IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 6, 2018, accepted September 26, 2018, date of publication October 5, 2018, date of current version October 31, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2874098

Research on Malicious JavaScript Detection Technology Based on LSTM

YONG FANG[1], CHENG HUANG[1], LIANG LIU[1], AND MIN XUE[2]
[1]College of Cybersecurity, Sichuan University, Chengdu 610207, China
[2]College of Electronics and Information, Sichuan University, Chengdu 610065, China

Corresponding author: Cheng Huang (opcodesec@gmail.com)

This work was supported in part by the Fundamental Research Funds for the Central Universities under Grant 20826041B4249 and Grant 20826041B4252.

ABSTRACT The attacker injects malicious JavaScript into web pages to achieve the purpose of implanting Trojan horses, spreading viruses, phishing, and obtaining secret information. By analyzing the existing researches on malicious JavaScript detection, a malicious JavaScript detection model based on LSTM (Long Short-Term Memory) is proposed. Features are extracted from the semantic level of bytecode, and the method of word vector is optimized. It can distinguish malicious JavaScript code and combat obfuscated code effectively. Experiments showed that the accuracy of detection model based on LSTM is 99.51%, and the F1-score is 98.37%, which is better than the existing model based on Random Forest and SVM algorithm.

INDEX TERMS JavaScript, malicious code detection, bytecode, word vector, LSTM.

I. INTRODUCTION

With the Web application occupying the mainstream market with Browser/Server (B/S) architecture, browsers and web pages have become an essential channel for the spread of malicious code. Attackers use website code vulnerability, third-party application vulnerabilities, browser vulnerabilities, and operating system vulnerabilities to perform cross-site scripting attacks on websites, inject web Trojans, tamper with web pages, Phishing, and steal personal information. According to the "2017 China Cyber Security Report" [1] published by the Information and Network Security Department of the Rising UN Home Information Center in January 2018, Rising "cloud security" intercepted a total of 80.11 million malicious URLs worldwide in 2017. Among them, there are 42.75 million linked websites and 37.35 million fraud websites. The total number of malicious URLs in China is 13.5 million, which is second only to the 26.684 million in the United States. The malicious content contained in malicious web pages which could easily expose visitors to network attacks unwittingly, such as virus transmission, Trojan implantation, information leakage, etc. Their malicious code is mainly script language such as JavaScript and VBScript. To avoid detection, these malicious scripts are also obfuscated in different encode methods. Therefore, this article proposed an LSTM-based JavaScript malicious code detection method; the main work is as follows:

1) The paper studies the extraction of bytecode sequences based on the V8 engine. However, because the DOM and BOM objects in the browser environment cannot be identified, it is proposed to import the jsdom package library of Nodejs for simulation.
2) The paper presents word-vector extraction of bytecode based on the word2vec model. To make it more suitable for the current detection model, the TF-IDF algorithm is proposed to optimize the rejection of high-frequency words.
3) The study introduces the Long Short-Term Memory algorithm to detect JavaScript malicious code. The effectiveness of the proposed model was evaluated by constructing a detection model and setting up an experimental environment.

II. RELATED RESEARCH

Malicious JavaScript code detection method is an integral part of malicious web page detection, and there are many research results at present. The most two important factors are the feature extraction and detection model construction.

There are roughly six types of malicious code features extracted from different layers. They are feature codes, code statistics features, sequence features, graph features, code semantic features, and opcode frequency features. The well-known Snort intrusion detection system exploits pattern matching to perform attack detection [2]. The statistical characteristics of code refer to the static analysis of malicious code, and the keyword frequency, special character frequency, and the code length in malicious code.

In 2009, Likarish *et al.* [3] selected 65 statistical features from each code file as input, including the number of readable sequences, the frequency of each JavaScript keyword, the length of the script, the average number of characters per line, and the Unicode symbol. The number etc. evaluates the readability of the code and then exploits machine learning to detect the obfuscated malicious JavaScript code. Typical sequence characteristics include system call sequence characteristics, system state sequence characteristics, and (risk) function call sequences. For example, Kim *et al.* [4] proposed dynamic analysis based on malware behavior sequence (API call sequence) and sequence comparison algorithm to implement malware detection and classification. Graph features refer to flow-based analysis abstractions for variously related graphs, such as control-flow graphs (CFG), data flow diagram, key API graphs, etc., such as Fan *et al.* [5] using static analysis techniques to source code Abstracting into a function call graph, and then using the graph editing distance to analyze the similarity of the malicious code. Code semantic features are abstract features from the perspective of instruction semantics. Such as Christodorescu *et al.* [6] through the static analysis of instruction semantic information, the use of ternary operators to construct features to deal with the confusion of garbage code insertion and other means. The operating code frequency characteristic refers to the frequency distribution of the calculated operation code after the code is disassembled into an operation code (also called machine code). For example, Zhanjun [7] first disassembled malicious codes into opcodes and then used N-gram algorithm to extract features. Dynamic analysis methods with machine learning algorithms are used to detect malicious JavaScript with different models [8]–[11], but they did not provide any bytecode features from JavaScript code.

About detection technology, from the traditional pattern-based matching to the now-fabricated machine-based learning, the detection of malicious code is moving towards a more automatic and intelligent direction. The requirement for detection results are not only the ability to accurately and ultimately identify the known types of attacks must be able to fight against all kinds of potential and suspicious attacks.

III. MALICIOUS JAVASCRIPT FEATURE EXTRACTION

A. BYTECODE SEQUENCE EXTRACTION BASED ON V8 ENGINE

V8 is Google's open source high-performance JavaScript engine, written in C++ and used in Google Chrome, the open source browser from Google. V8 can run standalone, or can be embedded into any C++ application. There are two main reasons for using the V8 engine for bytecode extraction:

The V8 compiler is lazy. If a function is defined but not invoked, V8 will not interpret it, effectively reducing the impact of redundant code.

The V8 engine introduced the bytecode interpreter not long ago. At present, there is no any research on JavaScript malicious code with V8 bytecode technology.

1) SIMULATION OF DOM AND BOM OBJECTS

Most JavaScript codes in web pages need to implement web page interaction effects through DOM objects, and BOM objects to control some browser behaviors, but the V8 engine only parses the built-in JavaScript objects and methods [12] for objects in the browser environment. DOM and BOM are not recognized (the Nodejs environment provides a Global object), that is, when NodeJs directly executes document.getElementById statement, it will report an error because the document object cannot be found, and the execution of the subsequent statement stops the sequence of bytecode.

The solution is to introduce the packaging library jsdom of Node.js, which is a pure JavaScript implementation of many web standards, especially the WHATWG DOM and HTML standards, which can simulate the browser rendering engine to parse documents into DOM. It could be realized by adding the following code at the beginning of the code:

```
DOMSeg = '"const jsdom = require("jsdom");
const {JSDOM } = jsdom;
const document = (new JSDOM()).window.document;
const window = (new JSDOM()).window;'"
```

Although the new code redefine the document object and use document.getElementById('elementId').value to get the property methods value of the object. But there is an error reporting that the value property does not exist because a specific elementId element cannot be found. For this kind of situation, this article modified the source code of jsdom, for such a method does not need to return true and exact objects, but directly returns a custom object. The modified part of the document.getElementById method code is as follows:

```
function customObject(i) {
this.value = i;
this.innerHTML = i;
...
}
Document.prototype.getElementById =
function getElementById(elementId) {
    return new customObject(elementId);
};
```

2) BYTECODE SEQUENCE EXTRACTION

There are 174 bytecode in V8, including operators (eg. Add, Typeof), and attribute loaders (eg. LdaNamedProperty). The operands of the register include both input and output and are specified by their bytecode. The Ignition bytecode interpreter exploits registers r0, r1, r2..... and accumulator registers, but almost all bytecode use accumulator registers.

Currently, there are two ways to generate bytecode using the V8 engine: 1) In D8 or Node.js (requires 8.3 or later) environment, print by adding –print-bytecode to the command line parameters; 2) For Chrome, to launch Chrome program from the command line, use –js-flags="–print-bytecode" to print. The advantage of the former is that there is no need to

launch the browser, just run the JavaScript code directly, but you need to simulate the DOM and BOM objects; the latter is the opposite.

Take a simple piece of JavaScript code as an example. Use nodejs to execute it. At the same time, add the –print-bytecode command line parameter to print its bytecode. The result is shown in Figure 1.

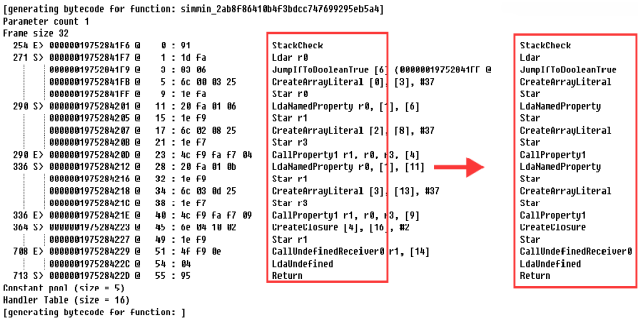

FIGURE 1. Sample bytecode sequence.

TABLE 1. Occurrence frequency of Top10 bytecode.

Bytecode	The proportion	Meaning
Star	24.73%	Register-accumulator transfer
Wide	22.53%	Extended width operands
CreateClosure	9.12%	Close distribution
StaCurrentContextSlot	7.45%	Context operation
LdaConstant	6.75%	Load accumulator
LdaSmi	6.35%	Load accumulator
Load accumulator	5.16%	Context operation
Context operation	1.71%	Property loading operation
Ldar	1.53%	Register-accumulator transfer
Register-accumulator transfer	1.43%	Global
other	13.24%	None

By counting the number of occurrences of bytecode in all sample bytecode, Table 1 lists the top 10 bytecode that have the highest frequency of occurrence, including the proportion of occurrences and their meaning.

B. WORD VECTOR EXTRACTION BASED ON WORD2VEC

1) BASIC PRINCIPLES OF WORD2VEC

Word2vec is a word vector feature training tool open sourced by Google in 2013 [13]. It combines the advantages of neural network language model and log linear, and exploits Distributed Representation as a representation of a word vector that expresses a word with a continuous, dense vector.

Word2vec provides Continuous Bag-Of-Words (CBOW) and Skip-gram training models for the calculation of word vectors. It also provides two sets of optimization methods, Hierarchy Softmax and Negative Sampling, to provide training effectiveness of word vector. Therefore word2vec has a total of four training frameworks: CBOW+HS, Skig-gram+HS, CBOW+NS, and Skip-gram+NS.

The schematic diagrams of the CBOW model and the Skip-gram model are shown in Fig. 2. They all contain an input layer, a projection layer, and an output layer.

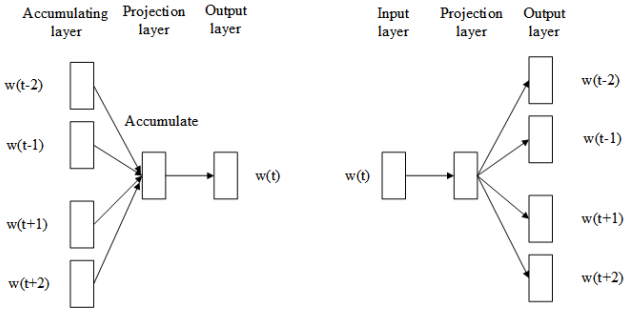

FIGURE 2. CBOW model (left) and Skip-gram model (right).

Among them, the CBOW model predicts the current word w_t on the premise that the current word context $w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}$ is known; the exact opposite of the Skip-gram model is to predict its context $w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}$ on the premise that the current word w_t is known [14].

The following takes sample $(Context(w), w)$ as an example to introduce the principle of the CBOW model based on the Hierarchy Softmax optimization method. Where $(Context(w), w)$ can be expressed as formula (1):

$$Context(w) = (w_1, \ldots, w_2, \ldots, w_{2c}) \tag{1}$$

There are c words before and after word$w$.

The input to the CBOW model is the word vector $v(w_1), v(w_2) \ldots, v(w_{2c}) \in R^m$ of $2c$ words in $Context(w)$, where m represents the dimension of the word vector. The projection layer accumulates $2c$ words vector of the input layer, as shown in formula (2):

$$X_w = \sum_{i=t-c}^{t+c} v(w_i) \in R^m \tag{2}$$

The output layer is a Huffman tree. The words appearing in the corpus are the leaf nodes of the binary tree. The number of occurrences of each word in the corpus is the weight of the corresponding node. In this Huffman tree, there are N child nodes (the size of N is the number of words in the corpus) and $N - 1$ non-leaf nodes. The vector corresponding to the leaf node is its word vector, and the vector corresponding to the non-leaf node is the auxiliary vector.

The CBOW model is known as the context of the current word w, so as to predict the current word, its objective function can be expressed as $p(w|Context(w))$.

The basic idea of Hierarchical Softmax is that for the word w in the dictionary D, there is a unique path p^w from the root node to the leaf node (i.e. the word w) in the Huffman tree, and each branch on the path is treated as a double classification. With a certain probability, the cumulative multiplier of these probabilities is the value of the objective function. The calculation of the objective function is based on such rules:

1) From the root node to the leaf node, each branch experienced in the middle is a binary classification;
2) All left nodes are marked as negative and the label is 1; all right nodes are marked as positive and label is 0;

3) The path from the root node to the corresponding child node of the word w is p^w;
4) The node in path p^w is $p_1^w, p_2^w, \ldots, p_{l^w}^w$, and the number is l^w, where is the root node $p_{l^w}^w$ and $p_{l^w}^w$ is the node corresponding to the word w;
5) The Huffman code from the root node to the corresponding child node of the word w is $d_2^w, d_3^w, \ldots, d_{l^w}^w \in \{0, 1\}$, d_j^w indicating the encoding corresponding to the node j in path p^w, where the root node does not correspond to the encoding;
6) The vector corresponding to the non-leaf node in path p^w is $\theta_1^w, \theta_2^w, \ldots, \theta_{l^w-1}^w \in R^m$, θ_j^w indicating the vector corresponding to the node j in path p^w.

According to logistic regression, the probability that a node is classified as positive is $\sigma(\mathbf{x}_w^{\mathbf{T}}\theta) = \frac{1}{1+e^{-\mathbf{x}_w^{\mathbf{T}}\theta}}$, and that of negative is $1-\sigma(\mathbf{x}_w^{\mathbf{T}}\theta)$, so the objective function of the CBOW model can be expressed as formula (3):

$$p(w|Context(w)) = \prod_{j=2}^{l^w} p(d_j^w|\mathbf{x}_w, \theta_{j-1}^w) \tag{3}$$

$$p(d_j^w|\mathbf{x}_w, \theta_{j-1}^w) = \begin{cases} \sigma(\mathbf{x}_w^{\mathbf{T}}\theta_{j-1}^w), & d_j^w = 0; \\ 1 - \sigma(\mathbf{x}_w^{\mathbf{T}}\theta_{j-1}^w), & d_j^w = 1, \end{cases} \tag{4}$$

2) WORD VECTOR EXTRACTION AND OPTIMIZATION

Word2vec is a tool designed for natural language processing. It is based on some word features in the native language environment. For example, in a large corpus, some common words (such as the stop words "of" and "yes") are very frequently appearing. These words provide very little useful information, and the word vectors correspond to these words. No significant changes occur when training on many samples. Therefore, for the processing of such high-frequency words in the corpus, word2vec exploits a sub-sampling method to increase the training speed. The specific approach is to give a word frequency threshold parameter, and the word w will be discarded with the probability of formula (5)

$$prob(w) = 1 - (\sqrt{\frac{t}{f(w)}} + \frac{t}{f(w)}) \tag{5}$$

$$f(w) = \frac{counter(w)}{\sum_{u \subset D} counter(u)}, \quad w \subset D \tag{6}$$

Therefore, when the current word w is processed, a value of $ran = \sqrt{\frac{t}{f(w)}} + \frac{t}{f(w)}$ is calculated, and then a random real number r of (0, 1) is generated. If $r > ran$, the current word w is discarded. Since the probability of producing a random number smaller than ran in the interval (0, 1) is $1 - ran$, the above approach is equivalent to rejecting the word w with the probability of $1 - ran$.

The rejection of high-frequency words in Word2vec with a certain probability is not applicable to the model in this article for two reasons:

1) The sample of the word vector training in this paper is not a natural language, but a sequence of bytecode. There is no stop word like the natural language environment. That is, each bytecode provides useful information.
2) After observing the bytecode generated by malicious JavaScript code, it was found that the frequency of some bytecode is very high. In the past research based on opcodes to detect malicious code, some malicious codes were found to be used to avoid detection. A large number of repeated instructions, so the performance of high-frequency words has a specific influence on the classification effect of the model.

Therefore, the goal of the optimization in this paper is to keep the effective high-frequency words as much as possible and discard the invalid high-frequency words. The measurement of the high-frequency words is performed through the TF-IDF algorithm. The three core concepts in the TF-IDF algorithm are TF (Term Frequency), DF (Document Frequency), and IDF (Inverse Document Frequency). TF is the frequency of a feature word appearing in a document, it can well represent the importance of the current feature word to a document; DF refers to the frequency of a feature word appearing in the document set, it can well represent the distribution characteristics of the current feature word in the document set; IDF refers to the frequency of reverse documents. It focuses on the ability to distinguish the current feature words. The TD-IDF algorithm is currently the most commonly used eigenvalue weight calculation method. Its calculation formula can be expressed as formula (7):

$$Weight(w) = TF(w) * \log(\frac{1}{DF(w)} + 0.01) \tag{7}$$

The principle that the original word2vec rejects high-frequency words is that the higher the word frequency is, the higher the probability of rejection is. The optimized model introduces TD-IDF weights, which reduces the probability of discarding valid high-frequency words. The final rejection probability function can be expressed as formula (8):

$$prob(w) = \frac{1}{t\sqrt{10^{Weight(w)} - 0.01} + 10^{Weight(w)}} \tag{8}$$

IV. MALICIOUS JAVASCRIPT DETECTION MODEL

A. MODEL TRAINING AND DETECTION PROCESS

The detailed flow of the malicious JavaScript detection model at the training and detection stage is shown in Figure 3.

In the training phase, the model first preprocesses the malicious sample set and the normal sample set, removes the duplicate samples in the sample set, and performs JavaScript code extraction on the sample files in the HTML format. DOM and BOM objects are called in the JavaScript code. Perform object simulation. Then use the V8 parsing engine under the Nodejs platform to parse the JavaScript code to generate bytecode, remove the redundant parts while extracting the bytecode sequences, and mark them as text files. Then use the tagged bytecode sequence file to train the improved word2vec model and generate word vectors. Finally, the processed word

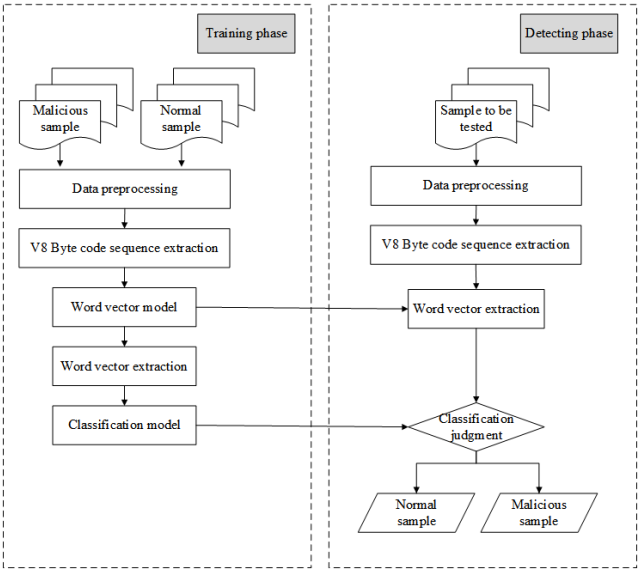

FIGURE 3. Flow chart of model training and detection.

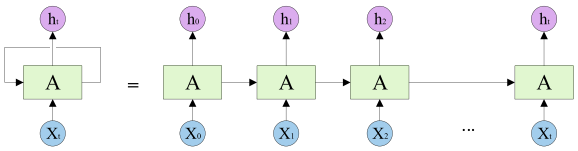

FIGURE 4. RNN logical structure.

vector is used as the input of neural network LSTM to train the JavaScript malicious code detection model.

In the detection stage, the data preprocessing and the V8 bytecode extraction is all in the same training phase. The word vector extraction part extracts the word vectors according to the word vector model trained in the training phase. The final classification decision part is trained according to the training phase. The JavaScript malicious code detection model performs the determination of malicious code and normal code.

B. BASIC PRINCIPLES OF LSTM

Long Short-Term Memory (LSTM) is an improved type of Recurrent Neural Networks (RNN) to resolve the problem that RNN cannot handle long-distance dependence [15]. The traditional neural network model is based on the assumption that the input from the input layer to the hidden layer to the output layer is independent of each other. However, such assumptions are unreasonable for the handling of many practical problems, because most of the information has the chronological and sequential nature, that is, the task of information at the moment to influence this moment. RNN has emerged to solve this problem; it can make information in the network again. The logical structure of the RNN is shown in Figure 4.

Theoretically, RNN can process any length of sequence data, but in practical applications, RNN cannot be implemented ideally because of the long-range dependence of vanishing gradient and exploding gradient [16]. The disappearance of the gradient means that if the gradient is small (less than 1), the gradient will drop to almost no effect on the reference parameter after multiple layers of iterative calculations, similar to $(0.99)^{100} \rightarrow 0$; a gradient explosion means that if the gradient is larger (greater than 1), it passes through multiple layers. After iterative calculation, the gradient will rise to very large, similar to $(1.01)^{100}$ is not small.

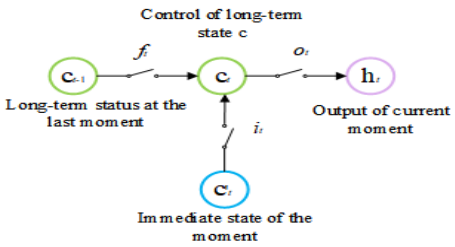

FIGURE 5. LSTM unit control diagram.

The LSTM improves the RNN by introducing a controlled self-loop to create a path that allows the gradient to continue flowing for long periods of time. The original RNN is very sensitive to short-term input because it has only one state in the hidden layer. LSTM saves the long-term state by adding a cell state and introduces three "gates" to control the state of the cell. , as shown in Figure 5.

1) input gate i_t: determine the current time network input X_t saved to the unit state c_t;
2) Oblivion gate f_t: Determine the unit status of the last moment c_{t-1} How much is reserved to the current moment c_t;
3) Output gates o_t: Control unit status c_t How much of the current output value h_i is output to the LSTM.

C. LSTM-BASED CLASSIFICATION MODEL

LSTM-based classification model training steps as below:

1) Before starting training the LSTM classification model, prepare the initial training sample set A0.
2) Perform network configuration on the classification model, including activation function, loss function, and optimization function.
3) Arrange the parameters of the classification model, including calculating the number of samples and training periods when the primary gradient falls.
4) Calculate the output value of each neuron in advance, namely f_t, i_t, c_t, o_t and h_t.
5) Calculate the error based on the loss function. At the beginning of training, the output value will not be consistent with the predicted value, and the error term value of each neuron needs to be calculated according to the loss function.
6) According to the gradient guide of the loss function and the optimization function, the network weight parameter is updated. Similar to the traditional RNN, the back propagation calculation of LSTM error terms includes

both spatial and temporal layers. At the spatial level, the error term is propagated to the upper layer of the network. At the time level, the current time before t is calculated. The moment of error.

7) Repeat steps 4) - 6) until the error is less than the given value to form the final classification model.

The specific training flow chart of the LSTM classification model is shown in Figure 6.

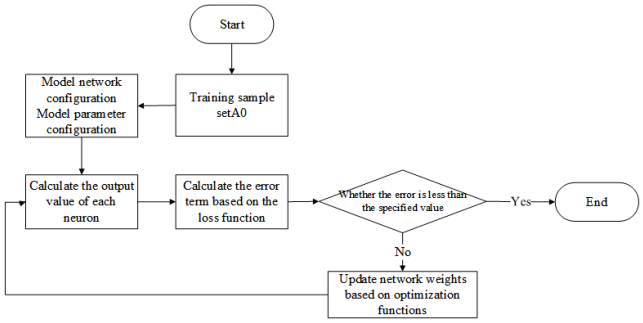

FIGURE 6. LSTM classification model training flowchart.

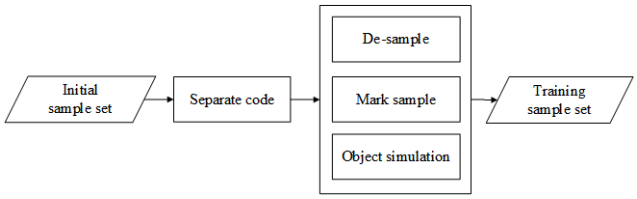

FIGURE 7. Data preparation module function design.

V. EXPERIMENTAL RESULTS AND ASSESSMENT

A. DATA SETS AND DATA PREPROCESSING

In order to evaluate the effectiveness of the proposed model, we crawled 23,993 files for the experimental data sets, including 20,000 benign files and 3993 malicious files. The benign files were extracted from different Web pages of the top 200 Alexa domains (https://www.alexa.com/topsites), and malicious files were crawled from malicious platform VX Heavens (http://vxvault.net/ViriList.php) and the code repository GitHub (https://github.com/geeksonsecurity/js-malicious-dataset).

We use 10-fold cross validation to evaluate the detection accuracy. The data sets are randomly partitioned into 10 equal-size parts. At each iteration, one single part is taken as the testing data, and the other parts are used as training data. The final result is the average over all iterations.

The experiment steps of malicious JavaScript detection model at the training and detection stage are shown in Figure 3. Before training, you need to filter duplicate code samples, label mark code samples, and simulate DOM and BOM objects. The functional design of the data preparation module is shown in Figure 7. And detail steps are described as below:

1) JavaScript code separation: For HTML files, you need to separate JavaScript code and save it as a JavaScript file. BeautifulSoup is a very useful web parsing library in Python. It is used here to extract the JavaScript code in the <script> tag.
2) Code Sample Deduplication: The sample sets contain a large number of duplicate code files, especially in malicious samples, in order to better verify the accuracy and effectiveness of the model, the duplicate sample files need to be removed.
3) Sample Mark: The malicious JavaScript code sample is marked as malicious and the normal JavaScript code sample is marked as benign.
4) DOM and BOM Object Simulation: Install the Nodejs module jsdom and make certain modifications to the jsdom source code to suit the current conditions.

B. MODEL VALIDITY

In order to evaluate the effect of proposed model, the experiment introduced multiple algorithms to get the highest accurate. The four algorithms contain random forest, SVM, Naive Bayes and LSTM algorithm. In this paper, we train the LSTM for 39 epochs, the batch_size is set to 32, and embedding_dims is 128.

The total results for four classification algorithms are shown in Table2. We use accuracy, precision, recall and other normal evaluation method to classify the malicious and benign code.

TABLE 2. Test results of different classification algorithms.

Classification algorithm	Accuracy	Precision	Recall	Value of F_1
Random forest	98.97%	99.84%	93.10%	96.35%
SVM	97.45%	98.28%	83.99%	90.57%
Naive Bayes	95.41%	90.33%	76.80%	83.02%
LSTM	99.51%	99.55%	97.21%	98.37%

From Table 2, the accuracy, precision, recall, and value of the LSTM algorithm are all higher than those of the SVM algorithm. Although the accuracy of the LSTM algorithm is slightly lower than random forest algorithm.

Overall, the LSTM algorithm has the best classification performance among these four algorithms, followed by the random forest algorithm and Naive Bayes algorithm. The above four algorithms are based on the ROC curve (Receiver Operating Characteristic Curve) as shown in Figure 8.

As can be seen from Figure. 8, the area under the ROC curve (AUC) of the LSTM algorithm is also larger than that of the random forest algorithm, SVM algorithm, and Naive Bayes algorithm, which is proved that malicious JavaScript detection technology based on LSTM is useful, with highly accurate.

Next, we downloaded the previous data sets and further evaluate the effectiveness of malicious JavaScript detection model. Wang Wei and other students from Beijing Jiaotong University disclosed that 685 malicious codes that were disclosed on the research team's website (http://infosec.bjtu.edu.cn/wangwei/?page_id=85). Then, we can

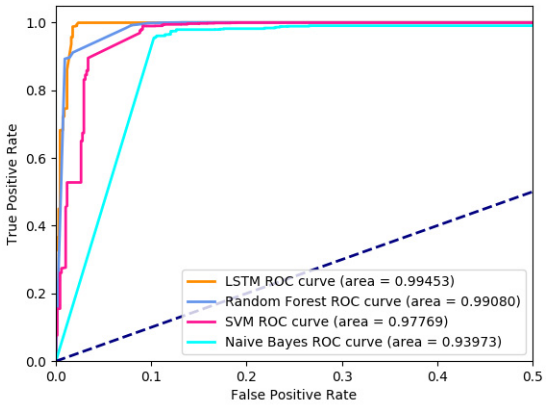

FIGURE 8. ROC curves for different classification algorithms.

combine the malicious JavaScript codes and normal benign codes as test data sets to further evaluate the model. 685 confused malicious codes and 700 normal codes were selected. We used the previous datasets (23,993 files) to train the model, and then automatic classify these current test data (1385 files).

At last, the LSTM-based JavaScript malicious code detection model has achieved an accuracy rate of 98.19%, an accuracy rate of 98.53%, a recall rate of 97.81%, which proved that the model can effectively detect confused with malicious JavaScript code.

VI. CONCLUSION

Based on the existing detection technology, this paper proposes a JavaScript malicious code detection model based on LSTM. To combat the obfuscation technique in malicious code, we analyzed it from the bytecode level based on the V8 parsing engine and used the optimized word2vec algorithm to extract word vector features. Finally, we used the deep learning algorithm LSTM to classify JavaScript malicious code. By testing the detection model and comparing with other classification algorithms, the feasibility and effectiveness of the LSTM-based JavaScript malicious code detection model proposed in this paper are verified. However, the number of bytecode extracted based on the V8 engine is nearly 100,000, even if the amount of code is small. These bytecode contain a lot of redundancy. Although this model selects a part of the bytecode for analysis, other bytecode may also contribute to the detection results. Therefore, the bytecode still has a valid sequence which need to improve.

REFERENCES

[1] National Information Center. (2018). *China Network Security Report 2017*. [Online]. Available: http://www.rising.com.cn/2018/baogao2017/2017.pdf

[2] L. Hua *et al.*, "Technique of detecting malicious executables via behavioural and binary signatures," *Appl. Res. Comput.*, vol. 28, no. 3, pp. 1127–1129, 2011.

[3] P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques," in *Proc. 4th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2010, pp. 47–54.

[4] H. Kim, J. Kim, Y. Kim, I. Kim, K. J. Kim, and H. Kim, "Improvement of malware detection and classification using API call sequence alignment and visualization," *Cluster Comput.*, vol. 20, no. 4, pp. 1–9, 2017.

[5] Y. Fan *et al.*, "Malware detection based on graph edit distance," *Wuhan Univ. Nat. Sci. Ed.*, vol. 59, no. 5, pp. 454–457, 2013.

[6] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proc. IEEE Symp. Secur. Privacy*, May 2005, pp. 32–46.

[7] L. Zhanjun, *Research on Static Malicious Code Detection Method Based on Opcode Sequence*. Harbin Institute of Technology, Harbin, China, 2013.

[8] K. Borgolte, C. Kruegel, and G. Vigna, "Meerkat: Detecting website defacements through image-based object recognition," in *Proc. USENIX Secur. Symp.*, 2015, pp. 595–610.

[9] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proc. ACM 19th Int. Conf. World Wide Web.*, 2010, pp. 281–290.

[10] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, "WebWitness: Investigating, categorizing, and mitigating malware download paths," in *Proc. USENIX Secur. Symp.*, 2015, pp. 1025–1040.

[11] B. Eshete *et al.*, "EKHunter: A counter-offensive toolkit for exploit kit infiltration," in *Proc. NDSS*, 2015, pp. 1–15.

[12] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the incoherencies in Web browser access control policies," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 463–478.

[13] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 26, 2013, pp. 3111–3119.

[14] Z. Lian, "The working principle and application of Word2vec," *SCI-Tech Inf. Develop. Economy*, vol. 25, no. 2, pp. 145–148, 2015.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[16] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1310–1318.

YONG FANG received the Ph.D. degree from Sichuan University, Chengdu, China, in 2010. He is currently a Professor with the College of Cyber security, 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 Cyber security, Sichuan University. His current research interests include Web security, span network, social privacy, and artificial intelligence.

LIANG LIU received the M.A. degree from Sichuan University, Chengdu, China, in 2010. He is currently an Assistant Professor at the College of Cyber security, Sichuan University, China. His current research interests include malicious detection, network security, and system security and artificial intelligence.

MIN XUE received the B.Eng. degree in information engineering from Sichuan University, Wuhan, China, in 2015. 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.