 61, 68]. We believe that it may be that the different obfuscation distributions of benign and malicious samples make machine learning rely on features related to obfuscation for classification. However, they do not or only briefly discuss the impact when applying machine learning. We consider that obfuscation must have a significant impact on machine learning due to its own characteristics, which seriously hinders its application in reality. Thus we explore how obfuscation actually affects machine learning. Specifically, we aim to answer four research questions (RQs):

- *RQ1: What impact does obfuscation have on static machine learning malicious JavaScript detectors?*
- *RQ2: Are the commonly used measures to mitigate the impact of obfuscation effective?*
- *RQ3: What is the root cause of obfuscation affecting static machine learning malicious JavaScript detectors?*
- *RQ4: How does obfuscation affect real-world static malicious JavaScript detectors?*

As necessary preparation works before the experiments, we create the original dataset and obfuscated dataset. The original dataset is similar to the composition of the dataset in previous work [60]. The malicious samples come from Hynek Petrak [9], GeeksOnSecurity [15], and VirusTotal [22]. The benign samples come from the 150k JavaScript Dataset [57]. Since previous work has validated these data, we consider them to be ground truth. Then we create the obfuscated dataset by obfuscating all the samples in the original dataset with six common obfuscation techniques.

To answer RQ1, we evaluate the performance of four detectors and five machine learning models on obfuscated samples, as well as the performance of the detectors when the training data shows a bias about obfuscation type. To answer RQ2, we study the effectiveness of improving the quality of the dataset and extracting features using deep learning models to mitigate the effects of obfuscation. To answer RQ3, we find the root cause by visualizing the distribution of vectors, analyzing the feature spaces, and investigating the distances between vector sets. To answer RQ4, we perform an experimental investigation by submitting benign and malicious samples obfuscated by different obfuscators to VirusTotal for scanning.

Through the results of our experiments, we find that obfuscation has a serious impact on static machine learning malicious JavaScript detectors. These detectors can not work well on obfuscated samples. Furthermore, when there is a bias about obfuscation between the two classes in the training set, these detectors tend to detect obfuscation rather than malicious behaviors. What's worse, the common measures to mitigate obfuscation such as improving the quality of the dataset and introducing deep learning models are not effective. The root cause is that the feature spaces of existing detectors can only reflect shallow differences in code, not about the nature of benign and malicious, which can be easily affected by the differences brought by obfuscation. Moreover, we find that obfuscation has a similar effect on real-world static malicious JavaScript detectors.

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

- We conduct exhaustive experiments to investigate the effect of obfuscation on malicious JavaScript machine learning detectors based on static analysis features. We find that obfuscation can significantly interfere with the detector's determination, the bias of obfuscation in the training set induces the detectors to detect obfuscation rather than malicious behaviors.
- We explore the root cause of obfuscation affecting static machine learning malicious JavaScript detectors by vector visualization, feature spaces analysis, and distances between vector sets investigation. We find that the feature spaces of existing detectors can only reflect shallow differences in code, not about the nature of benign and malicious, which can be easily affected by the differences brought about by obfuscation.
- We evaluate obfuscation on real-world detectors using VirusTotal as a representative. The results show that obfuscation has a similar effect on these detectors, which allows malicious samples to evade detection.

The remainder of this paper is organized as follows. Section 2 presents our motivation to conduct this research. Section 3 introduces the background of JavaScript code transformation techniques and machine-learning-based malicious JavaScript detection. Section 4 describes the design of our study. Section 5 presents our experiments and the results. Section 6 discusses some inspiration from our findings and some limitations of this paper. Section 7 introduces some related works. Section 8 concludes this paper.

2 MOTIVATION

Machine learning methods based on static analysis have been applied to the detection of malicious JavaScript samples with high accuracy [30, 33, 35, 36, 42, 59, 61]. However, malicious JavaScript generally utilizes obfuscation techniques to disguise its malicious behavior.

It is tempting to assume that machine-learning-based detectors achieve such results by relying on features that distinguish between obfuscated and unobfuscated scripts. For example, Fass et al. [35] propose a static machine-learning-based approach, which leverages lexical, *abstract syntax tree* (AST), control and data flow information, achieving over 99% accuracy on their dataset. As they mention, their malicious dataset comes from the German Federal Office for Information Security [6], Hynek Petrak [9], Kafeine DNC [17], GeeksOnSecurity [15], and VirusTotal [22]. Most of these samples are obfuscated. Their benign dataset comes from Tranco top 10,000 websites [55], Microsoft products, open source games, web frameworks, and Atom [1]. As they describe, over 40% of the samples obtained from the web pages are either minified or obfuscated according to Skolka et al. [64] and the majority of benign JavaScript from Microsoft products are also obfuscated. Nevertheless, according to Moog et al. [53], there is a difference between the obfuscation techniques used for these samples and those used for malicious samples, while obfuscated benign samples from Microsoft in the dataset of Fass et al. represent a relatively small percentage of the overall quantity. Furthermore, as also mentioned in [53], the way script obfuscation is evolving, these changes are likely to affect machine learning methods that rely on training data.

Unfortunately, Fass et al. [35] and other related work [30, 33, 36, 42, 59, 61, 68] did not consider or only briefly discussed the effects of obfuscation. This situation leads us to fully investigate the effects of obfuscation on static malicious JavaScript detectors based on machine learning.

3 BACKGROUND

In this section, we briefly introduce common JavaScript code transformation techniques and machine-learning-based static malicious JavaScript detectors.

3.1 JavaScript Code Transformations

Code transformations are commonly used to hide the code logic in JavaScript, which includes obfuscation and minification.

Obfuscation makes code difficult to understand and analyze. The common obfuscation techniques can be summarized as follows [16, 45, 70]:

- **Variable obfuscation** randomly turns meaningful variable, method, and constant names into meaningless gibberish-like strings, reducing code readability, such as to single characters or hexadecimal strings.
- **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. We refer to it and variable obfuscation collectively as *data obfuscation*.
- **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, 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.
- **Self defending** inserts self-testing code into the script to prevent formatting and variable renaming operations. If the code is formatted, it will not run properly.
- **Polymorphic mutation** makes JavaScript code automatically mutate 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.

Minification refers to minimizing the code in a script file, which reduces the readability of the code and of course improves the loading speed of the website at the same time. Common minification techniques include removing unnecessary spaces, line breaks, etc. from the code, or processing potentially publicly available code for sharing, compressing, and converting some call logic into a few lines of code [4, 11]. In a broader sense, it can also be seen as a kind of obfuscation. In the following, we collectively refer to these techniques as obfuscation techniques.

We transform JavaScript code based on the following commonly used obfuscation and minification tools:

- **JavaScript-Obfuscator** [10] is a powerful and free obfuscator for JavaScript, containing a variety of features.
- **gnirts** [7] is used to obfuscate string literals in JavaScript code. It mangles string literals by using some code instead of hexadecimal escape.
- **JSObfu** [12] is a JavaScript obfuscator written in Ruby, which randomizes as much as possible and removes string constants that are easily-signaturable.
- **JavaScript Minifier** [19] is an easy-to-use tool for minifying JavaScript code.
- **Google closure compiler** [3] is a tool for making JavaScript download and run faster, which contains a variety of advanced optimization ways to minify JavaScript code.

We select the five obfuscation tools as they are publicly available, popularity, and recognized by other studies. The five tools are very convenient to use. *JavaScript-Obfuscator*, *gnirts*, and *JSObfu* are open source projects on github with high stars. *JavaScript Minifier* and *Google closure compiler* are developed by the famous companies **Toptal** and **Google**, which are widely used by developers. Moreover, these tools are recognized by other researchers. They have been used by works [53, 64] published in top conferences.

Based on these tools, we generalize six obfuscation techniques, which are *data obfuscation*, *control flow flattening*, *dead code injection*, *self defending*, *debug protection*, and *minification*. For some tools that implement the same technique, we use only one of them.

3.2 Malicious JavaScript Detection based on Static Analysis and Machine Learning

Nowadays, machine learning has been applied to malicious JavaScript detection and has shown good effectiveness.

Rieck et al. [59] present *CUJO* for automatic detection of drive-by-download attacks. Both static and dynamic features of JavaScript files on a website are extracted by using n-grams, and *Support Vector Machines* (SVM) is applied for the detection, achieving 94% accuracy in detecting drive-by download attacks. Curtsinger et al. [30] propose *ZOZZLE*, utilizing hierarchical features of the JavaScript abstract syntax tree and Bayesian detector to detect JavaScript malware. They claim that *ZOZZLE* has a low false positive rate of 0.00003%. Canali et al. [27] implement *Prophiler*, detecting malicious web pages by leveraging features derived from the HTML contents of web pages, the associated JavaScript code, and corresponding URLs. Laskov et al. [51] detect JavaScript-bearing malicious PDF documents based on the lexical analysis. Xu et al. [71] propose

JStill to detect obfuscated malicious JavaScript based on the static analysis of function invocation. Wang et al. [66] implement *JDSC*, which use features of lexical analysis, program structures, and risky function calls to detect JavaScript malware. Seshagiri et al. [63] propose *AMA* to detect malicious code through static code analysis of web pages. Stock et al. [65] present *KIZZLE*, leveraging anti-virus signatures for exploit kits. Kar et al. [46] model SQL queries as a graph of tokens and use the centrality measure of nodes as features to detect SQL attack. Fass et al. [36] present *JAST* that uses n-grams of sequential nodes produced from AST to identify patterns indicative of malicious behavior. They evaluate different classifiers and determine that random forest is the best, with a high detection accuracy of 99.5%. Afterwards, Fass et al. propose *JSTAP* [35], a modular static malicious JavaScript detector that leverages lexical, syntax, control-flow, and data-flow information. They extract features either by constructing n-grams of nodes or by combining the AST node type with its corresponding identifier/literal value, and a random forest is used as classifier. By combining different modules, the accuracy of *JStap* can reach 99.7%. Alazab et al. [24] employ several features and machine-learning techniques to detect obfuscation in JavaScript before classifying the code as benign or malicious.

Some works leverage deep learning to conduct feature learning. Wang et al. [68] propose a framework for malicious JavaScript detection based on deep learning and logistic regression. They use stacked denoising auto encoder to extract features and logistic regression as a classifier. Ndichu et al. [54] utilizes AST of JavaScript for representation and Doc2Vec to conduct feature learning to detect JavaScript-based attacks. Fang et al. [32] use Bi-LSTM network and syntactic unit sequences from AST to detect malicious JavaScript. Huang et al. [42] present *JSContana* that uses context analysis based on dynamic word embeddings and TextCNN [73] as the classification module and achieve 99.0% in AUC-score. Rozi et al. [61] detect malicious JavaScript based on the AST features processed by a *graph convolutional neural network* (GCN) [48] and apply an attention layer to improve the performance. Fang et al. [33] propose a static detection model based on graph neural network, *JStrong*, which utilizes data flow and control flow information through PDG and learns the features through the graph neural network. Hwang et al. [43] propose a deobfuscation processing and a variant of Stacked denoising Autoencoder-Logistic Regression, which extracting features from obfuscated samples, to detect obfuscated malicious JavaScript.

While some of these works [24, 32, 33, 35, 42, 43, 54, 71] mention the detection ability on obfuscated scripts, no work discusses the impact of changes in obfuscation on the detectors.

4 METHODOLOGY

4.1 Research Questions

In this work, we aim to answer the following research questions:

- RQ1: What impact does obfuscation have on static machine learning malicious JavaScript detectors?
- RQ2: Are the commonly used measures to mitigate the impact of obfuscation effective?
- RQ3: What is the root cause of obfuscation affecting static machine learning malicious JavaScript detectors?
- RQ4: How does obfuscation affect real-world static malicious JavaScript detectors?

4.2 Datasets

4.2.1 Original Dataset. The original dataset is the one that contains samples that are directly from the wild and have not been manually transformed. Our original dataset consists of samples from different sources. The malicious samples in the original dataset consist of the malware collection of Hynek Petrak [9], exploit kits from GeeksOnSecurity [15], and the additional samples from VirusTotal [22]. The benign samples come from the 150k JavaScript Dataset published by Raychev et al. [57], consisting of 150,000 JavaScript files, and the scripts crawled from Alexa Top-10k websites, consisting of over 60,000 scripts. Although these data were collected in 2015-2017, which are not the most up-to-date data, the oldness of the data do not affect our research, as we explore the qualitative impact of obfuscation on machine learning detectors and reobfuscate these samples. The details of the original dataset are shown in Table 1.

4.2.2 Obfuscated Dataset. Even though previous studies have proposed some approaches to detect obfuscation [53, 64], we cannot be 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, we create our obfuscated dataset.

As mentioned in Section 3.1, we select six techniques based on the transformation tools to transform the samples in original dataset. We refer to these six techniques as **obfuscators** in the following. Accordingly, these transformations are referred to as **obfuscations**. We generate our obfuscated dataset by applying these six obfuscators to obfuscate all samples in the original dataset. None of these obfuscators are able to transform all samples. We select a subset of 21,000 samples each from benign and malicious samples that are successfully obfuscated for all six obfuscators. That is, our obfuscated dataset contains a total of 294,000 samples, where there are equal numbers of benign and malicious samples. The benign and malicious class each contains seven subsets of 21,000 samples that are unobfuscated and obfuscated by the six obfuscators, respectively.

Table 1: Original 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

4.3 Malicious JavaScript Detectors

The objects of our study are static machine learning-based malicious JavaScript detectors. We select four detectors, which are state-of-the-art and include a variety of static analysis techniques such as lexical analysis, AST-based analysis, CFG-based analysis, and

Table 2: FNR and FPR (%) of four detectors on obfuscated samples

Detectors	Baseline (Unobfuscated)		*Control flow flattening*		*Data obfuscation*		*Dead code injection*		*Debug protection*		*Self defending*		*Minification*		Average	
	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR
CUJO	0.3	13.5	0	92.9	0	100	0	100	0	99.7	6.9	75.2	4.7	14.3	1.9	80.4
ZOZZLE	0.6	0.6	1.2	0.8	11.7	0.8	12.2	1.1	17.9	0.1	9.0	0.2	8.0	0.7	10.0	0.6
JAST	0.2	0.2	4.4	3.4	0.1	34.5	0	34.2	0	30.2	18.5	25.6	17.4	1.3	5.9	21.5
JSTAP	0.5	0	10.9	0.2	20.3	1.7	21.8	1.2	28.1	0	23.4	0.1	29.4	0	22.3	0.5

PDG-based analysis. Moreover, they are the most influential, with relevant papers published in high quality conferences. In addition, they are open source, while other tools are not and are difficult to reproduce. Therefore, we choose these four detectors as the main objects to be evaluated in our experiments, and we believe that their performance is representative:

- *CUJO* [59] uses n-grams features from both static and dynamic analysis to detect malware. In this paper, we conduct experiments only with *CUJO*'s static part. We use the re-implementation of *CUJO* provided by Fass et al. [14] in our experiments.
- *ZOZZLE* [30] detects malicious JavaScript based on the hierarchical features of ASTs. In our experiments, we use the *ZOZZLE* re-implemented by Fass et al. [21].
- *JAST* [36] extracts n-grams features from ASTs to detect malicious JavaScript. We directly use the project available on GitHub [8].
- *JSTAP* [35] 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 experiments. We use their implementation available on GitHub [13].

The dataset used in *CUJO* consists of 220,083 URLs of *Alexa-200k* and *Surfing*, and 609 samples obtained from Cova et al. [29]. The samples in the dataset of *ZOZZLE* are obtained by scanning URLs by *NOZZLE* [56] and extracting *Alexa.com* top 50 URLs, containing 919 malicious contents and 7,976 benign contents. The dataset in *JAST* is provided by the German Federal Office for Information Security, containing JavaScript files extracted from emails, Microsoft products, games, web frameworks, and the source code of Atom. It comprises 20,246 benign and 85,059 malicious JavaScript samples. The dataset of *JSTAP* contains 131,448 malicious samples and 141,768 benign samples. Among them, malicious samples are from German Federal Office for Information Security, Hynek Petrak [9], Kafeine DNC [17], GeeksOnSecurity [15], and VirusTotal [22]. Benign samples are from Tranco top 10,000 websites [55], Microsoft products, open source games, web frameworks, and Atom. Compared to these datasets, our dataset is comparable to the dataset of *JSTAP* and superior to the datasets used in *CUJO*, *ZOZZLE*, and *JAST* in terms of size and diversity. Overall, our dataset is large enough and contains samples from diverse sources. It is sufficient to support our experiments.

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 32 GB RAM. To reduce the statistical effects of random factors, we use five-fold cross-validation when training the models, and our experimental results are obtained by running each experiment five times and then averaging the results.

5 EXPERIMENTS AND RESULTS

5.1 RQ1: What Impact Does Obfuscation Have on Static Machine Learning Malicious JavaScript Detectors?

Although it is natural to suspect that the obfuscation of source code can affect machine learning detectors base on static features, solid experimental confirmation is needed. We conduct the following experiments to figure out how obfuscation affects the machine learning detector. This is important because machine learning is increasingly being used for malicious JavaScript detection, while the impact of obfuscation is rarely discussed, which can have a detrimental effect on its practical application.

5.1.1 Detectors Performance on Obfuscated Samples. First, we consider what happens when a trained detector detects obfuscated scripts in practice. We randomly select 10,000 samples each from the malicious and benign samples in our original dataset as the training set, and 5,000 as the validation set. Then, we train four detectors with these samples based on the description in their papers and the steps provided in the open source projects. Next, the performance of four trained detectors are evaluated on 5,000 unobfuscated samples from original dataset with equal numbers of benign and malicious samples, and versions that are obfuscated by each of the six obfuscators.

We focus on the metrics commonly used to evaluate malware detectors: *false negative rate* (FNR) and *false positive rate* (FPR). A good malware detector should have both a low FNR and a low FPR under various scenarios. Intuitively, obfuscation can interfere with the detector's analysis and thus allow malicious scripts to escape detection, which can lead to an increase in the FNR. There is also the possibility that benign samples use the same obfuscation as malicious samples, causing the detector to classify benign samples as malicious, thus increasing the FPR.

Table 2 shows the FNR and FPR of four detectors evaluated on the unobfuscated samples and the six obfuscated versions. The results show that obfuscation affect different detectors in different ways. For example, *CUJO* and *JAST* have a significantly higher FPR on obfuscated samples, increasing by 66.9% and 21.3% on average

Table 3: FNR and FPR (%) of five machine learning models on obfuscated samples

Detectors	Baseline (Unobfuscated)		*Control flow flattening*		*Data obfuscation*		*Dead code injection*		*Debug protection*		*Self defending*		*Minification*		Average	
	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR
SVM	3.9	1.2	6.2	1.7	1.6	3.8	16.5	4.0	15.8	0.6	13.3	1.1	7.6	5.6	12.6	2.8
Logistic Regression	26.8	1.8	44.9	1.7	59.2	0.3	60.4	0.3	55.0	0.2	49.0	0.4	38.1	4.2	51.1	1.2
Bernoulli Naive Bayes	32.9	0.8	31.5	1.7	20.6	36.6	21.7	45.8	32.4	0.8	32.8	1.0	43.5	0.5	30.4	14.4
Decision Tree	1.1	1.6	8.8	2.2	10.3	2.8	9.2	2.3	40.6	1.7	42.0	1.7	1.5	1.9	18.7	2.1
Random Forest	0.6	0.6	1.2	0.8	11.7	0.8	12.2	1.1	17.9	0.1	9.0	0.2	8.0	0.7	10.0	0.6

compared to the baseline, while *ZOZZLE* and *JSTAP* achieve a significantly higher FNR, increasing by 9.4% and 21.8% on average compared to the baseline. Moreover, different obfuscators have different effects on the same detector. Some obfuscators have a large impact and some produce a less pronounced impact. However, they affect the same detector with the same bias. For example, all six obfuscators increase *JSTAP*'s FNR significantly, while having little effect on its FPR. In general, obfuscation can seriously interfere with the detection of the detector.

5.1.2 Different Machine Learning Algorithms. The feature extraction methods for these four detectors are different, but the machine learning algorithms they used are all *Random Forest*. We wonder if different machine learning algorithms will perform differently. From Table 2 we can see that *ZOZZLE* performs best on obfuscated samples, so we use the method of *ZOZZLE* to extract features and then apply different machine learning algorithms utilizing the implementation in a Python library, *scikit-learn* [20], in the final classification stage, including *Support Vector Machine* (*SVM*), *Logistic Regression*, *Bernoulli Naive Bayes*, and *Decision Tree*.

Table 3 shows the FNR and FPR of different machine learning algorithms based on the features extracted by *ZOZZLE* running on the unobfuscated and obfuscated samples. The results illustrate that the bias affected by obfuscation is the same for different machine learning algorithms. Here all of them mainly are affected in terms of the FNR mainly. Moreover, *Random Forest* performs the best. In the following, we consider that different machine learning algorithms perform with the same characteristics by default and do not discuss a specific machine learning algorithm any more.

5.1.3 Biased Training Sets. In the practical application of machine learning detectors, there will also be a bias in the training data. For example, benign samples are mostly unobfuscated, while malicious samples are mostly obfuscated, or both benign and malicious samples are obfuscated, but in different ways. This makes us wonder whether the detector could be affected by the bias of the obfuscation in the training set, causing the detector to actually detect obfuscation instead of detecting malicious behaviors. Here we consider two extreme scenarios, i.e., a training set with all unobfuscated benign samples and all obfuscated malicious samples, and a training set with all obfuscated benign samples and all unobfuscated malicious samples. The detectors trained with such training sets are used to detect unobfuscated benign samples, obfuscated benign samples, unobfuscated malicious samples, and obfuscated malicious samples, respectively. The size of the training set is still 20,000 samples with an equal number of benign and malicious samples, and the number

Table 4: Average accuracy (%) of detectors trained on biased datasets detecting unobfuscated and obfuscated samples

Detectors	Bias in training set	Test samples: Unobfuscated Benign	Obfuscated Benign	Unobfuscated Malicious	Obfuscated Malicious
CUJO	Malicious Obfuscated	88.2	14.3	82.6	100
	Benign Obfuscated	20.5	94.7	94.8	34.9
ZOZZLE	Malicious Obfuscated	99.6	19.8	55.4	100
	Benign Obfuscated	32.8	99.9	100	53.7
JAST	Malicious Obfuscated	100	16.7	23.6	100
	Benign Obfuscated	20.0	99.9	100	25.6
JSTAP	Malicious Obfuscated	100	18.6	14.0	100
	Benign Obfuscated	22.4	100	100	14.3

of test samples in each category is 5,000. Since we are identifying a single category of samples, here we choose the metric accuracy.

Table 4 shows the average accuracy of the detector trained with the bias dataset detecting unobfuscated benign samples, obfuscated benign samples, unobfuscated malicious samples, and obfuscated malicious samples. Note that the samples for testing use the same obfuscator as in the training set, and the accuracy is obtained by averaging the accuracy of detecting the six kinds of obfuscated samples. The results show a consistent performance of the four detectors. When only malicious samples are obfuscated in the training set, detectors classify unobfuscated benign samples and obfuscated malicious samples with high accuracy, and classify obfuscated benign samples and unobfuscated malicious samples with low accuracy. The opposite is true when only benign samples are obfuscated in the training set. This confirms our conjecture that the detector will use obfuscation as its basis for classification when obfuscation appears significantly biased in the training set.

From the above experiments, we can conclude that obfuscation can have a serious impact on malicious JavaScript detectors based on static analysis and machine learning. These detectors show a significant increase in FNR and FPR on obfuscated samples. Moreover, the detectors will tend to detect obfuscation rather than malicious behaviors when there is a clear bias about obfuscation between benign and malicious samples in the dataset for training these detectors.

Finding 1: The malicious JavaScript detectors based on static analysis and machine learning perform poorly on obfuscated samples, with a significant increase in FNR and FPR, and the bias of obfuscation in the training set induces the detectors to detect obfuscation rather than malicious behaviors.

5.2 RQ2: Are the Common Measures to Mitigate the Impact of Obfuscation Effective?

From the above experiments we can see that obfuscation does interfere with the detectors significantly. Mitigating the effect of obfuscation on the detectors can be seen as improving the robustness of the detectors. For machine learning-based detectors, it is natural to think of two ways to improve their robustness: improving the quality of the dataset and optimizing the method of extracting features, which are common ideas. Correspondingly, we consider two approaches: one is to add similar obfuscated samples to the training set, and the other is to use the currently hot deep learning methods to extract features.

5.2.1 Training and Testing Detectors on Samples with Same Types of Obfuscation. Here we evaluate four detectors in an extreme scenario, where the data from the training and test sets are all obfuscated by the same obfuscator. The sizes of the training and test sets are kept the same as in the previous experiments.

Table 5 shows the results of detecting the same kind of obfuscated samples with a detector trained on the obfuscated samples. We can see a significant improvement in the performance of all four detectors compared to the detectors trained on all unobfuscated samples, even comparable to baseline. The only exception is *CUJO*, which has a 13.4% increase in FNR compared to baseline, but still has a significant performance improvement compared to training on unobfuscated samples. This indicates that adding the same type of obfuscated data when training the detector helps to improve the performance of the detector in detecting obfuscated data.

5.2.2 Training and Testing Detectors on Samples with Different Types of Obfuscation. Our experimental setup above is a very ideal scenario. It is natural to think that this may be a palliative, not a cure for the problem, i.e., once the obfuscation type of the samples to be analyzed is different from that in the training set, the detector may once again be ineffective. To verify this we evaluate the performance of the four detectors in scenarios with different types of obfuscation in the training and test sets.

Figure 1(c) shows the FNR and FPR of the four detectors when the data in the training and test sets are of different obfuscation types. The FNR and FPR are obtained by averaging the results of various different obfuscation type combinations (30 combinations per detector). The results are consistent with our guess. The detector returns to the performance when trained on unobfuscated samples and detecting obfuscated samples. From Figure 1(a) and Figure 1(c) we can see that the FNR and FPR of each detector are very similar in the two scenarios.

5.2.3 BERT Variants. In the following we evaluate the detectors after introducing deep learning methods. Deep learning approaches have made impressive progress in code representation, especially BERT variants, which has been able to generate fairly robust representations of the semantics of code. We select three state-of-art influential BERT variants for programming language, which are CodeBERT [37], GraphCodeBERT [40], and UniXcoder [39]. We generate the code representation directly using the pre-trained models Microsoft provides [5]. Then *Random Forest* model is used as the classifier, which is in consistent with the above experiments. From Table 6, we find that these three models exhibit very similar characteristics on obfuscated samples, mainly an increase in the FPR. They also exhibit similar characteristics on the same type of obfuscated samples, with a close increase FPR. On average, there is only a small increase in the FNR of the three models on the obfuscated samples compared to baseline, while a significant increase in the FPR, which increases by 77.2%, 75,6%, and 69.0%, respectively. This indicates that deep learning-based models like BERT variants are also unable to address the impact of obfuscation and are even more affected than the four detectors.

From the above experiments, we can conclude that among the common measures to mitigate obfuscation impact, improving the quality of the dataset, such as adding the same type of obfuscated samples to the dataset used to train the detector, can improve the performance of the detector on the obfuscated data. However, it is a palliative, not a cure for the problem brought by obfuscation. The detectors still does not work well with different types of obfuscation in the training and test sets. As for deep learning related methods, such as BERT variants, we do not see a trend that would solve this problem and they can be even more severely affected.

Finding 2: The common measures to mitigate the impact of obfuscation can not work well. Improving the quality of the dataset is just a palliative for the problem. Deep learning models do not appear to solve this problem effectively either, and are even more affected by obfuscation.

5.3 RQ3: What Is the Root Cause of Obfuscation Affecting Static Machine Learning Malicious JavaScript Detectors?

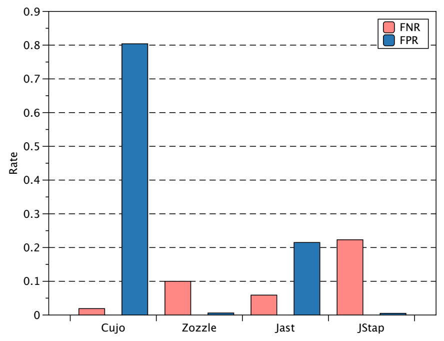
In this RQ, we explore the root cause of obfuscation affecting malicious JavaScript detectors that are based on static analysis and machine learning.

First let us revisit these four detectors we choose. *CUJO* utilizes lexical analysis. *ZOZZLE* and *JAST* analyse the code based on AST. *JSTAP* leverages not only lexical analysis and AST, but also control flow and data flow information. *CUJO* directly uses the features after lexical analysis, and the other three filter the features based on frequency. Although there are differences, we can consider them as the same class of methods. We consider this way of extracting features, both lexical analysis and pipelines based on AST, as well as leveraging control flow and data flow information, can be easily messed up by obfuscation.

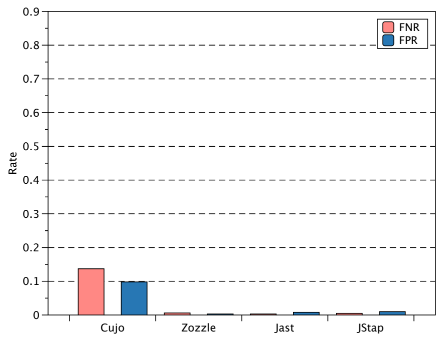
5.3.1 Vectors Visualization. To show the interference of obfuscation on the features extracted by the detector more intuitively, we choose *JSTAP* trained on unobfuscated samples as a representative to visualize the corresponding feature vectors of the test samples in the classification phase. The visualization is performed by first normalizing the high-dimensional vector features so that the sizes are all in the same range. Then PCA is used to reduce the dimensionality to low dimensions, and the t-SNE is used to reduce it to

Table 5: FNR and FPR (%) of four detectors trained and tested on same type of obfuscated samples

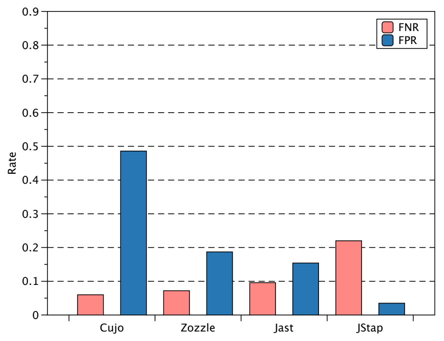
Detectors	*Control flow flattening*		*Data obfuscation*		*Dead code injection*		*Debug protection*		*Self defending*		*Minification*		Average	
	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR
CUJO	5.3	7.2	20.9	7.3	20.5	9.7	21.9	14.1	12.7	10.5	0.6	9.8	13.7	9.8
ZOZZLE	0.8	0.4	0.9	0.2	0.4	0.2	0.6	0.1	0.4	0.2	0.4	0.9	0.6	0.3
JAST	0.5	0.3	0.3	1.5	0.3	1.7	0.2	0.5	0.2	0.3	0.3	0.4	0.3	0.8
JSTAP	0.9	0	0.7	0.7	0.5	0.5	0.2	3.9	0.4	1.1	0.4	0	0.5	1.0



(a) Trained on unobfuscated samples and tested on obfuscated samples



(b) Trained and tested on same type of obfuscated samples



(c) Trained and tested on different types of obfuscated samples

Figure 1: Average FNR and FPR (%) of four detectors trained on unobfuscated samples and tested on obfuscated samples, trained and tested on same type of obfuscated samples, and trained and tested on different types of obfuscated samples

Table 6: FNR and FPR (%) of three BERT variants on obfuscated samples

Model	Baseline (Unobfuscated)		*Control flow flattening*		*Data obfuscation*		*Dead code injection*		*Debug protection*		*Self defending*		*Minification*		Average	
	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR
CodeBERT	0.3	0.3	1.4	81.6	0.1	98.2	0.1	98.4	0	97.6	0.1	85.4	11.8	3.7	2.3	77.5
GraphCodeBERT	0.1	0.3	1.0	81.5	0	99.2	0	99.2	0	97.8	0.9	68.5	8.6	9.4	1.8	75.9
UniXcoder	0.1	0.4	3.2	71.6	0.2	96.8	0.1	98.2	0.1	85.8	1.9	60.8	9.5	3.0	2.5	69.4

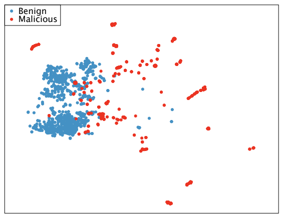
two dimensions to plot it out. The reason for applying both PCA and t-SNE is that PCA is faster, while the visualization result after t-SNE dimensionality reduction is more intuitive. Combining the two for two-step dimensionality reduction balances the speed and visualization effectiveness.

From Figure 2 we can clearly see that the obfuscation changes the distribution of the vectors to a large extent. The vector distribution of the samples obfuscated by *control flow flattening* and *minification* changes less, while the vector distribution of the samples obfuscated by *debug protection* and *self defending* changes greatly. This explains to some extent the better performance of the detectors in *control flow flattening* and *minification* and the poorer performance in *debug protection* and *self defending* in the Table 2. Overall, it is because of the obfuscation that the distribution of feature vectors changes significantly, and the performance of the detector on these samples naturally decreases.

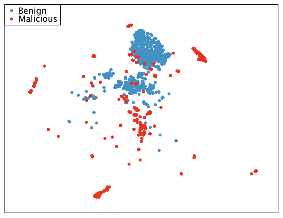
5.3.2 *Feature Analysis.*

Next we explore the scenario where obfuscated samples as training data. In such a case the ability of the detector to detect obfuscated samples is enhanced. We conjecture that the obfuscated benign and malicious samples are also different enough to be distinguished, and the detector is given a new feature set to identify this difference.

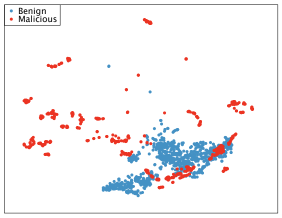
To verify this, we extract the ten most important features from *JSTAP* trained on unobfuscated samples and trained on obfuscated samples (we choose *Data obfuscation* as a representative) based on *Random Forest* model, respectively. As shown in Table 7, these two sets of features are completely different. They exhibit very different characteristics. For example, the most important feature obtained by training on unobfuscated samples is *[MemberExpression, Identifier, Identifier, Identifier]*. This often represents calling members of an object. Whereas, the most important feature obtained by training on obfuscated samples is *[Literal, ObjectExpression, Property, Literal]*. This usually indicates the definition of the object. These results indicate that these features extracted by the detector are to find the difference between the two types of samples rather than concerning the nature of benign and malicious.



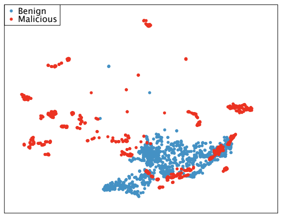
(a) Unobfuscated



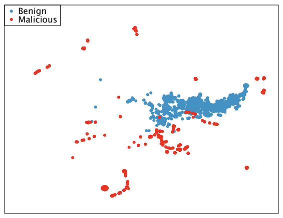
(b) Control flow flattening



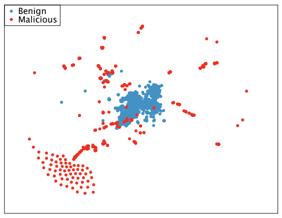
(c) Data obfuscation



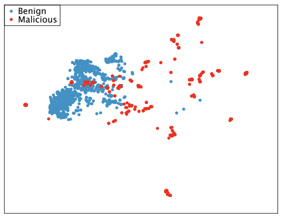
(d) Dead code injection



(e) Debug protection



(f) Self defending



(g) Minification

Figure 2: Vectors of benign and malicious samples obfuscated by different obfuscators

Table 7: The ten most important features of *JSTAP* trained on unobfuscated and obfuscated samples

Trained on unobfuscated samples	Trained on obfuscated samples
[MemberExpression, Identifier, Identifier, Identifier]	*[Literal, ObjectExpression, Property, Literal]*
[MemberExpression, Identifier, Identifier, ObjectExpression]	*[MemberExpression, MemberExpression, Identifier, CallExpression]*
[MemberExpression, MemberExpression, Identifier, Identifier]	*[MemberExpression, Identifier, CallExpression, Identifier]*
[Identifier, Identifier, MemberExpression, MemberExpression]	*[Literal, Property, Literal, CallExpression]*
[CallExpression, MemberExpression, Identifier, Identifier]	*[SequenceExpression, CallExpression, FunctionExpression, Identifier]*
[AssignmentExpression, MemberExpression, Identifier, Identifier]	*[Identifier, Literal, ObjectExpression, Property]*
[Identifier, Identifier, ObjectExpression, Property]	*[BlockStatement, VariableDeclaration, ExpressionStatement, FunctionExpression]*
[Identifier, ObjectExpression, Property, Identifier]	*[ExpressionStatement, FunctionExpression, BlockStatement, VariableDeclaration]*
[CallExpression, MemberExpression, MemberExpression, Identifier]	*[CallExpression, Identifier, Literal, ObjectExpression]*
[ExpressionStatement, CallExpression, MemberExpression, MemberExpression]	*[CallExpression, Identifier, Literal, Property]*

Table 8: The number of malicious samples that eight antivirus engines integrated with VirusTotal detect out of 1,500 samples after obfuscated by different obfuscators

Obfuscators	AV1	AV2	AV3	AV4	AV5	AV6	AV7	AV8	Average FNR (%)
Data obfuscation	876	880	827	876	879	878	875	875	40.4
Control flow flattening	920	901	620	744	721	717	721	717	48.2
Dead code injection	875	879	834	878	879	877	877	877	40.4
Self defending	957	937	1240	711	702	703	697	694	43.2
Debug protection	941	908	541	682	682	680	677	675	50.5
Minification	1000	1002	852	986	964	963	965	963	34.2
Unobfuscated	1446	1455	1433	1446	1461	1457	1455	1454	0.8

5.3.3 *Distances Between Vector Sets.*

In addition, we calculate the distances between the vector sets to quantitatively show the differences between the samples of different categories. The equation is as follows:

$$Center(V) = \frac{\sum_{i=1}^{n} v_i}{n}, \tag{1}$$

$$Distance(V_1, V_2) = EuclideanDistance(V_1, V_2), \tag{2}$$

where V is a vector set, $v_i \in V$, and $n = |V|$.

Note that we use the feature space of *JSTAP*. The distances between the obfuscated benign and malicious, unobfuscated benign and malicious, unobfuscated and obfuscated, and different types of obfuscated samples are obtained. The final distance is obtained by averaging the distances over the samples of the six obfuscation types.

As shown in Figure 3, the distance between the obfuscated benign and malicious samples is significantly smaller than the distance between the unobfuscated benign and malicious samples, and also significantly smaller than the distance between the unobfuscated and obfuscated samples, and the distance between different types of obfuscated samples. This echoes the experimental results above and further indicates that with such feature spaces, the difference obfuscation brings to the samples misleads the detector to identify obfuscation rather than benign and malicious.

To conclude, the root cause of obfuscation affecting malicious JavaScript detectors that are based on static analysis and machine learning is that detectors identify differences between different classes of samples, and the differences introduced by obfuscation

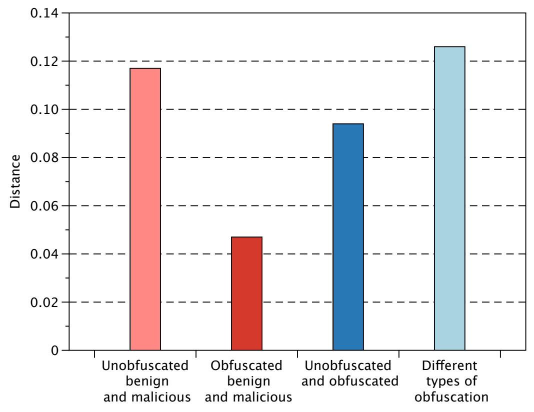


Figure 3: Distance between different vectors sets

can override the differences between benign and malicious samples, preventing detectors from working well on obfuscated samples.

Finding 3: The feature spaces of existing detectors can only reflect shallow differences in code, not about the nature of benign and malicious, which can be easily affected by the differences brought by obfuscation.

5.4 RQ4: How Does Obfuscation Affect Real-world Static Malicious JavaScript Detectors?

Finally, we explore the impact of obfuscation on the real-world static malicious JavaScript detectors. VirusTotal [22] is integrated with a variety of anti-virus engines and is very often used in real-world applications, and many researchers use VirusTotal's detection results as labels for their data. We conduct the following experiment to see the change of VirusTotal's results before and after obfuscation.

We submit 1,500 benign samples, 1,500 malicious samples, and the corresponding samples obfuscated by obfuscators to VirusTotal for scanning. For benign samples, the detectors in VirusTotal are correctly recognized. There are also almost no false positives after obfuscation. The exception is *self defending*, where four detectors show a large number of false positives on the samples obfuscated by it, with false positive rates over 13%. Although only this obfuscator appears in this situation, it indicates the existence of a specific obfuscator by which benign samples are obfuscated can lead to false positives in the detectors.

For malicious samples, 1,462 are successfully submitted, and no detector can detect all of them. We select eight detectors with high accuracy, which include detectors using machine learning techniques and signature-based techniques. As VirusTotal discourages comparisons between anti-virus tools [2], we choose the detectors with similar performance, and we do not mention the names of these detectors below.

Table 8 shows the number of malicious samples detected by each detector. From the table, we can see a similar trend to the above experiments. The performance of all eight detectors decreases to varying degrees on the obfuscated samples. Different obfuscation methods affect different detectors to different degrees. Overall, obfuscation has a similar effect, increasing the detectors' false negative rates by 40%-50% on average. *Minification*'s effect is a little weaker, causing the false negative rates to rise by about 30%.

To summarize, malicious samples can evade detection by anti-virus engines after obfuscation. This indicates that obfuscation has a similar impact on real-world detectors as our experiments above. Moreover, it can cause some false positives, as VirusTotal performs on benign samples obfuscated by *self defending*. The anti-virus community needs to further enhance its ability to capture features that are truly about maliciousness, whether they are obfuscated or not.

Finding 4. Obfuscation has the similar effect on the anti-virus engines in VirusTotal, which can lead to malicious JavaScript evading detection by using obfuscation.

6 DISCUSSION AND LIMITATIONS

6.1 Discussion

We show that the static machine learning-based malicious JavaScript detectors have high false negative rates on obfuscated samples. Experiments demonstrate that obfuscation can greatly affect the detector, and it can even be said that the features brought by obfuscation override the original features of the sample. Adding obfuscated samples to the training of the detector can improve its detection ability on the same obfuscation type, but it is not robust enough. The performance varies greatly with the distribution of different kinds of obfuscated data. We argue that the detector does not learn the features of the sample that are really relevant to the nature of the sample, and in some cases performs well on obfuscated data because some features are obtained that are useful for classifying these scripts but are not indicative of whether they are malicious. The results of machine learning detectors are difficult to trust for realistic situations with complex obfuscated data.

However, in the face of the volume of malicious JavaScript on the Web, the application of machine learning can significantly reduce human costs. Moreover, due to the expensive overhead and inapplicability of dynamic analysis in some cases, the application of machine learning techniques based on static analysis in the field of malicious JavaScript detection is necessary. For these problems we observe, we suggest that they can be solved from the following points.

Feature extraction. We can explore new feature extraction methods to extract features that cannot be changed by obfuscation and that are related to benign or malicious nature, such as abstraction slicing of code to filter out key fragments. We suggest that the code can be split and regrouped. The code is split into fine-grained, atomized representations. Then the most important parts of the code are obtained by regrouping these representations. Such groups are used as features to classify benign and malicious JavaScript files. We believe that such features are more reflective of the essence of the code, which are less susceptible to obfuscation. Using such features can naturally improve the robustness of the detector against obfuscation.

Dataset quality. We can enhance the quality of the dataset by collecting as many types of data as possible, and data cleaning and pre-processing should be performed meticulously.

Combining dynamic analysis. Dynamic analysis can reveal the behavior of the tested samples more clearly. Combining dynamic analysis when conditions allow and the overhead is acceptable. In practice, we suggest that filtering out suspicious samples with static methods first, and then determining them accurately with dynamic methods, which can greatly reduce the impact of obfuscation.

6.2 Limitations

We do not conduct a detailed analysis of all malicious JavaScript detectors based on static analysis and machine learning. In fact, the results will certainly vary for different machine learning methods in the face of the situations set up in our experiments.

Moreover, we do not discuss the obfuscation of the up-to-date website scripts with its impact on the machine learning detector, whose obfuscation distribution may be different from the settings in our experiments, and the resulting impact should be different as well. Yet, our experiments are reliable and our findings are valuable. We reveal the effects of obfuscation on static machine learning-based malicious JavaScript detectors. Even though the distribution of obfuscation in real-world websites is different from the dataset we construct, it is only a quantitative difference, not a qualitative one. Hence the construction method of our obfuscated dataset does not affect the validity of our conclusions.

7 RELATED WORK

7.1 Obfuscation Studies

Many studies of obfuscation have been proposed, and these studies cover various programming languages.

Some works aim to evaluate the effectiveness of obfuscation. Ceccato et al. [28] evaluate 44 obfuscations on the Java code. They find that code obfuscation has effect on all code metrics they select. Hammad et al. [41] conduct a large-scale study to assess the effects of obfuscation on Android apps and anti-malware products. They discover that code obfuscation impacts Android anti-malware products significantly. Wang et al. [67] explore the necessity of applying obfuscation to iOS apps against malicious reverse engineering for protecting mobile apps. Their results show that software obfuscation can provide effective protection with modest cost.

There are also some works proposing new obfuscation techniques for JavaScript. Fass et al. [34] introduce *HideNoSeek*, which rewrites ASTs of malicious samples into benign ones to evade detection. Romano et al. [60] propose *Wobfuscator*, which transforms selected parts of behaviors implemented in JavaScript into WebAssembly to evade detection. These new techniques further increase the impact of obfuscation.

7.2 Obfuscation Detection

Some existing works lay more emphasis on detecting obfuscation. Kaplan et al. [45] present *NoFus*, which is a static obfuscation classifier based on the AST of JavaScript code. AL-Taharwa et al. [23] propose *JSOD*, which is designed to detect readable versions of obfuscation by using syntactic and contextual features. Sarker et al. [62] leverage the differences in browser API features between dynamic and static analysis to reveal the behavior of obfuscation. These studies are characterized by targeted approaches designed for obfuscation. Furthermore, their results indicate that the correlation between obfuscation and maliciousness is weak.

Other works investigate the obfuscation applied on JavaScript code in real-world. Likarish et al. [52] train classifiers to detect malicious JavaScript and obfuscation. They find that some websites choose to compress their JavaScript code, and this obfuscated code is the most likely to generate a false positive. Xu et al. [70] measure the usage of obfuscation in real-world malicious JavaScript samples and explore the effectiveness of 20 anti-virus software against obfuscation. They find all popular anti-virus products can be effectively evaded by various obfuscation techniques. Skolka et al. [64] study the obfuscation in a large-scale scripts from 100,000 websites by leveraging a neural network-based classifier. They show that obfuscation is very common and is used for a variety of purposes. Moog et al. [53] conduct an in-depth study of obfuscation on both client-side JavaScript and library code from npm in the wild. They define two random forest-based classifiers to detect obfuscated samples, benefiting from AST-based features. Their results demonstrate the popularity of obfuscation and the differences between obfuscation techniques in benign and malicious scripts.

8 CONCLUSION

In this paper, we investigate the impact of obfuscation on machine learning malicious JavaScript detectors based on static analysis features in depth. We conduct sufficient experiments and find that obfuscation can indeed interfere with the detectors' judgment to a great extent, with a significant increase in FNR and FPR. Moreover, when there is a clear bias about obfuscation between benign and malicious samples in the training sets, detectors can be induced to detect obfuscation rather than malicious behaviors. What's worse, the common measures to mitigate obfuscation impact, such as improving the quality of dataset by adding the relevant type of obfuscated samples and introducing state-of-the-art deep learning models for programming language, are not effective. Through our in depth analysis, we believe that the root cause of obfuscation affecting those detectors is that detectors essentially identify differences between different classes of samples, and the differences introduced by obfuscation can override the differences between benign and malicious samples, preventing detectors from working well on obfuscated samples. Finally, we observe a similar effect of obfuscated data on static detectors in VirusTotal, which indicates that this problem is widespread. The anti-virus community needs to pay attention to this issue, and it could have serious implications for users of machine learning detectors based on static features, which are being applied more and more.

9 DATA AVAILABILITY STATEMENT

We make the code and dataset publicly available at https://doi.org/10.5281/zenodo.7977493 [58].

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported in part by the National Natural Science Foundation of China (Grant No. 62272181).

REFERENCES

[1] 2023. Atom: A Hackable Text Editor for the 21st Century. https://atom.io/
[2] 2023. AV Comparative Analyses. https://blog.virustotal.com/2012/08/av-comparative-analyses-marketing-and.html
[3] 2023. Closure-compiler. https://github.com/google/closure-compiler
[4] 2023. Closure Compiler: Advanced Compilation. https://developers.google.com/closure/compiler/docs/api-tutorial3
[5] 2023. CodeBERT. https://github.com/microsoft/CodeBERT
[6] 2023. German Federal Office for Information Security. https://www.bsi.bund.de/EN
[7] 2023. Gnirts: Obfuscate String Literals in JavaScript Code. https://github.com/anseki/gnirts
[8] 2023. JaSt - JS AST-Based Analysis. https://github.com/Aurore54F/JaSt
[9] 2023. JavaScript Malware Collection. https://github.com/HynekPetrak/javascript-malware-collection
[10] 2023. JavaScript-obfuscator: A Powerful Obfuscator for JavaScript and Node.js. https://github.com/javascript-obfuscator/javascript-obfuscator
[11] 2023. JavaScript Reference. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference
[12] 2023. JSObfu: Obfuscate JavaScript (Beyond Repair) with Ruby. https://github.com/rapid7/jsobfu
[13] 2023. JStap: A Static Pre-Filter for Malicious JavaScript Detection. https://github.com/Aurore54F/JStap
[14] 2023. Lexical-jsdetector. https://github.com/Aurore54F/lexical-jsdetector
[15] 2023. Malicious JavaScript Dataset. https://github.com/geeksonsecurity/js-malicious-dataset
[16] 2023. Malware with Your Mocha: Obfuscation and Anti Emulation Tricks in Malicious JavaScript. https://www.yumpu.com/s/0PX6x19R5gw0KWvt
[17] 2023. MDNC - Malware Don't Need Coffee. https://malware.dontneedcoffee.com/
[18] 2023. Microsoft Digital Defense Report. https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RWMFIi
[19] 2023. Online JavaScript Minifier Tool and Compressor, with Fast and Simple API Access. https://www.toptal.com/developers/javascript-minifier
[20] 2023. Scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/
[21] 2023. Syntactic-jsdetector. https://github.com/Aurore54F/syntactic-jsdetector
[22] 2023. VirusTotal - Analyze Suspicious Files and URLs to Detect Types of Malware. https://www.virustotal.com/
[23] Ismail Adel Al-Taharwa, Hahn-Ming Lee, Albert B. Jeng, Kuo-Ping Wu, Cheng-Seen Ho, and Shyi-Ming Chen. 2015. JSOD: JavaScript Obfuscation Detector. *Secur. Commun. Networks* 8, 6 (2015), 1092–1107. https://doi.org/10.1002/sec.1064
[24] Ammar Alazab, Ansam Khraisat, Moutaz Alazab, and Sarabjot Singh. 2022. Detection of Obfuscated Malicious JavaScript Code. *Future Internet* 14, 8 (2022), 217. https://doi.org/10.3390/fi14080217
[25] Mohamed Alsharnouby, Furkan Alaca, and Sonia Chiasson. 2015. Why Phishing Still Works: User Strategies for Combating Phishing Attacks. *International Journal of Human-Computer Studies* 82 (2015), 69–82. https://doi.org/10.1016/j.ijhcs.2015.05.005
[26] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP'17)*. 559–578. https://doi.org/10.1109/SP.2017.68
[27] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. 2011. 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)*. 197–206. https://doi.org/10.1145/1963405.1963436
[28] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. 2015. A Large Study on the Effect of Code Obfuscation on the Quality of Java Code. *Empirical Software Engineering* 20 (2015), 1486–1524. https://doi.org/10.1007/s10664-014-9321-0
[29] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. 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)*. 281–290. https://doi.org/10.1145/1772690.1772720
[30] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Security Symposium*.
[31] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. 1388–1401. https://doi.org/10.1145/2976749.2978313
[32] Yong Fang, Cheng Huang, Yu Su, and Yaoyao Qiu. 2020. Detecting Malicious JavaScript Code Based on Semantic Analysis. *Computers & Security* 93 (2020), 101764. https://doi.org/10.1016/j.cose.2020.101764
[33] Yong Fang, Chaoyi Huang, Minchuan Zeng, Zhiying Zhao, and Cheng Huang. 2022. JStrong: Malicious JavaScript Detection Based on Code Semantic Representation and Graph Neural Network. *Computers & Security* 118 (2022), 102715. https://doi.org/10.1016/j.cose.2022.102715
[34] Aurore Fass, Michael Backes, and Ben Stock. 2019. Hidenoseek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. 1899–1913. https://doi.org/10.1145/3319535.3345656
[35] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: A Static Pre-filter for Malicious JavaScript Detection. In *Proceedings of the 35th Annual Computer Security Applications Conference, (ACSAC'19)*. 257–269. https://doi.org/10.1145/3359789.3359813
[36] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. 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)*, Vol. 10885. 303–325. https://doi.org/10.1007/978-3-319-93411-2_14
[37] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139
[38] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. 2018. Hiding in the Crowd: An Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *Proceedings of the 27th International Conference on World Wide Web (WWW'18)*. 309–318. https://doi.org/10.1145/3178876.3186097
[39] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL'22)*. 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499
[40] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *Proceedings of the 9th International Conference on Learning Representations (ICLR'21)*.
[41] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A Large-scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-malware Products. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 421–431. https://doi.org/10.1145/3180155.3180228
[42] Yunhua Huang, Tao Li, Lijia Zhang, Beibei Li, and Xiaojie Liu. 2021. JSContana: Malicious JavaScript Detection Using Adaptable Context Analysis and Key Feature Extraction. *Computers & Security* 104 (2021), 102218. https://doi.org/10.1016/j.cose.2021.102218
[43] Shin-Jia Hwang and Tzu-Ping Chen. 2023. A Detector Using Variant Stacked Denoising Autoencoders with Logistic Regression for Malicious JavaScript with Obfuscations. In *Proceedings of the 25th International Computer Symposium (ICS'22)*. 374–386. https://doi.org/10.1007/978-981-19-9582-8_33
[44] Luca Invernizzi, Paolo Milani Comparetti, Stefano Benvenuti, Christopher Kruegel, Marco Cova, and Giovanni Vigna. 2012. Evilseed: A Guided Approach to Finding Malicious Web Pages. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP'12)*. 428–442. https://doi.org/10.1109/SP.2012.33
[45] Scott Kaplan, Benjamin Livshits, Benjamin Zorn, Christian Siefert, and Charlie Curtsinger. 2011. " NOFUS: Automatically Detecting"+ String. fromCharCode (32)+" ObFuSCateD". toLowerCase ()+" JavaScript Code. *Technical report, Technical Report MSR-TR 2011–57, Microsoft Research* (2011).
[46] Debabrata Kar, Suvasini Panigrahi, and Srikanth Sundararajan. 2016. SQLiGoT: Detecting SQL Injection Attacks Using Graph of Tokens and SVM. *Computers & Security* 60 (2016), 206–225. https://doi.org/10.1016/j.cose.2016.04.005
[47] 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 *Proceedings of the 26th International Conference on World Wide Web (WWW'17)*. 897–906. https://doi.org/10.1145/3038912.3052674
[48] Thomas N. Kipf and Max Welling. 2016. Semi-supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (2016). http://arxiv.org/abs/1609.02907
[49] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking Internet Malware. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP'12)*. 443–457. https://doi.org/10.1109/SP.2012.48
[50] 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 *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. 1714–1730. https://doi.org/10.1145/3243734.3243858
[51] Pavel Laskov and Nedim Srndic. 2011. Static Detection of Malicious JavaScript-bearing PDF Documents. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*. 373–382. https://doi.org/10.1145/2076732.2076785
[52] Peter Likarish, Eunjin Jung, and Insoon Jo. 2009. Obfuscated Malicious JavaScript Detection Using Classification Techniques. In *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE'09)*. 47–54. https://doi.org/10.1109/MALWARE.2009.5403020

[53] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'21)*. 569–580. https://doi.org/10.1109/DSN48987.2021.00065

[54] Samuel Ndichu, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. 2019. A Machine Learning Approach to Detection of JavaScript-based Attacks Using AST Features and Paragraph Vectors. *Applied Soft Computing* 84 (2019), 105721. https://doi.org/10.1016/j.asoc.2019.105721

[55] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS'19)*.

[56] Paruj Ratanaworabhan, V. Benjamin Livshits, and Benjamin G. Zorn. 2009. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of the 18th USENIX Security Symposium*. 169–186.

[57] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning Programs from Noisy Data. *ACM Sigplan Notices* 51, 1 (2016), 761–774. https://doi.org/10.1145/2837614.2837671

[58] Kunlun Ren. 2023. *Artifacts for the ISSTA 2023 Paper: An Empirical Study on the Effects of Obfuscation on Static Machine Learning-based Malicious JavaScript Detectors*. https://doi.org/10.5281/zenodo.7977493

[59] Konrad Rieck, Tammo Krueger, and Andreas Dewald. 2010. Cujo: Efficient Detection and Prevention of Drive-by-download Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*. 31–39. https://doi.org/10.1145/1920261.1920267

[60] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. 2022. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to Webassembly. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP'22)*. 1574–1589. https://doi.org/10.1109/SP46214.2022.9833626

[61] Muhammad Fakhrur Rozi, Tao Ban, Seiichi Ozawa, Sangwook Kim, Takeshi Takahashi, and Daisuke Inoue. 2021. JStrack: Enriching Malicious JavaScript Detection Based on AST Graph Analysis and Attention Mechanism. In *Proceedings of the 28th International Conference on Neural Information Processing (ICONIP'21)*, Vol. 13109. 669–680. https://doi.org/10.1007/978-3-030-92270-2_57

[62] Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. 2020. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *Proceedings of the 20th ACM Internet Measurement Conference (IMC'20)*. 648–661. https://doi.org/10.1145/3419394.3423616

[63] Prabhu Seshagiri, Anu Vazhayil, and Padmamala Sriram. 2016. AMA: Static Code Analysis of Web Page for the Detection of Malicious Scripts. *Procedia Computer Science* 93 (2016), 768–773.

[64] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *Proceedings of the 28th International Conference on World Wide Web (WWW'19)*. 1735–1746. https://doi.org/10.1145/3308558.3313752

[65] Ben Stock, Benjamin Livshits, and Benjamin Zorn. 2016. 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)*. 455–466. https://doi.org/10.1109/DSN.2016.48

[66] Junjie Wang, Yinxing Xue, Yang Liu, and Tian Huat Tan. 2015. JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'15)*. 109–120. https://doi.org/10.1145/2714576.2714620

[67] Pei Wang, Dinghao Wu, Zhaofeng Chen, and Tao Wei. 2019. Field Experience with Obfuscating Million-user iOS Apps in Large Enterprise Mobile Development. *Software: Practice and Experience* 49, 2 (2019), 252–273. https://doi.org/10.1002/spe.2648

[68] Yao Wang, Wan-dong Cai, and Peng-cheng Wei. 2016. A Deep Learning Approach for Detecting Malicious JavaScript Code. *Security and Communication Networks* 9, 11 (2016), 1520–1534. https://doi.org/10.1002/sec.1441

[69] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. 2015. Understanding Malvertising through Ad-injecting Browser Extensions. In *Proceedings of the 24th International Conference on World Wide Web (WWW'15)*. 1286–1295. https://doi.org/10.1145/2736277.2741630

[70] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study. In *Proceedings of the 7th International Conference on Malicious and Unwanted Software (MALWARE'12)*. 9–16. https://doi.org/10.1109/MALWARE.2012.6461002

[71] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2013. JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code. In *Proceedings of the third ACM Conference on Data and Application Security and Privacy (CODASPY'13)*. 117–128. https://doi.org/10.1145/2435349.2435364

[72] Yinxing Xue, Junjie Wang, Yang Liu, Hao Xiao, Jun Sun, and Mahinthan Chandramohan. 2015. Detection and Classification of Malicious JavaScript via Attack Behavior Modelling. In *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA'15)*. 48–59. https://doi.org/10.1145/2771783.2771814

[73] Ye Zhang and Byron C. Wallace. 2017. A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017*. 253–263. https://aclanthology.org/I17-1026/

[74] Yuchen Zhou and David Evans. 2015. Understanding and Monitoring Embedded Web Scripts. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP'15)*. 850–865. https://doi.org/10.1109/SP.2015.57

Received 2023-02-16; accepted 2023-05-03