Computers & Security 93 (2020) 101764

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Detecting malicious JavaScript code based on semantic analysis

Yong Fang[a], Cheng Huang[a,*], Yu Su[a], Yaoyao Qiu[b]

[a] College of Cybersecurity, Sichuan University, Chengdu, China
[b] College of Electronics and Information Engineering, Sichuan University, Chengdu, China

ARTICLE INFO

Article history:
Received 10 October 2019
Revised 16 February 2020
Accepted 18 February 2020
Available online 19 February 2020

Keywords:
Malicious JavaScript detection
Abstract syntax tree
Attention mechanism
Static analysis
Bi-LSTM
FastText

ABSTRACT

Web development technology has undergone tremendous evolution, the creation of JavaScript has greatly enriched the interactive capabilities of the client. However, attackers use the dynamics feature of JavaScript language to embed malicious code into web pages for the purpose of drive-by-download, redirection, etc. The traditional method based on static feature detection is difficult to detect the malicious code after obfuscation, and the method based on dynamic analysis has low efficiency. To overcome these challenges, this paper proposes a static detection model based on semantic analysis. The model firstly generates an abstract syntax tree from JavaScript source codes, then automatically converts them to syntactic unit sequences. FastText algorithm is introduced to training word vectors. The syntactic unit sequences are represented as word vectors which will be input into Bi-LSTM network with attention mechanism. The detection model with Bi-LSTM network with attention mechanism is the key to detect malicious JavaScript. We experimented with the dataset using a five-fold cross-validation method. Experiments showed that the model can effectively detect obfuscated malicious JavaScript code and improve the detection speed, with a precision of 0.977 and recall of 0.974.

1. Introduction

The steady development of the Internet technology greatly enriches the Internet application, which not only increasingly affects human's life in all aspects, but also has exposed more and more cybersecurity risks. Attackers can implement malicious behavior by injecting malicious JavaScript code into web pages, such as spreading Trojan viruses, obtaining user sensitive information, and mining data (Lubowicka, 2019.1; Sarmah et al., 2018; Trendmicro, 2019.1). According to the "2018 Annual Security Report" published by Tencent Anti-Virus Laboratory (antivirus lab, 2019.1), on the basis of the statistics of non-PE (portable executable) virus samples collected in 2018, VBS virus rank the first in annual average intercepted non-PE virus, accounting for 50.65% of all viruses, and the TOP2 is JavaScript virus, accounting for 23.21% of all viruses.

The malicious JavaScript code on the web pages is flexible and changeable. Attackers often use encryption or obfuscation techniques to avoid detection, which makes malicious code with strong concealment and causes the detection is hard to do. Some tools have the ability to deobfuscate a part of JavaScript code but not completely. That is to say, after the deobfuscation, it needs to be decoded manually. Thus, the code deobfuscation is complicated and costly. Web applications with openness and extensiveness make it play a vital role in propagating malicious code. Malicious web pages mainly inject malicious JavaScript code to achieve their goals. Therefore, malicious JavaScript code detection in web pages is of great significance. Aiming at the flood of malicious script code and the inefficiency of malicious code detection, this paper proposes a malicious JavaScript code detection model based on semantic analysis to improve the accuracy of model detection and reduce the resources and time cost by malicious code detection.

The malicious JavaScript code detection model based on semantic analysis has the following innovations:

(1) This paper introduces the semantic analysis method to parse the source code into an abstract syntax tree (AST) and traverse the abstract syntax tree to extract the syntactic unit sequences as the feature of the code.
(2) This paper adopts the FastText model for word vector training of syntactic unit sequences. By effectively learning affix information through subword, the FastText model achieves very good performance in training word vectors. Experiments show that the FastText model is more suitable for the corpus of the thesis than Word2Vec (Mikolov et al., 2013), which can improve the classification performance of the model.

* Corresponding author.
E-mail addresses: yongfangscu@gmail.com (Y. Fang), opcodesec@gmail.com (C. Huang), fishsu@stu.scu.edu.cn (Y. Su), fionsky911@gmail.com (Y. Qiu).

https://doi.org/10.1016/j.cose.2020.101764
0167-4048/

(3) This paper proposes a malicious JavaScript detection model based on Bi-LSTM(Bidirectional Long Short-Term Memory) network (Graves and Schmidhuber, 2005) with attention mechanism. The Bi-LSTM network can make full use of the context information of the sequence, and the attention mechanism introduced can capture the key fragments of the input sequence to improve the accuracy of detection model.

This paper has been organized in the following way. Section 2 gives a brief overview of the recent related work about malicious JavaScript detection and introduces the background of malicious JavaScript detection. Section 3 presents methods and techniques for malicious JavaScript detection. Section 4 analyses the experimental results of the model and the discussion. And conclusions are presented in Section 5.

2. Related work

Due to the serious damage caused by malicious JavaScript code, it has attracted broad attention to the security research area. At present, a number of network security scholars and experts have proposed a variety of methods in research articles to detect malicious JavaScript code. According to whether malicious JavaScript code is executed during the detection process, previous research work has been focused on three types of analysis methods: static analysis, dynamic analysis, and combination of static analysis and dynamic analysis.

2.1. Static analysis method

The static analysis method mainly matches the code to be detected with the malicious code feature database to determine whether the code contains suspicious keywords or fragments. It also extracts the semantic structure of the source code, key strings, and functions, and then builds a detection model using machine learning.

Curtsinger et al. (2011) extract the hierarchical features of the JavaScript abstract syntax tree to identify syntax elements, and use a Bayesian model to judge, which with highly predictive. Meanwhile, they propose a low-overhead solution named ZOZZLE for detecting and preventing JavaScript malware, that is fast enough to be deployed in the browser. For the detection of obfuscated code, (Seshagiri et al., 2016) propose a mostly static approach called AMA (Amrita Malware Analyzer), a framework capable of detecting the presence of malicious code through static code analysis of web pages. But this approach targets only few among many possible obfuscation strategies. Conventional approaches often employ signature and heuristic-based methods, which almost cannot detect new deformed malicious JavaScript code. Ndichu et al. (2018) adopt a machine-learning approach to feature learning called Doc2Vec, which is a neural network model that can learn context information from JavaScript code, then use these extracted features to judge the maliciousness of JavaScript code. Fass et al. (2018) present JaSt, a low-overhead solution that combines the extraction of features from the abstract syntax tree with a random forest classier to detect malicious JavaScript instances. Aiming to solve the problem of some model lack semantic information, they also propose JStap (Fass et al., 2019b), a modular static JavaScript detection system, which extends the detection capability of existing lexical and AST-based pipelines by leveraging control and data flow information. Shen et al. (2018) propose a novel method of detecting the JavaScript malware by using a high-level fuzzy Petri net (HLFPN) to extract malicious features from JavaScript code. Some attack consists of changing the constructs of malicious JavaScript samples to reproduce a benign syntax. So (Fass et al., 2019a) automatically rewrite the abstract syntax trees of malicious JavaScript inputs into existing benign ones. In a word, these detection works based on static methods are mainly focused on using machine learning methods and processing the original JavaScript code.

2.2. Dynamic analysis method

The dynamic analysis method often extracts behavioral features during code executioner and uses honeypots to simulate the browser environment to execute JavaScript code, then records and analyzes execution information which could provide additional details of malicious behavior.

Honeypot is a kind of computer resource used for detection and attack, so as to study and analyze the means and intention of attackers (Nawrocki et al., 2016). Cova et al. (2010) propose a low interaction honeypot tool named JSAND which uses the HtmlUnit with Rhino engine simulation environment to simulate the client. By extracting redirect target and number of times, string definition, the ratio of dynamic code (such as the *eval*s and *setTimeout*) execution frequency, the number of similar shellcode strings, the string manipulation functions (such as *concat* and *substring*) and some other features to detect malicious JavaScript code. Honeypot detection technique is highly accurate, but the detection speed is slow. In addition to simulating the browser environment to extract code characteristics, scholars have also studied other methods to extract code dynamic execution characteristics. Kim et al. (2012) propose JsSandBox framework, which is a dynamic analysis method that monitors and analyzes the behavior of malicious JavaScript code through Internal Function Hooking (IFH) and solves the problem that Application Programming Interface (API) hooking cannot monitor functions.

2.3. Combination analysis method

Static analysis is characterized by its simplicity of design and ease of use, but some malicious behavior on a malicious Web page is triggered only in a specific execution environment which causes these types of malicious code may not be detected. Dynamic analysis methods usually have disadvantages such as high resource cost and slow detection speed. Analyzing problems from multiple perspectives is called hybrid analysis. In recent years, there have been many studies(Habtamu, 2019; Wang et al., 2020) on hybrid analysis in various fields. So lots of scholars and experts put forward some methods combining static analysis and dynamic analysis on JavaScript detection.

Rieck et al. (2010) propose a system called CUJO that can automatically detect and prevent Web page attack. It is embedded on the Web agent to extract the static and dynamic characteristics of the code in real-time and analyze the malicious patterns with a support vector machine (SVM). Kapravelos et al. (2013) propose a malicious JavaScript code detection system Revolver which extracts features through the abstract syntax tree of the code, and combines with dynamic analysis to compare the similarity between code samples. The two samples with high similarity are classified into the same category, that is, the codes with high similarity with malicious codes are identified as malicious codes, and the codes with high similarity with normal codes are identified as normal codes. However, the system needs to manually check the results produced by the system and has not been fully automated yet. Wang et al. (2015) combine the dynamic and static analysis method to extract the code text information, program structure, dangerous function calls, and other characteristics, and input them into the machine learning model to identify malicious code. Besides, according to the attack feature vector and dynamic execution trajectory of the code, the identified malicious code is divided into eight known attack types.

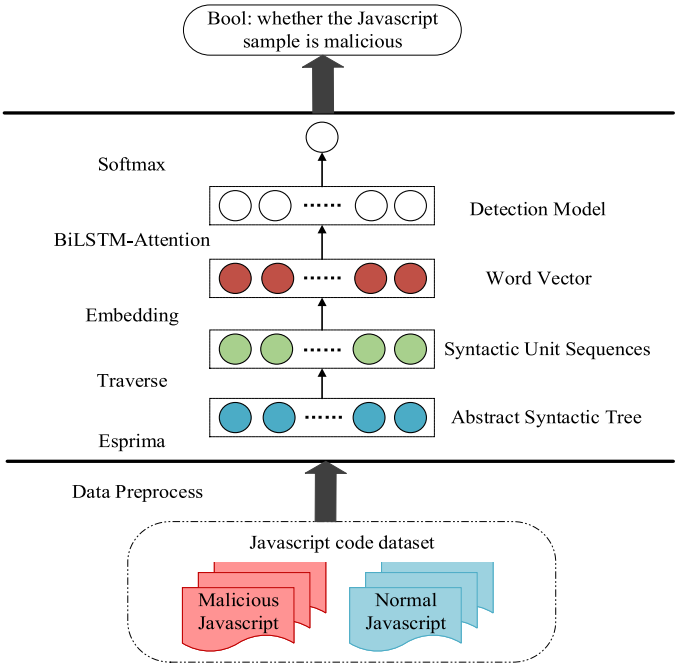

Fig. 1. The Framework for Detecting Malicious JavaScript Code Method in this Paper.

We use a static analysis method based on the following considerations. The dynamic analysis method significantly complicates the implementation, which requires high computational overhead like consuming many resources and causing a long execution time. And executing an attack code within the detector itself which could potentially be exploited by attackers. While static analysis has the advantages of high detection efficiency and small resource occupation. Especially for malicious code employing encryption and obfuscation techniques, there is no technology or tool that can completely and automatically in handle of de-obfuscation currently. So it is not easy to detect the code after obfuscating the code. In this paper, we propose a static analysis method based on semantic analysis with better detection of obfuscated code. We will describe it in detail in the next section.

3. Proposed method

The framework of our proposed method consists of two parts, the data processing and the detection model. As we can see from Fig. 1, we first parse the sample into an abstract syntax tree and traversed the nodes of the abstract syntax tree in a depth-first way to get syntactic unit sequences. Second, we use FastText model to train the syntactic unit sequences for generating word vectors. Finally, corresponding word vectors of the sample's syntactic unit sequences are inputted to the deep learning model which can analyze and detect malicious JavaScript code.

3.1. Syntactic unit sequences extraction

Abstract syntax tree is a tree representation of the abstract syntactic structure of source code (wikipedia, 2019). That is, for source code in a specific programming language, the statements in the source code are mapped to each node of the tree by constructing a syntax tree. The abstract syntax tree often used to code checking, analysis, conversion, etc. For example, the browser will convert the JavaScript source code into an abstract syntax tree through the JavaScript parser before executing. The abstract syntax tree is used intensively during semantic analysis which means it is a key component in semantic analysis. The syntax tree is useful for our method of static program analysis. In this paper, we use Esprima (Hidayat, 2018) to parse the JavaScript source code into an abstract syntax tree. Esprima is a high-performance JavaScript parser that takes a string representing JavaScript program and produces a syntax tree. It produces 69 different types of nodes including Program type nodes, Statement type nodes, Expression type nodes, Declaration type nodes, and Pattern type nodes, etc. Different code snippets are mapped into different types of nodes which we called syntactic unit as well. Take the following JavaScript code for example, the code is parsed by Esprima and mapped into an abstract syntax tree, then the abstract syntax tree is traversed by depth-first to generate the syntactic unit sequences.

Table 1
An Example of Syntactic Unit Sequences.

No.	Syntactic node type
1	Identifier
2	Literal
3	Identifier
4	MemberExpression
5	Literal
6	Identifier
7	CallExpression
8	VariableDeclarator
9	VariableDeclaration
10	Program

The process of generating abstrct syntax tree is shown in Fig. 2 (a) and extracting the syntactic nodes and visualizing them in a tree diagram is shown in Fig. 2 (b).

Among these syntactic nodes, Program is the root node, VariableDeclaration represents the variable declaration, VariableDeclarator is the description of the variable declaration, Identifier is named that used to name variables, and CallExpression is the function call expression, which represents the statement like *func("hello")*. MemberExpression is a member expression node that represents a reference object member. Literal represents a literal value, which may be a string, a boolean, a number, a null or a regular expression, and so on.

By depth-first traversing the nodes of the abstract syntax tree, the syntactic unit sequence text can be finally obtained for later training, as shown in Table 1.

3.2. Word vector extraction based on FastText

After extracting the syntactic unit sequences, the sequence is discriminated against by the text classification technique. In this process, the sequence needs to be converted into what can be processed by our model, namely, the word vector. We need to know that the result of the word vector model training determines the effectiveness of the deep learning model to a large extent. Therefore, for the corpus in this article, which type of word vector model is more suitable need to figure out.

Word2Vec is one of the most popular training tools for extracting word vector feature (Mikolov et al., 2013). It transforms the text content processing into a vector in the K-dimensional vector space and uses the similarity in the vector space to represent the semantic similarity of the text. The Word2Vec mainly contains two training models, the Skip-gram model and the Continuous Bag Of Words (CBOW), as shown in Fig. 3. Both Skip-gram and CBOW model converted into word vectors adopted weight the distance between the words in context and the middle word. It means a few word order relationship is retained, but not fully used.

Since the congeneric syntactic unit of JavaScript has similar affixes, such as declaration class and the expression class, as shown in Table. 2. The internal structure of the syntactic unit can be re-

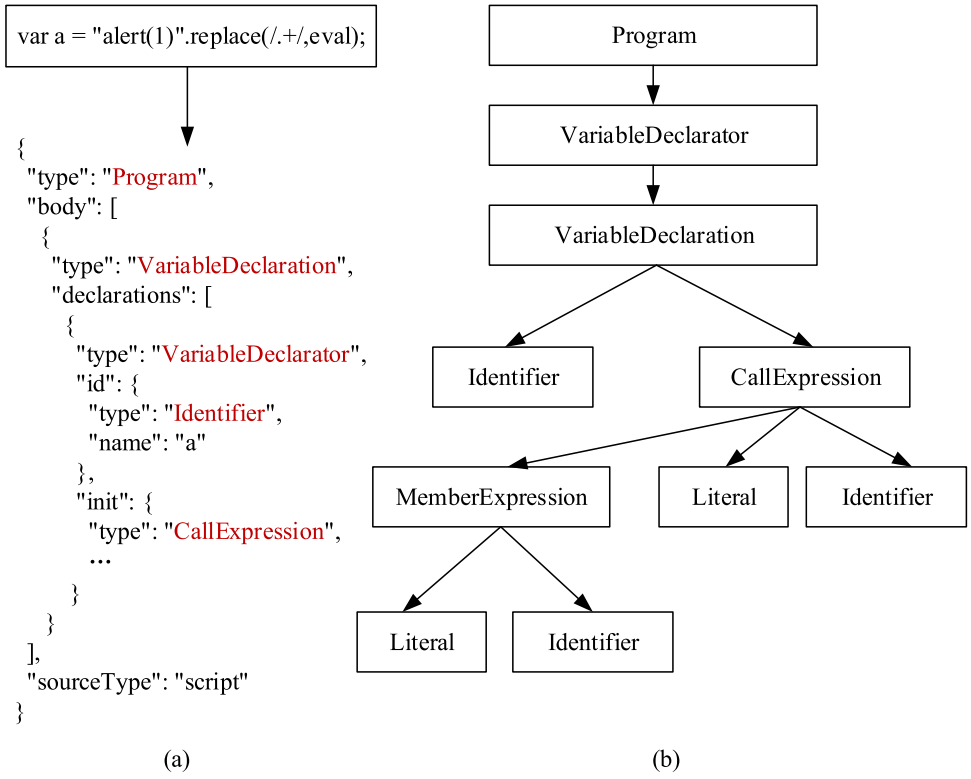

Fig. 2. An Example of Processing an Abstract Syntax Tree.

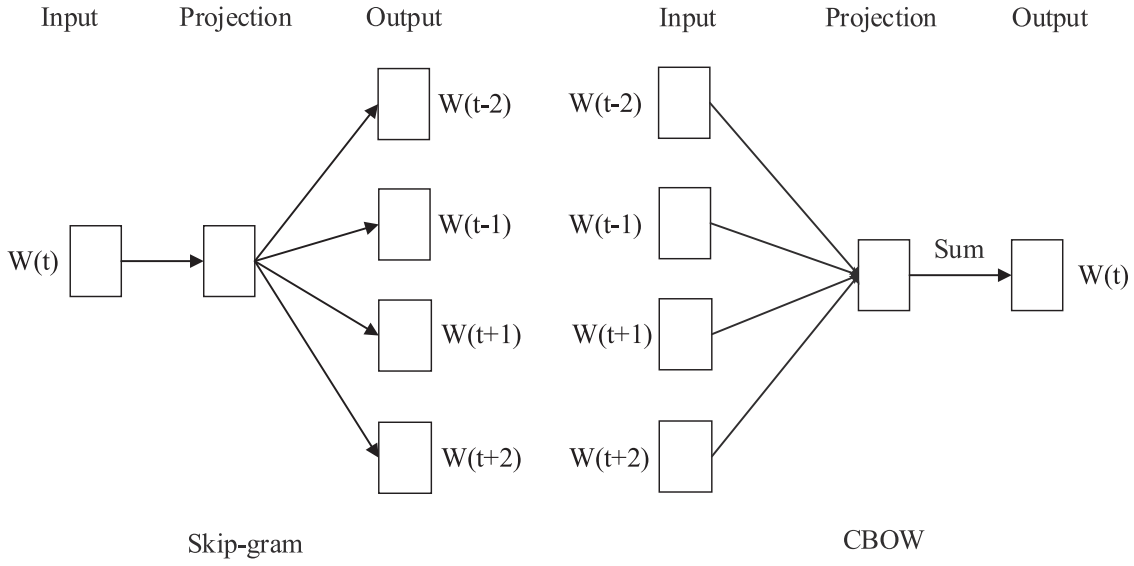

Fig. 3. Skip-gram Model(left) and CBOW Model(right).

flected by the semantics of the current node, which should be considered during training the word vector. So the word vector training of syntactic units sequence based on the FastText model is proposed in this paper.

FastText word vector model is an extension based on Word2Vec word vector model (Bojanowski et al., 2017). Word2Vec model ignores the internal structure of words, while FastText uses subword information, which represents each word into a character-level n-gram bag of words, and the word vector representation of a word is associated with each n-gram character. Take a syntactic unit "ClassBody" where n=3 as an example, the word can be represented as 9 n-gram character sets and a complete undivided word "<ClassBody>". The segmentation result is shown in Fig. 4. where "<" stands for beginning and ">" for the end, to distinguish prefixes and suffixes respectively.

For a word w, the n-gram character set of the word is expressed as $\mathcal{G}_w \subset \{1, \ldots, G\}$, the word vector of each n-gram g is denoted as z_g, the scoring function is as shown in Eq. (1).

$$s(w, c) = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g^{\top} \mathbf{v}_c \tag{1}$$

Another selection criterion is that the word vector similarity of similar words should be large, and the word vector similarity

Table 2
The Examples of the JavaScript Syntactic Units with the Same Suffix.

Affix	Examples
Statement	LabeledStatement ReturnStatement SwitchStatement
Declaration	ImportDeclaration ExportDeclaration VariableDeclaration ClassDeclaration
Element	RestElement SpreadElement
Expression	UpdateExpression AwaitExpression UnaryExpression BinaryExpression LogicalExpression

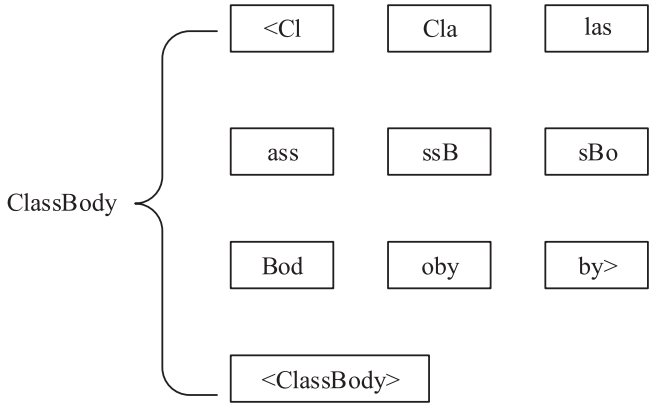

Fig. 4. The Participle Result of "ClassBody" When n-gram=3.

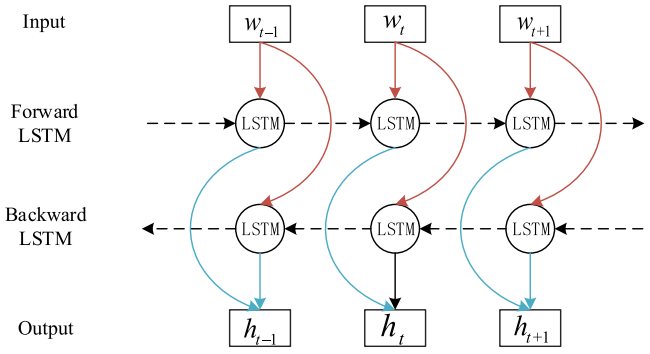

Fig. 5. The Bi-LSTM Network Structure.

of dissimilar words should be small. For example, *WhileStatement* and *ForStatement* have the same suffix *Statement* and *While* and *For* has similar semantics, but *WhileStatement* and *Identifier* are different suffixes. We can conclude the *WhileStatement* and *ForStatement* have a large similarity and the *WhileStatement* and *Identifier* have a small similarity.

A good word vector model can take care of these aspects. The fastText word vector model can enrich the word vector through the subword information and capture the effective content from the internal structure of the word. Therefore, the CBOW model of FastText is superior to the model in word2vec in this corpus. Experiments have also verified this conclusion.

3.3. Malicious JavaScript detection model

It is necessary to consider the characteristics of the input dataset when selecting a deep learning algorithm. One of the most obvious features of the syntactic unit sequence is that it is sequential. In order to use the syntactic unit sequences for reliable classification, a deep learning network that can mine the sequence relationship seems to be critical. Therefore, this paper selects the LSTM (long and short time memory) (Chung et al., 2014) network model to train the syntactic unit sequences of the code. In order to ensure that the sequence context is fully utilized, this paper uses Bi-LSTM two-way deep learning network.

RNN (recurrent neural network) can handle data with timing relationships, but long-distance dependencies will lead to gradient disappearance and gradient explosion problems (Hochreiter and Schmidhuber, 1997). The LSTM model is a special structure of RNN for solving long-distance dependency problems. LSTM stores long-term states in the cell state, and also introduces a gate structure to get the ability which reserve or forget the information to the cell state. LSTM has three gate structures, called input gate, output gate and forget gate. As the information enters the model, the structure will retain the information in accordance with the rules, forget the information that does not match, thus control the cell state.

The formula for the input gate, the oblivion gate, and the output gate is as follows:

$$\begin{aligned} i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \\ f_t &= \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \\ o_t &= \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \end{aligned} \tag{2}$$

Although the LSTM model solves the problem of long-distance and can effectively utilize the semantic dependence of the text above, in the syntactic unit sequences of the code, the text above and the text below are equally important and should be fully utilized. Graves and Schmidhuber (2005) used the LSTM unit instead of the RNN neuron to form a bidirectional neural network model based on the bidirectional RNN model proposed by Schuster and Paliwal (1997). The structure of the Bi-LSTM is shown in Fig. 5. In Fig. 5, the input of Bi-LSTM is not only into forward LSTM but into backward LSTM and the output has resulted from the two forward LSTM and backward LSTM.

Instead of using LSTM, the Bi-LSTM consists of forward LSTM network which can utilize semantic dependencies from the text above, and the backward LSTM network which can utilize the semantic dependencies from the text below. So the Bi-LSTM model can fully utilize the context information of the sequence. It is worth to be noted that malicious JavaScript code usually mixed more than one obfuscation techniques together. However, normal JavaScript code employes fewer obfuscation techniques which allow us to use this as a criterion for distinguishing between the two categories. As syntactic unit sequences represent JavaScript source code's semantic information and word vectors implicit location information of source code, context information of the sequences is key to accurately detect malicious JavaScript code.

Supposing there is a JavaScript code syntactic unit sequence S, it made up of n syntactic units, where $w_i (1 \le i \le n)$ represents the word vector of each syntactic node, as shown in Eq. (3). These syntactic units are inputted into the Bi-LSTM model.

$$S = (w_1, w_2, w_3, \ldots, w_n) \tag{3}$$

h_t represents the output at time t, which consists of a forward hidden vector $\overrightarrow{h_t}$ and a backward hidden vector $\overleftarrow{h_t}$, as shown in Eq. (4).:

$$h_t = \left[\overrightarrow{h}_t, \overleftarrow{h}_t\right] \tag{4}$$

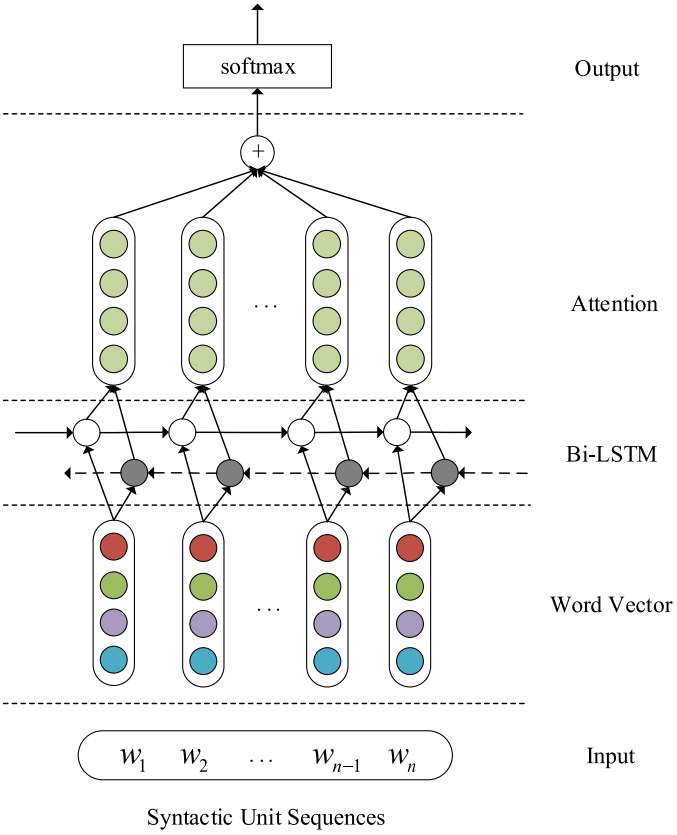

Fig. 6. The Structure of the Detection Model Network.

among them,

$$\overrightarrow{h_t} = \overrightarrow{LSTM}\left(w_t, \overrightarrow{h_{t-1}}\right)$$
$$\overleftarrow{h}_t = \overleftarrow{LSTM}\left(w_t, \overleftarrow{h_{t+1}}\right) \quad (5)$$

After passing through the Bi-LSTM network, the output H is represented as Eq. (6).

$$H = (h_1, h_2, \ldots, h_n) \quad (6)$$

For a sample of malicious JavaScript code in the wild, there are actually a lot of benign code fragments, while malicious code may be only a small fraction. Thus we think in identifying whether a sample to be malicious samples, different code to the contribution of the whole sample is not the same. So the attention mechanism (Vaswani et al., 2017) is introduced into the model, selecting important content from the syntactic unit sequences and focusing on, learning the more important information and ignoring other information not so important. Combined with the attention mechanism, the model can reduce the computation of high-dimensional input data, weight different syntactic units according to their importance, drop down the data dimension, and enable them to focus on the useful information related to the current output of the input data, so as to increase the accuracy of the output.

The overall network of the model in this paper is shown in Fig. 6.

The input layer is the extracted syntactic unit sequences. After training the word vector, the word vector representation of each syntactic unit is obtained. Then putting the word vector into the Bi-LSTM network, the output of this layer will be used as the input of the attention layer. From the Eq. (6)., the output H of the Bi-LSTM network can be obtained. After the output H_t at time t passes the tanh function, the vector u_t is obtained, and then the attention weight value a_t is obtained by softmax function, as the Eqs. (7) and (8). shows:

$$u_t = \tanh\left(w_{s_1} \cdot h_t + b\right) \quad (7)$$

$$a_t = \text{softmax}\left(u_t w_{s_2}\right) \quad (8)$$

Where w_{s_1} and w_{s_2} represent the parameter vector, b is the bias term, and finally the output of all LSTM networks are weighted and summed to obtain the output vector s of the attention layer, as shown in Eq. (9).:

$$s = \sum_{t=1}^{n} a_t H_t \quad (9)$$

Finally, the softmax function is used to calculate the probability distribution of the sequences belonging to each class, as shown in Eq. (10).:

$$p\left(y_{\text{pred}}|s\right) = \text{softmax}\left(W_S + b\right) \quad (10)$$

Where W and b are the learning parameters of the classifier.

Table 3
Experimental environment configuration.

Designation	Information
Operating System	Ubuntu 16.04.3 LTS
System Configuration	CPU:Intel i7-7700,memory 16G GPU:GeForce GTX 1060 6G
Python Library	Genism 3.1.0 FastText 0.8.3 Keras 2.1.1 Scikit-learn 0.19.1 Matplotlib 2.1.0
Node.Js	Node.js 9.4.0 esprima 4.0.1

4. Experimental evaluation

4.1. Experimental design

The model is built on Ubuntu. The sample preparation and syntactic unit sequences extraction are developed in the NodeJS written in JavaScript language. And the process of word vector training and model building are implemented by Python. The experimental environment configuration is shown in Table. 3.

In this paper, we crawled and collected 26,700 samples in total. By crawling the top 2000 websites from Alexa (Cooper, 2018), 21,700 normal samples were collected after cleaning. The malicious samples were from Github (HynekPetrak, 2017) and the publicly available data from Curtsinger et al. (2011), totaling 5000 after cleaning and preprocessing. Among these malicious samples, 942 samples are from Curtsinger et al. (2011) which all of them are employing obfuscation techniques and the rest are randomly selected from GithubHynekPetrak (2017) which are not sure for the fraction of obfuscated samples but the author said "All or most". We use 5-fold cross-validation to train our dataset on different models. We choose traditional machine learning algorithms as basic comparative experiments based on which algorithms are effective in malicious application detection filed (Wang et al., 2014). Cross-Validation is a very powerful tool in machine learning and deep learning. Different samples of data or different partitions of the data to form training and test set may cause the resulting performance evaluation to be optimistically biased. Cross-Validation can utilize data better and generally result in a less biased model. Because it ensures that every sample in the original dataset has been in training and test set. In addition, we divided the dataset into two parts: 80% of the samples are used for the training model, 20% of the samples are used as the test set. The detailed data distribution is shown in Table. 4. We use this 8: 2 segmented data for experiments to compare with Fang et al. (2018).

Table 4
Statistics of Comparative Experiment Dataset.

Dataset	Training set	Test set	Total
Normal	17,374	4326	21,700
Malicious	3986	1014	5000
Total	21,360	5340	26,700

The performance evaluation is done using statistical measures of binary classification, as details are described as follows. Detection accuracy is defined in Eq. (11). It is a proportion of instances that are correctly predicted by the classifier.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{11}$$

Where True Positive (TP) is the number of malicious samples that are correctly identified, True Negative (TN) is the number of benign samples that are correctly identified, False Negative (FN) is the number of malicious samples that are mistakenly classified as a benign sample and False Positive (FP) is the number of benign samples that are mistakenly classified as a malicious sample. Recall is the fraction of the total amount of relevant instances that were correctly predicted.

Recall and precision are defined in Eqs. (12) and (13):

$$Recall = \frac{TP}{TP + FN} \tag{12}$$

$$Precision = \frac{TP}{TP + FP} \tag{13}$$

F1-score is the harmonic mean of precision and recall and defined in Eq. (14):

$$F_1 = \frac{2 \cdot P \cdot R}{P + R} \tag{14}$$

4.2. Experimental results

The experiment compares the performance of four traditional machine learning algorithms that are logistic regression, support vector machine, naive bayesian and random forest and two deep learning algorithms LSTM and Bi-LSTM with Bi-LSTM combined with attention mechanism. All algorithms use the accuracy, precision, recall, and F1 value indicators to measure the results, as shown in Table. 5. Note that these indicators are weighted averages.

It can be seen from the table that the BiLSTM-Attention model works best, with a precision rate of 0.974 and a recall rate of 0.974. The performance of the Bi-LSTM model and Fang et al. (2018) which they used LSTM model to detect is closely followed, and the Random Forest and SVM perform well but relatively worse. The Naive Bayesian performance is slightly worse, the precision is only 0.771, and the recall is 0.718 respectively.

The BiLSTM-Attention model is superior to the Fang et al. (2018) and Bi-LSTM model in all aspects, demonstrating that the combination of Bi-LSTM and attention mechanism can use the context information effectively, extract the key segments in the code, and improve the classification performance of the model.

The ThreatBook Cloud Sandbox (Threatbook, 2019) integrates 25 authoritative anti-virus engines, including Rising, Avast, Microsoft MSE, etc. After upload test data to the ThreatBook Cloud Sandbox platform by open API, the uploaded files which are suspicious can be detected, and the scan results of 25 anti-virus engines can be easily obtained. In this paper, 1000 malicious samples and 1000 benign samples are selected randomly then we upload the 2000 samples to the ThreatBook Cloud Sandbox for scanning, and the best performing 7 detection engines are selected for comparison. The results are shown in Table. 6. Our model is not only better than the start-of-art methods but also better than the security vendors on the market.

As is shown in the Table. 6 that the best performance of the authoritative detection engine is Rising, and the detection rate is 0.978. Compared with the other 7 types of authoritative detection engines, the recall of our BiLSTM-Attention model is 0.993, which higher than other anti-virus engines, but 2 out of 1000 benign samples are judged to be malicious samples, the false alarm rate is 0.002, while other anti-virus engines are 0. In this paper, the syntactic unit sequence is the only one detection feature, which maybe leads to misclassification of the classifier. Compared with these authoritative anti-virus scanning engines, our proposed model still performs well.

Table 5
Comparison of Malicious JavaScript Detection Results with Different Classification Methods.

Dataset segmentation method	Model	Accuracy	Precision	Recall	F1-score
5-fold cross-validation	Naive Bayesian	0.718	0.771	0.718	0.738
	Logistic Regression	0.925	0.929	0.925	0.927
	Support Vector Machine	0.980	0.973	0.920	0.946
	Random Forest	0.982	0.973	0.930	0.951
	Bi-LSTM	0.971	0.974	0.971	0.972
	BiLSTM-Attention	0.974	0.977	0.974	0.975
80% training : 20% test	Fang et al. (2018)	0.9951	0.9955	0.9721	0.9837
	Bi-LSTM	0.992	0.993	0.992	0.992
	BiLSTM-Attention	0.993	0.994	0.993	0.993

Table 6
Results compared with different detection engines.

Antivirus engine	The number of checkout	Recall	The number of false positive	False positive rate
AVG	855	0.855	0	0
Tencent	916	0.916	0	0
Huorong	918	0.918	0	0
ESET	952	0.952	0	0
MSE	957	0.957	0	0
Avast	960	0.960	0	0
Rising	978	0.978	0	0
BiLSTM-Attention	985	0.985	2	0.002

Table 7
Processing time for each sample in different stage.

Model	Syntactic Unit Sequences Extractation	Model Load	Get Sequence Feature	Model Detection	total
BiLSTM-Attention	8.54ms	3.76ms	1.69ms	10.28ms	24.27ms
JSDC Wang et al. (2015)	\	\	\	\	79.30ms
JaSt Fass et al. (2018)	\	\	\	\	226.86ms

This paper calculates the total time taken for each data sample to generate syntactic unit sequences from code parsing, load trained models (including word vector models and detection models) and generate word vector features of sequences, and prediction, which as shown in Table. 7.

We tested on 300 data samples, the average time for each sample is only 24.27 ms. The detection speed is fast and the cost is small. Compared with the model that takes the least time in Wang et al. (2015), as shown in Table. 7. The time of the detection model combined with dynamic features often reaches more than 1 second. Therefore, the model we proposed has obvious advantages in efficiency.

5. Conclusion

Based on the current research, this paper proposes a semantic-based malicious JavaScript code detection model. In order to combat obfuscate malicious JavaScript code, this paper transforms the code into an abstract syntax tree which is converted to the syntactic unit sequences, selects the FastText algorithm to extract the word vector features, and then uses the Bi-LSTM model with attention mechanism to discriminate against malicious JavaScript code. By comparing with some machine learning methods and related works, the BiLSTM-Attention model is at the leading level in all aspects of the indicators, and the accuracy is up to 0.974, the recall reached 0.974 under 5-fold cross-validation. It indicates that the model can effectively distinguish between malicious code and benign code. We only compared the experiments of a single machine learning algorithm instead of ensemble multiple classifiers which usually vote to generate a final prediction on whether a sample is malicious Wang et al. (2018). In Dietterich (2000), it describes that "ensembles can often perform better than any single classifier". This will be the direction we will consider later. In addition, compared with 25 anti-virus engines, the model has an advantage in the recall rate, revealing the superiority of the extracted features and the validity of the model. However, the model has a certain false positive rate, which may be caused by the single detection feature. Besides, the time of model detection is calculated, which proves that the model has the advantages of small resources cost.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Yong Fang: Writing - review & editing, Methodology, Conceptualization, Methodology, Writing - review & editing. **Cheng Huang:** Conceptualization, Methodology, Writing - review & editing. **Yu Su:** Investigation, Methodology, Data curation, Writing - original draft. **Yaoyao Qiu:** Investigation, Methodology, Software, Writing - original draft.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China under Grant 61902265, and the Fundamental Research Funds for the Central Universities.

References

Bojanowski, P., Grave, E., Joulin, A., Mikolov, T., 2017. Enriching word vectors with subword information. Trans. Assoc. Comput.Ling. 5, 135–146.
Chung, J., Gulcehre, C., Cho, K., Bengio, Y., 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv:1412.3555.
Cooper, K., 2018. Alexa. https://www.alexa.com/.
Cova, M., Kruegel, C., Vigna, G., 2010. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In: Proceedings of the 19th International Conference on World Wide Web. ACM, pp. 281–290.
Curtsinger, C., Livshits, B., Zorn, B.G., Seifert, C., 2011. Zozzle: Fast and precise in-browser JavaScript malware detection.. In: USENIX Security Symposium. San Francisco, pp. 33–48.
Dietterich, T.G., 2000. Ensemble methods in machine learning. In: International Workshop on Multiple Classifier Systems. Springer, pp. 1–15.
Fang, Y., Huang, C., Liu, L., Xue, M., 2018. Research on malicious JavaScript detection technology based on lstm. IEEE Access 6, 59118–59125.
Fass, A., Backes, M., Stock, B., 2019a. Hidenoseek: camouflaging malicious JavaScript in benign asts.
Fass, A., Backes, M., Stock, B., 2019b. Jstap: a static pre-filter for malicious JavaScript detection.
Fass, A., Krawczyk, R.P., Backes, M., Stock, B., 2018. Jast: fully syntactic detection of malicious (obfuscated) javascript. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 303–325.
Graves, A., Schmidhuber, J., 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Netw. 18 (5–6), 602–610.
Habtamu, A., 2019. A Hybrid Analysis and Detection of Android Malware Using Machine Learning and Blockchain Technology. ASTU.
Hidayat, A., 2018. Esprima master documentation.
Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.
HynekPetrak, 2017. Javascript malware collection. https://github.com/HynekPetrak/javascript-malware-collection.
Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., Vigna, G., 2013. Revolver: an automated approach to the detection of evasive web-based malware. In: Presented as Part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13), pp. 637–652.
Kim, H.C., Choi, Y.H., Lee, D.H., 2012. Jssandbox: a framework for analyzing the behavior of malicious JavaScript code using internal function hooking.. KSII Trans. Internet Inf. Syst. 6 (2).
antivirus lab, T., 2019.1. 2018 annual safety report.
Lubowicka, K., 2019.1. Javascript malware steals card details from up to 300 new websites.
Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, pp. 3111–3119.
Nawrocki, M., Wählisch, M., Schmidt, T. C., Keil, C., Schönfelder, J., 2016. A survey on honeypot software and data analysis. arXiv:1608.062497.
Ndichu, S., Ozawa, S., Misu, T., Okada, K., 2018. A machine learning approach to malicious JavaScript detection using fixed length vector representation. In: 2018 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1–8.
Rieck, K., Krueger, T., Dewald, A., 2010. Cujo: efficient detection and prevention of drive-by-download attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference. ACM, pp. 31–39.
Sarmah, U., Bhattacharyya, D., Kalita, J.K., 2018. A survey of detection methods for XSS attacks. J. Netw. Comput. Appl. 118, 113–143.
Schuster, M., Paliwal, K.K., 1997. Bidirectional recurrent neural networks. IEEE Trans. Signal Process. 45 (11), 2673–2681.
Seshagiri, P., Vazhayil, A., Sriram, P., 2016. Ama: static code analysis of web page for the detection of malicious scripts. Procedia Comput Sci 93, 768–773.
Shen, V.R., Wei, C.-S., Juang, T.T.-Y., 2018. Javascript malware detection using a high-level fuzzy petri net. In: 2018 International Conference on Machine Learning and Cybernetics (ICMLC), 2. IEEE, pp. 511–514.
Threatbook, 2019. Threatbook cloud sandbox.
Trendmicro, 2019.1. Javascript malware in spam spreads ransomware, miners, spyware, worm.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. In: Advances in Neural Information Processing Systems, pp. 5998–6008.
Wang, J., Xue, Y., Liu, Y., Tan, T.H., 2015. JSDC: a hybrid approach for JavaScript malware detection and classification. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. ACM, pp. 109–120.
Wang, W., Li, Y., Wang, X., Liu, J., Zhang, X., 2018. Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. Future Gener. Comput. Syst. 78, 987–994.
Wang, W., Shang, Y., He, Y., Li, Y., Liu, J., 2020. Botmark: automated botnet detection with hybrid analysis of flow-based and graph-based traffic behaviors. Inf. Sci. 511, 284–296.
Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., Zhang, X., 2014. Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans. Inf. Forensics Secur. 9 (11), 1869–1882.
wikipedia, 2019. Abstract syntax tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree.

Yong Fang received the Ph.D degree from Sichuan University, Chengdu, China, in 2010. He is currently a Professor with college of Cybersecurity, Sichuan University, China. His research interests include network security,Web security, Internet of Things, Big Data and artificial intelligence.

Cheng Huang received the Ph.D degree from Sichuan University, Chengdu, China, in 2017. From 2014 to 2015, he was a visiting student at the School of Computer Science, University of California, CA, USA. He is currently an Assistant Research Professor at the college of Cybersecurity, Sichuan University, Chengdu, China. His current research interests include Web security, attack detection, artificial intelligence.

Yu Su received the B.Eng. degree in information security from SichuanUniversity, Chengdu, China, in 2018, where she is currently pursuing the M.A. degree with the College of Cybersecurity. Her current research interests include Web development and network security.

Yaoyao Qiu received the B.Eng. degree in information engineering from Sichuan University, Chengdu, China, in 2016. She is currently pursuing the M.A. degree with the College of Electronics and Information, Sichuan University. Her current research interests include Web development and network security.