JStrack: Enriching Malicious JavaScript Detection Based on AST Graph Analysis and Attention Mechanism

Rozi, Fakhrur Muhammad ; Ban, Tao ; Ozawa, Seiichi ; Kim, Sangwook ; Takahashi, Takeshi ; Inoue, Daisuke

(Citation)
Neural Information Processing:669-680

(Issue Date)
2021-12-07

(Resource Type)
conference paper

(Version)
Accepted Manuscript

(URL)
https://hdl.handle.net/20.500.14094/0100481143

JStrack: Enriching Malicious JavaScript Detection Based on AST Graph Analysis and Attention Mechanism

Muhammad Fakhrur Rozi[1,2], Tao Ban[1], Seiichi Ozawa[2], Sangwook Kim[2], Takeshi Takahashi[1], and Daisuke Inoue[1]

[1] National Institute of Information and Communications Technology
Koganei, Tokyo, Japan
{fakhrurrozi95,bantao,takeshi_takahashi,dai}@nict.go.jp
[2] Kobe University, Kobe, Hyogo, Japan
ozawasei@kobe-u.ac.jp, kim@eedept.kobe-u.ac.jp

Abstract. Malicious JavaScript is one of the most common tools for attackers to exploit the vulnerability of web applications. It can carry potential risks such as spreading malware, phishing, or collecting sensitive information. Though there are numerous types of malicious JavaScript that are difficult to detect, generalizing the malicious script's signature can help catch more complex JavaScripts that use obfuscation techniques. This paper aims at detecting malicious JavaScripts based on structure and attribute analysis of abstract syntax trees (ASTs) that capture the generalized semantic meaning of the source code. We apply a graph convolutional neural network (GCN) to process the AST features and get a graph representation via neural message passing with neighborhood aggregation. The attention layer enriches our method to track pertinent parts of scripts that may contain the signature of malicious intent. We comprehensively evaluate the performance of our proposed approach on a real-world dataset to detect malicious websites. The proposed method demonstrates promising performance in terms of detection accuracy and robustness against obfuscated samples.

Keywords: Cyber security · Malicious JavaScript · Abstract Syntax Tree · Graph neural network

1 Introduction

Javascript payload injection into legitimate or fake websites has been one of the largest attack on the web. The malicious script can exploit the vulnerability of the web applications to perform a drive-by download attack [2] or cross-site scripting (XSS) [19]. When the attack is succesful, attackers distribute malware to clients, which can cause damage such as sensitive data leakage, wire transfer, or integrating into distributed denial-of-service (DDoS) attacks [3]. For instance, one of the most famous examples of XSS vulnerability is the Myspace Samy worm by Samy Kamkar in 2005 [9]. He exploited a vulnerability on the target

that could give him priviledge to store a JavaScript payload on his Myspace profile. Moreover, web technology improvement helps attackers use the latest method to avoid detection, such as the obfuscation techniques.

Researchers have identified the malicious JavaScript payload, which is typically used by attackers as part of a web security attack. A variety of detection systems has been proposed that use JavaScript features to detect malicious intent. We can take many approaches to create a detection system for malicious JavaScript, such as strings, function calls, bytecode sequences, abstract syntax tree (ASTs), outputs of dynamic analysis tools. Among these features, AST gives the most notably excellent performance. Fass et al. [6] use this feature for their static analysis and use the N-gram model to detect malicious obfuscated JavaScripts. However, their work focused on the frequency analysis of the specific patterns with the connection between syntactic units of AST feature ignored. We have to analyze it at the tree level instead of the sequence level when we want to capture the semantic meaning of the code.

We propose JStrack, a malicious JavaScript detection system using a graph-based approach on the AST features to capture the whole semantic meaning which has not been considered in previous works. We hypothesize that the style of malicious code tends to be better structured due to the decryption or deobfuscation process that should exist inside the code instead of having an abstract structure. Analyzing the whole AST as a graph structure also gives us more information about the actual intent of the source code. To capture that information thoroughly, we use a supervised graph neural network (GNN), known as a graph convolutional neural network (GCN) model. This model can capture the connections between nodes in the graph structures and formulate them as vectorial features to be used in a neural network model. Moreover, we try to combine it with the attention layer to know which parts of AST carry a significant information to detect malicious JavaScript code.

To summarize, our contributions are as follow:

- We introduce JStrack, a static analysis method, to detect malicious JavaScript using the AST features as a graph. We applied GCN to capture the typical structure and attribute of the AST representation from malicious JavaScript samples. The GCN model is built by stacking multiple convolutional layers to be used as a layer-wise linear model in our detection system.
- We track the suspicious part of the AST graph, which corresponds to the actual JavaScript code, by using the attention layer in our proposed model. The attention scores give us significant code segments that can lead us to the signature of a malicious script.
- We evaluate our proposed approach using real-world malicious samples and collected JavaScript files from the top domain list as benign. We show that our graph-based approach can accurately detect malicious JavaScript even with the presence of the obfuscation techniques to evade the detection system. Moreover, our approach detects the obfuscation pattern of AST-graph by observing the similarity of graph structures and attributes among malicious or benign samples.

The rest of the paper is organized as follows. Section 2 provides the background of JavaScript-based attack and related works. Then, we will explain how we parse JavaScript code to get the AST representation and how we construct the graph based on that. Section 3 explains our proposed approach, which uses a graph-based model to extract the AST feature. Section 4 presents our experiment and evaluation result of our JStrack in Section 5. Finally, we provide our concluding remarks.

2 Background and Related Works

In this section, we explain the background of JavaScript-based attacks and how attackers use obfuscation technique to hide their malicious intent. We also give an overview of the AST feature as an abstract representation of JavaScript and the derivation of the graph from the characteristics of the AST features.

2.1 JavaScript-based attack

According to Web Technology surveys [16], JavaScript is the most used client-side programming language on websites, reaching about 97.4%. Because of that, malicious JavaScript code is one of the most common web security vulnerabilities that are frequently found in buttons, text, images, or pop-up pages. For instance, if a website does not sanitize angle brackets (`< >`), attackers can insert `<script></script>` to inject payload, which this tag instructs the browser to execute the JavaScript between them [21]. The injected script can be triggered when a single HTTP request runs the malicious payload and attackers did not store it anywhere on the website or when a site saves and renders it unsanitized [21].

The malicious JavaScript code generally contains some function calls that attackers usually use to execute their intended action. Examples of function calls include `document.write()`, `eval()`, `unenscape()`, `SetCookie()`, `GetCookie()`, or `newActiveXObject()` [7]. Attackers will activate the malicious payload by altering the document object model (DOM) to drop the malware or steal users' sensitive data. Due to many malicious samples have these functions, we can assume that this part of the code gives more important information about the maliciousness of code. However, in practice, attackers hide the malicious code by particular means to take advantage of the security flaw. It won't be easy to detect such kinds of payload that it can bypass the system. In addition, they utilize obfuscation techniques to hide their malicious code, making it harder to find the signature.

2.2 Related Works

Previous researches have thoroughly explored the machine learning-based method for detecting malicious JavaScript. They used various features of JavaScript and

applied a different approach to increase the performance. Ndichu et al. [12] applied the FastText model to detect the malicious JavaScript based on AST features. They tried to deobfuscate the source code to catch the identical actual malicious payload before modeling. However, their approach handles the short relationship between syntactic units in AST that they forgot to consider the edge connection. Besides that, Fass et al. [6] did a similar work that they proposed a syntactical analysis approach using a low-overhead solution that mixes AST feature extraction sequences and a random forest classifier model.

Differently, Rozi et al. [14] used bytecode sequences as the main features of JavaScript code, which is the middle language between machine and high-level code. Due to the super long problem in the bytecode sequence, they used a deep pyramid convolutional neural network (DPCNN) that contains a pyramid shape network to get a more straightforward representation. The limitation is that they have to declare all possible DOM objects in every sample to generate the sequences.

Moreover, Song et al. [15] and Fang et al. [5] used recurrent neural networks (RNNs) architectures to capture the semantic meaning of JavaScript. Song et al. [15] tried to use the Program Dependency Graph (PDG), AST, and control flow diagram (CFG), which preserve the semantic information of JavaScript. However, Fang et al. [5] only relied on AST features to capture the sequence patterns of syntactic unit sequences. Both of them applied Bidirectional Long-Short Term Memory (BiLSTM) and Long-Short Term Memory (LSTM) to learn the long-term dependencies.

3 Proposed approach

To overcome such challenges from malicious JavaScript, we propose a detection system that can predict the label of a given source code, whether it is malicious or benign. Our proposed approach uses AST as the feature of JavaScript that can define the style and semantic meaning of the source code. By analyzing this feature, we can capture the malicious intent based on the typical structure and attribute of the AST graph. We use GCN to learn the graph to have the generalization of malicious and benign samples.

3.1 Overview

We can see the entire detection system framework in Figure 1. It begins with a JavaScript file that we want to predict the malicious intent. After that, we parse it using a parser to get the AST representation, describing how programmers write the code. The output is a JSON format file where each record is a syntactic unit object based on ESTree standardization [4]. We can construct graph objects from a JSON file as a simplification of its data structure. The graph generator creates syntactic unit types as finite nodes, and the hierarchical connection among nodes is an edge of the AST graph. Next, we create two matrices, feature matrix $\mathbf{X}$ and adjacency $\mathbf{A}$, representing the feature value of

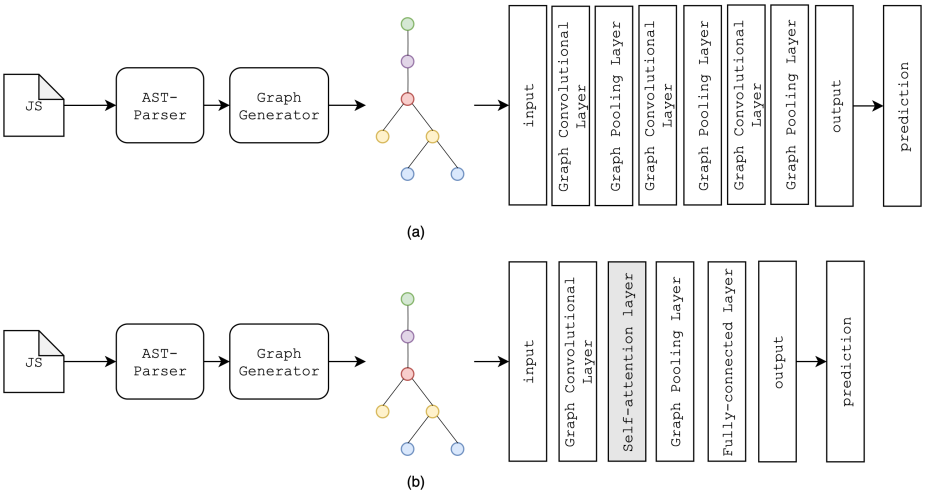

Fig. 1. The overview of proposed approach. (a) The original architecture consists of three layers of convolutional and pooling layers. (b) The combination of GCN and attention mechanism to locate the suspicious codes of JavaScript. To get the whole information of nodes, we put the pooling layer after attention layer before going to fully-connected layer.

each node and all connections of edges, respectively. The GCN is similar to the convolutional neural network (CNN) in that it consists of two main layers, the convolutional and pooling layers. The difference is that GCN applies these layers on a graph structure to get a suitable vector representation for the graph. The output is the prediction score to determine the JavaScript label.

3.2 AST graph construction

We often find many systems around us that use graph representation to solve many problems. Graph representation can render a complex system become more structured so that the problem will be easier to solve. A graph is a ubiquitous data structure and universal language consisting of a collection of objects, including a set of interactions between pairs of objects [8].

Formally, we can define graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ as a set of nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. (u, v) denotes an edge going from node $u \in V$ to node $v \in V$ [8]. We can represent a finite graph $\mathcal{G}$ in a squared matrix called adjacency matrix $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$. Each row and column indicates all nodes that a finite graph $\mathcal{G}$ has. Furthermore, edges represent entries in $\mathbf{A}$ where $\mathbf{A}[u, v] = 1$ if $(u, v) \in \mathcal{E}$ and otherwise $\mathbf{A}[u, v] = 0$. Matrix $\mathbf{A}$ will not necessarily be symmetric if graph $\mathcal{G}$ has directed edges. Some graphs also have weighted edges, where the entries in the adjacency matrix are real-values. Besides that, a graph may have an attribute or feature information for each node that using a real-valued matrix $\mathbf{X}^{|\mathcal{V}|\times m}$ where m is the feature size of nodes, and the ordering of the nodes is consistent with the adjacency matrix

A. In some cases, edges also have real-valued features in addition to discrete edge types.

We can use a graph-based approach to represent the AST feature with a tree graph structure. AST is a top-down parsing structure in which each syntactic unit has at least one hierarchical connection where the root is always a 'program' type. Based on that, we consider each syntactic unit as a node and hierarchical link as an edge. Using graph representation simplifies the AST feature in a fixed form to help the feature extraction process. This representation also allows us to capture the big picture of the source code, which shows the complexity yet the programmer's obfuscation style.

3.3 Learning AST graph feature

Suppose we have $G = \{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, ..., \mathcal{G}_N\}$, a set of all graphs in our dataset. We can define a graph $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i)$ consisting of nodes $\mathcal{V}$ and edges $\mathcal{E}$. In our problem, we assume our target for the model is $t \in \{0, 1\}$ which 0 as benign and 1 as the malicious.

Graph Convolutional Neural Networks. The basic idea of GCN is actually from convolutional neural networks (CNNs), where it also uses the convolution and pooling function for getting feature information of each node in the graph. Originally, Kipf et al. proposed GCN to solve semi-supervised classification tasks such as graph Laplacian regularization include label propagation [22], manifold regularization [1], and deep semi-supervised embedding [20]. The basic idea is to generate embedding information of nodes via neural message passing to aggregate information from all neighborhoods. GCN consists of a stack of graph convolution layers, where a point-wise non-linearity follows each layer. The number of layers is the farthest distance that node features can travel. The number of layers also influence the performance. More layers are not guaranteed to get a good result because it makes the aggregation less meaningful if it goes further.

The multi-layer network in GCN follows layer-wise propagation rule:

$$\mathbf{H}^{(l+1)} = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right). \tag{1}$$

Where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ is the adjacency matrix of the undirected graph $\mathcal{G}$ with added self-connections. $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$ and $\mathbf{W}^{(l)}$ is trainable weight matrix in specific layer. $\mathbf{H}(l) \in \mathbb{R}^{N \times m}$ is the matrix of activations in the l^{th} layer with m is the feature size of nodes; $\mathbf{H}^{(0)} = \mathbf{X}$. $\sigma(\cdot)$ stands for an activation function, such as the $\mathrm{ReLU}(\cdot) = \max(0, \cdot)$.

Attention mechanism. This mechanism is basically about paying more focus on some component that significantly influences the system. Precisely, the attention function map a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors [17]. The computation of attention function as follows:

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V} \quad (2)$$

where **Q, K, V** are query, key, and value matrices, respectively. d_k is the key of dimensions.

In this work, the attention mechanism can leverage the learning process of GCN by giving attention weight to concentrate selectively on a discrete aspect of the graph convolutional layer. We use a self-attention layer to handle long-range dependencies and have lower complexity than other layer types (e.g., convolutional or recurrent).

4 Experiments

In this section, we present our experiments to evaluate our proposed approach for detecting malicious JavaScript samples. We evaluated our framework's performance by adjusting the maximum number of nodes in each graph. Then, we compared our results with some related works that have a similar task. Finally, we give some analysis discussion to find out our limitations.

4.1 Setup

Dataset. We collect malicious and benign JavaScript datasets, where the malicious samples are from two different sources due to the difficulties of getting the real-world dataset. For our malicious samples, we mixed the dataset from Rozi et al. [14] and Ndichu et al. [12] that use some different time stamps of files from 2015 until 2017. We also confirmed that all those datasets are dangerous scripts based on the VirusTotal scanner [18]. Meanwhile, we collected JavaScript codes for benign samples by scrapping from the top domain list on the Majestic website [10], and we combined it with the benign dataset from SRILAB [13]. We consider all JavaScript codes inside popular websites as safe code without any attacking intent.

We split our dataset into two parts: training and testing. We used the training dataset for the learning purpose of our graph learning model. Otherwise, we evaluated our model with the testing dataset. We conducted 10-folds cross-validation to see our model's average performance that generalizes to an independent dataset. Because of that, the proportion between training and testing is 80% and 20%, respectively. Table 1 summarizes the number of JavaScript files that we use in our experiments.

Hyper-parameters and setup. We set optimal hyper-parameters to conduct our experiments to control the learning process. We used the Adam algorithm optimization with a 0.01 learning rate and 32 for the batch size. In addition, the feature size of the convolutional layer in GCN is 32 and using rectified linear unit (ReLU) as the activation function. For the pooling layer, we used a 50% ratio to downsample the matrix node.

Table 1. The description of our dataset that is used for training and testing process.

Label \ Dataset	Training	Testing	**Total**
Benign	97,361	24,341	121,702
Malicious	31,560	7,890	39,450
Total	128,921	32,231	**161,152**

Unlike the usual deep learning model, adding more layers does not correlate with the performance. When we work with the GNNs, this model will significantly lose the ability to learn if we have too deep layers, where we call this problem over-smoothing [23]. The main idea of over-smoothing is that all node representations look identical and uninformative after too many message passing rounds due to too many layers. Zhou et al. [22] recommended using between 2 and 4 layers to achieve an optimal solution. Therefore, we used the middle range number, three layers, in our experiments.

Moreover, we applied a data loader with disjoint mode for creating mini-batches of data in graph learning. It represents a batch of graphs with a disjoint union that gives us one big graph [11]. Figure 2 illustrates how the disjoint loader works.

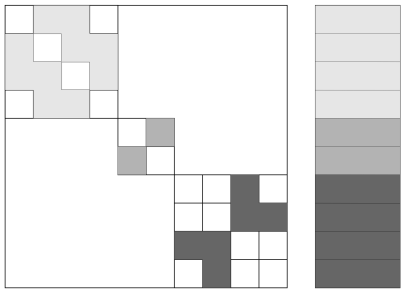

Fig. 2. Disjoint loader is a method to load dataset in graph learning process that represents batch of graphs via disjoint union. It uses zero-based indices to keep track of the different graphs.

5 Evaluation and Discussion

Due to the memory capacity reason, we could not include all nodes in the learning process. Because of that, we evaluated six different maximum nodes of the AST graph: 50, 100, 200, 500, 1000, and 2000. This experiment aims to find the sufficient nodes that we need to detect the maliciousness of JavaScript. Table

2 shows the performances (precision, recall, F1 score) for each maximum nodes setting. We can see that the performance of our method will increase in line with the number of nodes in the AST graph that we can capture. This result is in accordance with our hypothesis that AST nodes give an abstraction of the source code where all nodes give essential information. However, using 2000 nodes still give high performance even though we did not include all information. It is because AST uses the hierarchical structure that each node has summarized its successor.

Table 2. Overall performances of our detection system using graph-based approach on accuracy, precision, recall, F1 score, and AUC.

Max Nodes	Accuracy	Precision	Recall	F1 score	AUC
50	0.9864	0.9872	0.9878	0.9875	0.9878
100	0.9877	0.9881	0.9901	0.9891	0.9901
200	0.9906	0.9929	0.9937	0.9933	0.9937
500	0.9933	0.9940	0.9956	0.9948	0.9956
1000	**0.9941**	0.9953	0.9965	0.9959	0.9965
2000	0.9940	**0.9956**	**0.9971**	**0.9963**	**0.9971**

Table 3 shows the comparison between previous works and our proposed method. GCN has around 98% in terms of F1 score for our dataset with the maximum 50 nodes of the AST graph. Meanwhile, adding attention layers before fully connected layers can improve the performance by 99%. Our approaches outperform the previous works that use the FastText model based on frequency analysis of syntactic AST units. Even though the difference is relatively small, our proposed method can predict the part of the source code which gives more attention to detect malicious intent. This information will be valuable for further analysis of malicious code. Figure 4 is one of the malicious samples in our dataset that shows the attention score for each node in a graph. Moreover, the bytecode sequences feature cannot be implemented on every JavaScript samples because we have to declare all possible DOM objects.

Moreover, we found in our experiments that the malicious JavaScript has its obfuscation technique to hide the actual source code. Figure 3 (a) shows the graph visualization of malicious JavaScript code. The structure of the AST graph for malicious JavaScript has many repetitions of the subgraph that we rarely find in benign samples. Some similar styles appear many times within the same time range, indicating that attackers consistently use their obfuscation function that normal programmers will not use. On the other hand, most benign samples in Figure 3 (b) have an arbitrary structure of AST and inconsistent subgraph patterns. This result is in line with our hypothesis that benign JavaScript mostly does not use obfuscation techniques, or if it has obfuscated parts, it uses more complicated methods to protect from reverse engineering.

Table 3. Performance comparison with closely related works.

Model	Feature	F1
DPCNN[14]	Bytecode Sequence	0.9684
DPCNN+LSTM[14]	Bytecode Sequence	0.9657
DPCNN+BiLSTM[14]	Bytecode Sequence	0.9683
LSTM[12]	AST	0.9234
FastText[12]	AST	0.9873
GCN (3-layers;max 50 nodes)	AST	0.9875
GCN (w/ attention; max 50 nodes)	**AST**	**0.9935**

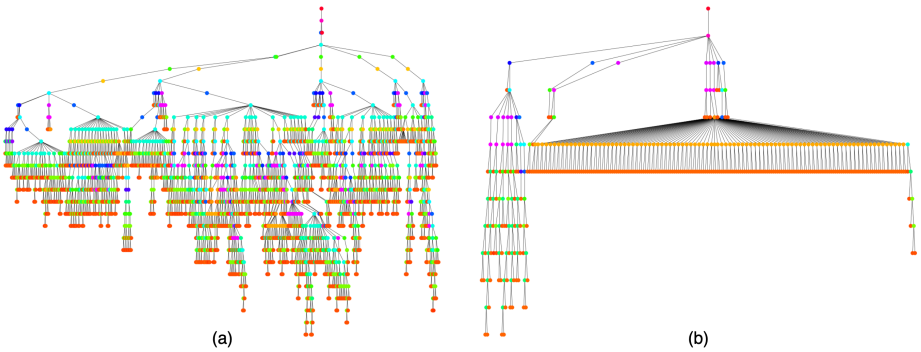

Fig. 3. A sample of AST graph that is constructed from a benign (a) and malicious (b) JavaScript file.

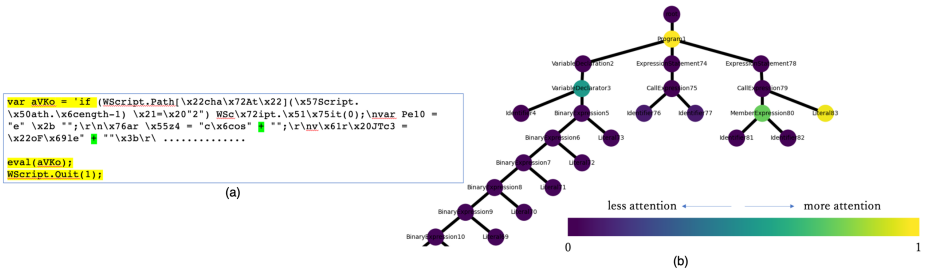

Fig. 4. (a) A malicious sample where the highlight parts are the vital parts to execute the code. (b) The AST representation of the malicious code that each node has a color represents the attention score. Some nodes have high scores that correlate to the vital part of malicious code.

However, there are two limitations to our proposed method that we are considering. First, we lose detailed information about malicious code due to using the AST feature to represent JavaScript. In the AST graph, we merely use the syntactic units and omit component details for each unit, which may contain the essential information for our detection system. Then, the use of deep/machine learning does not always consider uncertainty in the prediction task. It relies on statistical assumptions about the distribution of the dataset to train the model. Consequently, adversaries-based attacks can exploit the machine learning model to disrupt the analysis process and make false detection.

6 Conclusions and Future Works

In this paper, we proposed an alternative approach to detect malicious JavaScript based on the analysis of AST representation. The syntactical structure of JavaScript can give more comprehensive information about the source code's semantic meaning to capture the generalization of malicious signatures to overcome future attacks. GCN successfully encodes the whole AST graph via a neural message from its local neighborhood that leads to high detection performance. Additionally, the attention layers also help us locate suspicious parts of the malicious samples, significantly contributing to the detection system. As future plan, we will extend our research for future work to detect malicious websites based on encoded JavaScript information. We will explore more about other JavaScript features that probably increase the performance.

Acknowledgements

This research was partially supported by the Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Scientific Research (B) 21H03444.

References

1. Belkin, M., Niyogi, P., Sindhwani, V.: Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. J. Mach. Learn. Res. **7**, 2399–2434 (Dec 2006)
2. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceedings of the 19th International Conference on World Wide Web. p. 281–290. WWW '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1772690.1772720, https://doi.org/10.1145/1772690.1772720
3. Douligeris, C., Mitrokotsa, A.: Ddos attacks and defense mechanisms: classification and state-of-the-art. Computer Networks **44**(5), 643–666 (2004). https://doi.org/https://doi.org/10.1016/j.comnet.2003.10.003, https://www.sciencedirect.com/science/article/pii/S1389128603004250
4. The estree spec. https://github.com/estree/estree, accessed: 2021-01-20
5. Fang, Y., Huang, C., Liu, L., Xue, M.: Research on malicious javascript detection technology based on lstm. IEEE Access **6**, 59118–59125 (2018)

6. Fass, A., Krawczyk, R.P., Backes, M., Stock, B.: Jast: Fully syntactic detection of malicious (obfuscated) javascript. In: DIMVA (2018)
7. Gupta, S., Gupta, B.: Enhanced xss defensive framework for web applications deployed in the virtual machines of cloud computing environment. Procedia Technology **24**, 1595–1602 (2016). https://doi.org/https://doi.org/10.1016/j.protcy.2016.05.152, https://www.sciencedirect.com/science/article/pii/S2212017316302419, international Conference on Emerging Trends in Engineering, Science and Technology (ICETEST - 2015)
8. Hamilton, W.L.: Graph representation learning. Synthesis Lectures on Artificial Intelligence and Machine Learning **14**(3), 1–159 (2020)
9. Kamkar, S.: phpwn: Attacking sessions and pseudo-random numbers in php. In: Blackhat (2010)
10. Majestic. https://majestic.com/, accessed: 2021-01-26
11. Data modes. https://graphneural.network/data-modes/, accessed: 2021-04-17
12. Ndichu, S., Kim, S., Ozawa, S.: Deobfuscation, unpacking, and decoding of obfuscated malicious javascript for machine learning models detection performance improvement. CAAI Transactions on Intelligence Technology **5** (06 2020)
13. Raychev, V., Bielik, P., Vechev, M., Krause, A.: Learning programs from noisy data. SIGPLAN Not. **51**(1), 761–774 (Jan 2016)
14. Rozi, M.F., Kim, S., Ozawa, S.: Deep neural networks for malicious javascript detection using bytecode sequences. In: 2020 International Joint Conference on Neural Networks (IJCNN). pp. 1–8 (2020)
15. Song, X., Chen, C., Cui, B., Fu, J.: Malicious javascript detection based on bidirectional lstm model. Applied Sciences **10**(10) (2020). https://doi.org/10.3390/app10103440, https://www.mdpi.com/2076-3417/10/10/3440
16. Usage statistics of javascript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript, accessed: 2021-05-08
17. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, u., Polosukhin, I.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. p. 6000–6010. NIPS'17, Curran Associates Inc., Red Hook, NY, USA (2017)
18. Virustotal. https://www.virustotal.com/gui/, accessed: 2021-01-15
19. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. pp. 171–180 (2008). https://doi.org/10.1145/1368088.1368112
20. Weston, J., Ratle, F., Collobert, R.: Deep learning via semi-supervised embedding. In: Proceedings of the 25th International Conference on Machine Learning. p. 1168–1175. ICML '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1390156.1390303, https://doi.org/10.1145/1390156.1390303
21. Yaworski, P.: Real-world bug hunting: A field guide to web hacking **14**(3) (2019)
22. Zhou, K., Dong, Y., Wang, K., Lee, W.S., Hooi, B., Xu, H., Feng, J.: Understanding and Resolving Performance Degradation in Graph Convolutional Networks. arXiv e-prints arXiv:2006.07107 (Jun 2020)
23. Zhu, X., Ghahramani, Z., Lafferty, J.D.: Semi-supervised learning using gaussian fields and harmonic functions. In: Fawcett, T., Mishra, N. (eds.) Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA. pp. 912–919. AAAI Press (2003), http://www.aaai.org/Library/ICML/2003/icml03-118.php