Obfuscated Malicious JavaScript Detection by Machine Learning

Jinkun Pan[1, a], Xiaoguang Mao[1, 2]

[1] College of Computer, National University of Defense Technology, Changsha, 410073, China

[2] Laboratory of Science and Technology on Integrated Logistics Support, National University of Defense Technology, Changsha, 410073, China

[a]email: pan_jin_kun@163.com

Keywords: Malicious JavaScript Detection; Machine Learning; Obfuscation; Dynamic Trace; Semantic-based Deobfuscation; Trace Pattern

Abstract. In recent years, malicious JavaScript code has become more and more pervasive and been used by attackers to perform their attacks on the Web. To evade the detection of defense measures, various kinds of obfuscation techniques have been applied by the malicious script, taking advantage of the dynamic nature of JavaScript language. In this paper, we propose a new machine-learning based detection approach aiming at defeating such evasion attempts. Dynamic execution traces are recorded to capture all behaviors performed by the malicious script, including the dynamic generated code. Semantic-based deobfuscation is used to simplify the traces to get more concise and more essential instructions. None-ordered and none-concessive trace patterns are extracted from the deobfuscated traces to represent the intrinsic features for malicious scripts. We evaluated our approach with a large number of dataset collected from the Internet. The empirical results demonstrate that our approach is able to detect obfuscated malicious JavaScript code both effectively and efficiently.

Introduction

Nowadays, various kinds of Web sites rely on JavaScript for better user experience and enhanced functionality. However, JavaScript can also be used by attackers to trick the victim to click a malicious link pointed to a malicious page, or to exploit security vulnerabilities in the environment of the victim. To defeat such attacks, many detection and mitigation methods have been proposed. Some considered using high-interaction and low-interaction honeypots [1, 2] to attract, record and analyze JavaScript attacks. Some aim at off-line analysis of JavaScript code [3, 4]. Recent work has integrated machine learning techniques into malicious JavaScript detection [5-11]. However, existing approaches have not paid adequate attention for the problem of obfuscation. Obfuscation is a kind of code transformation that makes the code unintelligible, including data obfuscation, logic obfuscation, encoding obfuscation and so on. It is prevalently adopted by malicious code to conceal the malicious intent and to escape the detection of defense measures.

In this paper, we propose a new machine-learning based detection approach for obfuscated malicious JavaScript. We use the dynamic execution trace of the JavaScript program as the primary data from which we extract representative features for machine learning. This avoids several obfuscation techniques only effective for static analysis. As for the dynamic code unfolding, real malicious script has to be deobfuscated anyway in order to fulfil its intent, which will be captured by the execution trace eventually. We adopt semantic-based deobfuscation technique to simplify the trace. The semantic-based deobfuscation technique was originally proposed for simplifying obfuscated JavaScript code automatically. As we do not need to recover the program in the form of source code for further human investigation, we just use the deobfuscation slicing to simplify the trace. On the one hand, the trace is deobfuscated, thus will be more powerful and useful to extract features that represent intrinsic characteristics of malicious code. On the other hand, the trace is simplified, which decreases the time and space overhead to process it. We extract none-ordered and none-consecutive sequence patterns from the trace as the features for machine learning. Compared to previously used n-gram patterns [8, 11], it could help defeat logic obfuscation such as insertion of

© 2016. The authors - Published by Atlantis Press

irrelevant instructions or switching instruction orders.

To evaluate our approach, we investigated a large dataset collected from the Internet. The best performance of our approach achieves 99.64% accuracy, 97.74% precision, 93.05% recall and 95.34% F-measure. Our approach runs without incurring too much overhead, which might be used to implement a real-time detection tool in the practical setting.

JavaScript Obfuscation

In order to escape the detection of defense measures and to hide the malicious intent, obfuscation techniques have been applied prevalently by malicious JavaScript code. We summarize these techniques into five categories as follows and present examples for each category in Fig. 1.

Category	Normal	Obfuscated
Randomization	var mystring = "Hello!";	var _dfeakia1f92 //_gcvdseaepk = "Hello!"; //_gpqkjk3424pkl

Logic Structure	document.write("Hello!");	var i = 99;
		if(i==1){
		document.write("Aha!");
		}
		document.write("Hello!");
		i++;

Encoding	document.write("Hello!");	var s = “\x48\x65\x6c\x6c\x6f\x21”; document.write(s);
Data	document.write("Hello!");	var a = “He”; var b = “ll”; var c = “o!”; document.write(a + b + c);
Code Unfolding	document.write("Hello!");	var s = "document.write(\"Hello!\")"; eval(s);

Fig. 1. JavaScript obfuscation examples

Approach

To detect obfuscated malicious JavaScript, we use machine learning approaches and the overview of such process is shown in Fig. 2.

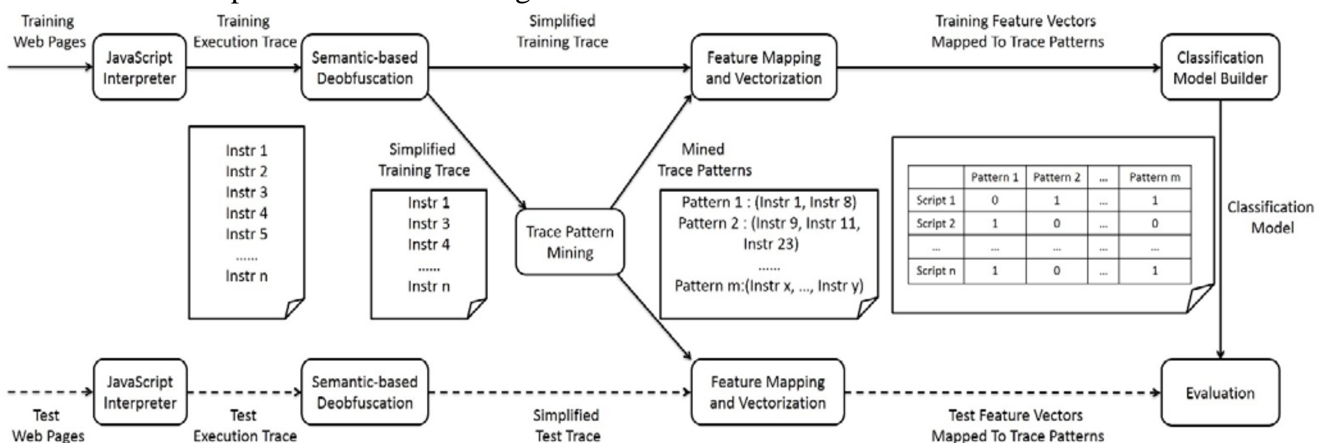


Fig. 2. Approach overview

Execution Trace Collection. In our work, we adopt byte-code level dynamic analysis. Byte-code level analysis can take advantage of the compiler because those obfuscation methods aiming at confusing human or source-code level tools will be removed or revealed after compilation. To collect the trace of JavaScript execution, we instrument the open-source Web browser Mozilla Firefox. The traces are in the form of JavaScript Opcode, which is an intermediate byte-code level representation during the Web page rendering process.

Semantics-based Deobfuscation. To mitigate the problem caused by code unfolding obfuscation, we adopt the semantics-based deobfuscation proposed by Lu and Debray [12, 13], which simplifies a given script by removing deobfuscation-related instructions and producing a semantically equivalent script. The motivation comes from the intuition behind the deobfuscation process. Although the original source code cannot be recovered exactly, we can produce the code that is semantics equivalent to the original one because it can be expected that any obfuscation and deobfuscation process should be semantics preserving. With the help of semantics-based deobfuscation, we can gain benefits from two aspects. On the one hand, the deobfuscated instructions are more essential to characterize the malicious codes, the features extracted from which are more useful to guide the machine learning algorithms to achieve better performance. On the other hand, the deobfuscated instructions are more concise to process, which save the time and space cost of machine learning algorithms.

To perform the semantic-based deobfuscation, the definition of semantic equivalence is essential. Consider the analysis of the potential malicious script, observational equivalence is a practical definition. That is to say, if two scripts interact with the outer environment through the same way, they will be considered equivalent. Because native calls are the only approach to interact with the outer environment, two scripts are considered equivalent if they have the same native call sequences with the same arguments. In this way, we can simplify the process of deobfuscation by identifying codes that affect the native calls both directly and indirectly.

To identify instructions that affect native functions, deobfuscation slicing is used, which is based on dynamic slicing. For a given variable in the execution trace of a script, dynamic slicing identifies which instructions actually affect the value of such variable.

Trace Pattern Mining. Given the deobfuscated JavaScript execution traces, we would like to extract representative features from them to characterize script intent and help detect malicious scripts. We believe the relationship among execution instructions in the trace contains the information of the behavior of the program. Therefore, the trace patterns are mined from instruction sequences, but not limited to ordered and consecutive sequences like n-gram does. Malicious programs frequently insert irrelevant garbage instruction and change the order of instructions to obfuscate the logic of the program. Any detection approach based on ordered and consecutive sequences could be fooled and fail to detect the malicious behavior.

To mine trace patterns from instruction sequences, we use Apriori algorithm, which is a classical approach to compute patterns and association rules, to generate trace patterns. To select appropriate trace patterns as features for our classifier, the following two criteria are used:

1) The frequency of the trace pattern should be high enough in the dataset used for training. If it rarely occurs, it will be considered as noise and excluded from our features.

2) The trace pattern should be a significant indicator distinguishing malicious and benign scripts from each other. That is to say, it should be frequent in malicious script and infrequent in benign script, or vice versa.

Considering the first term, only frequent trace patterns are extracted. In the Apriori algorithm, there is a parameter named support, which is the number of a pattern occurs in all data. We call a trace pattern frequent if its support is above a minimum support value.

To satisfy the second criteria, for a given pattern P_i, we define the term significance (denoted as Sig) as follows:

$$Sig(P_i) = \begin{cases} \dfrac{NBCount(P_i)+\varepsilon}{NMCount(P_i)+\varepsilon}, NBCount(P_i) \geq NmCount(P_i) \\ \dfrac{NMCount(P_i)+\varepsilon}{NBCount(P_i)+\varepsilon}, NBCount(P_i) < NmCount(P_i) \end{cases}$$

In this formula, $NBCount(P_i)$ is the normalized count for the benign dataset and $NMCount(P_i)$ is the normalized count for the malicious dataset. The normalized count for a dataset D refers to the frequency of the trace pattern appears in D divided by the number of scripts in D. ε is a small constant to avoid the divide-by-zero error. The normalization helps us factor out the problem caused

by imbalance size of benign and malicious datasets.

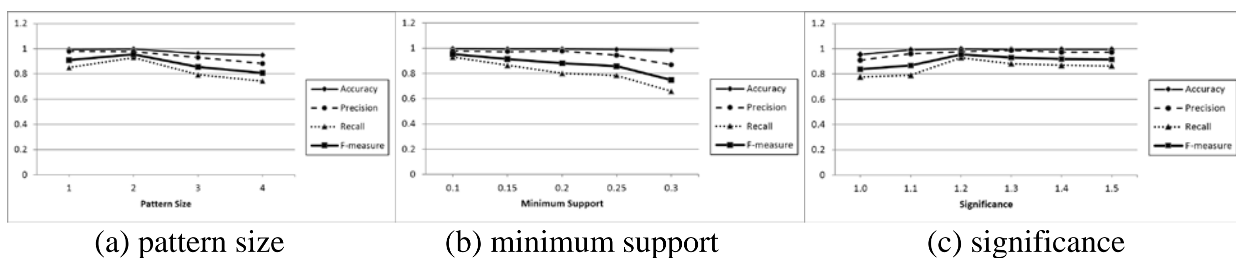
Experiment

Data Collection. To the best of our knowledge, there is no standard benchmarks of malicious JavaScript that are publicly available. Therefore, we constructed our own datasets of benign and malicious scripts. For the benign dataset, we crawled 10,000 scripts from the Web sites listed in the Alexa top 500. From each site, 20 scripts are fetched randomly. Every script was validated with VirusTotal. The malicious dataset was created by collecting samples from different malware repositories. The malicious samples contain various types of scripts that attack browser vulnerabilities, attack the PDF reader, attack the Java Runtime Environment, attack the Flash plugin, attack based on multimedia, and so on. VirusTotal assisted with manual inspection was used to ensure they were malicious. In the end, a total of 1,000 samples were collected.

Evaluation Approach. For evaluation, we used Accuracy, Precision, Recall and F-measure, which are widely used metrics in machine learning. In addition, stratified 10-fold cross validation was employed, where the dataset was separated to 10 sets randomly. In each set, the proportion of malicious to benign scripts is maintained. Each one of them was used as testing dataset and the rest ones were used as training dataset, thus 10 combinations were evaluated. This process is repeated 10 times, and the average result of 100 runs is reported.

One issue we should further consider is data imbalance which many real-world applications suffer from. Data imbalance means one class data appears more than another class data, which causes the machine learning methods biased towards the majority class. To deal with this problem, there are two common approaches: re-sampling the data and re-weighting the data. Re-sampling changes the number of instances in different classes to make them in the same level. Re-weighting changes the weight of instances in different classes. We evaluated both two methods and re-weighting performs better, therefore re-weighting was used in all experiments. A weight according to the proportion of malicious to benign scripts was assigned to each malicious sample in the training data. The testing data was maintained the same ratio of malicious to benign scripts as in the original dataset.

Parameter Selection. To evaluate our approach with different parameter settings, we tested several key parameters with controlled experiments, including pattern size, minimum support, significance and machine learning algorithms.



(a) pattern size (b) minimum support (c) significance

Fig.3. Parameter selection results

1) Pattern Size: Small values of pattern size can lead to bias caused by inadequate relationship information, while large values of pattern can result in high variance because of over-fitting. Increasing pattern causes the feature space increase exponentially, therefore the maximum size is up to 4 due to the limit of the memory. As we can see the result presented in Fig. 3(a), the best pattern size is 2.

2) Minimum Support: The minimum support is represented as the percentage of all mined patterns. Suppose the number of all mined patterns is n, and the minimum support is s, then all patterns less than n*s will be filtered out. The result of varying minimum support is presented in Fig. 3(b). The performance gets worse as we increase the minimum support. The reason might be larger support value filters more patterns which can be useful. With this trend, we also wanted to evaluate smaller support values; however, smaller support means more patterns retained, which is limited by the memory size. Therefore, 0.1 is set to be the best minimum support value.

3) Minimum Significance: Larger significance causes less but more significant patterns to be retained. The effect of minimum significance on the performance is shown in Fig. 3(c). As we can see, the value 1.2 is the best one which balances the significance and the number of useful patterns.

4) Machine Learning Algorithm: The impact of different machine learning algorithms was evaluated by several classic and representative algorithms. Table 1 presents the effectiveness for our approach with J48 decision tree, Naive Bayes, Logistic Regression and linear SVM. We used the implementation of these algorithms in python package scikit-learn with default settings. Among them, linear SVM achieves the best overall performance.

Table 1. Performance of different algorithms

Algorithm	Accuracy	Precision	Recall	F-measure
J48	0.9948	0.9777	0.8875	0.9304
Naïve Bayes	0.9282	0.3147	0.7029	0.4347
Logistic Regression	0.9937	0.9970	0.8420	0.9130
SVM	0.9964	0.9774	0.9305	0.9534

Comparison. To demonstrate the effectiveness of our work, we compared it with two state-of-the-art methods, namely JSAND [5] and Opcode Analysis [8]. JSAND is publicly available as an online service; therefore, it uses different datasets for training. Opcode Analysis uses n-grams of dynamic traces and improves CUJO [11] through a bounded but high coverage feature space. We implemented Opcode Analysis and evaluated it using the same training and testing dataset as our approach. We selected n = 3 as the best setting after several evaluation for different n. To demonstrate the effectiveness of the semantic-based deobfuscation used by our approach, we also considered our approach without using deobfuscation. Our approach used the best parameters among previous evaluated ones. The comparison result is shown in Table 2. JSAND has the best precision, but the recall is very low. Opcode Analysis performs better than our approach without deobfuscation, but worse than our approach with full phases. Among all approaches, our approach with full phases achieves the best overall performance.

Table 2. Performance of different approaches

Approach	Accuracy	Precision	Recall	F-measure
JSAND	0.6987	1.0000	0.3005	0.3005
Opcode Analysis	0.9685	0.9557	0.8251	0.8856
Ours-NoDeobfuscation	0.9556	0.9103	0.7760	0.8378
Ours-Full	0.9964	0.9774	0.9305	0.9534

Runtime Overhead. We recorded the running time of different phases of our approach, which is shown in Table 3. The main overheads come from the deobfuscation phase. Although the total time of the deobfuscation for all 11,000 scripts is 3,153.65 seconds, the average time for each one is just 286.7 milliseconds, which is practically acceptable. The average time for each script including all phases demonstrates the potential of our approach being used as a real-time detection tool, which can be integrated in the browser.

Table 3. Running time of different phases

Phase	Number	Total (s)	Average (ms)
Trace Collection	11,000	758.37	68.94
Deobfuscation	11,000	3,153.65	286.70
Pattern Mining	11,000	1,366.52	124.23
Training	9,900*100	324.21	0.33
Testing	1,100*100	28.13	0.26

Conclusion

As malicious JavaScript code on the Web spreads and armed with more sophisticated evasion techniques such as obfuscation, malicious script detection techniques should be improved to defeat such threats in an effective and efficient way. We propose our approach targeting for detection of obfuscated malicious JavaScript code and evaluate it with real-world samples from the Internet. The

results show that our approach is able to improve the performance of obfuscated malicious script detection without incurring too much overhead. In the future, we plan to deal with more sophisticated code obfuscation techniques.

Acknowledgments

This research was supported in part by grants from National Natural Science Foundation of China (Nos. 61379054, 61502015 and 91318301), and Program for New Century Excellent Talents in University.

References

[1] A. Büscher, M. Meier, and R. Benzmüller, Throwing a monkeywrench into web attackers plans, Communications and Multimedia Security, 2010, pp. 28-39

[2] M. T. Qassrawi and H. Zhang, Detecting malicious web servers with honeyclients, Journal of Networks, 2011, 6, pp. 145-152

[3] S. Karanth, S. Laxman, P. Naldurg, R. Venkatesan, J. Lambert, and J. Shin, ZDVUE: prioritization of javascript attacks to discover new vulnerabilities, Proceedings of the 4th ACM workshop on Security and artificial intelligence, 2011, pp. 31-42

[4] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, Rozzle: De-cloaking internet malware, IEEE Symposium on Security and Privacy, 2012, pp. 443-457

[5] M. Cova, C. Kruegel, and G. Vigna, Detection and analysis of drive-by-download attacks and malicious JavaScript code, Proceedings of the 19th international conference on World Wide Web, 2010, pp. 281-290

[6] D. Canali, M. Cova, G. Vigna, and C. Kruegel, Prophiler: a fast filter for the large-scale detection of malicious web pages, Proceedings of the 20th international conference on World Wide Web, 2011, pp. 197-206

[7] J. Wang, Y. Xue, Y. Liu, and T. H. Tan, JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification, Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, 2015, pp. 109-120

[8] G. K. Jayasinghe, J. S. Culpepper, and P. Bertok, Efficient and effective realtime prediction of drive-by download attacks, Journal of Network and Computer Applications, 2014, 38, pp. 135-149

[9] M. Heiderich, T. Frosch, and T. Holz, Iceshield: Detection and mitigation of malicious websites with a frozen dom, Recent Advances in Intrusion Detection, 2011, pp. 281-300

[10] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert, ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection, USENIX Security Symposium, 2011, pp. 33-48

[11] K. Rieck, T. Krueger, and A. Dewald, Cujo: efficient detection and prevention of drive-by-download attacks, Proceedings of the 26th Annual Computer Security Applications Conference, 2010, pp. 31-39

[12] G. Lu and S. Debray, Automatic simplification of obfuscated JavaScript code: A semantics-based approach, Proceedings of the Sixth International Conference on Software Security and Reliability, 2012, pp. 31-40

[13] G. Lu, K. Coogan, and S. Debray, Automatic simplification of obfuscated JavaScript code, Information Systems, Technology and Management, 2012, 1, pp. 348-359